Universität Trier
Zentrum für Informations-, Medien- und Kommunikationstechnologie

Bernhard Baltes-Götz & Johannes Götz

Einführung in das Programmieren mit Java 13

2020 (Rev. 210424)

Herausgeber: Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK)
an der Universität Trier
Universitätsring 15
D-54286 Trier
WWW: zimk.uni-trier.de
E-Mail: zimk@uni-trier.de

Autoren: Bernhard Baltes-Götz & Johannes Götz
E-Mail: baltes@uni-trier.de

Copyright © 2020; ZIMK

Vorwort

Dieses Manuskript basiert auf der Begleitlektüre zum Java-Einführungskurs, den das Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK) an der Universität Trier im Wintersemester 2019/2020 angeboten hat.

Inhalte und Lernziele

Die von der Firma **Sun Microsystems** (mittlerweile von der Firma **Oracle** übernommen) entwickelte und 1995 veröffentlichte Programmiersprache Java ist zwar mit dem Internet groß geworden, hat sich jedoch mittlerweile als universelle, für vielfältige Zwecke einsetzbare Lösung etabliert, die als de-facto - Standard für die *plattformunabhängige* Entwicklung gelten kann. Unter den objektorientierten Programmiersprachen hat Java den größten Verbreitungsgrad, und das objektorientierte Paradigma der Software-Entwicklung hat sich praktisch in der gesamten Branche als *State of the Art* etabliert.

Die Entscheidung der Firma Sun, Java beginnend mit der Version 6 als **Open Source** unter die GPL (*General Public License*) zu stellen, ist in der Entwicklerszene positiv aufgenommen worden und trägt zum anhaltenden Erfolg der Programmiersprache bei.

Allerdings steht Java nicht ohne Konkurrenz da. Nach dem fehlgeschlagenen Versuch, Java unter der Bezeichnung *J++* als Windows-Programmiersprache zu etablieren, hat die Firma Microsoft mittlerweile mit der Programmiersprache C# für das .NET-Framework ein ebenbürtiges Gegenstück erschaffen (siehe z. B. Baltes-Götz 2019). Beide Konkurrenten inspirieren sich gegenseitig und treiben so den Fortschritt voran.

Außerdem sind mittlerweile neben Java etliche weitere Sprachen zur Entwicklung von Programmen für die Java-Laufzeitumgebung entstanden (z. B. Clojure, Groovy, JRuby, Jython, Kotlin, Scala). Sie bieten dieselbe Plattformunabhängigkeit wie Java und können teilweise alternative Programmiertechniken wie das funktionale Programmieren (früher) unterstützen, weil sie nicht zur Abwärtskompatibilität verpflichtet sind. Diese Vielfalt (vergleichbar mit der Wahlfreiheit von Programmiersprachen für die .NET - Plattform) ist grundsätzlich zu begrüßen. Allerdings ist Java für allgemeine Einsatzzwecke nicht zuletzt wegen der großen Verbreitung und Unterstützung weiterhin zu bevorzugen. Nachhaltig relevante Programmiertechniken sind früher oder später auch in Java verfügbar. So ist z. B. das funktionale Programmieren seit der Version 8 auch in Java möglich.

Das Manuskript beschränkt sich auf die *Java Standard Edition* (JSE) zur Entwicklung von Anwendersoftware für Arbeitsplatzrechner, auf die viele weltweit populäre Softwarepakete setzen (z. B. IBM SPSS Statistics, Matlab). Daneben gibt es sehr erfolgreiche Java-Editionen bzw. - Frameworks für unternehmensweite oder serverorientierte Lösungen. Neben der *Java Enterprise Edition* (JEE), die jüngst von der Firma Oracle an die Open Source Community (vertreten durch die Eclipse Foundation) übergeben wurde, ist hier vor allem das Spring-Framework zu erwähnen. Eher auf dem Rückzug ist die *Java Micro Edition* (JME) für Kommunikationsgeräte mit beschränkter Leistung.

Moderne Smartphones und Tablets zählen nicht mehr zu den Geräten mit beschränkter Leistung. Sofern diese Geräte das Betriebssystem Android benutzen, kommt auch hier zur Software-Entwicklung häufig die Programmiersprache Java zum Einsatz (siehe z. B. Baltes-Götz 2018).

Im Manuskript geht es nicht um Kochrezepte zur schnellen Erstellung effektvoller Programme, sondern um die systematische Einführung in das Programmieren mit Java. Dabei werden wichtige Konzepte und Methoden der Software-Entwicklung vorgestellt, wobei die objektorientierte Programmierung einen großen Raum einnimmt.

Voraussetzungen bei den Teilnehmern

- **Programmierkenntnisse**
  Programmierkenntnisse werden *nicht* vorausgesetzt. Leser *mit* Programmiererfahrung werden sich bei den ersten Kapiteln eventuell etwas langweilen.
- **EDV-Plattform**
  Wir arbeiten im Kurs mit PCs unter Microsoft Windows. Weil dabei ausschließlich Java-Software zum Einsatz kommt, ist die Plattformunabhängigkeit jedoch garantiert.

Software zum Üben

Für die unverzichtbaren Übungen verwenden wir das Java SE Development Kit in den Versionen 8, 11 und 13 sowie die Entwicklungsumgebung *IntelliJ IDEA* der Firma JetBrains in der Community Edition 2019.2. Die genannte Software ist kostenlos für alle signifikanten Plattformen (z. B. Linux, MacOS, UNIX, Windows) im Internet verfügbar. Nähere Hinweise zum Bezug, zur Installation und zur Verwendung folgen im Manuskript.

Manuskript und Dateien zum Kurs

Die aktuelle Version dieses Manuskripts ist zusammen mit den behandelten Beispielen und Lösungsvorschläge zu den Übungsaufgaben auf dem Webserver der Universität Trier von der Startseite (http://www.uni-trier.de/) ausgehend folgendermaßen zu finden:

IT-Services > Downloads & Broschüren >
Programmierung > Einführung in das Programmieren mit Java

Leider blieb zu wenig Zeit für eine sorgfältige Kontrolle des Textes, sodass einige Fehler und Mängel verblieben sein dürften. Entsprechende Hinweise an die Mail-Adresse

baltes@uni-trier.de

werden dankbar entgegengenommen.[1]

Trier und Bruchsal, im März 2020 Bernhard Baltes-Götz und Johannes Götz

[1] Für zahlreiche Hinweise auf mittlerweile behobene Fehler möchten wir uns bei Paul Frischknecht, Andreas Hanemann, Peter Krumm, Michael Lehnen, Lukas Nießen, Rolf Schwung und Jens Weber herzlich bedanken.

Inhaltsverzeichnis

VORWORT III

INHALTSVERZEICHNIS V

1 EINLEITUNG 19

1.1 Beispiel für die objektorientierte Software-Entwicklung mit Java 19
1.1.1 Objektorientierte Analyse und Modellierung 19
1.1.2 Objektorientierte Programmierung 25
1.1.3 Algorithmen 26
1.1.4 Startklasse und main() - Methode 27
1.1.5 Zusammenfassung zum Abschnitt 1.1 29

1.2 Java-Programme ausführen 29
1.2.1 Java-Laufzeitumgebung installieren 29
1.2.2 Konsolenprogramme ausführen 34
1.2.3 Ausblick auf Anwendungen mit grafischer Bedienoberfläche 35
1.2.4 Ausführung auf einer beliebigen unterstützten Plattform 38

1.3 Die Java-Softwaretechnik 38
1.3.1 Herkunft und Bedeutung der Programmiersprache Java 39
1.3.2 Quellcode, Bytecode und Maschinencode 40
1.3.3 Die Standardklassenbibliothek der Java-Plattform 42
1.3.4 Java-Editionen für verschiede Einsatzszenarien 43
1.3.5 Update- und Lizenzpolitik von Oracle 44
1.3.6 Wichtige Merkmale der Java-Softwaretechnik 45
1.3.6.1 Objektorientierung 45
1.3.6.2 Portabilität 45
1.3.6.3 Sicherheit 46
1.3.6.4 Robustheit 47
1.3.6.5 Einfachheit 47
1.3.6.6 Multithreading 48
1.3.6.7 Netzwerkunterstützung 48
1.3.6.8 Performanz 48
1.3.6.9 Beschränkungen 48

1.4 Übungsaufgaben zum Kapitel 1 49

2 WERKZEUGE ZUM ENTWICKELN VON JAVA-PROGRAMMEN 51

2.1 Aktuelles OpenJDK installieren 51
2.1.1 OpenJDK 13 der Firma Oracle 52
2.1.2 OpenJDK-Distributionen mit langfristiger Update-Versorgung 52

2.2 Java-Entwicklung mit JDK und Texteditor 53
2.2.1 Editieren 53
2.2.2 Übersetzen 55
2.2.3 Ausführen 57
2.2.4 Suchpfad für class-Dateien 58
2.2.5 Programmfehler beheben 59

2.3 IntelliJ IDEA Community installieren 61

2.4 Java-Entwicklung mit IntelliJ IDEA 63
2.4.1 Erster Start 63
2.4.2 Projekt anlegen 65

2.4.3 Quellcode-Editor 69
2.4.3.1 Syntaxvervollständigung 69
2.4.3.2 Code-Inspektion und Quick-Fixes 71
2.4.3.3 Live Templates 72
2.4.3.4 Orientierungshilfen 73
2.4.3.5 Refaktorieren 75
2.4.3.6 Sonstige Hinweise 75
2.4.4 Übersetzen und Ausführen 75
2.4.5 Sichern und Wiederherstellen 77
2.4.6 Konfiguration 77
2.4.6.1 Verfügbare SDKs einrichten 77
2.4.6.2 Struktur des aktuellen Projekts 80
2.4.6.3 Einstellungen für IntelliJ oder das aktuelle Projekt 81
2.4.6.4 Einstellungen für neue Projekte 82
2.4.7 Übungsprojekte zum Kurs verwenden 84

2.5 OpenJFX und Scene Builder installieren 84

2.6 Übungsaufgaben zum Kapitel 2 87

3 ELEMENTARE SPRACHELEMENTE 89

3.1 Einstieg 89
3.1.1 Aufbau eines Java-Programms 89
3.1.2 Projektrahmen zum Üben von elementaren Sprachelementen 90
3.1.3 Syntaxdiagramme 92
3.1.3.1 Klassendefinition 93
3.1.3.2 Methodendefinition 94
3.1.4 Hinweise zur Gestaltung des Quellcodes 95
3.1.5 Kommentare 96
3.1.6 Namen 99
3.1.7 Vollständige Klassennamen und Import-Deklaration 100

3.2 Ausgabe bei Konsolenanwendungen 101
3.2.1 Ausgabe einer (zusammengesetzten) Zeichenfolge 102
3.2.2 Formatierte Ausgabe 102

3.3 Variablen und Datentypen 105
3.3.1 Strenge Compiler-Überwachung bei Java-Variablen 105
3.3.2 Variablennamen 107
3.3.3 Primitive Typen und Referenztypen 108
3.3.4 Klassifikation der Variablen nach Zuordnung 110
3.3.5 Eigenschaften einer Variablen 111
3.3.6 Primitive Datentypen in Java 112
3.3.7 Vertiefung: Darstellung von Gleitkommazahlen im Arbeitsspeicher des Computers 114
3.3.7.1 Binäre Gleitkommadarstellung 114
3.3.7.2 Dezimale Gleitkommadarstellung 116
3.3.8 Variablendeklaration, Initialisierung und Wertzuweisung 118
3.3.9 Blöcke und Sichtbarkeitsbereiche für lokale Variablen 121
3.3.10 Finalisierte lokale Variablen 123
3.3.11 Literale 124
3.3.11.1 Ganzzahlliterale 124
3.3.11.2 Gleitkommaliterale 126
3.3.11.3 boolean-Literale 127
3.3.11.4 char-Literale 127
3.3.11.5 Zeichenfolgenliterale 128
3.3.11.6 Referenzliteral null 130

3.4 Eingabe bei Konsolenprogrammen 130
3.4.1 Die Klassen Scanner und `Simput` 130
3.4.2 Eine globale Bibliothek mit Simput in IntelliJ einrichten 133

3.5 Operatoren und Ausdrücke 136
3.5.1 Arithmetische Operatoren 137
3.5.2 Methodenaufrufe 140
3.5.3 Vergleichsoperatoren 141
3.5.4 Identitätsprüfung bei Gleitkommawerten 142
3.5.5 Logische Operatoren 144
3.5.6 Vertiefung: Bitorientierte Operatoren 147
3.5.7 Typumwandlung (Casting) bei primitiven Datentypen 148
3.5.7.1 Automatische erweiternde Typanpassung 148
3.5.7.2 Explizite Typumwandlung 149
3.5.8 Zuweisungsoperatoren 150
3.5.9 Konditionaloperator 153
3.5.10 Auswertungsreihenfolge 153
3.5.10.1 Regeln 154
3.5.10.2 Operatorentabelle 156

3.6 Über- und Unterlauf bei numerischen Variablen 157
3.6.1 Überlauf bei Ganzzahltypen 158
3.6.2 Unendliche und undefinierte Werte bei den Typen float und double 160
3.6.3 Unterlauf bei den Gleitkommatypen 162
3.6.4 Vertiefung: Der Modifikator strictfp 164

3.7 Anweisungen (zur Ablaufsteuerung) 165
3.7.1 Überblick 165
3.7.2 Bedingte Anweisung und Fallunterscheidung 166
3.7.2.1 if-Anweisung 166
3.7.2.2 if-else - Anweisung 167
3.7.2.3 switch-Anweisung 172
3.7.2.4 IntelliJ-Ausführungskonfiguration 175
3.7.2.5 Vertiefung: switch-Ausdruck 176
3.7.3 Wiederholungsanweisung 178
3.7.3.1 Zählergesteuerte Schleife (for) 179
3.7.3.2 Iterieren über die Elemente von Arrays oder Kollektionen 180
3.7.3.3 Bedingungsabhängige Schleifen 182
3.7.3.4 Endlosschleifen 184
3.7.3.5 Schleifen(durchgänge) vorzeitig beenden 184

3.8 Entspannungs- und Motivationseinschub: GUI-Standarddialoge 186

3.9 Übungsaufgaben zum Kapitel 3 190
Abschnitt 3.1 (Einstieg) 190
Abschnitt 3.2 (Ausgabe bei Konsolenanwendungen) 190
Abschnitt 3.3 (Variablen und Datentypen) 191
Abschnitt 3.4 (Eingabe bei Konsolen) 192
Abschnitt 3.5 (Operatoren und Ausdrücke) 192
Abschnitt 3.6 (Über- und Unterlauf bei numerischen Variablen) 193
Abschnitt 3.7 (Anweisungen (zur Ablaufsteuerung)) 194

4 KLASSEN UND OBJEKTE 197

4.1 Überblick, historische Wurzeln, Beispiel 198
4.1.1 Einige Kernideen und Vorzüge der OOP 198
4.1.1.1 Datenkapselung und Modularisierung 198
4.1.1.2 Vererbung 200
4.1.1.3 Polymorphie 202
4.1.1.4 Realitätsnahe Modellierung 203
4.1.2 Strukturierte Programmierung und OOP 203
4.1.3 Auf-`Bruch` zu echter Klasse 204

4.2 Instanzvariablen (Felder) 207
4.2.1 Sichtbarkeitsbereich, Existenz und Ablage im Hauptspeicher 208
4.2.2 Deklaration mit Modifikatoren für den Zugriffsschutz und für andere Zwecke 209

4.2.3 Automatische Initialisierung auf den Voreinstellungswert 212
4.2.4 Zugriff in klasseneigenen und fremden Methoden 212
4.2.5 Finalisierte Instanzvariablen 214

4.3 Instanzmethoden 215
4.3.1 Methodendefinition 216
4.3.1.1 Modifikatoren 217
4.3.1.2 Rückgabewert und return-Anweisung 218
4.3.1.3 Namen 219
4.3.1.4 Formalparameter 219
4.3.1.5 Methodenrumpf 224
4.3.2 Methodenaufruf und Aktualparameter 224
4.3.3 Debug-Einsichten zu (verschachtelten) Methodenaufrufen 227
4.3.4 Methoden überladen 232

4.4 Objekte 234
4.4.1 Referenzvariablen deklarieren 234
4.4.2 Objekte erzeugen 235
4.4.3 Konstruktoren 237
4.4.4 Instanzinitialisierer 240
4.4.5 Objekte aus der Fabrik 241
4.4.6 Abräumen überflüssiger Objekte durch den Garbage Collector 242
4.4.7 finalize() und Cleaner 243
4.4.8 Objektreferenzen verwenden 245
4.4.8.1 Rückgabe mit Referenztyp 245
4.4.8.2 this als Referenz auf das aktuelle Objekt 246

4.5 Klassenvariablen und -methoden 246
4.5.1 Klassenvariablen 246
4.5.2 Wiederholung zur Kategorisierung von Variablen 248
4.5.3 Klassenmethoden 249
4.5.4 Statische Initialisierer 251

4.6 Rekursive Methoden 252

4.7 Komposition 254

4.8 Mitgliedsklassen und lokale Klassen 257
4.8.1 Mitgliedsklassen 257
4.8.1.1 Innere Klassen 258
4.8.1.2 Statische Mitgliedsklassen 260
4.8.2 Lokale Klassen 261

4.9 Bruchrechnungsprogramm mit JavaFX-GUI 263
4.9.1 JavaFX-Projekt mit dem OpenJDK 8 anlegen 263
4.9.2 Bedienoberfläche bzw. FXML-Datei mit dem Scene Builder gestalten 265
4.9.3 Klasse `Bruch` einbinden 271
4.9.4 Controller-Klasse vervollständigen 272
4.9.5 Programmstart 275
4.9.6 OpenJDK/OpenJFX 13 276

4.10 Übungsaufgaben zum Kapitel 4 280

5 ELEMENTARE KLASSEN 287

5.1 Arrays 287
5.1.1 Array-Variablen deklarieren 288
5.1.2 Array-Objekte erzeugen 288
5.1.3 Kovariante Einbindung von Arrays in die Klassenhierarchie 290
5.1.4 Arrays verwenden 291
5.1.5 Array-Kopien mit neuer Länge erstellen 292
5.1.6 Nützliche Methoden in der Klasse Arrays 292

5.1.7 Beispiel: Beurteilung des Java-Pseudozufallszahlengenerators 292
5.1.8 Initialisierungslisten 294
5.1.9 Objekte als Array-Elemente 295
5.1.10 Mehrdimensionale Arrays 295

5.2 Klassen für Zeichenfolgen 297
5.2.1 Die Klasse String für konstante Zeichenfolgen 298
5.2.1.1 Erzeugen von String-Objekten 298
5.2.1.2 String als WORM - Klasse 299
5.2.1.3 Interner String-Pool und Identitätsvergleich 299
5.2.1.4 Methoden für String-Objekte 301
5.2.1.5 Vertiefung: Aufwand beim Inhalts- bzw. Referenzvergleich 304
5.2.2 Die Klassen StringBuilder und StringBuffer für veränderliche Zeichenfolgen 306
5.2.3 Mehrzeilige Textblöcke 308

5.3 Verpackungsklassen für primitive Datentypen 309
5.3.1 Wrapper-Objekte erstellen 309
5.3.2 Auto(un)boxing 310
5.3.3 Empfehlungen zur Verwendung von Wrapper-Objekten 311
5.3.3.1 Identitätsoperator vermeiden 312
5.3.3.2 Wrapper-Objekte nicht für variable Werte verwenden 313
5.3.4 Konvertierungsmethoden 313
5.3.5 Konstanten für Grenz- bzw. Spezialwerte 314
5.3.6 Character-Methoden zur Zeichen-Klassifikation 315

5.4 Aufzählungstypen 315
5.4.1 Einfache Enumerationstypen 316
5.4.2 Erweiterte Enumerationstypen 318

5.5 Übungsaufgaben zum Kapitel 5 319
Abschnitt 5.1 (Arrays) 319
Abschnitt 5.2 (Klassen für Zeichen) 321
Abschnitt 5.3 (Verpackungsklassen für primitive Datentypen) 323
Abschnitt 5.4 (Aufzählungstypen) 324

6 PAKETE UND MODULE 325

6.1 Pakete 327
6.1.1 Pakete erstellen 327
6.1.1.1 package-Deklaration und Paketordner 327
6.1.1.2 Standardpaket 328
6.1.1.3 Unterpakete 329
6.1.1.4 Paketunterstützung in IntelliJ 330
6.1.1.5 Konventionen für weltweit eindeutige Paketnamen 332
6.1.2 Pakete verwenden 333
6.1.2.1 Verfügbarkeit der class-Dateien (Klassenpfad) 333
6.1.2.2 Typen aus fremden Paketen ansprechen 335
6.1.2.3 Startklassen im Paket 337
6.1.3 Traditionelle jar-Dateien (Java 8) 339
6.1.3.1 Eigenschaften von Archivdateien 340
6.1.3.2 Archivdateien mit dem JDK-Werkzeug jar erstellen 341
6.1.3.3 Archivdateien verwenden 342
6.1.3.4 Ausführbare jar-Dateien 343
6.1.3.5 Ausführbare jar-Datei für ein Projekt mit OpenJFX 8 344

6.2 Module 347
6.2.1 *Moduldeklarationsdatei module-info.java* 349
6.2.1.1 Modulnamen 349
6.2.1.2 requires-Deklaration 350
6.2.1.3 exports-Deklaration 351
6.2.1.4 uses- und provides-Deklaration 352
6.2.1.5 opens-Deklaration 352

6.2.2 Quellcode-Organisation 353
6.2.3 Übersetzung in ein explodiertes Modul und den Moduldeskriptor 356
6.2.4 Modulpfad 357
6.2.5 Ausführen 358
6.2.6 Modulare jar-Dateien 359
6.2.7 Modulunterstützung in IntelliJ IDEA 361
6.2.7.1 Modul `de.uni_trier.zimk.util` 363
6.2.7.2 Modul `de.uni_trier.zimk.matrain` 366
6.2.7.3 Hauptmodul `de.uni_trier.zimk.ba` 368
6.2.8 Eigenständige Anwendungen mit maßgeschneiderter Laufzeitumgebung 371
6.2.9 Kompatibilität und Migration 372
6.2.9.1 Automatische Module 373
6.2.9.2 Das unbenannte Modul 373
6.2.9.3 Notlösung 374
6.2.9.4 Moduldeklaration zu vorhandenem Quellcode erstellen 374
6.2.10 Das modulare API der Java Standard Edition 375
6.2.11 Modul-Taxonomie 376

6.3 Zugriffsschutz 377
6.3.1 Sichtbarkeit von Top-Level - Typen 377
6.3.2 Sichtbarkeit von Typmitgliedern 379

6.4 Übungsaufgaben zum Kapitel 6 380

7 VERERBUNG UND POLYMORPHIE 383

7.1 Definition einer abgeleiteten Klasse 385

7.2 Optionale und erzwungene Beschränkungen beim (Ver)erben 386
7.2.1 Finale Klassen 386
7.2.2 Keine Mehrfachvererbung 386

7.3 Der Zugriffsmodifikator protected 386

7.4 Basisklassenkonstruktoren und Initialisierungsmaßnahmen 387

7.5 Überschreiben und Überdecken 389
7.5.1 Überschreiben von Instanzmethoden 389
7.5.2 Überdecken von statischen Methoden 391
7.5.3 Finalisierte Methoden 392
7.5.4 Felder überdecken 393

7.6 Verwaltung von Objekten über Basisklassenreferenzen 394

7.7 Polymorphie 395

7.8 Abstrakte Methoden und Klassen 396

7.9 Das Liskovsche Substitutionsprinzip 398

7.10 Unerwünschte Abhängigkeiten durch Vererbung 399
7.10.1 Risiken für abgeleitete Klassen 399
7.10.2 Nachteile für potentielle Basisklassen 401

7.11 Übungsaufgaben zum Kapitel 7 402

8 GENERISCHE KLASSEN UND METHODEN 405

8.1 Generische Klassen 405
8.1.1 Vorzüge und Verwendung generischer Klassen 405

8.1.1.1 Veraltete Technik mit Risiken und Umständlichkeiten 405
8.1.1.2 Generische Klassen bringen Typsicherheit und Bequemlichkeit 406
8.1.2 Technische Details und Komplikationen 408
8.1.2.1 Typlöschung und Rohtyp 408
8.1.2.2 Spezialisierungsbeziehungen bei parametrisierten Klassen und bei Arrays 410
8.1.2.3 Keine Array-Kreation mit einem nicht-reifizierbaren Elementtyp 411
8.1.2.4 Serienparameter mit einem parametrisierten Typ 413
8.1.3 Definition von generischen Klassen 414
8.1.3.1 Unbeschränkte Typformalparameter 414
8.1.3.2 Beschränkte Typformalparameter 419

8.2 Generische Methoden 422

8.3 Wildcard-Datentypen 425
8.3.1 Beschränkte Wildcard-Typen 426
8.3.1.1 Beschränkung nach oben 426
8.3.1.2 Beschränkung nach unten 427
8.3.2 Unbeschränkte Wildcard-Typen 428
8.3.3 Verwendungszwecke für Wildcard-Datentypen 428

8.4 Einschränkungen der Generizitätslösung in Java 429
8.4.1 Konkretisierung von Typformalparametern nur durch Referenztypen 429
8.4.2 Typlöschung und die Folgen 429
8.4.2.1 Keine Typparameter bei der Definition von statischen Mitgliedern 429
8.4.2.2 Keine Kreation von Objekten aus einer per Typformalparameter bestimmten Klasse 430
8.4.2.3 Verwendung einer per Typformalparameter konkretisierten generischen Klasse 431

8.5 Übungsaufgaben zum Kapitel 8 432

9 INTERFACES 435

9.1 Überblick 435
9.1.1 Beispiel 435
9.1.2 Primärer Verwendungszweck 436
9.1.3 Mögliche Bestandteile 437

9.2 Interfaces definieren 438
9.2.1 Kopf einer Schnittstellen-Definitionen 439
9.2.2 Vererbung bzw. Erweiterung bei Schnittstellen 439
9.2.3 Schnittstellen-Methoden 439
9.2.3.1 Abstrakte Instanzmethoden 440
9.2.3.2 Instanzmethoden mit default-Implementierung 440
9.2.3.3 Statische Methoden 443
9.2.3.4 Private Interface-Methoden 444
9.2.4 Konstanten 445
9.2.5 Statische Mitgliedstypen 445
9.2.6 Zugriffsschutz bei Schnittstellen 446
9.2.7 Marker - Interfaces 447

9.3 Interfaces implementieren 447
9.3.1 Mehrere Schnittstellen implementieren 448
9.3.2 Geerbte Interface-Implementationen 449
9.3.3 Implementieren von Schnittstellen und Sichtbarkeit für Klassen 451

9.4 Interfaces als Referenzdatentypen verwenden 452

9.5 Annotationen 453
9.5.1 Definition 454
9.5.2 Zuweisung 455
9.5.3 Runtime-Annotationen per Reflexion auswerten 456
9.5.4 API-Annotationen 457

9.6 Übungsaufgaben zum Kapitel 9 **459**

10 JAVA COLLECTIONS FRAMEWORK **461**

10.1 Arrays versus Kollektionen **461**

10.2 Zur Rolle von Schnittstellen beim JCF-Design **463**
10.2.1 Das Interface Collection<E> mit Basiskompetenzen 464
10.2.2 Optionale Operationen 468

10.3 Listen **468**
10.3.1 Das Interface List<E> 469
10.3.2 Beispiel 471
10.3.3 Listenarchitekturen 472
10.3.3.1 Array als Hintergrundspeicher 472
10.3.3.2 Verkette Objekte 473
10.3.4 Leistungsunterschiede und Einsatzempfehlungen 474

10.4 Iteratoren **476**

10.5 Mengen **478**
10.5.1 Das Interface Set<E> 479
10.5.2 Leistungsvorteil bei der Existenzprüfung 481
10.5.3 Hashtabellen 481
10.5.4 Balancierte Binärbäume 483
10.5.5 Interfaces für geordnete Mengen 486

10.6 Abbildungen **490**
10.6.1 Das Interface Map<K,V> 490
10.6.2 Die Klasse HashMap<K,V> 494
10.6.3 Interfaces für Abbildungen mit geordneten Schlüsseltypen 496
10.6.4 Die Klasse TreeMap<K,V> 499

10.7 Vergleich der Kollektionsarchitekturen **500**

10.8 Warteschlangen **501**

10.9 Nützliche Methoden in der Klasse Collections **501**

10.10 Übungsaufgaben zum Kapitel 10 **503**

11 AUSNAHMEBEHANDLUNG **507**

11.1 Unbehandelte Ausnahmen **508**

11.2 Ausnahmen abfangen **510**
11.2.1 Die try-catch-finally - Anweisung 510
11.2.1.1 Ausnahmebehandlung per catch-Block 511
11.2.1.2 Aufräumarbeiten im finally-Block 515
11.2.2 Programmablauf bei der Ausnahmebehandlung 517
11.2.3 Diagnostische Ausgaben 519

11.3 Ausnahmeobjekte im Vergleich zur Fehlerkommunikation per Rückgabewert **520**

11.4 Ausnahmen und Fehler **525**
11.4.1 Error 526
11.4.2 Geprüfte und ungeprüfte Ausnahmen 528
11.4.2.1 Unterschiedliche Behandlung durch den Compiler 528
11.4.2.2 Eine schwierige Unterscheidung 529

11.5 Ausnahmen in einer eigenen Methode auslösen und ankündigen **530**

11.5.1 Ausnahmen auslösen (throw), ankündigen (throws) und dokumentieren 530
11.5.2 Pflicht zur Ausnahmebehandlung abschieben 533
11.5.3 Compiler-Intelligenz beim erneuten Werfen von abgefangenen Ausnahmen 534

11.6 Ausnahmen bei der Parameter-Validierung 534

11.7 Ausnahmen definieren 535

11.8 Freigabe von Ressourcen 538
11.8.1 Traditionelle Lösung per finally-Block 538
11.8.2 Try With Resources 539

11.9 Übungsaufgaben zum Kapitel 11 541

12 FUNKTIONALES PROGRAMMIEREN 543

12.1 Lambda-Ausdrücke 543
12.1.1 Traditionelle und moderne Realisation von Funktionsobjekten 544
12.1.1.1 Funktionale Schnittstellen 544
12.1.1.2 Anonyme Klassen 546
12.1.1.3 Compiler-Kompetenz statt Boilerplate-Code 550
12.1.2 Definition von Lambda-Ausdrücken 552
12.1.2.1 Formalparameterliste 552
12.1.2.2 Rumpf 553
12.1.2.3 Definitionsumgebungen 554
12.1.3 Methoden- und Konstruktorreferenzen 556
12.1.3.1 Methodenreferenzen 556
12.1.3.2 Konstruktorreferenzen 558

12.2 Ströme 559
12.2.1 Elementare Begriffe und Beispiel 559
12.2.2 Externe versus interne Iteration 561
12.2.3 Eigenschaften von Strömen 562
12.2.3.1 Datentyp der Elemente 562
12.2.3.2 Sequentiell oder parallel 563
12.2.4 Erstellung von Stromobjekten 563
12.2.4.1 Stromobjekt aus einer Kollektion (stream, parallelStream) 563
12.2.4.2 Stromobjekt aus einem Array oder aus einer Serie von Werten (stream, of) 563
12.2.4.3 Stromobjekte mit einer Sequenz ganzer Zahlen (range, rangeClosed) 564
12.2.4.4 Unendliche Ströme (iterate, generate) 564
12.2.4.5 Sonstige Erstellungsmethoden (lines) 565
12.2.5 Stromoperationen 565
12.2.5.1 Intermediäre und terminale Stromoperationen 566
12.2.5.2 Faulheit ist nicht immer dumm 567
12.2.5.3 Intermediäre Operationen 568
12.2.5.4 Terminale Operationen 573

12.3 Empfehlungen für erfolgreiches funktionales Programmieren 581
12.3.1.1 Deklarieren statt Kommandieren 581
12.3.1.2 Veränderliche Variablen vermeiden 582
12.3.1.3 Seiteneffekte vermeiden 582
12.3.1.4 Ausdrücke bevorzugen gegenüber Anweisungen 582
12.3.1.5 Verwendung von Funktionen höherer Ordnung 583

12.4 Übungsaufgaben zum Kapitel 12 584

13 GUI-PROGRAMMIERUNG MIT JAVAFX 587

13.1 Einordnung 587
13.1.1 Vergleich von Konsolen- und GUI-Programmen 587
13.1.2 Desktop-GUI-Lösungen in Java 589

13.2 Einstieg in JavaFX 590
13.2.1 Beispiel Anwesenheitsliste 591
13.2.2 Starten und Beenden einer JavaFX-Anwendung 594
13.2.3 Bühne, Szene und Szenengraph 596

13.3 Anwendung mit All-In-One - Architektur 596

13.4 Anwendung mit Model-View-Controller - Architektur (MVC) 600
13.4.1 Das Model-View-Controller - Konzept 600
13.4.2 Projekt anlegen 601
13.4.3 Model 602
13.4.4 GUI-Gestaltung per Scene Builder 603
13.4.5 FXML 605
13.4.6 Controller 607
13.4.7 Anwendungsklasse 610

13.5 Properties mit Änderungssignalisierung und automatischer Synchronisation 611
13.5.1 Properties 612
13.5.1.1 Traditionelle JavaBean-Eigenschaften 612
13.5.1.2 Property-Klassen von JavaFX 613
13.5.1.3 Vermeidung von überflüssigen Objektkreationen 616
13.5.2 Invalidierungs- und Veränderungsereignisse 617
13.5.3 Automatische Synchronisation von Property-Objekten 618
13.5.3.1 Uni- und bidirektionale Synchronisation von Property-Objekten 618
13.5.3.2 Property-Objekt an ein Berechnungsergebnis binden 620
13.5.3.3 Beobachtbare Listen 624

13.6 Layoutmanager 628
13.6.1 GridPane 629
13.6.1.1 Beispiel 629
13.6.1.2 GridPane-Eigenschaften zur Gestaltung von Abständen 630
13.6.1.3 Anzeigeeinstellungen für Kindelemente (layout constraints) 630
13.6.1.4 Dynamische Platzverteilung auf mehrere Spalten bzw. Zeilen 632
13.6.2 AnchorPane 632
13.6.3 HBox und VBox 633
13.6.4 BorderPane 634
13.6.5 FlowPane 636
13.6.6 StackPane 637

13.7 Elementare Steuerelemente 638
13.7.1 Label 638
13.7.2 Button 640
13.7.3 Einzeiliges Texteingabefeld 642
13.7.4 Umschalter 644
13.7.4.1 Kontrollkästchen 644
13.7.4.2 Radioschalter 646

13.8 Übungsaufgaben zum Kapitel 13 647

14 EIN- UND AUSGABE ÜBER DATENSTRÖME 649

14.1 Grundlagen 649
14.1.1 Datenströme 649
14.1.2 Beispiel 650
14.1.3 Klassifikation der Stromverarbeitungsklassen 651
14.1.4 Aufbau und Verwendung der Transformationsklassen 652
14.1.5 Zum guten Schluss 654

14.2 Verwaltung von Dateien und Verzeichnissen 656
14.2.1 Dateisystemzugriffe über das NIO.2 - API 657
14.2.1.1 Repräsentation von Dateisystemeinträgen 657
14.2.1.2 Existenzprüfung 659

14.2.1.3 Verzeichnis anlegen 660
14.2.1.4 Datei explizit erstellen 660
14.2.1.5 Attribute von Dateisystemobjekten ermitteln 661
14.2.1.6 Zugriffsrechte für Dateien ermitteln 662
14.2.1.7 Attribute ändern 662
14.2.1.8 Über Verzeichniseinträge iterieren 663
14.2.1.9 Datei und Ordner kopieren 663
14.2.1.10 Umbenennen und Verschieben 664
14.2.1.11 Löschen 665
14.2.1.12 Informationen über Dateisysteme ermitteln 665
14.2.1.13 Weitere Optionen 666
14.2.2 Dateisystemzugriffe über die Klasse File aus dem Paket java.io 667
14.2.2.1 Verzeichnis anlegen 667
14.2.2.2 Dateien explizit erstellen 668
14.2.2.3 Informationen über Dateien und Ordner 668
14.2.2.4 Atribute ändern 669
14.2.2.5 Verzeichnisinhalte auflisten 669
14.2.2.6 Umbenennen 670
14.2.2.7 Löschen 670

14.3 Klassen zur Verarbeitung von Byte-Strömen 671
14.3.1 Die OutputStream-Hierarchie 671
14.3.1.1 Überblick 671
14.3.1.2 FileOutputStream 672
14.3.1.3 OutputStream mit Dateianschluss per NIO.2 - API 674
14.3.1.4 DataOutputStream 675
14.3.1.5 BufferedOutputStream 676
14.3.1.6 PrintStream 677
14.3.2 Die InputStream-Hierarchie 680
14.3.2.1 Überblick 680
14.3.2.2 FileInputStream 681
14.3.2.3 InputStream mit Dateianschluss per NIO.2 - API 682
14.3.2.4 DataInputStream 683

14.4 Klassen zur Verarbeitung von Zeichenströmen 683
14.4.1 Die Writer-Hierarchie 684
14.4.1.1 Überblick 684
14.4.1.2 Brückenklasse OutputStreamWriter 685
14.4.1.3 FileWriter 688
14.4.1.4 BufferedWriter 689
14.4.1.5 PrintWriter 690
14.4.1.6 BufferedWriter mit Dateianschluss per NIO.2 - API 692
14.4.2 Die Reader-Hierarchie 694
14.4.2.1 Überblick 694
14.4.2.2 Brückenklasse InputStreamReader 694
14.4.2.3 FileReader und BufferedReader 695
14.4.2.4 BufferedReader mit Dateianschluss per NIO.2 - API 696

14.5 Zahlen und Zeichenfolgen aus einer Textdatei lesen 697

14.6 Objektserialisierung 701
14.6.1 Objektserialisierung und Sicherheit 701
14.6.2 Beispiel für eine serialisierbare Klasse 702
14.6.3 Versionskontrolle und Kompatibilitätsprobleme 703
14.6.4 Objekte in eine Datei schreiben und von dort lesen 704
14.6.5 Von der Serialisierung ausgeschlossene Felder 705
14.6.6 Mehr Kontrolle und Eigenverantwortung 706
14.6.7 Serialisierung und Vererbung 707
14.6.8 Bewertung der Objektserialisierung und mögliche Alternativen 708

14.7 Daten lesen und schreiben über die NIO.2 - Klasse Files 709
14.7.1 Öffnungsoptionen 710

14.7.2 Lesen und Schreiben von kleinen Dateien 710
14.7.3 Datenstrom zu einem Path-Objekt erstellen 711
14.7.4 MIME-Type einer Datei ermitteln 712
14.7.5 Stream<String> mit den Zeilen einer Textdatei erstellen 713

14.8 Empfehlungen zur Ein- und Ausgabe 713
14.8.1 Ausgabe in eine Textdatei 713
14.8.2 Textzeilen einlesen 714
14.8.3 Zahlen und andere Tokens aus einer Textdatei lesen 715
14.8.4 Eingabe von der Konsole 716
14.8.5 Objekte (de)serialisieren 716
14.8.6 Werte mit primitiven Datentypen in eine Binärdatei schreiben 717
14.8.7 Werte mit primitiven Datentypen aus einer Binärdatei lesen 717

14.9 Übungsaufgaben zum Kapitel 14 718

15 MULTITHREADING 723

15.1 Start und Ende eines Threads 724
15.1.1 Die Klasse Thread 724
15.1.2 Das Interface Runnable 729

15.2 Threads koordinieren 731
15.2.1 Fehlerhafte oder veraltete Daten 731
15.2.1.1 Fehlerhafte Daten wegen nicht-atomarer Operationen 731
15.2.1.2 Veraltete Daten im lokalen Cache eines Threads 732
15.2.2 Per Monitor synchronisierte Code-Bereiche 733
15.2.2.1 Synchronisierte Methoden und Blöcke 733
15.2.2.2 Koordination per wait(), notify() und notifyAll() 735
15.2.3 Explizite Lock-Objekte 737
15.2.3.1 Interface Lock und Klasse ReentrantLock 737
15.2.3.2 Koordination per await(), signal() und signalAll() 740
15.2.4 Automatisierte Thread-Koordination für Produzenten-Konsumenten - Konstellationen 742
15.2.4.1 BlockingQueue<E> 742
15.2.4.2 PipedOutputStream und PipedInputStream 744
15.2.5 Klassen zur Thread-Synchronisation 746
15.2.5.1 Semaphore 746
15.2.5.2 CountDownLatch 748
15.2.5.3 CyclicBarrier 749
15.2.5.4 Phaser 751
15.2.6 Nicht-blockierende Koordination 753
15.2.6.1 Modifikator volatile 754
15.2.6.2 Atomare Variablen 754

15.3 Direkt gestartete Threads verwalten 756
15.3.1 Weck mich, wenn Du fertig bist (join) 756
15.3.2 Andere Threads unterbrechen, fortsetzen oder abbrechen 758
15.3.2.1 Unterbrechen und fortsetzen 758
15.3.2.2 Abbrechen 758
15.3.3 Thread-Lebensläufe 760
15.3.3.1 Scheduling und Prioritäten 760
15.3.3.2 Zustände von Threads 761
15.3.4 Deadlock 762

15.4 Aufgaben per Threadpool erledigen 763
15.4.1 ExecutorService 763
15.4.2 Interface Callable<V> 765
15.4.3 Threadpools mit Timer-Funktionalität 768

15.5 Fork-Join - Framework 769
15.5.1 Direkte Verwendung des Fork-Join - Frameworks 769
15.5.2 Parallele Aggregatoperationen mit Strömen 773

15.6 Thread-sichere Kollektionen im Paket java.util.concurrent 774

15.7 Threads und JavaFX 777
15.7.1 JavaFX-Komponenten aus einem Hintergrund-Thread modifizieren 777
15.7.2 Das JavaFX-Multithreading - API 778
15.7.3 Die Klasse Task<V> 780

15.8 Weitere Thread-Themen 782
15.8.1 Daemon-Threads 782
15.8.2 Thread-Gruppen 783

15.9 Übungsaufgaben zum Kapitel 15 783

ANHANG 785

A. Operatorentabelle 785

B. Lösungsvorschläge zu den Übungsaufgaben 787
Kapitel 1 (Einleitung) 787
Werkzeuge zum Entwickeln von Java-Programmen) 789
Kapitel 3 (Elementare Sprachelemente) 789
Abschnitt 3.1 (Einstieg) 789
Abschnitt 3.2 (Ausgabe bei Konsolenanwendungen) 790
Abschnitt 3.3 (Variablen und Datentypen) 791
Abschnitt 3.4 (Eingabe bei Konsolen) 792
Abschnitt 3.5 (Operatoren und Ausdrücke) 793
Abschnitt 3.6 (Über- und Unterlauf bei numerischen Variablen) 794
Abschnitt 3.7 (Anweisungen (zur Ablaufsteuerung)) 794
Kapitel 4 (Klassen und Objekte) 796
Kapitel 5 (Elementare Klassen) 798
Abschnitt 5.1 (Arrays) 798
Abschnitt 5.2 (Klassen für Zeichen) 798
Abschnitt 5.3 (Verpackungsklassen für primitive Datentypen) 799
Abschnitt 5.4 (Aufzählungstypen) 799
Kapitel 6 (Pakete und Module) 799
Kapitel 7 (*Vererbung und Polymorphie*) 800
Kapitel 8 (Generische Klassen und Methoden) 801
Kapitel 9 (*Interfaces*) 801
Kapitel 10 (Java Collections Framework) 802
Kapitel 11 (Ausnahmebehandlung) 802
Kapitel 12 (Funktionales Programmieren) 803
Kapitel 13 (*GUI-Programmierung mit JavaFX*) 804
Kapitel 14 (Ein- und Ausgabe über Datenströme) 805
Kapitel 15 (Multithreading) 806

LITERATUR 809

INDEX 813

1 Einleitung

Im ersten Kapitel geht es zunächst um die Denk- und Arbeitsweise der objektorientierten Programmierung. Danach wird Java als Software-Technologie vorgestellt.

1.1 Beispiel für die objektorientierte Software-Entwicklung mit Java

In diesem Abschnitt soll eine Vorstellung davon vermittelt werden, was ein Computerprogramm (in Java) ist. Dabei kommen einige Grundbegriffe der Informatik zur Sprache, wobei wir uns aber nicht unnötig lange von der Praxis fernhalten wollen.

Ein Computerprogramm besteht im Wesentlichen (von Medien und anderen Ressourcen einmal abgesehen) aus einer Menge von wohlgeformten und wohlgeordneten *Definitionen* und *Anweisungen* zur Bewältigung bestimmter Aufgaben. Ein Programm muss ...

- den betroffenen Anwendungsbereich **modellieren**
  Beispiel: In einem Programm zur Verwaltung einer Spedition sind z. B. Kunden, Aufträge, Mitarbeiter, Fahrzeuge, Einsatzfahrten, (Ent-)ladestationen etc. und kommunikative Prozesse (als Nachrichten zwischen beteiligten Akteuren) zu repräsentieren.
- **Algorithmen** realisieren, die in endlich vielen Schritten und unter Verwendung von endlich vielen Betriebsmitteln (z. B. Speicher, CPU-Leistung) bestimmte Ausgangszustände in akzeptable Zielzustände überführen.
  Beispiel: Im Speditionsprogramm muss u.a. für jede Tour zu den meist mehreren (Ent-)ladestationen eine optimale Route ermittelt werden (hinsichtlich Kraftstoffverbrauch, Fahrtzeit, Mautkosten etc.).

Wir wollen präzisere und komplettere Definitionen zum komplexen Begriff eines Computerprogramms den Informatik-Lehrbüchern überlassen (siehe z. B. Goll & Heinisch 2016) und stattdessen ein Beispiel im Detail betrachten, um einen Einstieg in die Materie zu finden.

Bei der Suche nach einem geeigneten Java-Einstiegsbeispiel tritt ein Dilemma auf:

- Einfache Beispiele sind angenehm, aber für das Programmieren mit Java nicht besonders repräsentativ. Z. B. ist von der Objektorientierung außer einem gewissen Formalismus nichts vorhanden.
- Repräsentative Java-Programme eignen sich in der Regel wegen ihrer Länge und Komplexität (aus der Sicht des Anfängers) nicht für den Einstieg. Beispielsweise können wir das eben zur Illustration einer realen Aufgabenstellung verwendete, aber potentiell sehr aufwändige Speditionsverwaltungsprogramm jetzt nicht vorstellen.

Wir analysieren ein Beispielprogramm, das trotz angestrebter Einfachheit nicht auf objektorientiertes Programmieren (OOP) verzichtet. Seine Aufgabe besteht darin, elementare Operationen mit Brüchen auszuführen (z. B. Kürzen, Addieren), womit es etwa einem Schüler beim Anfertigen der Hausaufgaben (zur Kontrolle der eigenen Lösungen) nützlich sein kann. Das Beispiel wird in sukzessive ausgebauter Form im Kurs noch oft verwendet, sodass sich eine genauere Betrachtung lohnt.

1.1.1 Objektorientierte Analyse und Modellierung

Einer objektorientierten Programmierung geht die **objektorientierte Analyse** der Aufgabenstellung voran mit dem Ziel einer Modellierung durch kooperierende **Klassen**. Man identifiziert per **Abstraktion** die beteiligten **Objektsorten** und definiert für sie jeweils eine **Klasse**.

In unserem Bruchrechnungsbeispiel ergibt sich bei der objektorientierten Analyse, dass vorläufig nur eine Klasse zum Modellieren von Brüchen benötigt wird (Name: `Bruch`). Beim möglichen

Ausbau des Programms zu einem Bruchrechnungstrainer kommen jedoch weitere Klassen hinzu (z. B. `Aufgabe`, `Schüler`).

Eine Klasse ist gekennzeichnet durch:

- **Eigenschaften bzw. Zustände**
  Manche Eigenschaften bzw. Zustände gehören zu den *Objekten* bzw. *Instanzen* der Klasse (z. B. Zähler und Nenner eines Bruchs), manche gehören zur Klasse selbst (z. B. Anzahl der bei einem Programmeinsatz bereits erzeugten Brüche). Im letztlich entstehenden Programm landet jede Eigenschaft in einer sogenannten **Variablen**. Darunter versteht man in der Datenverarbeitung einen benannten Speicherplatz, der Werte eines bestimmten Typs (z. B. ganze Zahlen, Zeichenfolgen) aufnehmen kann. Variablen zur Repräsentation der Eigenschaften von Objekten oder Klassen werden in Java meist als **Felder** bezeichnet.
- **Handlungskompetenzen**
  Analog zu den Eigenschaften sind auch die Handlungskompetenzen entweder individuellen Objekten bzw. Instanzen oder der Klasse selbst zugeordnet. Im Beispiel ...
    - hat ein Bruchobjekt z. B. die Fähigkeit zum Kürzen seiner Darstellung,
    - kann die Klasse z. B. über die Anzahl der bereits erzeugten Brüche informieren.

  Im letztlich entstehenden Programm sind die Handlungskompetenzen durch sogenannte **Methoden** repräsentiert. Diese ausführbaren Programmbestandteile enthalten die oben angesprochenen Algorithmen. Die Kommunikation zwischen Klassen und Objekten besteht darin, ein Objekt oder eine Klasse aufzufordern, eine bestimmte Methode auszuführen.

Eine Klasse kann …

- einerseits **Bauplan für konkrete Objekte** sein, die im Programmablauf je nach Bedarf erzeugt und mit der Ausführung bestimmter Methoden beauftragt werden,
- andererseits aber auch **Akteur** sein (Methoden ausführen und aufrufen).

Weil der Begriff *Klasse* gegenüber dem Begriff *Objekt* dominiert, hätte man eigentlich die Bezeichnung *klassenorientierte Programmierung* wählen sollen. Allerdings gibt es nun keinen ernsthaften Grund, die eingeführte Bezeichnung *objektorientierte Programmierung* zu ändern.

Dass jedes Objekt gleich in eine Klasse („Schublade“) gesteckt wird, mögen die Anhänger einer ausgeprägt individualistischen Weltanschauung bedauern. Auf einem geeigneten Abstraktionsniveau betrachtet lassen sich jedoch die meisten Objekte der realen Welt ohne großen Informationsverlust in Klassen einteilen. Bei einer definitiv nur *einfach* zu besetzenden Rolle kann eine Klasse zum Einsatz kommen, die *nicht* zum Instanzieren (Erzeugen von Objekten) gedacht ist, sondern als Akteur.

Dass Zähler und Nenner die zentralen **Eigenschaften** eines `Bruch`-Objekts sind, bedarf keiner näheren Erläuterung. Sie werden in der Klassendefinition durch Felder zum Speichern von ganzen Zahlen (Java-Datentyp **int**) mit den folgenden Namen repräsentiert:

- `zaehler`
- `nenner`

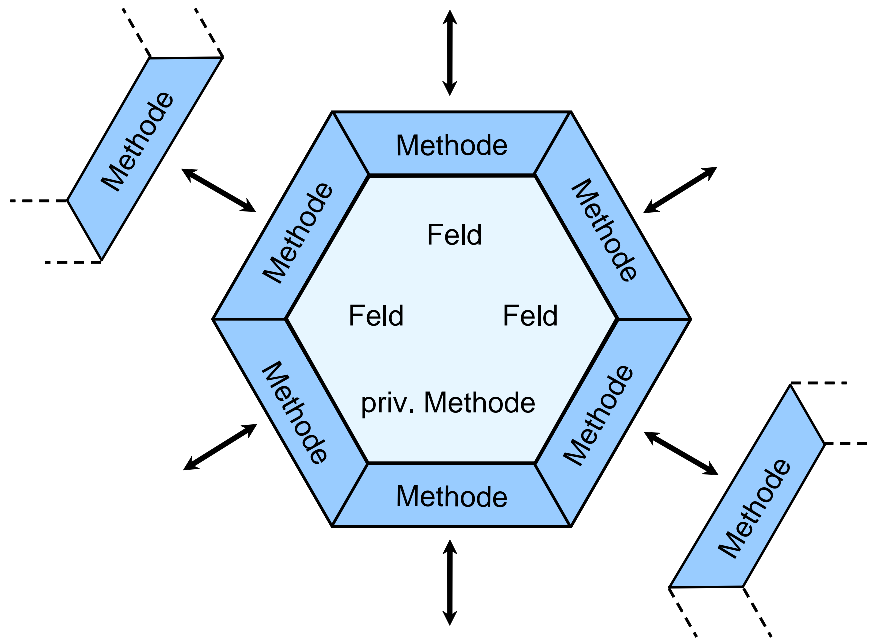
Auf die oben als Möglichkeit genannte *klassenbezogene* Eigenschaft mit der Anzahl bereits erzeugter Brüche wird vorläufig verzichtet.

Im objektorientierten Paradigma ist jede Klasse für die Manipulation ihrer Eigenschaften selbst verantwortlich. Diese sollten **eingekapselt** und so vor dem direkten Zugriff durch fremde Klassen geschützt sein. So kann sichergestellt werden, dass nur sinnvolle Änderungen der Eigenschaften auftreten. Außerdem wird aus später zu erläuternden Gründen die Produktivität der Software-Entwicklung durch die Datenkapselung gefördert.

Demgegenüber sind die **Handlungskompetenzen** (Methoden) einer Klasse in der Regel von anderen Akteuren (Klassen, Objekten) ansprechbar, wobei es aber auch *private* Methoden für den ausschließlich internen Gebrauch gibt. Die *öffentlichen* Methoden einer Klasse bilden ihre **Schnittstelle** zur Kommunikation mit anderen Klassen. Man spricht auch vom **API** (*Application Programming Interface*) einer Klasse.

Die folgende, an Goll & Heinisch (2016) angelehnte Abbildung zeigt für eine Klasse ...

- im gekapselten Bereich ihre Felder sowie eine private Methode
- die Kommunikationsschnittstelle mit den öffentlichen Methoden



Die **Objekte** (Exemplare, Instanzen) einer Klasse, d.h. die nach diesem Bauplan erzeugten Individuen, sollen in der Lage sein, auf eine Reihe von **Nachrichten** mit einem bestimmten Verhalten zu reagieren. In unserem Beispiel sollte die Klasse `Bruch` z. B. eine Methode zum Kürzen besitzen. Dann kann einem konkreten `Bruch`-Objekt durch Aufrufen dieser Methode die Nachricht zugestellt werden, dass es Zähler und Nenner kürzen soll.

Sich unter einem `Bruch` ein Objekt vorzustellen, das Nachrichten empfängt und mit einem passenden Verhalten beantwortet, ist etwas gewöhnungsbedürftig. In der realen Welt sind Brüche, die sich selbst auf ein Signal hin kürzen, nicht unbedingt alltäglich, wenngleich möglich (z. B. als didaktisches Spielzeug). Das objektorientierte Modellieren eines Anwendungsbereichs ist nicht unbedingt eine direkte Abbildung, sondern eine *Rekonstruktion*. Einerseits soll der Anwendungsbereich im Modell gut repräsentiert sein, andererseits soll eine möglichst stabile, gut erweiterbare und wiederverwendbare Software entstehen.

Um (Objekten aus) fremden Klassen trotz Datenkapselung die Veränderung einer Eigenschaft zu erlauben, müssen entsprechende Methoden (mit geeigneten Kontrollmechanismen) angeboten werden. Unsere `Bruch`-Klasse sollte also über Methoden zum Verändern von Zähler und Nenner verfügen (z. B. mit den Namen `setzeZaehler()` und `setzeNenner()`). Bei einer gekapselten Eigenschaft ist auch der direkte *Lesezugriff* ausgeschlossen, sodass im `Bruch`-Beispiel auch noch Methoden zum Ermitteln von Zähler und Nenner erforderlich sind (z. B. mit den Namen `gibZaehler()` und `gibNenner()`). Eine konsequente Umsetzung der Datenkapselung erzwingt also eventuell eine ganze Serie von Methoden zum Lesen und Setzen von Eigenschaftswerten.

Mit diesem Aufwand werden aber gravierende Vorteile realisiert:

- **Stabilität**
  Die Eigenschaften sind vor unsinnigen und gefährlichen Zugriffen geschützt, wenn Veränderungen nur über die vom Klassendesigner sorgfältig entworfenen Methoden möglich sind. Treten trotzdem Fehler auf, sind diese relativ leicht zu identifizieren, weil nur wenige Methoden verantwortlich sein können.
- **Produktivität**
  Durch Datenkapselung wird die **Modularisierung** unterstützt, sodass bei der Entwicklung großer Softwaresysteme zahlreiche Programmierer möglichst reibungslos zusammenarbeiten können. Der Klassendesigner trägt die Verantwortung dafür, dass die von ihm entworfenen Methoden korrekt arbeiten. Andere Programmierer müssen beim Verwenden einer Klasse lediglich die Methoden der Schnittstelle kennen. Das Innenleben einer Klasse kann vom Designer nach Bedarf geändert werden, ohne dass andere Programmbestandteile angepasst werden müssen. Bei einer sorgfältig entworfenen Klasse stehen die Chancen gut, dass sie in mehreren Software-Projekten genutzt werden kann (**Wiederverwendbarkeit**). Besonders günstig ist die Recycling-Quote bei den Klassen der Java-Standardbibliothek (siehe Abschnitt 1.3.3), von denen alle Java-Programmierer regen Gebrauch machen.

Nach obigen Überlegungen sollten die Objekte der Klasse `Bruch` folgende Methoden beherrschen:

- `setzeZaehler(int z)`, `setzeNenner(int n)`
  Ein Objekt wird beauftragt, seinen `zaehler` bzw. `nenner` auf einen bestimmten Wert zu setzen. Ein direkter Zugriff auf die Eigenschaften soll fremden Klassen nicht erlaubt sein (Datenkapselung). Bei dieser Vorgehensweise kann das Objekt z. B. verhindern, dass sein Nenner auf 0 gesetzt wird.
  Wie die Beispiele zeigen, wird dem Namen einer Methode eine in runden Klammern eingeschlossene, eventuell leere *Parameterliste* angehängt. Methodenparameter, mit denen wir uns noch ausführlich beschäftigen werden, haben einen Namen (bei `setzeNenner()` z. B. `n`) und einen Datentyp. Im Beispiel erlaubt der Datentyp **int** ganze Zahlen als Werte.
- `gibZaehler()`, `gibNenner()`
  Ein `Bruch`-Objekt wird beauftragt, den Wert seiner Zähler- bzw. Nenner-Eigenschaft mitzuteilen. Diese Methoden sind im Beispiel erforderlich, weil bei gekapselten Eigenschaften weder schreibende noch lesende Direktzugriffe möglich sind.
- `kuerze()`
  Ein Objekt wird beauftragt, `zaehler` und `nenner` zu kürzen. Welcher Algorithmus dazu benutzt wird, bleibt dem Objekt bzw. dem Klassendesigner überlassen.
- `addiere(Bruch b)`
  Ein Objekt wird beauftragt, den als Parameter übergebenen Bruch zum eigenen Wert zu addieren. Wir werden uns noch ausführlich damit beschäftigen, wie man beim Aufruf einer Methode ihr Verhalten durch die Übergabe von Parametern (Argumenten) steuert.
- `frage()`
  Ein Objekt wird beauftragt, `zaehler` und `nenner` beim Anwender via Konsole (Eingabeaufforderung) zu erfragen.
- `zeige()`
  Ein Objekt wird beauftragt, `zaehler` und `nenner` auf der Konsole anzuzeigen.

In realen (komplexeren) Programmen wird keinesfalls *jedes* gekapselte Feld über ein Methodenpaar zum Lesen und geschützten Schreiben für die Außenwelt zugänglich gemacht.

Beim Eigenschaftsbegriff ist eine (ungefährliche) Zweideutigkeit festzustellen, die je nach Anwendungsbeispiel mehr oder weniger spürbar wird (beim Bruchrechnungsbeispiel überhaupt nicht). Man kann unterscheiden:

- real definierte, meist gekapselte Felder
  Diese sind für die Außenwelt (für andere Klassen) irrelevant und unbekannt. In diesem Sinn wurde der Begriff oben eingeführt.
- nach außen dargestellte Eigenschaften
  Eine solche Eigenschaft ist über Methoden zum Lesen und/oder Schreiben zugänglich und *nicht* unbedingt durch ein *einzelnes* Feld realisiert.

Wir sprechen im Manuskript meist über *Felder* und *Methoden*, wobei keinerlei Mehrdeutigkeit besteht.

Man verwendet für die in einer Klasse definierten Bestandteile oft die Bezeichnung **Member**, gelegentlich auch die deutsche Übersetzung **Mitglieder**. Unsere Klasse `Bruch` hat folgende Member:

- Felder
  `zaehler`, `nenner`
- Methoden
  `setzeZaehler()`, `setzeNenner()`, `gibZaehler()`, `gibNenner()`, `kuerze()`, `addiere()`, `frage()` und `zeige()`

Von kommunizierenden Objekten und Klassen mit Handlungskompetenzen zu sprechen, mag als übertriebener Anthropomorphismus (als Vermenschlichung) erscheinen. Bei der Ausführung von Methoden sind Objekte und Klassen selbstverständlich streng determiniert, während Menschen bei Kommunikation und Handlungsplanung ihren freien Willen einbringen, Spontanität, Kreativität und auch Emotionen besitzen. Fußball spielende Roboter (als besonders anschauliche Objekte aufgefasst) zeigen allerdings mittlerweile schon recht weitsichtige und auch überraschende Spielzüge. Was sie noch zu lernen haben, sind vielleicht Strafraumschwalben, absichtliches Handspiel etc. Nach diesen Randbemerkungen kehren wir zum Programmierkurs zurück, um möglichst bald freundliche und kompetente Objekte definieren zu können.

Um die durch objektorientierte Analyse gewonnene Modellierung eines Anwendungsbereichs standardisiert und übersichtlich zu beschreiben, wurde die **Unified Modeling Language** (UML) entwickelt, die bevorzugt mit Diagrammen arbeitet.[1] Hier wird eine Klasse durch ein Rechteck mit drei Abschnitten dargestellt:

- Oben steht der **Name** der Klasse.
- In der Mitte stehen die **Eigenschaften** (**Felder**).
  Hinter dem Namen einer Eigenschaft gibt man ihren Datentyp an (z. B. **int** für ganze Zahlen).
- Unten stehen die **Handlungskompetenzen** (**Methoden**).
  In Anlehnung an eine in vielen Programmiersprachen (wie z. B. Java) übliche Syntax zur Methodendefinition gibt man für die Argumente eines Methodenaufrufs sowie für den Rückgabewert (falls vorhanden) den Datentyp an. Was mit dem letzten Satz genau gemeint ist, werden Sie bald erfahren.

Für die `Bruch`-Klasse erhält man die folgende Darstellung:

[1] Während die UML im akademischen Bereich nachdrücklich empfohlen wird, ist ihre Verwendung in der Software-Branche allerdings noch entwicklungsfähig, wie empirische Studien gezeigt haben (siehe z. B. Baltes & Diehl 2014, Petre 2013).

```
Bruch
-----------------------------
zaehler: int
nenner: int
-----------------------------
setzeZaehler(int zpar)
setzeNenner(int npar):boolean
gibZaehler():int
gibNenner():int
kuerze()
addiere(Bruch b)
frage()
zeige()
```

Sind bei einer Anwendung *mehrere* Klassen beteiligt, dann sind auch die *Beziehungen* zwischen den Klassen wesentliche Bestandteile des Modells. In einem UML-Klassendiagramm können u.a. die folgenden Beziehungen zwischen Klassen (bzw. zwischen den Objekten von Klassen) dargestellt werden:

- Spezialisierung bzw. Vererbung („Ist-ein - Beziehung“)
  Beispiel: Ein Lieferwagen ist ein spezielles Auto.
- Komposition („Hat - Beziehung“)
  Beispiel: Ein Auto hat einen Motor.
- Assoziation („Kennt - Beziehung“)
  Beispiel: Ein (intelligentes, autonomes) Auto kennt eine Liste von Parkplätzen.

Nach der sorgfältigen Modellierung per UML muss übrigens die Programmierung nicht am Punkt null beginnen, weil UML-Entwicklungswerkzeuge üblicherweise Teile des Quellcodes automatisch aus dem Modell erzeugen können. Die kostenpflichtige Ultimate-Version der im Kurs bevorzugte Java-Entwicklungsumgebung IntelliJ IDEA (siehe Abschnitt 2.4) unterstützt die UML-Modellierung und erstellt automatisch den Quellcode zu einem UML-Diagramm. Das mit IntelliJ Ultimate erstellte Diagramm der Klasse `Bruch` zeigt für die Klassen-Member auch den Typ (Feld bzw. Methode) sowie den Zugriffsschutz an (privat bzw. öffentlich):

c Bruch	
f zaehler	int
f nenner	int
m setzeZaehler(int)	void
m setzeNenner(int)	boolean
m gibZaehler()	int
m gibNenner()	int
m kuerze()	void
m addiere(Bruch)	void
m frage()	void
m zeige()	void

Die fehlende UML-Unterstützung der Community Edition von IntelliJ wird sich im Kurs *nicht* nachteilig auswirken.

Weiterführende Informationen zur objektorientierten Analyse und Modellierung bieten z. B. Balzert (2011) und Booch et al. (2007).

1.1.2 Objektorientierte Programmierung

In unserem einfachen Beispielprojekt soll nun die Klasse `Bruch` in der Programmiersprache Java kodiert werden, wobei die Felder (Eigenschaften) zu deklarieren und die Methoden zu implementieren sind. Es resultiert der sogenannte **Quellcode**, der in einer Textdatei namens **Bruch.java** untergebracht werden muss.

Zwar sind Ihnen die meisten Details der folgenden Klassendefinition selbstverständlich jetzt noch fremd, doch sind die Variablendeklarationen und Methodenimplementationen als zentrale Bestandteile leicht zu erkennen. Außerdem sind Sie nach den ausführlichen Erläuterungen zur Datenkapselung sicher an der technischen Umsetzung interessiert. Die beiden Felder (`zaehler`, `nenner`) werden durch eine **private**-Deklaration vor direkten Zugriffen durch fremde Klassen geschützt. Demgegenüber werden die Methoden über den Modifikator **public** für die Verwendung in klassenfremden Methoden freigegeben. Für die Klasse selbst wird mit dem Modifikator **public** die Verwendung in beliebigen Java-Programmen erlaubt.[1]

```
public class Bruch {
    private int zaehler;    // wird automatisch mit 0 initialisiert
    private int nenner = 1;

    public void setzeZaehler(int z) {
        zaehler = z;
    }

    public boolean setzeNenner(int n) {
        if (n != 0) {
            nenner = n;
            return true;
        } else
            return false;
    }

    public int gibZaehler() {
        return zaehler;
    }

    public int gibNenner() {
        return nenner;
    }

public void kuerze() {
   // Größten gemeinsamen Teiler mit dem Euklidischen Algorithmus bestimmen
   if (zaehler != 0) {
      int az = Math.abs(zaehler);
      int an = Math.abs(nenner);
      while (az != an)
         if (az > an)
           az = az - an;
         else
           an = an - az;
      zaehler = zaehler / az;
      nenner = nenner / az;
   } else
      nenner = 1;
}
```

[1] Bei der Zugriffsberechtigung spielen in Java die Pakete (und ab Java 9 zusätzlich die Module) eine wichtige Rolle. Jede Klasse gehört zu einem Paket, und per Voreinstellung (ohne Vergabe eines Zugriffsmodifikators) haben die anderen Klassen im selben Paket Zugriff auf eine Klasse und ihre Member (Felder und Methoden). In der Regel sollten die Felder einer Klasse vor dem Zugriff durch andere Klassen im selben Paket geschützt sein.

```
    public void addiere(Bruch b) {
        zaehler = zaehler * b.nenner + b.zaehler * nenner;
        nenner = nenner * b.nenner;
        kuerze();
    }
    public void frage() {
        int n;
        do {
            System.out.print("Zähler: ");
            setzeZaehler(Simput.gint());
        } while (Simput.checkError());
        do {
            System.out.print("Nenner: ");
            n = Simput.gint();
            if (n == 0 && !Simput.checkError())
                System.out.println("Der Nenner darf nicht null werden!\n");
        } while (n == 0);
        setzeNenner(n);
    }

    public void zeige() {
        System.out.printf("   %d\n -----\n   %d\n", zaehler, nenner);
    }
}
```

Allerdings ist das Programm schon zu umfangreich für die bald anstehenden ersten Gehversuche mit der Software-Entwicklung in Java.

Wie Sie bei späteren Beispielen erfahren werden, dienen in einem objektorientierten Programm beileibe nicht alle Klassen zur Modellierung des Aufgabenbereichs. Es sind auch Objekte aus der Welt des Computers zu repräsentieren (z. B. Fenster der Bedienoberfläche, Netzwerkverbindungen, Störungen des normalen Programmablaufs).

1.1.3 Algorithmen

Am Anfang von Abschnitt 1.1 wurden mit der *Modellierung des Anwendungsbereichs* und der *Realisierung von Algorithmen* zwei wichtige Aufgaben der Software-Entwicklung genannt, von denen die letztgenannte bisher kaum zur Sprache kam. Auch im weiteren Verlauf des Manuskripts wird die explizite Diskussion von Algorithmen (z. B. hinsichtlich Korrektheit, Terminierung und Aufwand) keinen großen Raum einnehmen. Wir werden uns intensiv mit der Programmiersprache Java sowie der zugehörigen Standardbibliothek beschäftigen und dabei mit möglichst einfachen Beispielprogrammen (Algorithmen) arbeiten. Damit die Beschäftigung mit Algorithmen im Kurs nicht ganz fehlt, werden wir im Rahmen des Bruchrechnungsbeispiels alternative Verfahren zum Kürzen von Brüchen betrachten.

Unser Einführungsbeispiel verwendet in der Methode `kuerze()` den bekannten und nicht gänzlich trivialen **Euklidischen Algorithmus**, um den größten gemeinsamen Teiler (GGT) von Zähler und Nenner eines Bruchs zu bestimmen, durch den zum optimalen Kürzen beide Zahlen zu dividieren sind. Im Euklidischen Algorithmus wird die leicht zu beweisende Aussage genutzt, dass für zwei natürliche Zahlen (1, 2, 3, …) u und v ($u > v > 0$) der GGT gleich dem GGT von v und (u - v) ist:

Ist t ein Teiler von u und v, dann gibt es natürliche Zahlen t_u und t_v mit $t_u > t_v$ und

$$u = t_u \cdot t \quad \text{sowie} \quad v = t_v \cdot t$$

Folglich ist t auch ein Teiler von (u - v), denn:

$$u - v = (t_u - t_v) \cdot t$$

Ist andererseits t ein Teiler von u und (u – v), dann gibt es natürliche Zahlen t_u und t_d mit $t_u > t_d$ und

$$u = t_u \cdot t \quad \text{sowie} \quad (u - v) = t_d \cdot t$$

Folglich ist t auch ein Teiler von v:

$$u - (u - v) = v = (t_u - t_d) \cdot t$$

Weil die Paare (u, v) und $(v, u - v)$ dieselben Mengen gemeinsamer Teiler besitzen, sind auch die größten gemeinsamen Teiler identisch.

Beim Übergang von

$$(u, v) \quad \text{mit } u > v > 0$$

zu

$$(v, u - v) \quad \text{mit } v > 0 \text{ und } u - v > 0$$

wird die größere von den beiden Zahlen durch eine kleinere ersetzt, während der GGT identisch bleibt.

Wenn v und $(u - v)$ in einem Prozessschritt identisch werden, ist der GGT gefunden. Das muss nach endlich vielen Schritten passieren, denn:

- Solange die beiden Zahlen im aktuellen Schritt k noch *verschieden* sind, resultieren im nächsten Schritt k+1 zwei neue Zahlen mit einem kleineren Maximum.
- Alle Zahlen bleiben > 0.
- Das Verfahren endet in endlich vielen Schritten, eventuell mit $v = u - v = 1$.

Weil die Zahl 1 als trivialer Teiler zugelassen ist, existiert zu zwei natürlichen Zahlen immer ein größter gemeinsamer Teiler, der eventuell gleich 1 ist.

Diese Ergebnisse werden in der Methode `Kuerze()` folgendermaßen ausgenutzt:

> Es wird geprüft, ob Zähler und Nenner identisch sind. Trifft dies zu, ist der GGT gefunden (identisch mit Zähler und Nenner). Anderenfalls wird die größere der beiden Zahlen durch deren Differenz ersetzt, und mit diesem vereinfachten Problem startet das Verfahren neu.

Man erhält auf jeden Fall in endlich vielen Schritten zwei identische Zahlen und damit den GGT, der eventuell gleich 1 ist.

Der beschriebene Algorithmus eignet sich dank seiner Einfachheit gut für das Einführungsbeispiel, ist aber in Bezug auf den erforderlichen Berechnungsaufwand nicht optimal. In einer Übungsaufgabe zu Abschnitt 3.7 sollen Sie eine erheblich effizientere Variante implementieren.

1.1.4 Startklasse und main() - Methode

Bislang wurde im Einführungsbeispiel aufgrund einer objektorientierten Analyse des Aufgabenbereichs die Klasse `Bruch` entworfen und in Java realisiert. Wir verwenden nun die Klasse `Bruch` in einer Konsolenanwendung zur Addition von zwei Brüchen. Dabei bringen wir einen Akteur ins Spiel, der in einem einfachen sequentiellen Handlungsplan `Bruch`-Objekte erzeugt und ihnen Nachrichten zustellt, die (zusammen mit dem Verhalten des Anwenders) den Programmablauf voranbringen.

In diesem Zusammenhang ist von Bedeutung, dass es in *jedem* Java - Programm eine **Startklasse** geben muss, die eine Methode mit dem Namen **main()** in ihrem klassenbezogenen Handlungsrepertoire besitzt. Beim Starten eines Programms wird die (aufgrund des Startkommandos bekannte) Startklasse aufgefordert, ihre Klassenmethode **main()** auszuführen. Wegen der besonderen Rolle dieser Methode ist die Bezeichnung *Hauptmethode* durchaus berechtigt.

Es bietet sich an, die oben angedachte Handlungssequenz des Bruchadditionsprogramms in der obligatorischen **main()** - Methode der Startklasse unterzubringen.

Obwohl prinzipiell möglich, ist es nicht sinnvoll, die auf Wiederverwendbarkeit hin konzipierte Klasse `Bruch` mit der Startmethode für eine sehr spezielle Anwendung zu belasten. Daher definieren wir eine zusätzliche Klasse namens `Bruchaddition`, die nicht als Bauplan für Objekte dienen soll und auch kaum Recycling-Chancen besitzt. Ihr Handlungsrepertoire kann sich auf die Klassenmethode **main()** zur Ablaufsteuerung im Bruchadditionsprogramm beschränken. Indem wir eine *neue* Klasse definieren und dort `Bruch`-Objekte verwenden, wird u.a. gleich demonstriert, wie leicht das Hauptergebnis unserer bisherigen Arbeit (die Klasse `Bruch`) für verschiedene Projekte genutzt werden kann.

In der `Bruchaddition` - Methode **main()** werden zwei Objekte (Instanzen) aus der Klasse `Bruch` erzeugt und mit der Ausführung verschiedener Methoden beauftragt. Beim Erzeugen der Objekte ist eine spezielle Methode der Klasse `Bruch` beteiligt, der sogenannte **Konstruktor** (siehe unten):

Quellcode

```
class Bruchaddition {
    public static void main(String[] args) {
        Bruch b1 = new Bruch(), b2 = new Bruch();

        System.out.println("1. Bruch");
        b1.frage();
        b1.kuerze();
        b1.zeige();

        System.out.println("\n2. Bruch");
        b2.frage();
        b2.kuerze();
        b2.zeige();

        System.out.println("\nSumme");
        b1.addiere(b2);
        b1.zeige();
    }
}
```

Ein- und Ausgabe

```
1. Bruch
Zähler: 20
Nenner: 84
    5
  -----
    21


2. Bruch
Zähler: 12
Nenner: 36
    1
  -----
    3


Summe
    4
  -----
    7
```

Wir haben zur Lösung der Aufgabe, ein Programm für die Addition von zwei Brüchen zu erstellen, zwei Klassen mit folgender Rollenverteilung definiert:

- Die Klasse `Bruch` enthält den Bauplan für die wesentlichen Akteure im Aufgabenbereich. Dort alle Eigenschaften und Handlungskompetenzen von Brüchen zu konzentrieren, hat folgende Vorteile:
  - Die Klasse kann in verschiedenen Programmen eingesetzt werden (Wiederverwendbarkeit). Dies fällt vor allem deshalb so leicht, weil die Objekte sowohl Handlungskompetenzen (Methoden) als auch die erforderlichen Eigenschaften (Felder) besitzen.
    Wir müssen bei der Definition dieser Klasse ihre allgemeine Verfügbarkeit explizit mit dem Zugriffsmodifikator **public** genehmigen. Per Voreinstellung ist eine Klasse nur im eigenen Paket (siehe Kapitel 6) verfügbar.
  - Beim Umgang mit den `Bruch`-Objekten sind wenige Probleme zu erwarten, weil nur klasseneigene Methoden direkten Zugang zu kritischen Eigenschaften haben (Datenkapselung). Sollten doch Fehler auftreten, sind die Ursachen in der Regel schnell identifiziert.
- Die Klasse `Bruchaddition` dient *nicht* als Bauplan für Objekte, sondern enthält eine Klassenmethode **main()**, die beim Programmstart automatisch aufgerufen wird und dann für ei-

nen speziellen Einsatz von Bruch-Objekten sorgt. Mit einer Wiederverwendung des Bruchaddition-Quellcodes in anderen Projekten ist kaum zu rechnen.

In der Regel bringt man den Quellcode jeder Klasse in einer eigenen Datei unter, die den Namen der Klasse trägt, ergänzt um die Namenserweiterung **.java**, sodass im Beispielsprojekt die Quellcodedateien **Bruch.java** und **Bruchaddition.java** entstehen. Weil die Klasse Bruch mit dem Zugriffsmodifikator **public** definiert wurde, *muss* ihr Quellcode in einer Datei mit dem Namen **Bruch.java** gespeichert werden (siehe unten). Es wäre erlaubt, aber nicht sinnvoll, den Quellcode der Klasse Bruchaddition ebenfalls in der Datei **Bruch.java** unterzubringen.

Wie aus den beiden vorgestellten Klassen bzw. Quellcodedateien ein ausführbares Programm entsteht, erfahren Sie im Abschnitt 2.2.

1.1.5 Zusammenfassung zum Abschnitt 1.1

Im Abschnitt 1.1 sollten Sie einen ersten Eindruck von der Software-Entwicklung mit Java gewinnen. Alle dabei erwähnten Konzepte der objektorientierten Programmierung und die technischen Details der Realisierung in Java werden bald systematisch behandelt und sollten Ihnen daher im Moment noch keine Kopfschmerzen bereiten. Trotzdem kann es nicht schaden, an dieser Stelle einige Kernaussagen von Abschnitt 1.1 zu wiederholen:

- Vor der Programmentwicklung findet die objektorientierte Analyse der Aufgabenstellung statt. Dabei werden per Abstraktion die beteiligten Klassen identifiziert.
- Ein Programm besteht aus Klassen. Unsere Beispielprogramme zum Erlernen von elementaren Java-Sprachelementen werden oft mit einer einzigen Klasse auskommen. Praxisgerechte Programme bestehen in der Regel aus mehreren Klassen.
- Eine Klasse ist charakterisiert durch Eigenschaften (Felder) und Handlungskompetenzen (Methoden).
- Eine Klasse dient in der Regel als Bauplan für Objekte, kann aber auch selbst aktiv werden (Methoden ausführen und aufrufen).
- Ein Feld bzw. eine Methode ist entweder den Objekten einer Klasse oder der Klasse selbst zugeordnet.
- In den Methodendefinitionen werden Algorithmen realisiert. Dabei kommen selbst erstellte Klassen zum Einsatz, aber auch vordefinierte Klassen aus diversen Bibliotheken.
- Im Programmablauf kommunizieren die Akteure (Objekte und Klassen) durch den Aufruf von Methoden miteinander, wobei in der Regel noch „externe Kommunikationspartner“ (z. B. Benutzer, andere Programme) beteiligt sind.
- Beim Programmstart wird die Startklasse vom Laufzeitsystem aufgefordert, die Methode **main()** auszuführen. Ein Hauptzweck dieser Methode besteht oft darin, Objekte zu erzeugen und somit „Leben auf die objektorientierte Bühne zu bringen“.

1.2 Java-Programme ausführen

Wer sich schon jetzt von der Nützlichkeit des im Abschnitt 1.1 vorgestellten Bruchadditionsprogramms überzeugen möchte, findet eine ausführbare Version an der im Vorwort angegebenen Stelle im Ordner

...\BspUeb\Einleitung\Bruchaddition\Konsole

1.2.1 Java-Laufzeitumgebung installieren

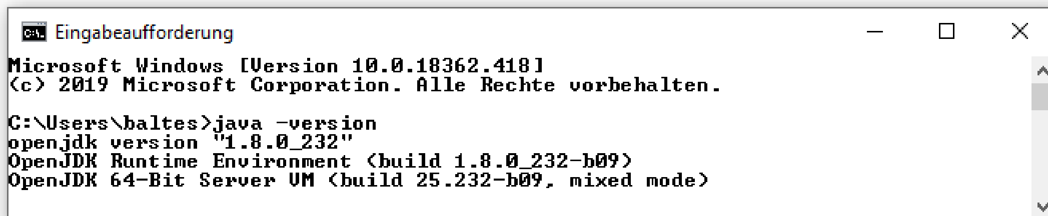
Um das Programm auf einem Rechner ausführen zu können, muss dort eine **Java Virtual Machine** (JVM), die auch als **Java Runtime Environment** (JRE) bezeichnet wird, mit hinreichend aktueller

Version (ab Java 8) installiert sein. Mit den technischen Grundlagen und Aufgaben dieser Ausführungsumgebung für Java-Programme werden wir uns im Abschnitt 1.3.2 beschäftigen.

Um unter Windows festzustellen, ob eine JVM installiert ist, und welche Version diese besitzt, startet man eine Eingabeaufforderung und schickt dort das Kommando

```
>java -version
```

ab, z. B.:



Die Firma Oracle liefert seit ihrer Änderung ihrer Lizenzpolitik im Jahr 2019 keine langfristig durch Updates unterstützte und frei verwendbare JVM mehr aus (siehe Abschnitt 1.3.5). Glücklicherweise sind einige IT-Firmen in die Presche gesprungen. Dabei wird meist das umfassende **Java Development Kit** (JDK) geliefert, das neben einer JVM z. B. auch den Quellcode der Java-Standardbibliothek und einen Java-Compiler enthält. Damit geht es geht über den Bedarf eines *Anwenders* von Java-Software hinaus, was aber bis auf ca. 100 MB verschwendeten Massenspeicherplatz keine Nachteile hat.

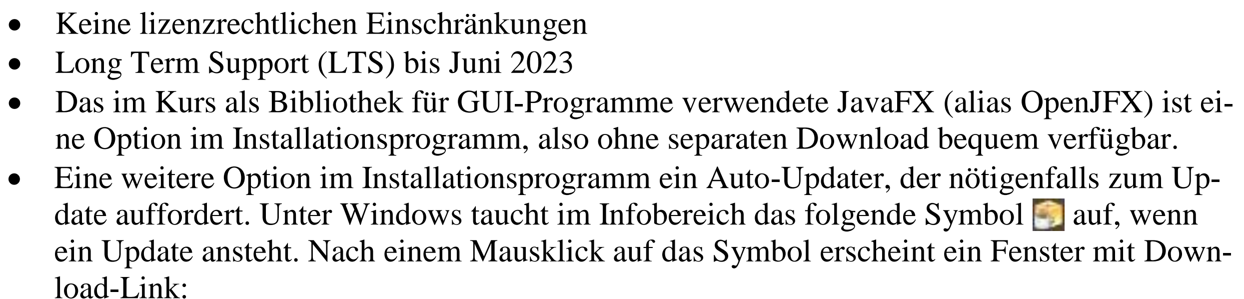
Um eine zeitgemäße Java-Laufzeitumgebung bequem und ohne Lizenzunsicherheit auf einen Windows-Rechner zu befördern, kommt die vom Open Source - Projekt **ojdkbuild**, das von der Firma **Red Hat** gesponsert wird, auf der Webseite

https://github.com/ojdkbuild/ojdkbuild

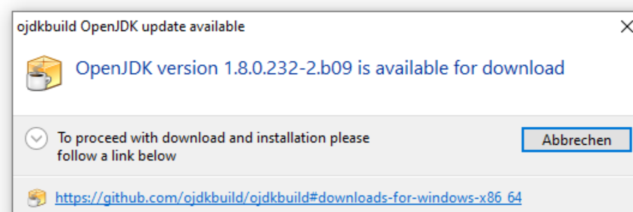
angebotene OpenJDK-Distribution für Java 8 in Frage:[1]

java-1.8.0-openjdk-1.8.0.232-1.b09.ojdkbuild.windows.x86_64.msi

Diese Distribution hat folgende Vorteile.



- Keine lizenzrechtlichen Einschränkungen
- Long Term Support (LTS) bis Juni 2023
- Das im Kurs als Bibliothek für GUI-Programme verwendete JavaFX (alias OpenJFX) ist eine Option im Installationsprogramm, also ohne separaten Download bequem verfügbar.
- Eine weitere Option im Installationsprogramm ein Auto-Updater, der nötigenfalls zum Update auffordert. Unter Windows taucht im Infobereich das folgende Symbol auf, wenn ein Update ansteht. Nach einem Mausklick auf das Symbol erscheint ein Fenster mit Download-Link:



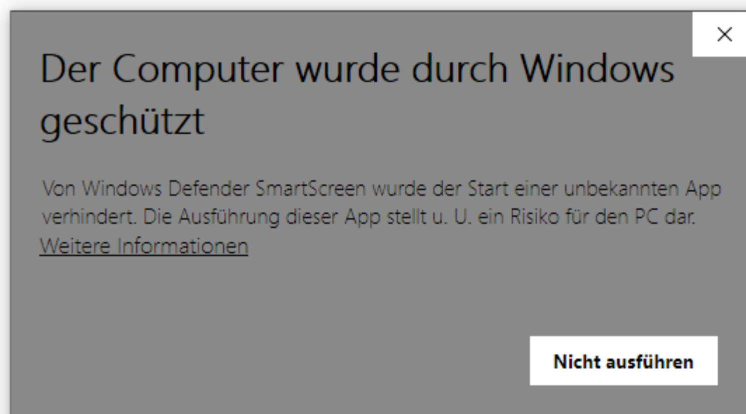
[1] Eine in der Java-Szene seit Jahrzehnten gepflegte Marotte besteht in zwei parallelen Versionierungen. Für die Version 8 erscheint in vielen Dateinamen die Version 1.8. Diese Spiel begann mit Java 2 (alias 1.2) im Jahr 1998. Mit der Version 9 hat zumindest die Firma Oracle die Doppel-Versionierung offenbar aufgegeben.

Eine heruntergeladene und ausgeführte MSI-Datei installierte die neue Version und entfernt die alte.

Obwohl nach Java 8 auch Java 11 als LTS-Version erschienen ist, bestehen Argumente für die beschriebene Distribution:

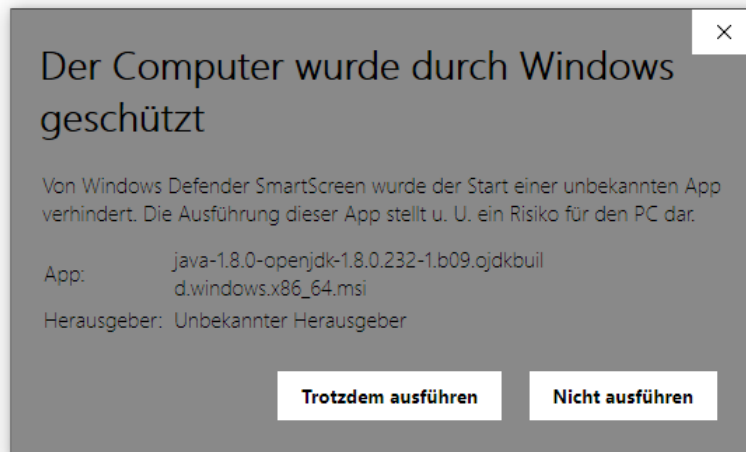
- Java 8 ist aktuell (Oktober 2019) die Version mit der größten Verbreitung.[1] Das liegt auch daran, dass in Java 9 mit dem Modul-System eine wesentlich Architekturveränderung Einzug gehalten hat, die von vielen Entwicklern und Anwendern noch mit Vorsicht betrachtet wird. Bei der Software-Entwicklung mit IntelliJ IDEA kommt das OpenJDK 8 als Laufzeitumgebung dann in Frage, wenn wir uns auf minimale Voraussetzungen auf der Kundenseite beschränken und keine JVM mit unserer Anwendung ausliefern wollen. Um Neuerungen der Java-Technik (Programmiersprache, Standardbibliothek, Laufzeitumgebung) nutzen zu können, werden wir später auch die aktuelleren Java-Versionen 11 und 13 verwenden (siehe Abschnitt 2.1).
- In den ebenfalls verfügbaren ojdkbuild-Distributionen mit Java 11 oder 13 fehlen die GUI-Bibliothek JavaFX und der Auto-Updater. Die sind zwar grundsätzlich nachrüst- bzw. verzichtbar, doch das OpenJDK 8 - Paket aus dem ojdkbuild-Projekt macht beim Einstieg in die Java-Technik weniger Umstände.

Nachdem die Installation per Doppelklick auf die heruntergeladene MSI-Datei gestartet worden ist, meldet unter Windows 10 eventuell die SmartScreen-Funktion Bedenken gegen das Installationsprogramm an:

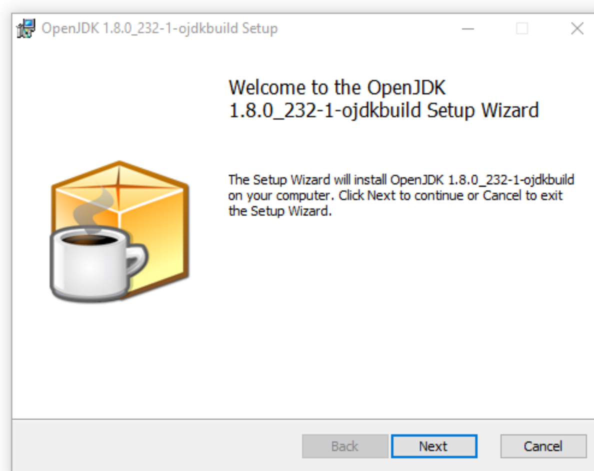


Vorsichtige Menschen lassen in dieser Situation die Datei zunächst von einem Virenschutzprogramm überprüfen. Weil es sich um ein großes Archiv mit ca. 70.000 Dateien handelt, nimmt die Prüfung einige Zeit in Anspruch. Nach bestandenem Test klickt man zunächst auf den Link **Weitere Informationen** und dann auf den Schalter **Trotzdem ausführen**:

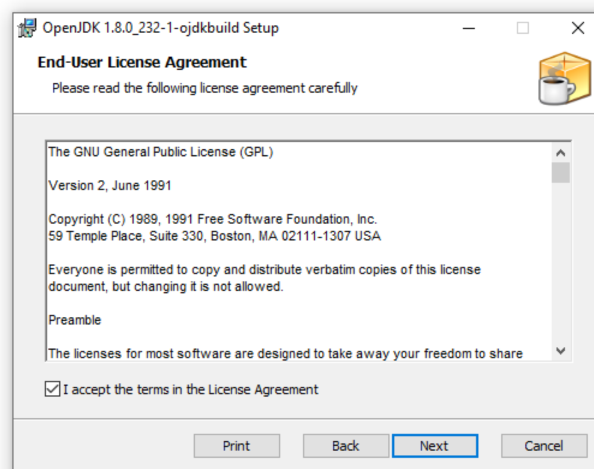
[1] https://sdtimes.com/java/report-java-8-remains-the-most-dominant-version-of-java/



Nach der freundlichen Begrüßung im ersten Dialog des OpenJDK-Installationsprogramms



werden die Lizenzbedingungen vorgelegt:



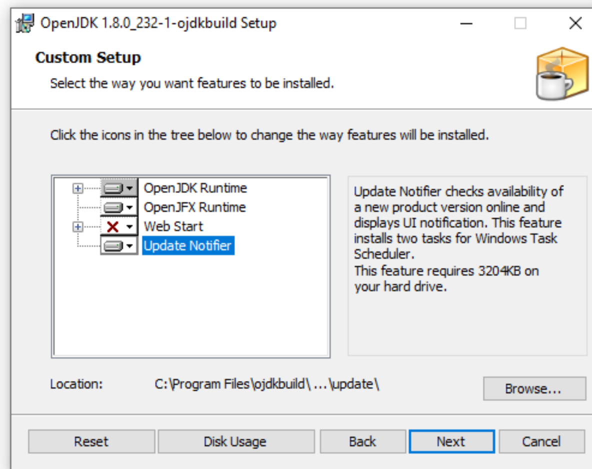
Die vorgelegte Lizenzvereinbarung mit dem Namen

GNU GPL, Version 2, mit CLASSPATH-Ausnahme

ist beim OpenJDK üblich und erlaubt eine liberale, auch kommerzielle Nutzung.[1]

[1] https://github.com/ojdkbuild/ojdkbuild/blob/master/LICENSE, https://openjdk.java.net/legal/gplv2+ce.html

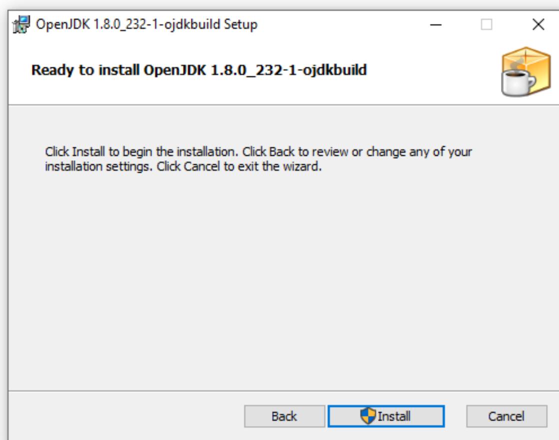
Im Dialog **Custom Setup** sollten Sie die **OpenJFX Runtime** und den **Update Notifier** aktivieren:

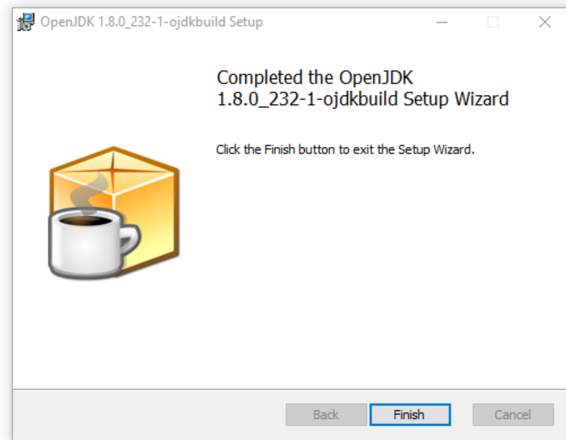


Außerdem kann hier der Installationsordner eingestellt werden. Wir belassen es bei der Voreinstellung:

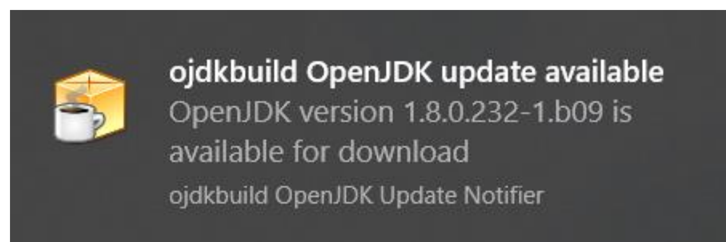
C:\Program Files\ojdkbuild\java-1.8.0-openjdk-1.8.0.232-1

Nach einem Klick auf **Install** und einer positiven Antwort auf die Nachfrage der Benutzerkontensteuerung von Windows (UAC) ist die Installation schnell erledigt:

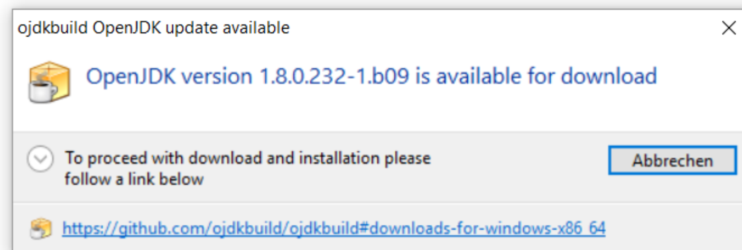




Ist auch der Update Notifier installiert worden, sollte bei einem anstehenden Update ein Infofenster in der unteren rechten Bildschirmecke erscheinen, z. B.:



Ein Klick auf die Information öffnet das folgende Fenster mit Link zur Download-Seite:



1.2.2 Konsolenprogramme ausführen

Nach der Installation einer Java-Laufzeitumgebung (JVM) machen wir uns endlich daran, das Bruchadditionsprogramm zu starten. Kopieren Sie von der oben angegebenen Quelle

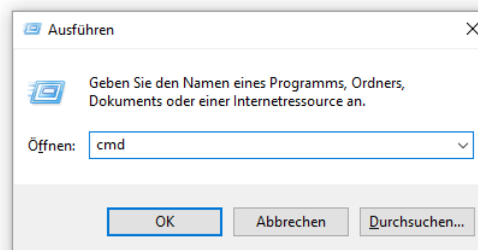
...\BspUeb\Einleitung\Bruchaddition\Konsole

die Dateien **Bruch.class**, **Bruchaddition.class** und **Simput.class** mit ausführbarem Java-Bytecode (siehe Abschnitt 1.3.2) auf einen eigenen Datenträger.

Weil die Klasse `Bruch` wie viele andere im Manuskript verwendete Beispielklassen mit konsolenorientierter Benutzerinteraktion die nicht zur Java-Standardbibliothek gehörige Klasse `Simput` verwendet, muss auch die Klassendatei **Simput.class** übernommen werden. Sobald Sie die zur Vereinfachung der Konsoleneingabe (*Simple Input*) für den Kurs entworfene Klasse `Simput` in eigenen Programmen einsetzen sollen, wird sie näher vorgestellt. Im Abschnitt 2.2.4 lernen Sie eine Möglichkeit kennen, die in mehreren Projekten benötigten **class**-Dateien zentral abzulegen und durch eine passende Definition der Windows-Umgebungsvariablen CLASSPATH allgemein verfügbar zu machen. Dann muss die Datei **Simput.class** nicht mehr in den Ordner eines Projekts kopiert werden, um sie dort nutzen zu können.

Gehen Sie folgendermaßen vor, um die Klasse `Bruchaddition` zu starten:

- Öffnen Sie ein Konsolenfenster, z. B. so:
  - Tastenkombination **Windows + R**
  - Befehl **cmd** eintragen und mit OK ausführen lassen:



- Wechseln Sie zum Ordner mit den **class**-Dateien, z. B.:

```
>u:
>cd \Eigene Dateien\Java\BspUeb\Einleitung\Bruchaddition\Konsole
```

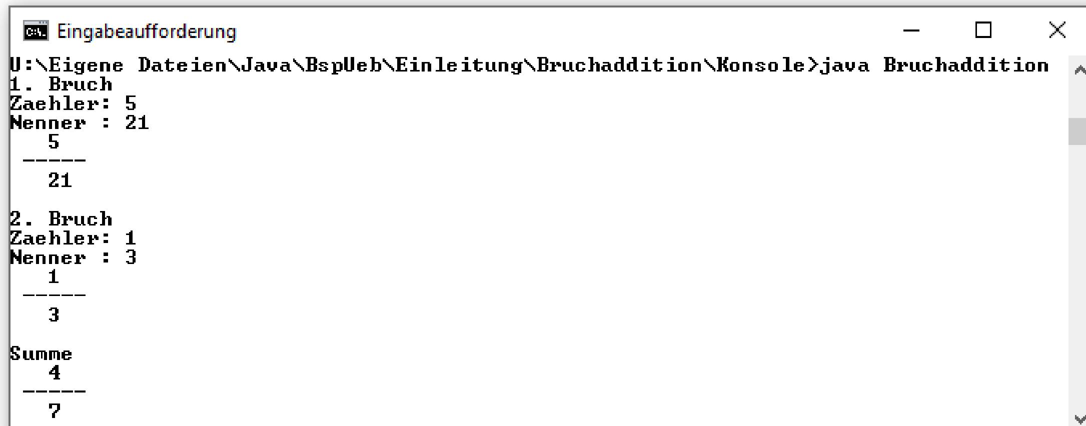
- Starten Sie die Java Runtime Environment über das Programm **java.exe**, und geben Sie als Kommandozeilenargument die Startklasse an, wobei die Groß/Kleinschreibung zu beachten ist:

```
>java Bruchaddition
```

  Damit das zur Java-Laufzeitumgebung gehörende Programm **java.exe** wie im Beispiel ohne Angabe des Installationsordners aufgerufen werden kann, muss der Installationsordner im die Windows-Umgebungsvariable PATH eingetragen worden sein, was bei der im Abschnitt

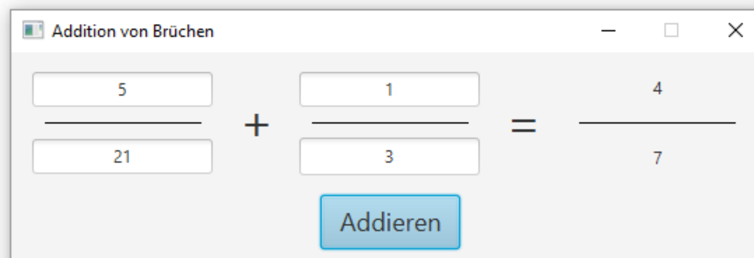
1.2.1 beschriebenen Installation automatisch geschieht. Wie man nötigenfalls einen fehlenden PATH-Eintrag manuell vornehmen kann, wird im Abschnitt 2.2.2 beschrieben.

Ab jetzt sind Bruchadditionen kein Problem mehr:



1.2.3 Ausblick auf Anwendungen mit grafischer Bedienoberfläche

Das obige Beispielprogramm arbeitet der Einfachheit halber mit einer konsolenorientierten Ein- und Ausgabe. Nachdem wir im Manuskript in dieser übersichtlichen Umgebung grundlegende Java-Sprachelemente kennengelernt haben, werden wir uns natürlich auch mit der Programmierung von grafischen Bedienoberflächen beschäftigen. Im folgenden Programm zur Addition von Brüchen wird die oben definierte Klasse `Bruch` verwendet, wobei anstelle ihrer Methoden `frage()` und `zeige()` jedoch grafikorientierte Techniken zum Einsatz kommen:

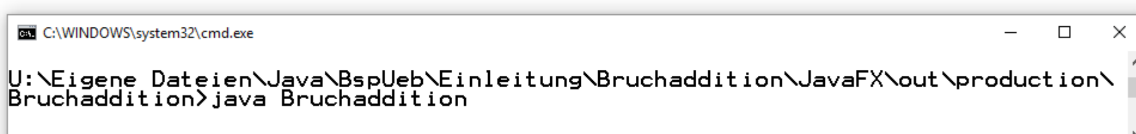


Mit dem Quellcode zur Gestaltung der grafischen Bedienoberfläche könnten Sie im Moment noch nicht allzu viel anfangen. Nach der Lektüre des Manuskripts werden Sie derartige Anwendungen aber mit Leichtigkeit erstellen, zumal die Erstellung grafischer Bedienoberflächen durch die GUI-Technologie JavaFX (alias OpenJFX) und den Fensterdesigner Scene Builder erleichtert wird (siehe Abschnitt 2.5).

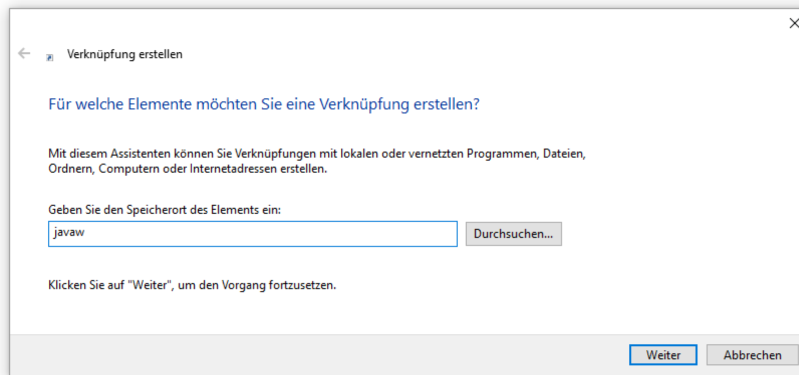
Zum Ausprobieren des Programms startet man mit Hilfe einer Java-Laufzeitumgebung *mit* OpenJFX-Unterstützung (vgl. Abschnitt 1.2.1 zur Installation) aus dem Ordner

...\BspUeb\Einleitung\Bruchaddition\JavaFX\out\production\Bruchaddition

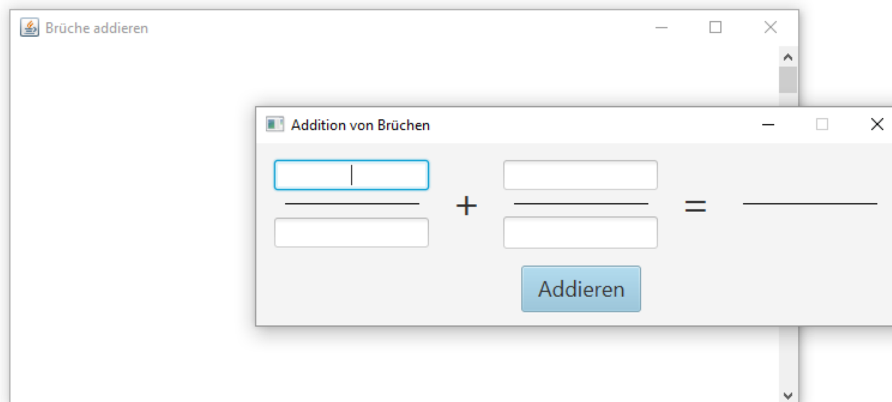
die Klasse `Bruchaddition`:



Um das Programm unter Windows per Doppelklick starten zu können, legt man eine Verknüpfung zum konsolenfreien JVM-Startprogramm **javaw.exe** an, z. B. über das Kontextmenü zu einem Fenster des Windows-Explorers (Befehl **Neu > Verknüpfung**):

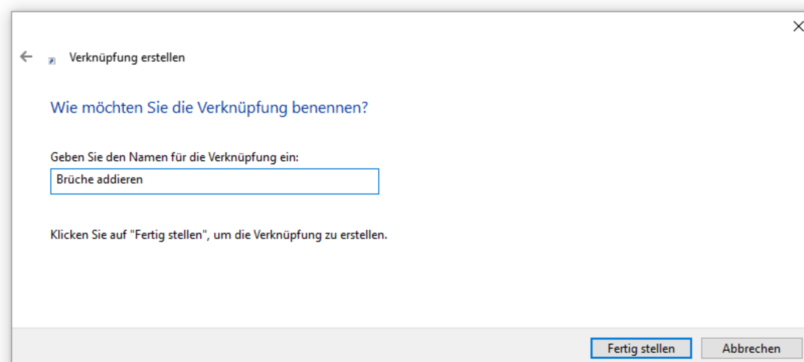


Weil das Programm keine Konsole benötigt, sondern ein Fenster als Bedienoberfläche anbietet, verwendet man bei der Link-Definition als JVM-Startprogramm die Variante **javaw.exe** (mit einem *w* am Ende des Namens). Bei Verwendung von **java.exe** als JVM-Startprogramm würde zusätzlich zum Bruchadditionsprogramm ein leeres Konsolenfenster erscheinen:

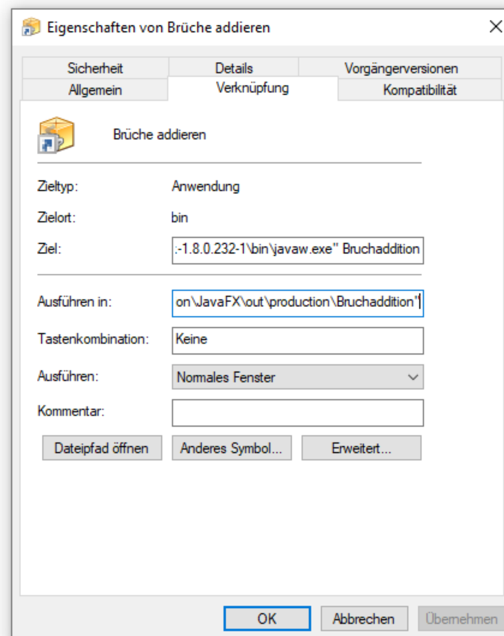


Während das Konsolenfenster beim normalen Programmablauf leer bleibt, erscheinen dort bei einen Laufzeitfehler hilfreiche diagnostische Ausgaben. Daher ist ein Programmstart *mit* Konsolenfenster (per **java.exe**) bei der Fehlersuche durchaus sinnvoll.

Im nächsten Dialog des Assistenten für neue Verknüpfungen trägt man den gewünschten Namen der Link-Datei ein:



Im Eigenschaftsdialog zur **fertiggestellten** Verknüpfungsdatei ergänzt man in Feld **Ziel** hinter **javaw.exe** den Namen der Startklasse und trägt im Feld **Ausführen in** den Ordner ein, in dem sich die Klasse befindet, z. B.:



Nun genügt zum Starten des Programms ein Doppelklick auf die Verknüpfung:

Im eben beschriebenen Verfahren muss die verwendete JVM eine JavaFX-Unterstützung enthalten, was z. B. nach den im Abschnitt 1.2.1 beschriebenen Installation der OpenJDK-Distribution 8 aus dem **ojdkbuild**-Projekt der Fall ist. Ist z. B. die ohne JavaFX (alias OpenJFX) ausgelieferte Open-JDK-Distribution 13.0.1 der Firma Oracle in

C:\Program Files\Java\OpenJDK-13

und die hier

https://gluonhq.com/products/javafx/

frei verfügbare JavaFX-Distribution 13.0.1 der Firma Gluon in[1]

C:\Program Files\Java\OpenJFX-SDK-13

installiert (vgl. Abschnitt 2.5), dann taugt das folgende Kommando zum Starten des Programms

```
>"C:\Program Files\Java\OpenJDK-13\bin\javaw.exe" --module-path "C:\Program
Files\Java\OpenJFX-SDK-13\lib" --add-modules javafx.controls,javafx.fxml
Bruchaddition
```

aus einer Konsole, die auf den Ordner mit der Datei **Bruchaddition.class** positioniert ist. Den Anwendern sollte dieses Kommando z. B. mit Hilfe einer Verknüpfung erspart werden.

Professionelle Java-Programme werden oft als Java-Archivdatei (mit der Namenserweiterung **.jar**, siehe Abschnitte 6.1.3 und 6.2.6) ausgeliefert und sind unter Windows nach einer korrekten JVM-Installation über einen Doppelklick auf diese Datei zu starten. Im Ordner

...\BspUeb\Einleitung\Bruchaddition\JavaFX\out\artifacts

finden Sie die (von IntelliJ erstellte) Datei **Bruchaddition.jar** mit dem grafischen Bruchadditionsprogramm. Es kann auf einem Windows-Rechner per Doppelklick gestartet werden, wenn z. B. ...

---

[1] Im Manuskript werden die Bezeichnungen *JavaFX* und *OpenJFX* synonym verwendet.

- die im Abschnitt 1.2.1 beschriebene Installation der OpenJDK 8 - Distribution aus dem ojdkbuild-Projekt ausgeführt wurde (inklusive OpenJFX),
- in der Registry ...
  - der Schlüssel
    HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.jar
    im Standardwert den Eintrag **jarfile** besitzt,
  - der Schlüssel
    HKEY_LOCAL_MACHINE\SOFTWARE\Classes\jarfile\shell\open\command
    im Standardwert den folgenden Eintrag besitzt:
    "C:\Program Files\ojdkbuild\java-1.8.0-openjdk-1.8.0.232-1\bin\javaw.exe" -jar "%1" %*

Nach einem Doppelklick auf die folgende Datei werden die beschriebenen Registry-Änderungen vorgenommen:

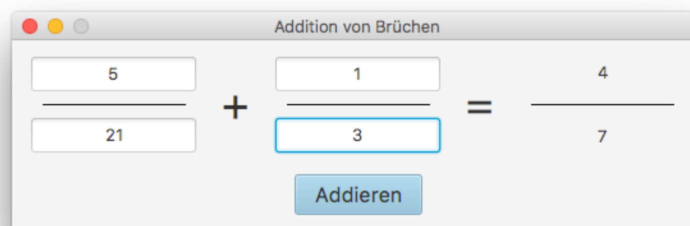
...\BspUeb\Einleitung\Bruchaddition\JavaFX\out\artifacts\ojdkbuild.reg

Die Datei sollte nicht leichtfertig angewendet werden, weil sich die Reaktion des betroffenen Rechners auf einen Doppelklick auf eine **jar**-Datei ändert.

Professionelle Java-Programme bringen oft eine JVM mit (z. B. basierend auf dem OpenJDK) und sind damit nicht darauf angewiesen, dass sich auf dem Kundenrechner eine JVM befindet. Damit bürden sich die Anbieter professioneller Java-Programme aber die Pflicht auf, Sicherheitsupdates für die integrierte JVM zu liefern. Seit Java 9 ermöglicht das JPMS (siehe Abschnitt 6.2) die Erstellung einer angepassten modularen Laufzeitumgebung, die ausschließlich vom Programm benötigte Module enthält (siehe Abschnitt 6.2.8). Das spart sehr viel Platz und reduziert die Gefahr, von einer entdeckten Sicherheitslücke betroffen zu sein. Für kleinere Programme bleibt es aber eine sinnvolle Option, dem Anwender der (meist kostenlosen) Software die Installation einer JVM zu überlassen (siehe Abschnitt 1.2.1). Dann entfällt für den Programmanbieter die Verpflichtung, auf Sicherheitsprobleme in der Java-Laufzeitumgebung zu reagieren. Auch in der Java - Software-Technik kommen Sicherheitsprobleme vor, allerdings vergleichsweise selten (siehe dazu Abschnitt 1.3.6.3).

1.2.4 Ausführung auf einer beliebigen unterstützten Plattform

Dank der Portabilität (Binärkompatibilität) von Java können wir z. B. das im letzten Abschnitt vorgeführte, unter Windows entwickelte Bruchadditionsprogramm auch unter anderen Betriebssystemen ausführen, z. B. unter Mac OS X. Wird die am Ende von Abschnitt 1.2.3 erwähnte Datei **Bruchaddition.jar** auf einen Mac mit installierter Java-Laufzeitumgebung ab Version 8 (inkl. JavaFX-Unterstützung) kopiert, dann lässt sich das Programm dort per Doppelklick starten. Es erscheint die vertraute Bedienoberfläche mit dem Mac OS X - üblichen Fensterdekor:



1.3 Die Java-Softwaretechnik

Bisher war von der Programmiersprache Java und gelegentlich etwas ungenau vom Laufzeitsystem die Rede. Nach der Lektüre dieses Abschnitts werden Sie ein gutes Verständnis von den **drei Säulen der Java-Softwaretechnik** besitzen:

- Die **Programmiersprache** mit dem **Compiler**, der Quellcode in Bytecode wandelt
- Die **Standardklassenbibliothek** mit ausgereiften Lösungen für (fast) alle Routineaufgaben
- Die **Java Virtual Maschine** (JVM) mit zahlreichen Funktionen bei der Ausführung von Bytecode (z. B. optimierender JIT-Compiler, Klassenlader, Sicherheitsüberwachung)

1.3.1 Herkunft und Bedeutung der Programmiersprache Java

Weil auf der indonesischen Insel Java eine auch bei Programmierern hoch geschätzte Kaffee-Sorte wächst, kam die in diesem Manuskript vorzustellende Programmiersprache Gerüchten zufolge zu ihrem Namen.

Java wurde ab 1990 von einem Team der Firma Sun Microsystems unter Leitung von James Gosling entwickelt (siehe z. B. Gosling et al. 2019). Nachdem erste Pläne zum Einsatz in Geräten aus dem Bereich der Unterhaltungselektronik (z. B. Set-Top-Boxen für TV-Geräte) wenig Erfolg brachten, orientierte man sich stark am boomenden Internet. Das zuvor auf die Darstellung von Texten und Bildern beschränkte WWW (Word Wide Web) wurde um die Möglichkeit bereichert, kleine Java-Programme (*Applets* genannt) von einem Server zu laden und ohne lokale Installation im Fenster des Internet-Browsers auszuführen. Ein erster Durchbruch gelang 1995, als die Firma Netscape die Java-Technologie in die Version 2.0 ihres WWW-Navigators integrierte. Kurze Zeit später wurden mit der Version 1.0 des Java Development Kits Werkzeuge zum Entwickeln von Java-Applets und -Anwendungen frei verfügbar.

Mittlerweile hat sich Java als sehr vielseitig einsetzbare Programmiersprache etabliert, die als de-facto - Standard für die plattformunabhängige Entwicklung gelten kann und wohl von allen Programmiersprachen den größten Verbreitungsgrad besitzt. Diesen Eindruck vermittelt jedenfalls eine Auswertung der folgenden Ranglisten, die sich hinsichtlich der verwendeten Datenquellen und vorgenommenen Gewichtungen deutlich unterscheiden:

- TIOBE Programming Community Index (Oktober 2019, Java-Rangplatz: 1)
  http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
- PYPL PopularitY of Programming Language (Oktober 2019, Java-Rangplatz: 2)
  http://pypl.github.io/PYPL.html
- RedMonk Programming Language Rankings (Juni 2019, Java-Rangplatz: 2)
  https://redmonk.com/sogrady/2019/07/18/language-rankings-6-19/
- IEEE Spectrum: The Top Programming Languages (September 2019, Java-Rangplatz: 2)
  https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019

Mit einem durchschnittlichen Rangplatz von 1,75 (die Statistik-Puristen mögen mir diese Auswertungstechnik verzeihen) steht Java vor der härtesten Konkurrenz:

Sprache	Mittlerer Rang
Java	1,75
Python	2
JavaScript	4,5
C++	4,75
C	5
C#	5,25

Außerdem ist Java relativ leicht zu erlernen und daher für den Einstieg in die professionelle Programmierung eine gute Wahl.

Die Java-Designer haben sich stark an den Programmiersprachen **C** und **C++** orientiert, sodass sich Umsteiger von diesen sowohl im Windows- als auch im Linux/UNIX - Bereich verbreiteten Sprachen schnell in Java einarbeiten können. Wesentliche Ziele bei der Weiterentwicklung waren Ein-

fachheit, Robustheit, Sicherheit und Portabilität. Auf die Darstellung der Verwandtschaftsbeziehungen von Java zu den anderen Programmiersprachen bzw. Softwaretechnologien wird hier verzichtet (siehe z. B. Goll et al 2000, S. 15). Jedoch sollen wichtige Eigenschaften beschrieben werden, weil sie eventuell relevant sind für die Entscheidung zum Einsatz der Sprache und zur Kursteilnehme bzw. Manuskriptlektüre.

1.3.2 Quellcode, Bytecode und Maschinencode

Im Abschnitt 1.1 haben Sie Java als eine Programmiersprache kennengelernt, die Ausdrucksmittel zur Modellierung des Anwendungsbereichs und zur Formulierung von Algorithmen bereitstellt. Unter einem *Programm* wurde dabei der vom Entwickler zu formulierende *Quellcode* verstanden. Während *Sie* derartige Texte bald mit Leichtigkeit lesen und begreifen werden, kann die CPU (*Central Processing Unit*) eines Rechners nur einen maschinenspezifischen Satz von Befehlen verstehen, die als Folge von Nullen und Einsen (= *Maschinencode*) formuliert werden müssen. Die ebenfalls CPU-spezifische Assembler-Sprache stellt eine für Menschen besser lesbare Form des Maschinencodes dar. Mit dem Assembler- bzw. Maschinenbefehl

```
mov eax, 4
```

einer CPU aus der x86-Familie wird z. B. der Wert 4 in das EAX-Register (ein Speicherort im Prozessor) geschrieben. Die CPU holt sich einen Maschinenbefehl nach dem anderen aus dem Hauptspeicher und führt ihn aus, wobei heutzutage (2020) die CPU eines handelsüblichen Arbeitsplatzrechners (mit GHz-Taktfrequenz und zahlreichen Kernen/Threads) mehrere hundert Milliarden Befehle pro Sekunde (*Instructions Per Second*, IPS) schafft.[1] Ein Quellcode-Programm muss also erst in Maschinencode übersetzt werden, damit es von einem Rechner ausgeführt werden kann. Dies geschieht bei Java aus Gründen der Portabilität und Sicherheit in zwei Schritten.

Übersetzen: Quellcode → Bytecode

Der (z. B. mit einem beliebigen Texteditor verfasste) Quellcode wird vom **Compiler** in einen maschinen-unabhängigen **Bytecode** übersetzt. Dieser besteht aus den Befehlen einer von der Firma Sun Microsystems bzw. dem Nachfolger Oracle definierten **virtuellen Maschine**, die sich durch ihren vergleichsweise einfachen Aufbau gut auf aktuelle Hardware-Architekturen abbilden lässt. Wenngleich der Bytecode von den heute üblichen Prozessoren noch nicht direkt ausgeführt werden kann, hat er doch bereits die meisten Verarbeitungsschritte auf dem Weg vom Quell- zum Maschinencode durchlaufen. Sein Name geht darauf zurück, dass die Instruktionen der virtuellen Maschine jeweils genau ein Byte (= 8 Bit) lang sind.

Ansätze zur Entwicklung von *realen* Java-Prozessoren, die Bytecode direkt (in Hardware) ausführen können, haben bislang keine nennenswerte Bedeutung erlangt. Die CPU-Schmiede ARM, deren Prozessoren auf mobilen und eingebetteten Systemen stark verbreitetet sind, hat eine Erweiterung namens *Jazelle DBX* (*Direct Bytecode eXecution*) entwickelt, die zumindest einen großen Teil der Bytecode-Instruktionen in Hardware unterstützt.[2] Allerdings macht das auf Geräten mit ARM-Prozessor oft eingesetzte (und überwiegend mit der Ausführung von Java-Software beschäftigte) Betriebssystem Android der Firma Google von Jazelle DBX keinen Gebrauch. In aktuellen ARM-Prozessoren spielt die mittlerweile als veraltet und überflüssig betrachtete Jazelle-Erweiterung keine Rolle mehr (Langbridge 2014, S. 48).

Den im kostenlosen **Java Development Kit** (**JDK**), das von der Firma Oracle und der Java Community entwickelt wird enthaltenen Compiler **javac.exe** setzen auch manche Java-

---

[1] https://de.wikipedia.org/wiki/Instruktionen_pro_Sekunde
https://en.wikipedia.org/wiki/Instructions_per_second

[2] http://en.wikipedia.org/wiki/Jazelle

Entwicklungsumgebungen im Hintergrund ein, z. B. die im Kurs bevorzugte Entwicklungsumgebung IntelliJ IDEA. Mit den JDK-Lizenzbedingungen werden wir uns im Abschnitt 1.3.5 beschäftigen. Die OpenJDK 8 - Distribution aus dem ojdkbuild-Projekt haben wir schon im Abschnitt 1.2.1 installiert. Im Abschnitt 2.1 folgt noch die OpenJDK-Version 13.

Quellcode-Dateien tragen in Java die Namenserweiterung **.java**, Bytecode-Dateien die Erweiterung **.class**.

Interpretieren: Bytecode → Maschinencode

Abgesehen von den seltenen Systemen mit realem Java-Prozessor muss für jede Betriebssystem/CPU - Kombination mit Java-Unterstützung ein (naturgemäß plattformabhängiger) **Interpreter** erstellt werden, der den Bytecode zur Laufzeit in die jeweilige Maschinensprache übersetzt. Man verwendet die eben im Sinne eines Quasi-Hardware-Designs eingeführte Bezeichnung *virtuelle Maschine* (***J****ava* ***V****irtual* ***M****achine*, *JVM*) auch für die an der Ausführung von Java-Bytecode beteiligte Software, also sozusagen für die Emulation des Java-Prozessors in Software. Man benötigt also für jede reale Maschine eine partiell vom jeweiligen Betriebssystem und von der konkreten CPU abhängige JVM, um den Java-Bytecode auszuführen. Diese Software wird meist in der Programmiersprache C++ realisiert.

Für viele Desktop-Betriebssysteme (Linux, MacOS, Solaris, Windows) ist eine Java-Laufzeitumgebung kostenlos verfügbar. Über eine lange Zeit haben die Firma Sun und der Aufkäufer Oracle ein zur Ausführung, aber nicht zur Entwicklung von Java-Programmen geeignetes Softwarepaket namens **Java Runtime Environment** (**JRE**) kostenlos zur Verfügung gestellt und durch Updates unterstützt. Für Entwickler wurde das **Java Development Kit** (inklusive Compiler) kostenlos angeboten. Mit Java 8 endet die JRE-Verteilung durch Oracle, und seit April 2019 darf die JRE 8 der Firma Oracle nur noch für private Zwecke kostenlos genutzt werden. Beginnend mit der Java-Version 9 ist nur noch das JDK-Paket verfügbar, das aber auch eine Java-Laufzeitumgebung enthält. Java-Entwickler sind mit dem JDK gut bedient, und Java-Anwender müssen lediglich einen irrelevanten Massenspeicher-Mehrverbrauch hinnehmen. Außerdem liefert die Firma Oracle nur noch 6 Monate lang (bis zum Erscheinen der nächsten Hauptversion) kostenlose Updates für ein JDK. Einige Java-Versionen erhalten aber über die 6 Monate hinaus eine ca. 5 Jahre dauernde Langzeitunterstützung (LTS), also eine Versorgung durch Updates. Das ist zugesichert für die Versionen 8 und 11 und geplant für die Version 17. Während bei Oracle der LTS auf die private Nutzung beschränkt ist, gewähren andere Firmen den LTS auch für kommerziell genutzte JDK-Installationen (siehe Abschnitt 1.3.5).

Die wichtigsten Komponenten der Java-Laufzeitumgebung sind:

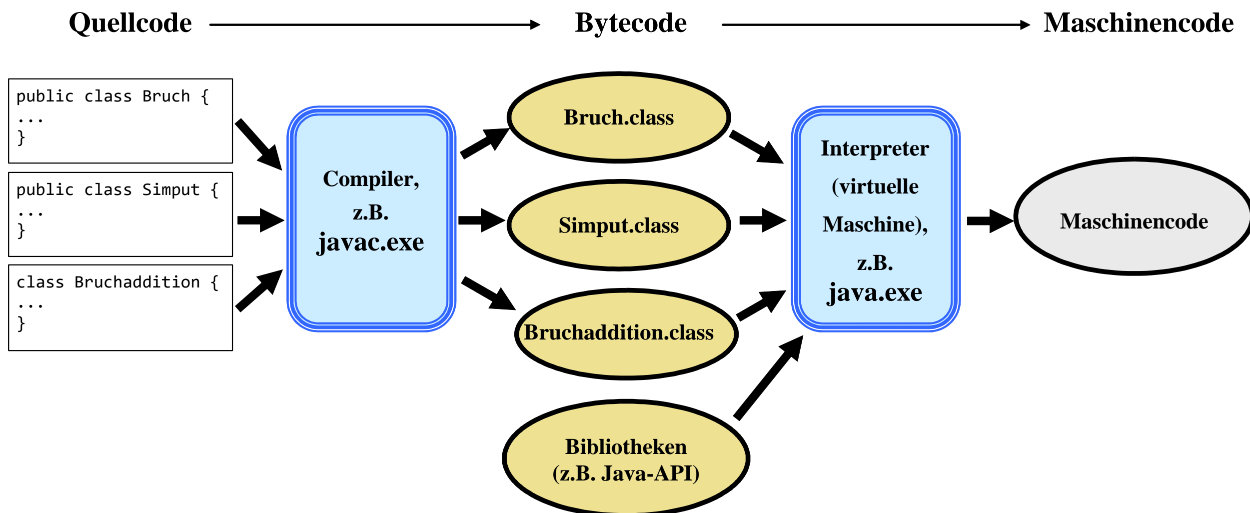
- JVM
  Neben der Bytecode-Übersetzung erledigt die JVM bei der Ausführung eines Java-Programms noch weitere Aufgaben, z. B.:
  - Der Klassenlader befördert die vom Programm benötigten Klassen in den Speicher und nimmt dabei eine Bytecode-Verifikation vor, um potentiell gefährliche Aktionen zu verhindern.
  - Die Speicherverwaltung entfernt automatisch die im Programmablauf überflüssig gewordenen Objekte (Garbage Collection).
- Java-Standardbibliothek mit Klassen für (fast) alle Routineaufgaben (siehe Abschnitt 1.3.3)

Wie Sie bereits aus Abschnitt 1.2 wissen, startet man unter Windows mit **java.exe** bzw. **javaw.exe** die Ausführungsumgebung für ein Java-Programm (mit Konsolen- bzw. Fensterbedienung) und gibt als Parameter die Startklasse des Programms an.

Mittlerweile kommen bei der Ausführung von Java-Programmen leistungssteigernde Techniken (**J**ust-**in**-**T**ime - Compiler, **HotSpot** - Compiler mit Analyse des Laufzeitverhaltens) zum Einsatz,

welche die Bezeichnung *Interpreter* fraglich erscheinen lassen. Allerdings ändert sich nichts an der Aufgabe, aus dem plattformunabhängigen Bytecode den zur aktuellen CPU passenden Maschinencode zu erzeugen. So wird wohl keine Verwirrung gestiftet, wenn in diesem Manuskript weiterhin vom *Interpreter* die Rede ist.

In der folgenden Abbildung sind die beiden Übersetzungen auf dem Weg vom Quell- zum Maschinencode durch den Compiler **javac.exe** (aus dem JDK) und den Interpreter **java.exe** (aus der JVM) am Beispiel des Bruchrechnungsprojekts (vgl. Abschnitt 1.1) im Überblick zu sehen:



1.3.3 Die Standardklassenbibliothek der Java-Plattform

Damit die Programmierer nicht das Rad (und ähnliche Dinge) ständig neu erfinden müssen, bietet die Java-Plattform eine Standardbibliothek mit fertigen Klassen für nahezu alle Routineaufgaben, die oft als **API** (***A****pplication* ***P****rogram* ***I****nterface*) bezeichnet wird. Im Manuskript werden Sie zahlreiche API-Klassen kennenlernen; eine vollständige Behandlung ist wegen des enormen Umfangs unmöglich und auch nicht erforderlich.

Bevor man selbst eine Klasse oder Methode entwickelt, sollte man unbedingt die Standardbibliothek auf die Existenz einer Lösung untersuchen, denn die Lösungen in der Standardbibliothek ...

- sind leistungsoptimiert und sorgfältig getestet,
- werden ständig weiterentwickelt.

Durch die Verwendung der Standardbibliothek steigert man in der Regel die Qualität der entstehenden Software, spart viel Zeit und verbessert auch noch die Lesbarkeit des Quellcodes, weil die Lösungen der Standardbibliothek vielen Entwicklern vertraut sind (Bloch 2018, S. 267ff).

Wir halten fest, dass die Java-Technologie einerseits auf einer Programmiersprache basiert, dass andererseits aber die Funktionalität im Wesentlichen von einer umfangreichen Standardbibliothek beigesteuert wird, deren Klassen in jeder virtuellen Java-Maschine zur Verfügung stehen.

Die Java-Designer waren bestrebt, sich auf möglichst wenige, elementare Sprachelemente zu beschränken und alle damit formulierbaren Konstrukte in der Standardbibliothek unterzubringen. Es resultierte eine sehr kompakte Sprache (siehe Gosling et al. 2019), die nach ihrer Veröffentlichung im Jahr 1995 lange Zeit nahezu unverändert blieb.

Neue Funktionalitäten werden in der Regel durch eine Erweiterung der Java-Klassenbibliothek realisiert, sodass hier erhebliche Änderungen stattfinden. Einige Klassen sind mittlerweile als *deprecated* (überholt, nicht mehr zu benutzen) eingestuft worden. Gelegentlich stehen für eine Aufgabe

verschiedene Lösungen aus unterschiedlichen Entwicklungsstadien zur Verfügung (z. B. bei den Multithreading-Lösungen für die nebenläufige Programmausführung).

Mit der 2004 erschienenen Version 5 (alias 1.5) hat auch die Programmiersprache Java substantielle Veränderungen erfahren (z. B. generische Typen, Auto-Boxing). Auch die Version 8 (alias 1.8) hat mit der funktionalen Programmierung (den Lambda-Ausdrücken) eine wesentliche Erweiterung der Programmiersprache Java gebracht.

In Kurs bzw. Manuskript steht zunächst die Programmiersprache Java im Vordergrund. Mit wachsender Kapitelnummer geht es aber auch darum, wichtige Pakete der Standardbibliothek mit Lösungen für Routineaufgaben kennenzulernen, z. B.:

- Kollektionen zur Verwaltung von Listen, Mengen oder (Schlüssel-Wert) - Tabellen
- Lesen und Schreiben von Dateien
- Multithreading

Neben der sehr umfangreichen Standardbibliothek, die integraler Bestandteil der Java-Plattform ist, sind aus diversen Quellen unzählige Java-Klassen für diverse Problemstellungen verfügbar.

1.3.4 Java-Editionen für verschiede Einsatzszenarien

Weil die Java-Plattform so mächtig und vielgestaltig geworden ist, wurden drei Editionen für spezielle Einsatzfelder definiert, wobei sich vor allem die jeweiligen Standardklassenbibliotheken unterscheiden:

- **Java Standard Edition (JSE)** zur Entwicklung von Software für Arbeitsplatzrechner
  Darauf wird sich das Manuskript beschränken.
- **Java Enterprise Edition (JEE)** für unternehmensweite oder serverorientierte Lösungen
  Bei der Java Enterprise Edition (JEE) kommt exakt dieselbe Programmiersprache wie bei der Java Standard Edition (JSE) zum Einsatz. Für die erweiterte Funktionalität sorgt eine entsprechende Variante der Standardklassenbibliothek. Beide Editionen verfügen über eine eigenständige Versionierung, wobei die JSE meist eine etwas höhere Versionsnummer besitzt (aktuell im Oktober 2019: JSE 13 und JEE 8). Die JEE ist im Herbst 2017 von der Firma Oracle an die Open Source Community (vertreten durch die Eclipse Foundation) übergeben worden. Als Alternative zur JEE für die Entwicklung von Java-Unternehmenslösungen hat sich das *Spring-Framework* etabliert.[1]
- **Java Micro Edition (JME)** für Kommunikationsgeräte und eingebettete Lösungen
  Diese Edition wurde einst für Mobiltelefone mit beschränkter Leistung konzipiert. Bei heutigen Smartphones kann aber von eingeschränkter Leistung kaum noch die Rede sein, und die JME ist dementsprechend ins Hintertreffen geraten. Neben großer Skepsis zu den Überlebenschance der JME gibt es Bestrebungen zu einer Neuausrichtung für den Einsatz bei eingebetteten Lösungen (Stichwort: Internet der Dinge).[2]

Wir werden uns im Manuskript weder mit der JEE noch mit der JME beschäftigen, doch sind erworbene Java-Programmierkenntnisse natürlich dort uneingeschränkt verwendbar, und elementare Klassen der JSE-Standardbibliothek sind auch in den anderen Editionen verfügbar.

Weil sich die Standardklassenbibliotheken der Editionen unterschieden, muss man z. B. vom *JSE-API* sprechen, wenn man die JSE-Standardbibliothek meint. Im Manuskript wird gelegentlich die Bezeichnung *Java-API* verwendet, wenn eine Aussage für alle Java-Edition gelten soll.

---

[1] Siehe: https://de.wikipedia.org/wiki/Spring_(Framework)

[2] Siehe: http://www.javaworld.com/article/2848210/java-me/java-me-8-and-the-internet-of-things.html

Im Marktsegment der Smartphones und Tablet-Computer hat sich eine Entwicklung vollzogen, welche die ursprüngliche Konzeption der Java-Editionen durcheinander gewirbelt hat. Einfache Mobiltelefone wurden von Smartphones mit GHz-Prozessoren verdrängt. Während die Firma Apple bisher in ihrem iPhone- und iPad-Betriebssystem **iOS** *keine* Java-Unterstützung bietet, hat der Konkurrent Google in seinem Smartphone - und Tablet - Betriebssystem **Android** Java lange als Standardsprache zur Anwendungsentwicklung eingesetzt. Neuerdings tendiert Google zur Programmiersprache Kotlin, die vom IntelliJ-Urheber JetBrains entwickelt wird, doch ist kein Ende des Java-Supports in Android abzusehen. Kotlin wird wie Java in Bytecode für eine virtuelle Maschine übersetzt, sodass eine hohe Kompatibilität zwischen den beiden Programmiersprachen besteht. In Android kommt eine alternative Bytecode-Technik zum Einsatz mit einer virtuellen Maschine namens **Dalvik** (bis Android 4.4) bzw. **ART** (seit Android 5.0).[1]

Es spricht für das Potential von Java, dass diese Sprache auch eine sehr wichtige Rolle bei der Entwicklung von Android-Apps spielt. Android trägt erheblich zur Attraktivität von Java bei, denn Android hat auf dem Markt für Smartphone-Betriebssysteme einen Marktanteil von ca. 85% erreicht[2] und zeigt auch auf dem Tablet-Markt eine ähnliche Dominanz.[3] Somit ist Android derzeit die am stärksten verbreitete Plattform für klientenseitige Java-Programmierung (Mednieks et al., 2013, S. 41).

Mit den Lernerfahrungen aus dem Kurs bzw. Manuskript können Sie zügig in die Software-Entwicklung für Android einsteigen, müssen sich aber mit einer speziellen Software-Architektur auseinandersetzen, die zum Teil aus der Smartphone-Hardware resultiert (z. B. kleines Display, Zwang zum Energiesparen wegen der begrenzten Akkukapazität). Zur Einführung in die Entwicklung von Android-Apps in Java bietet das ZIMK ein Manuskript an (Baltes-Götz 2018).

1.3.5 Update- und Lizenzpolitik von Oracle

Seit Java 7 ist das OpenJDK (mit Compiler, Laufzeitumgebung und Standardbibliothek) die offizielle Referenzimplementation der Java Standard Edition. An der Weiterentwicklung ist neben der Firma Oracle auch die Java Community beteiligt, zu der u.a. zahlreiche namhafte Firmen gehören (z. B. Red Hat[4], IBM[5], Microsoft[6]). Das OpenJDK steht unter der liberalen (GPLv2 & CPE) - Lizenz und darf zur Software-Entwicklung sowie im produktiven Einsatz frei verwendet werden.[7]

Allerdings ist die OpenJDK-Unterstützung durch Updates auf die 6 Monate bis zum Erscheinen der nächsten Java-Hauptversion beschränkt. Von der Firma Oracle ist stets ein aktuelles und sicheres OpenJDK zu beziehen, doch muss man alle 6 Monate auf eine neue Hauptversion umsteigen. Ein problemloses Update ist zwar wahrscheinlich, aber nicht garantiert. Wird das OpenJDK zusammen mit einer eigenen Anwendung ausgeliefert, muss den Kunden ein Update-Verfahren auf die neue Java-Hauptversion angeboten werden.

---

[1] Die in Smartphone-CPUs mit ARM-Design vorhandene reale Java-Maschine namens Jazelle DBX wird von Android ignoriert und von der Prozessor-Schmiede ARM mittlerweile als veraltet und überflüssig betrachtet (Langbridge 2014, S. 48). Aktuelle ARM-Prozessoren setzen auf einen Befehlssatz namens *ThumbEE*, der sich gut für die JIT - Übersetzung von Bytecode in Maschinencode eignet.

[2] Quelle: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[3] Quelle: https://www.statista.com/statistics/272446/global-market-share-held-by-tablet-operating-systems/

[4] https://jaxenter.de/red-hat-openjdk-java-82758
Red Hat wurde mittlerweile von IBM übernommen.

[5] https://developer.ibm.com/blogs/ibm-and-java-looking-forward-to-the-future/

[6] https://blogs.microsoft.com/blog/2019/08/19/microsoft-acquires-jclarity-to-help-optimize-java-workloads-on-azure/
https://www.theserverside.com/opinion/Microsoft-vs-IBM-A-major-shift-in-Java-support

[7] https://openjdk.java.net/legal/gplv2+ce.html

OpenJDK-Hauptversionen erscheinen seit der Version 9 in einem halbjährlichen Release-Zyklus, um die Weiterentwicklung zu beschleunigen.[1] Die jeweils aktuelle Version wird nur ein halbes Jahr lang (bis zum Erscheinen der nächsten Hauptversion) mit Updates versorgt. So sind wir aktuell (Oktober 2019) bei Version 13 angekommen.

Für einige Java-Versionen wird ein Long Term Support (LTS) zugesichert (aktuell für die Versionen 8 und 11, geplant für die Version 17), sodass die Versorgung mit (Sicherheits-)Updates über einen Zeitraum von ca. 5 Jahren garantiert ist. Damit bestehen für Entwickler und Anwender stabile Verhältnisse, und es fallen lediglich (Sicherheits-)Updates ohne Kompatibilitätsrisiko an. Während die Firma Oracle bei LTS-Versionen nach Ablauf des 6-monatigen Standard-Supports für die nichtprivate Verwendung Lizenzgebühren verlangt, versorgen andere Firmen die LTS-Versionen ohne Kosten und Lizenzeinschränkungen mit Updates. In der Regel werden dabei verschiedene Betriebssysteme unterstützt und bequeme Installationsprogramme geliefert (siehe Abschnitt 2.1.2).

Java ist trotz der neuen Lizenzpolitik der Firma Oracle eine frei verfügbare Entwicklungsplattform, wenngleich der Zugriff auf eine mit Updates versorgte Version nun für Entwickler und Anwender etwas mehr Aufmerksamkeit erfordert. Es ist in der IT-Industrie durchaus üblich, dass mit Open Source - Software Geld verdient wird, und man kann es der Firma Oracle nicht verdenken, dass sie sich auch um ihre Bilanzen kümmert.

1.3.6 Wichtige Merkmale der Java-Softwaretechnik

In diesem Abschnitt werden zentrale Merkmale der Java-Softwaretechnik beschrieben, wobei Vorgriffe auf die spätere Behandlung wichtiger Themen nicht zu vermeiden sind.

1.3.6.1 Objektorientierung

Java wurde als objektorientierte Sprache konzipiert und erlaubt im Unterschied zu hybriden Sprachen wie C++ oder Delphi außerhalb von Klassendefinitionen keine Anweisungen. In unserem Einleitungsbeispiel wurde einiger Aufwand in Kauf genommen, um einen realistischen Eindruck von objektorientierter Programmierung (OOP) zu vermitteln (siehe Abschnitt 1.1). Oft trifft man auf Einleitungsbeispiele, die zwar angenehm einfach aufgebaut sind, aber außer gewissen Formalitäten kaum Merkmale der objektorientierten Programmierung aufweisen. Hier wird die gesamt Funktionalität in die **main()** - Methode der Startklasse und eventuell in weitere statische Methoden der Startklasse gezwängt. Später werden auch wir solche pseudo-objektorientierten (POO-) Programme benutzen, um elementare Java-Sprachelemente in möglichst einfacher Umgebung kennenzulernen. Aus den letzten Ausführungen ergibt sich u.a., dass Java zwar eine objektorientierte Programmierweise nahelegen und unterstützen, aber nicht erzwingen kann.

Nachdem das objektorientierte Paradigma die Software-Entwicklung über Jahrzehnte dominiert hat, gewinnt das ältere, aber lange Zeit auf akademische Diskurse beschränkte **funktionale Paradigma** in den letzten Jahren an Bedeutung. Ein wesentlicher Grund ist seine gute Eignung für die zur optimalen Nutzung moderner Mehrkern-CPUs erforderliche nebenläufige Programmierung (Horstmann 2014b). Seit der Version 8 unterstützt Java wichtige Techniken bzw. Prinzipien der funktionalen Programmierung (z. B. Lambda-Ausdrücke).

1.3.6.2 Portabilität

Die im Abschnitt 1.3.2 beschriebene Übersetzungsprozedur führt zusammen mit der Tatsache, dass sich Bytecode-Interpreter für aktuelle IT-Plattformen relativ leicht implementieren lassen, zur guten Portabilität von Java. Man mag einwenden, dass sich der Quellcode vieler Programmiersprachen

---

[1] https://jaxenter.de/java-jdk-release-zyklus-75402

(z. B. C++) ebenfalls auf verschiedenen Rechnerplattformen kompilieren lässt. Diese **Quellcode-Portabilität** aufgrund weitgehend genormter Sprachdefinitionen und verfügbarer Compiler ist jedoch auf einfache Anwendungen mit textorientierter Benutzerschnittstelle beschränkt und stößt selbst dort auf manche Detailprobleme (z. B. durch verschiedenen Zeichensätze). C++ wird zwar auf vielen verschiedenen Plattformen eingesetzt, doch kommen dabei in der Regel **plattformabhängige Funktions- bzw. Klassenbibliotheken** zum Einsatz (z. B. GTK unter Linux, MFC unter Windows).[1] Bei Java besitzt hingegen bereits die zuverlässig in jeder JVM verfügbare Standardbibliothek mit ihren insgesamt ca. 4000 Klassen weitreichende Fähigkeiten für die Gestaltung grafischer Bedienoberflächen, für Datenbank- und Netzwerkzugriffe usw., sodass sich plattformunabhängige Anwendungen mit modernem Funktionsumfang und Design realisieren lassen.

Weil der von einem Java-Compiler erzeugte Bytecode von jeder JVM (mit passender Version) ausgeführt werden kann, bietet Java nicht nur Quellcode- sondern auch **Binärportabilität**. Ein Programm ist also ohne erneute Übersetzung auf verschiedenen Plattformen einsetzbar.

1.3.6.3 Sicherheit

Beim Design der Java-Technologie wurde das Thema *Sicherheit* gebührend berücksichtigt. Weil ein als Bytecode übergebenes Programm durch die beim Empfänger installierte virtuelle Maschine vor der Ausführung auf unerwünschte Aktivitäten geprüft wird, können viele Schadwirkungen verhindert werden.

Leider hat sich die Sicherheitstechnik der Java-Laufzeitumgebung im Jahr 2013 mehrfach als löchrig erwiesen. Von den Risiken, die oft voreilig und unreflektiert auf die *gesamte* Java-Technik bezogen wurden, waren allerdings überwiegend die von Webservern bezogenen *Applets* betroffen, die im Internet-Browser-Kontext mit Hilfe von Plugins ausgeführt werden. Diese (unabhängig von der Sicherheitsproblematik schon lange und oft totgesagten) Java-*Applets* sind strikt von lokal installierten Java-*Anwendungen* für Desktop-Rechner zu unterscheiden.[2] Noch weniger als Java-Anwendungen für Desktop-Rechner waren die außerordentlich wichtigen Server-Anwendungen (z. B. erstellt mit der Java Enterprise Edition oder alternativen Frameworks wie Spring) sowie die Java-Apps für Android-Geräte von der damaligen Sicherheitsmisere betroffen.

Generell ist natürlich auch Java-Software nicht frei von Sicherheitsproblemen und muss (wie das Betriebssystem, die Browser, der Virenschutz und viele andere Programme) stets aktuell gehalten werden. Das gilt insbesondere für JVM-Installationen.

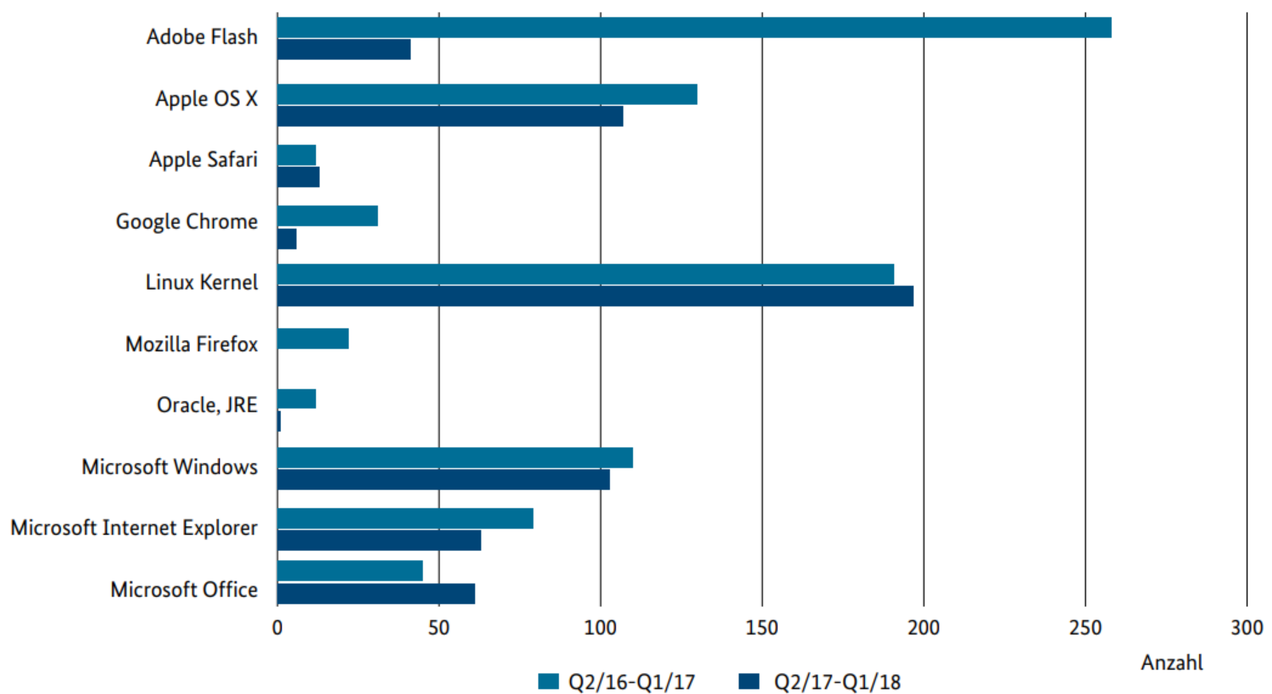
Offenbar hat die Firma Oracle aus den ärgerlichen und peinlichen Problemen des Jahres 2013 gelernt. Das Bundesamt für Sicherheit in der Informationstechnik stellt in seinem Jahresbericht 2018 zur IT-Sicherheit in Deutschland bei der Java-Laufzeitumgebung (Oracle JRE) relativ wenige kritische Schwachstellen (kritische CVE-Einträge) bei stark sinkender Tendenz fest:[3]

---

[1] Dass es grundsätzlich möglich ist, eine C++ - Klassenbibliothek mit umfassender Funktionalität (z. B. auch für die Gestaltung grafischer Bedienoberflächen) für verschiedene Plattformen herzustellen und so für Quellcode-Portabilität bei modernen, kompletten Anwendungen zu sorgen, beweist die Firma Trolltech mit ihrem Produkt Qt.

[2] Im Java-Kurs bzw. -Manuskript des Rechenzentrums der Uni Trier werden Applets seit ca. 10 Jahren nicht mehr behandelt.

[3] https://www.bmi.bund.de/SharedDocs/downloads/DE/publikationen/themen/it-digitalpolitik/bsi-lagebericht-2018.pdf?__blob=publicationFile&v=3
Im BSI-Jahresbericht 2019 ist leider kein vergleichbares Diagramm enthalten.



1.3.6.4 Robustheit

In diesem Abschnitt werden Gründe für die hohe Robustheit (Stabilität) von Java-Software genannt, wobei auch die anschließend noch separat behandelte Einfachheit eine große Rolle spielt.

Die Programmiersprache Java verzichtet auf Merkmale von C++, die erfahrungsgemäß zu Fehlern verleiten, z. B.:

- Pointer-Arithmetik
- Benutzerdefiniertes Überladen von Operatoren
- Mehrfachvererbung

Außerdem werden die Programmierer zu einer systematischen Behandlung der bei einem Methodenaufruf potentiell zu erwartenden Ausnahmefehler gezwungen.

Schließlich leistet die hohe Qualität der Java-Standardbibliothek einen Beitrag zur Stabilität der Software.

1.3.6.5 Einfachheit

Schon im Zusammenhang mit der Robustheit wurden einige komplizierte und damit fehleranfällige C++ - Bestandteile erwähnt, auf die Java bewusst verzichtet. Zur Vereinfachung trägt auch bei, dass Java keine Header-Dateien benötigt, weil die Bytecodedatei einer Klasse alle erforderlichen Metadaten enthält. Weiterhin kommt Java ohne Präprozessor-Anweisungen aus, die in C++ den Quellcode vor der Übersetzung modifizieren oder Anweisungen an die Arbeitsweise des Compilers enthalten können.[1]

Wenn man dem Programmierer eine Aufgabe komplett abnimmt, kann er dabei keine Fehler machen. In diesem Sinn wurde in Java der sogenannte **Garbage Collector** (dt.: *Müllsammler*) imple-

[1] Der Gerüchten zufolge im früher verbreiteten Textverarbeitungsprogramm *StarOffice* (Vorläufer der Open Source Programme *OpenOffice* und *LibreOffice*) über eine Präprozessor-Anweisung realisierte Unfug, im Quellcode den Zugriffsmodifikator **private** vor der Übergabe an den Compiler durch die schutzlose Alternative **public** zu ersetzen, ist also in Java ausgeschlossen.

mentiert, der den Speicher nicht mehr benötigter Objekte automatisch freigibt. Im Unterschied zu C++, wo die Freigabe durch den Programmierer zu erfolgen hat, sind damit typische Fehler bei der Speicherverwaltung ausgeschlossen:

- Ressourcenverschwendung durch überflüssige Objekte (Speicherlöcher)
- Programmabstürze beim Zugriff auf voreilig entsorgte Objekte

Insgesamt ist Java im Vergleich zu C/C++ deutlich einfacher zu beherrschen und damit für Einsteiger eher zu empfehlen.

Es existieren mehrere hochwertige Java-Entwicklungsumgebungen (z. B. *Eclipse*, *IntelliJ IDEA*, *NetBeans*), die das Erstellen des Quellcodes erleichtern (z. B. durch Code-Vervollständigung, Refaktorierung), Beiträge zur Qualitätssteigerung leisten (z. B. durch Code-Analyse, Testunterstützung) und meist kostenlos verfügbar sind.

1.3.6.6 Multithreading

Java unterstützt Anwendungen mit mehreren, parallel laufenden Ausführungsfäden (Threads). Solche Anwendungen bringen erhebliche Vorteile für den Benutzer, der z. B. mit einem Programm interagieren kann, während es im Hintergrund aufwändige Berechnungen ausführt oder auf die Antwort eines Netzwerk-Servers wartet. Andererseits kommt es vor, dass zwar nur *eine* Aufgabe ansteht (z. B. Transkodieren eines Films, Überprüfung vieler Dateien auf Schädlinge), dabei jedoch durch Beteiligung mehrerer Threads eine erhebliche Beschleunigung im Vergleich zum traditionellen Single-Thread-Betrieb erzielt werden kann. Weil mittlerweile Mehrkern- bzw. Mehrprozessor-Systeme üblich sind, wird für Programmierer die Multithreading-Beherrschung immer wichtiger.

Die zur Erstellung nebenläufiger Programme attraktive funktionale Programmierung wird in Java seit der Version 8 unterstützt.

1.3.6.7 Netzwerkunterstützung

Java ist gut vorbereitet zur Realisation von verteilten Anwendungen auf Basis des TCP/IP – Protokolls. Die Kommunikation kann über Sockets oder über höhere Protokolle wie z. B. **HTTP** (*Hypertext Transfer Protocol*) abgewickelt werden.

1.3.6.8 Performanz

Der durch Sicherheit (Bytecode-Verifikation), Stabilität (z. B. Garbage Collector) und Portabilität verursachte Performanznachteil von Java-Programmen (z. B. gegenüber C++) ist durch die Entwicklung leistungsfähiger virtueller Java-Maschinen mittlerweile weitgehend irrelevant geworden, wenn es nicht gerade um performanzkritische Anwendungen (z. B. Spiele) geht. Mit unserer Entwicklungsumgebung IntelliJ IDEA werden Sie eine komplett in Java erstellte, sehr komplexe und dabei flott agierende Anwendung kennenlernen.

1.3.6.9 Beschränkungen

Wie beim Designziel der Plattformunabhängigkeit nicht anders zu erwarten, lassen sich in Java-Programmen sehr spezielle Eigenschaften eines Betriebssystems schlecht verwenden (z. B. die Windows-Registrierungsdatenbank). Wegen der Einschränkungen beim freien Speicher- bzw. Hardware-Zugriff eignet sich Java außerdem kaum zur Entwicklung von Treiber-Software (z. B. für eine Grafikkarte). Für System- bzw. Hardware-nahe Programme ist z. B. C (bzw. C++) besser geeignet.

1.4 Übungsaufgaben zum Kapitel 1

1) Warum steigt die Produktivität der Software-Entwicklung durch objektorientiertes Programmieren?

2) Welche der folgenden Aussagen sind richtig bzw. falsch?

1. Die Programmiersprache Java ist relativ leicht zu erlernen, weil beim Design Einfachheit angestrebt wurde.
2. In Java muss jede Klasse eine Methode namens **main()** enthalten.
3. Die meisten aktuellen CPUs können Java-Bytecode direkt ausführen.
4. Java eignet sich für eine sehr breite Palette von Anwendungen, von Smartphone-Apps über Anwendungsprogramme für Arbeitsplatzrechner bis zur unternehmenswichtigen Server-Software.

2 Werkzeuge zum Entwickeln von Java-Programmen

In diesem Abschnitt werden kostenlose Werkzeuge zum Entwickeln von Java-Anwendungen beschrieben. Zunächst beschränken wir uns puristisch auf einen Texteditor und das **Java Development Kit (Standard Edition)**. In dieser sehr übersichtlichen „Entwicklungsumgebung“ werden die grundsätzlichen Arbeitsschritte und einige Randbedingungen besonders deutlich.

Anschließend gönnen wir uns erheblich mehr Luxus in Form der Open Source - Entwicklungsumgebung **IntelliJ IDEA Community**, die auf vielfältige Weise die Programmentwicklung unterstützt. IntelliJ IDEA bietet u.a.:

- einen Editor mit ...
    - farblicher Unterscheidung verschiedener Syntaxbestandteile
    - Syntaxvervollständigung
    - Unterschlängeln von Fehlern
    - usw.
- einen Debugger, der z. B. Programmänderungen im Testbetrieb erlaubt
- Assistenten zum automatischen Erstellen von Quellcode zu Routineaufgaben
- zahleiche Erweiterungen (ermöglicht durch eine flexible Plugin-Schnittstelle)

Anschließend werden die für Kursteilnehmer bzw. Leser empfohlenen Installationen beschrieben. Alle Pakete sind kostenlos für alle relevanten Betriebssysteme verfügbar.

2.1 Aktuelles OpenJDK installieren

Wir haben im Abschnitt 1.2.1 bereits ein Java Development Kit installiert, waren dabei aber in erster Linie an der enthaltenen Java Virtual Machine (JVM) interessiert. Unsere Entscheidung fiel auf das OpenJDK in der Version 8, zu der das ojdkbuild-Projekt eine komfortable Distribution pflegt mit folgenden Vorteilen:

- Kein relevante Lizenzeinschränkung
- Langzeit-Support bis Juni 2023
- JavaFX (alias OpenJFX) als Option enthalten
- Update-Unterstützung

Das JDK erscheint seit der Version 9 in einem halbjährlichen Release-Zyklus, um die Weiterentwicklung zu beschleunigen.[1] So sind wir aktuell (im Oktober 2019) bei der Version 13 angekommen. Obwohl der Abstand zwischen den Versionen 13 und 8 weniger gewaltig ist, als es der numerische Unterschied vermuten lässt, sind doch einige Neuerungen der Java-Softwaretechnik bzw. der Java-Programmiersprache für uns relevant, z. B.:

- das mit Java 9 eingeführte Modulsystem
- die mit Java 10 eingeführte Typinferenz für lokale Variablen
- die mit Java 11 eingeführte Erweiterung der Typinferenz für Lambda-Parameter
- die mit Java 12 (allerdings nur als Preview) eingeführten **Switch**-Ausdrücke
- die mit Java 13 (allerdings nur als Preview) eingeführten Textblöcke

Daher werden wir im Kurs auch mit dem aktuellen OpenJDK 13 arbeiten.

[1] https://jaxenter.de/java-jdk-release-zyklus-75402

2.1.1 OpenJDK 13 der Firma Oracle

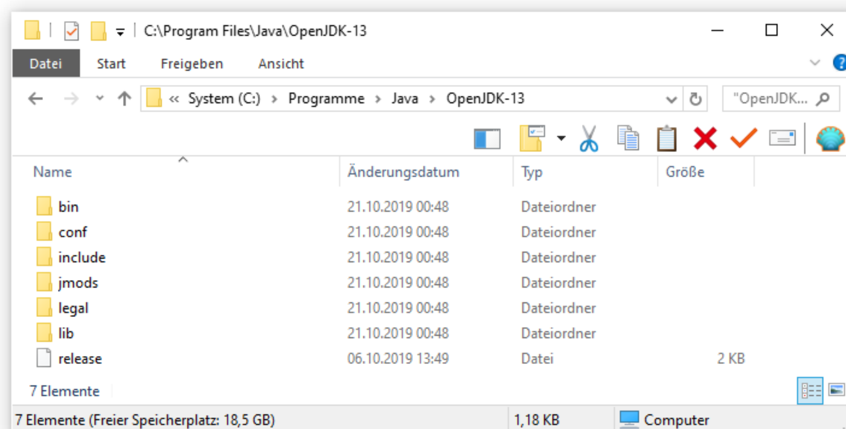
Die OpenJDK 13 - Distribution der Firma **Oracle** ist auf dieser Webseite

https://jdk.java.net/13/

frei verfügbar und ermöglicht es uns, die aktuelle, am 17.09.2019 erschienene, Java-Version 13 zu verwenden, die auch von unserer Entwicklungsumgebung IntelliJ IDEA 2019.2 unterstützt wird (siehe Abschnitt 2.3). Das OpenJDK von Oracle steht unter der (GPLv2 & CPE) - Lizenz und darf zur Software-Entwicklung sowie im produktiven Einsatz frei verwendet werden. Allerdings ist die Unterstützung durch Updates auf die 6 Monate bis zum Erscheinen der nächsten Java-Hauptversion beschränkt.

Eine OpenJDK-Installation auf Basis der Oracle-Distribution erfordert etwas Handarbeit, hat aber den Vorteil der extremen Transparenz. Anschließend wird die Installation unter Windows 10 (64 Bit) beschrieben:

- Laden Sie von der oben angegebenen Webseite das ZIP-Archiv mit der aktuellen Version herunter. Am 21.10.2019 wird die Version 13.0.1 in der Datei **openjdk-13.0.1_windows-x64_bin.zip** geliefert.
- Beim Auspacken des ZIP-Archivs entsteht der Ordner **jdk-13.0.1**.
- Erstellen Sie bei Bedarf den Ordner **C:\Program Files\Java**.
- Kopieren Sie den OpenJDK-Ordner unter dem Namen **OpenJDK-13** in den Ordner **C:\Program Files\Java**:



Eine OpenJDK - Distribution (von Oracle oder von einem alternativen Anbieter, siehe Abschnitt 2.1.2) enthält u.a. …

- den Java-Compiler **javac.exe**, der Java-Quellcode in Java-Bytecode übersetzt
- den Java-Interpreter **java.exe**, der Java-Programme ausführt (Bytecode in Maschinencode übersetzt)
- zahlreiche Werkzeuge (z. B. den Dokumentationsgenerator **javadoc.exe** und den Archivgenerator **jar.exe**)

2.1.2 OpenJDK-Distributionen mit langfristiger Update-Versorgung

Wer mit dem OpenJDK der Firma Oracle arbeitet, ist stets auf dem neuesten Stand, muss aber alle 6 Monate auf eine neue Hauptversion umsteigen. Für die LTS-Versionen (Long Term Support) von Java (aktuell die Versionen 8 und 11, geplant die Version 17) wird die Versorgung mit Updates über einen Zeitraum von ca. 5 Jahren zugesichert. Während die Firma Oracle für LTS-Versionen bei nicht-privater Verwendung Lizenzgebühren verlangt, ist der Service bei anderen Anbietern kostenlos zu haben (vgl. Abschnitt 1.3.5). Ohne Anspruch auf Vollständigkeit sollen genannt werden:

- **OpenJDK aus dem ojdkbuild-Projekt**
  Das von der Firma **Red Hat**, einem Anbieter professioneller Linux-Lösungen, gesponserte Open Source - Projekt **ojdkbuild** bietet auf der Webseite

  https://github.com/ojdkbuild/ojdkbuild

  u.a. OpenJDK - Distributionen mit den LTS-Versionen 8 und 11 an. Wir haben im Abschnitt 1.2.1 das OpenJDK 8 aus dem **ojdkbuild**-Projekt installiert.
- **Amazon Corretto**
  Die Firma **Amazon** bietet auf der Webseite

  https://aws.amazon.com/de/corretto/

  unter dem Namen Corretto OpenJDK - Distributionen mit den LTS-Versionsständen 8 und 11 an.

Leider besitzen die OpenJDK-Distributionen in der Regel keine automatische Update-Routine, sodass sich Entwickler und Anbieter über Sicherheitsupdates informieren müssen.

2.2 Java-Entwicklung mit JDK und Texteditor

2.2.1 Editieren

Um das Erstellen, Übersetzen und Ausführen von Java-Programmen ohne großen Aufwand üben zu können, erstellen wir das unvermeidliche **Hallo**-Programm, das vom bereits erwähnten POO-Typ ist (*pseudo*-objektorientiert):

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Hallo {
    public static void main(String[] args) {
        System.out.println("Hallo allerseits!");
    }
}
```

</td><td>

```
Hallo allerseits!
```

</td></tr>
</table>

Im Unterschied zu *hybriden* Programmiersprachen wie C++ und Delphi, die neben der objektorientierten auch die prozedurale Programmiertechnik erlauben, verlangt Java auch für solche Trivialprogramme eine **Klassendefinition**. Im Beispiel genügt eine einzige Klasse, die den Namen `Hallo` erhält. Es muss eine startfähige Klasse sein, weil eine solche in jedem Java-Programm benötigt wird. In der somit erforderlichen Methode **main()** erzeugt die Klasse `Hallo` aber keine Objekte, wie es die Startklasse `Bruchaddition` im Einstiegsbeispiel tat, sondern beschränkt sich auf eine Bildschirmausgabe.

Immerhin kommt dabei ein vordefiniertes Objekt (**System.out**) zum Einsatz, das durch Aufruf seiner **println()** - Methode mit der Ausgabe beauftragt wird. Durch einen Parameter vom Zeichenfolgentyp wird der Auftrag näher beschrieben.

Das POO-Programm eignet sich aufgrund seiner Kürze zum Erläutern wichtiger Regeln, an die Sie sich so langsam gewöhnen sollten. Alle Themen werden aber später noch einmal systematisch und ausführlich behandelt:

- Nach dem Schlüsselwort **class** folgt der frei wählbare Klassenname. Hier ist wie bei allen Bezeichnern zu beachten, dass Java streng zwischen Groß- und Kleinbuchstaben unterscheidet. Nach einer weitgehend eingehaltenen Konvention beginnt in Java ein Klassenname mit einem Großbuchstaben.

  Weil bei den Klassen der POO-Übungsprogramme im Unterschied zur eingangs vorgestellten `Bruch`-Klasse eine Nutzung durch andere Klassen *nicht* in Frage kommt, wird in der Klassendefinition auf den Modifikator **public** verzichtet. Manche Autoren von Java-

Beschreibungen entscheiden sich für die systematische Verwendung des **public**-Modifikators, z. B.:

```
public class Hallo {
    public static void main(String[] args) {
        System.out.println("Hallo allerseits!");
    }
}
```

Das vorliegende Manuskript orientiert sich am Verhalten der Java-Urheber: Gosling et al. (2019) lassen bei Startklassen, die nur von der JVM angesprochen werden, den Modifikator **public** systematisch weg. Später werden klare und unvermeidbare Gründe für die Verwendung des Klassen-Modifikators **public** beschrieben.

- Dem Kopf der Klassendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf.
- Weil die Klasse `Hallo` startfähig sein soll, muss sie eine Methode namens **main()** besitzen. Diese wird von der JVM beim Programmstart ausgeführt und dient bei „echten“ OOP-Programmen (direkt oder indirekt) dazu, Objekte zu erzeugen (siehe die Klasse `Bruchaddition` im Abschnitt 1.1.4).
- Die Definition der Methode **main()** wird von drei *obligatorischen* Schlüsselwörtern eingeleitet, deren Bedeutung Sie auch jetzt schon (zumindest teilweise) verstehen können:
    - **public**
      Wie eben erwähnt, wird die Methode **main()** beim Programmstart von der JVM gesucht und ausgeführt. Sie muss den Zugriffsmodifikator **public** erhalten. Anderenfalls reklamiert die JVM beim Startversuch:

```
Auswählen C:\WINDOWS\system32\cmd.exe

U:\Eigene Dateien\Java\BspUeb\Einleitung\Hallo\JDK>java Hallo
Fehler: Hauptmethode in Klasse Hallo nicht gefunden. Definieren Sie die Hauptmethode als:
   public static void main(String[] args):
oder eine JavaFX-Anwendung muss javafx.application.Application erweitern
```

    - **static**
      Mit diesem Modifikator wird **main()** als statische, d.h. der Klasse zugeordnete Methode gekennzeichnet. Im Unterschied zu den *Instanz*methoden der *Objekte* werden die statischen Methoden von der Klasse selbst ausgeführt. Die beim Programmstart automatisch ausgeführte **main()** - Methode der Startklasse muss auf jeden Fall durch den Modifikator **static** als Klassenmethode gekennzeichnet werden.
    - **void**
      Die Methode **main()** hat den Rückgabetyp **void**, weil sie keinen Rückgabewert liefert.[1]

  Die beiden Modifikatoren **public** und **static** stehen in beliebiger Reihenfolge am Anfang der Methodendefinition. Ihnen folgt der Rückgabetyp, der unmittelbar vor dem Methodennamen stehen muss.
- In der **Parameterliste** einer Methode, die dem Namen folgt und durch runde Klammern begrenzt wird, kann die gewünschte Arbeitsweise näher spezifiziert werden. Wir werden uns später ausführlich mit diesem wichtigen Thema beschäftigen und beschränken uns hier auf zwei Hinweise:

---

[1] Die Programmiersprachen C, C++ und C# besitzen ebenfalls eine Funktion bzw. Methode namens **main()** bzw. **Main()**, und dort wird (optional) der Rückgabetype **int** verwendet, der beim Verlassen des Programms die Übergabe eines Returncodes an das Betriebssystem erlaubt. In Java hat die **main()** - Methode obligatorisch den Rückgabetyp **void**, doch kann z. B. mit der statischen Methode **exit()** der Klasse **System** ein Returncode an das Betriebssystem übergeben werden (siehe Abschnitt 11.1).

- *Für Neugierige und/oder Vorgebildete*
  Der **main()** - Methode werden über einen Array mit **String**-Elementen die Spezifikationen übergeben, die der Anwender in der Kommandozeile beim Programmstart angegeben hat. In unserem Beispiel kümmert sich die Methode **main()** allerdings nicht um solche Anwenderwünsche.
- *Für Alle*
  Bei einer **main()** - Methode ist die im Beispiel verwendete Parameterliste obligatorisch, weil die JVM ansonsten die Methode beim Programmstart nicht erkennt und mit derselben Fehlermeldung wie bei einem fehlenden **public**-Modifikator reagiert (siehe oben). Den Parameter*namen* (im Beispiel: `args`) darf man allerdings beliebig wählen.

- Dem Kopf einer Methodendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf mit Variablendeklarationen und Anweisungen. Das minimalistische Beispielprogramm beschränkt sich auf eine einzige Anweisung, die einen Methodenaufruf enthält.
- In der **main()** - Methode unserer `Hallo`-Klasse wird die **println()** - Methode des vordefinierten Objekts **System.out** dazu benutzt, einen Text an die Standardausgabe zu senden. Zwischen dem Objekt- und dem Methodennamen steht ein Punkt. Bei einem Methodenaufruf handelt es sich um eine **Anweisung**, die folglich mit einem Semikolon abzuschließen ist.

Es dient der Übersichtlichkeit, zusammengehörige Programmteile durch eine **gemeinsame Einrücktiefe** zu kennzeichnen. Man realisiert die Einrückungen am einfachsten mit der Tabulatortaste, aber auch Leerzeichen sind erlaubt. Für den Compiler sind die Einrückungen irrelevant.

Schreiben Sie den Quellcode mit einem beliebigen Texteditor, unter Windows z. B. mit **Notepad**, und speichern Sie Ihr Quellprogramm unter dem Namen **Hallo.java** in einem geeigneten Verzeichnis, z. B. in

U:\Eigene Dateien\Java\BspUeb\Einleitung\Hallo\JDK

Beachten Sie bitte:

- Die Dateinamenserweiterung muss **.java** lauten.
- Der Dateinamensstamm (vor dem Punkt) sollte unbedingt mit dem Klassennamen übereinstimmen. Ein aus der Missachtung dieser Regel resultierendes Problem ist im Abschnitt 2.2.2 zu sehen. Die vom Compiler erzeugte Bytecode-Datei übernimmt auf jeden Fall den Namen der *Klasse*. Es resultiert also eine Namensabweichung zwischen Quellcode- und Bytecode-Datei, wenn die Quellcodedatei nicht den Namen der Klasse (mit angehängter Erweiterung **.java**) trägt.
- Unter Windows ist beim Dateinamen die Groß-/Kleinschreibung zwar irrelevant, doch sollte auch hier auf exakte Übereinstimmung mit dem Klassennamen geachtet werden.
- Bei einer Klasse mit dem Zugriffsmodifikator **public** (siehe unten) besteht der Compiler darauf, dass der Dateinamensstamm mit dem Klassennamen übereinstimmt (auch hinsichtlich der Groß-/Kleinschreibung).

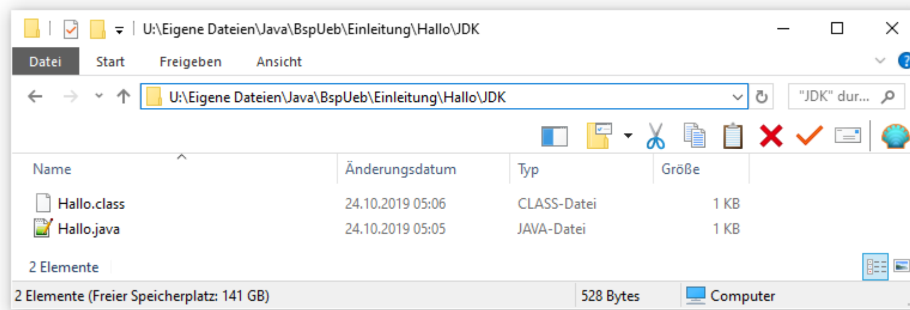
2.2.2 Übersetzen

Öffnen Sie ein Konsolenfenster (auch *Eingabeaufforderung* genannt), und wechseln Sie in das Verzeichnis mit dem neu erstellten Quellprogramm **Hallo.java**.

Lassen Sie das Programm vom OpenJDK 13 - Compiler **javac** übersetzen, z. B.:

```
>"C:\Program Files\Java\OpenJDK-13\bin\javac" Hallo.java
```

Falls beim Übersetzen keine Probleme auftreten, meldet sich der Rechner nach kurzer Arbeitszeit mit einem neuer Kommandoaufforderung zurück, und die Quellcodedatei **Hallo.java** erhält Gesellschaft durch die Bytecode-Datei **Hallo.class**, z. B.:



Damit unter Windows der Compiler ohne Pfadangabe von jedem Verzeichnis aus gestartet werden kann,

```
>javac Hallo.java
```

muss das **bin**-Unterverzeichnis der OpenJDK 13 - Installation (mit dem Compiler **javac.exe**) vorrangig in die Definition der Umgebungsvariablen PATH aufgenommen werden.[1] Nach der im Abschnitt 1.2.1 beschriebenen Installation der OpenJDK-Version 8 ist ein PATH-Eintrag mit einem Java-Compiler vorhanden. Soll auf jeden Fall der Compiler in

C:\Program Files\Java\OpenJDK-13\bin

genutzt werden, dann muss dieser Ordner vorrangig in die **PATH**-Definition aufgenommen werden, was unter Windows 10 z. B. so geschehen kann:

- Suchen: *Umgebungsvariablen*
- Starten: **Systemumgebungsvariablen bearbeiten**
  Bei der Umgebungsvariablen **PATH** erlaubt Windows eine System- und eine Benutzervariante, wobei die Systemvariable Vorrang hat. Enthält die (immer vorhandene) Systemvariable **PATH** einen Ordner mit **javac.exe** (z. B. aufgrund der im Abschnitt 1.2.1 beschriebenen Installation der OpenJDK-Version 8), dann muss diese Variable modifiziert werden, was anschließend beschrieben wird. Ansonsten genügt es, das **bin**-Unterverzeichnis der zu verwendenden JDK-Installation in die **PATH**-Benutzervariable einzutragen.
- Klick auf **Umgebungsvariablen**
- **Systemvariable PATH Bearbeiten**
- **Neuen** Eintrag mit **C:\Program Files\Java\OpenJDK-13\bin** erstellen und ganz **nach oben** befördern

Beim Kompilieren des Quellcodes zu einer Klasse A wird auch der Quellcode zu einer in A benutzten Klasse B neu übersetzt, wenn ...

- der Quellcode zu B im aktuellen Ordner vorhanden, der Bytecode zu B aber nicht verfügbar ist,
- der Quellcode zu B im aktuellen Ordner vorhanden und der Bytecode zu B verfügbar ist (im aktuellen Ordner vorhanden oder via CLASSPATH auffindbar, vgl. Abschnitt 2.2.4), wobei die Bytecode-Datei älter ist als die Quellcodedatei

Sind etwa im Bruchadditionsbeispiel die Quellcodedateien **Bruch.java** *und* **Bruchaddition.java** geändert worden, dann genügt der folgende Compiler-Aufruf, um *beide* Dateien neu zu übersetzen:

```
>javac Bruchaddition.java
```

[1] Bei der Software-Entwicklung mit IntelliJ IDEA wirkt sich ein Verzicht auf den PATH-Eintrag nicht aus.

Die benötigten Quellcode-Dateinamen (z. B. **Bruch.java**) konstruiert der Compiler aus den ihm bekannten Klassenbezeichnungen (z. B. `Bruch`). Bei Missachtung der Quellcodedatei-Benennungsregeln (siehe Abschnitt 2.2.1) muss der Compiler bei seiner Suche also scheitern.

Man kann den JDK-Compiler **javac.exe** über das Jokerzeichen * auch beauftragen, *mehrere* Quellcodedateien zu übersetzen, z. B.:

```
>javac *.java
```

Eine weitere Beschäftigung mit der Syntax von **javac.exe** ist nicht sinnvoll, weil wir komplexere Projekte mit Hilfe unserer Entwicklungsumgebung IntelliJ erstellen lassen. Wenn dabei in den Erstellungsprozess eingegriffen werden soll, bietet sich die Verwendung eines Erstellungswerkzeugs wie **Ant**, **Gradle** oder **Maven** an. Im Manuskript werden diese Werkzeuge aber nicht behandelt.

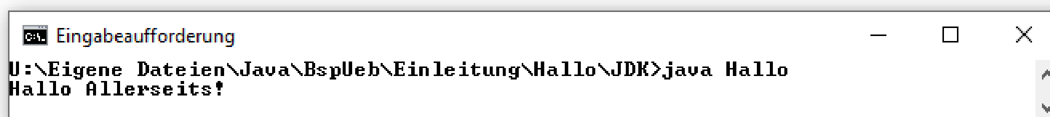
2.2.3 Ausführen

Wie Sie bereits wissen, wird zum *Ausführen* von Java-Programmen eine **Java Virtual Machine** (JVM) mit dem Interpreter **java.exe** und der Standardklassenbibliothek benötigt. Aufgrund der 2019 von der Firma Oracle geänderten Lizenzpolitik ist oft auf dem Rechner des Anwenders eine OpenJDK-Distribution installiert. Diese enthält auch eine JVM, sodass aus Anwendersicht eine OpenJDK-Installation äquivalent ist zur früher üblichen Installation einer puren Ausführungsumgebung (JRE).

Lassen Sie das Programm (bzw. die Klasse) **Hallo.class** von der JVM ausführen. Der Aufruf

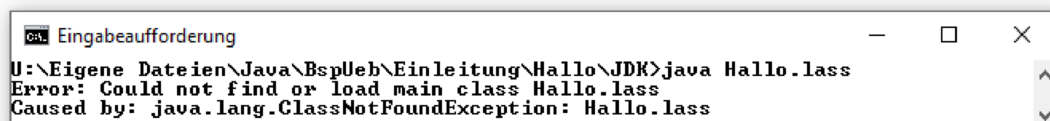
```
>java Hallo
```

sollte zum folgenden Ergebnis führen:



Beim Programmstart ist zu beachten:

- Beim Aufruf des Interpreters wird der Name der auszuführenden *Klasse* als Argument angegeben, nicht der zugehörige Dateiname. Wer den Dateinamen (samt Namenserweiterung **.class**) angibt, sieht eine Fehlermeldung:



- Weil beim Programmstart der Klassenname anzugeben ist, muss die Groß-/Kleinschreibung mit der Klassendeklaration übereinstimmen (auch unter Windows!). Java-Klassennamen beginnen meist mit großem Anfangsbuchstaben, und genau so müssen die Namen auch beim Programmstart geschrieben werden.

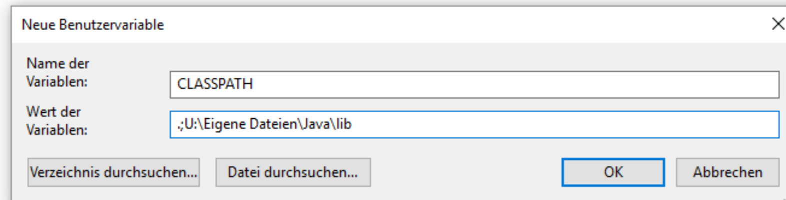
Wird **java.exe** ohne Pfad angesprochen, hängt es von der Windows-Umgebungsvariablen PATH ab,

- ob **java.exe** gefunden wird,
- welche Version zum Zug kommt, wenn mehrere Versionen von **java.exe** vorhanden sind.

Wie man unter Windows für einen korrekten PATH-Eintrag sorgt, wird im Abschnitt 2.2.2 beschrieben.

2.2.4 Suchpfad für class-Dateien

Compiler und Interpreter benötigen Zugriff auf die Bytecode-Dateien der Klassen, die im zu übersetzenden Quellcode bzw. im auszuführenden Programm angesprochen werden und nicht als Quellcode vorliegen. Mit Hilfe der Umgebungsvariablen CLASSPATH kann man eine Liste von Verzeichnissen, JAR-Archiven (siehe Abschnitt 6.1.3) oder ZIP-Archiven spezifizieren, die nach **class**-Dateien durchsucht werden sollen, z. B.:



Bei einer Verzeichnisangabe sind Unterverzeichnisse *nicht* einbezogen. Sollten sich z. B. für einen Compiler- oder Interpreter-Aufruf benötigte Dateien im Ordner **U:\Eigene Dateien\Java\lib\sub** befinden, werden sie aufgrund der CLASSPATH-Definition in obiger Dialogbox *nicht* gefunden.

Wie man unter Windows 10 eine Umgebungsvariable setzen kann, wird im Abschnitt 2.2.2 beschrieben.

Befinden sich alle benötigten Klassen entweder in der Standardbibliothek (vgl. Abschnitt 1.3.3) oder im aktuellen Verzeichnis, dann wird *keine* CLASSPATH-Umgebungsvariable benötigt. Ist sie jedoch vorhanden (z. B. von irgendeinem Installationsprogramm unbemerkt angelegt), dann werden außer der Standardbibliothek nur die Pfade in der CLASSTATH-Definition berücksichtigt. Dies führt zu Problemen, wenn in der CLASSPATH-Definition das aktuelle Verzeichnis *nicht* enthalten ist, z. B.:

```
Eingabeaufforderung
U:\Eigene Dateien\Java\BspUeb\Einleitung\Hallo\JDK>echo %classpath%
U:\Eigene Dateien\Java\lib

U:\Eigene Dateien\Java\BspUeb\Einleitung\Hallo\JDK>java Hallo
Error: Could not find or load main class Hallo
Caused by: java.lang.ClassNotFoundException: Hallo
```

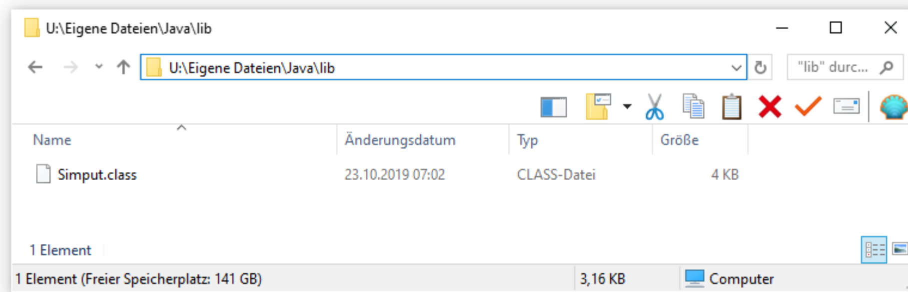
In diesem Fall muss das aktuelle Verzeichnis (z. B. dargestellt durch einen einzelnen Punkt, s.o.) in die CLASSPATH-Pfadliste aufgenommen werden, z. B.:

```
Eingabeaufforderung
U:\Eigene Dateien\Java\BspUeb\Einleitung\Hallo\JDK>echo %classpath%
.;U:\Eigene Dateien\Java\lib

U:\Eigene Dateien\Java\BspUeb\Einleitung\Hallo\JDK>java Hallo
Hallo Allerseits!
```

In vielen konsolenorientierten Beispielprogrammen des Manuskripts kommt die *nicht* zum Java-API gehörige Klasse **Simput.class** (siehe unten) zum Einsatz. Über die Umgebungsvariable CLASSPATH kann man dafür sorgen, dass der JDK-Compiler und der Interpreter die Klasse **Simput.class** finden. Dies gelingt z. B. unter Windows 7 oder 10 mit der oben abgebildeten Dialogbox **Neue Benutzervariable**, wenn Sie die Datei

...\BspUeb\Simput\Standardpaket\Simput.class

in den Ordner **U:\Eigene Dateien\Java\lib** kopiert haben:



Achten Sie in der Dialogbox **Neue Benutzervariable** unbedingt darauf, den aktuellen Pfad über einen Punkt in die CLASSPATH-Definition aufzunehmen.

Unsere Entwicklungsumgebung IntelliJ IDEA ignoriert die CLASSPATH-Umgebungsvariable, bietet aber eine alternative Möglichkeit zur Definition des Klassenpfads für ein Projekt (siehe Abschnitt 3.4.2).

Wenn sich nicht alle bei einem Compiler- oder Interpreter-Aufruf benötigten **class**-Dateien im aktuellen Verzeichnis befinden und auch nicht auf die CLASSPATH-Variable vertraut werden soll, können die nach **class**-Dateien zu durchsuchenden Pfade auch in den Startkommandos über die Option **-classpath** (abzukürzen durch **-cp**) angegeben werden, z. B.:

```
>javac -cp ".;U:\Eigene Dateien\java\lib" Bruchaddition.java
>java -cp ".;U:\Eigene Dateien\java\lib" Bruchaddition
```

Auch hier muss das aktuelle Verzeichnis ausdrücklich (z. B. durch einen Punkt) aufgelistet werden, wenn es in die Suche einbezogen werden soll.

Ein Vorteil der **-cp** - Option gegenüber der Umgebungsvariablen CLASSPATH besteht darin, dass für jede Anwendung eine eigene Suchliste eingestellt werden kann.

Bei Verwendung der **-cp** - Option wird eine eventuell vorhandene CLASSPATH-Umgebungsvariable für den gestarteten Compiler- oder Interpreter-Einsatz deaktiviert.

Die eben beschriebene Klassenpfadtechnik ist seit der ersten Java-Version im Einsatz, wird auch in Java-Versionen mit Modulsystem (ab Version 9) noch unterstützt und genügt unseren vorläufig sehr bescheidenen Ansprüchen beim Zugriff auf Bibliotheksklassen. Langfristig wird der Klassenpfad vermutlich durch den mit Java 9 eingeführten Modulpfad ersetzt (siehe Abschnitt 6.2.4).

2.2.5 Programmfehler beheben

Die vielfältigen Fehler, die wir mit naturgesetzlicher Unvermeidlichkeit beim Programmieren machen, kann man einteilen in:

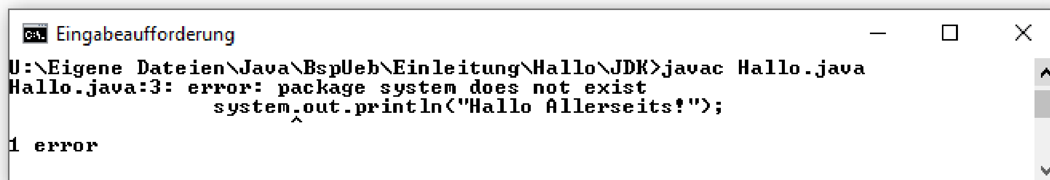
- **Syntaxfehler**
  Diese verstoßen gegen eine Syntaxregel der verwendeten Programmiersprache, werden vom Compiler gemeldet und sind daher relativ leicht zu beseitigen.
- **Logikfehler (Semantikfehler)**
  Hier liegt *kein* Syntaxfehler vor, aber das Programm verhält sich anders als erwartet, wiederholt z. B. ständig eine nutzlose Aktion („Endlosschleife") oder stürzt mit einem Laufzeitfehler ab. In jedem Fall sind die Benutzer verärgert.

Die Java-Urheber haben dafür gesorgt, dass möglichst viele Fehler vom Compiler aufgedeckt werden können. Während Syntaxfehler nur den Programmierer betreffen, automatisch entdeckt und leicht beseitigt werden können, verursachen Logikfehler für Entwickler *und* Anwender oft einen sehr großen Schaden. Simons (2004, S. 43) schätzt, dass viele Logikfehler tausendfach mehr Aufwand verursachen als der übelste Syntaxfehler.

Wir wollen am Beispiel eines provozierten Syntaxfehlers überprüfen, ob der JDK-Compiler hilfreiche Fehlermeldungen produziert. Wenn im `Hallo`-Programm der Klassenname **System** fälschlicherweise mit kleinem Anfangsbuchstaben geschrieben wird,

```
class Hallo {
   public static void main(String[] args) {
         system.out.println("Hallo Allerseits!");
   }
}
```

führt ein Übersetzungsversuch zu folgender Reaktion:



Weil sich der Compiler bereits unmittelbar hinter dem betroffenen Wort sicher ist, dass ein Fehler vorliegt, kann er die Schadstelle genau lokalisieren:

- In der ersten Fehlermeldungszeile liefert der Compiler den Namen der betroffenen Quellcodedatei, die Zeilennummer und eine Fehlerbeschreibung.
- Anschließend protokolliert der Compiler die betroffene Zeile und markiert die Stelle, an der die Übersetzung abgebrochen wurde.

Manchmal wird dem Compiler aber erst in einiger Distanz zur Schadstelle klar, dass ein Regelverstoß vorliegt, sodass statt der kritisierten Stelle eine frühere Passage zu korrigieren ist.

Im Beispiel fällt die Fehlerbeschreibung brauchbar aus, obwohl der Compiler falsch vermutet, dass mit dem verunglückten Bezeichner ein Paket (siehe unten) gemeint sei.

Weil sich in das simple Hallo-Beispielprogramm kaum ein Logikfehler einbauen lässt, betrachten wir die im Abschnitt 1.1 vorgestellte Klasse `Bruch`. Wird z. B. in der Methode `setzeNenner()` bei der Absicherung gegen Nullwerte das Ungleich-Operatorzeichen (!=) durch sein Gegenteil (==) ersetzt, ist keine Java-Syntaxregel verletzt:

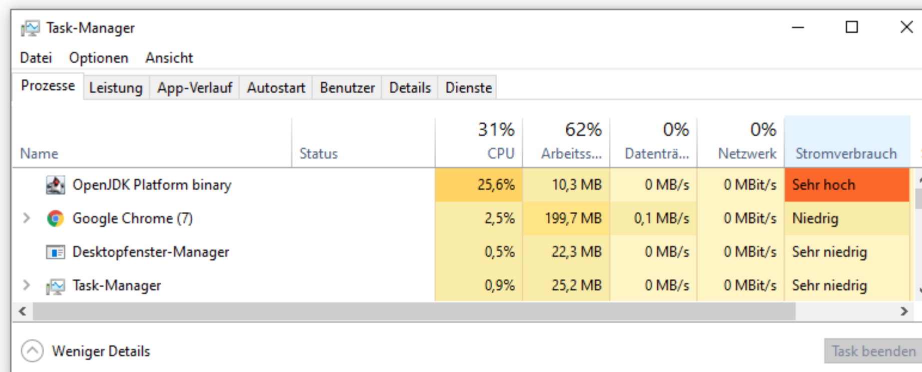
```
public boolean setzeNenner(int n) {
 if (n == 0) {
  nenner = n;
  return true;
 } else
  return false;
}
```

In der **main**() - Methode der folgenden Klasse `UnBruch` erhält ein „Bruch“ aufgrund der untauglichen Absicherung den kritischen Nennerwert 0 und wird anschließend zum Kürzen aufgefordert:

```
class UnBruch {
 public static void main(String[] args) {
  Bruch b = new Bruch();
  b.setzeZaehler(1);
  b.setzeNenner(0);
  b.kuerze();
 }
}
```

Das Programm lässt sich fehlerfrei übersetzen, zeigt aber ein unerwünschtes Verhalten. Es gerät in eine Endlosschleife (siehe unten) und verbraucht dabei reichlich Rechenzeit, wie der Windows-

Taskmanager (auf einem PC mit dem Intel-Prozessor Core i3 mit Dual-Core - Hyper-Threading-CPU, also mit 4 logischen Kernen) belegt. Das Programm kann aufgrund seiner Single-Thread-Technik nur *einen* logischen Kern nutzen und lastet diesen voll aus, sodass ca. 25% der CPU-Leistung verwendet werden:



Ein derart außer Kontrolle geratenes Konsolenprogramm kann man unter Windows z. B. mit der Tastenkombination **Strg+C** beenden.

2.3 IntelliJ IDEA Community installieren

Die Community Edition der Entwicklungsumgebung **IntelliJ IDEA** wird von der Firma JetBrains auf der Webseite

https://www.jetbrains.com/idea/download/

unter der Apache-Lizenz 2.0 kostenlos angeboten. Wir werden im Manuskript meist den zweiten Namensbestandteil weglassen und von der Entwicklungsumgebung *IntelliJ* reden.

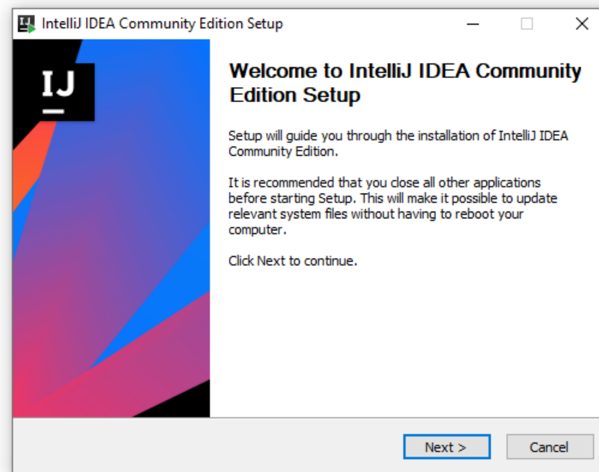
Während dieses Manuskript im Verlauf eines Wintersemesters entstanden ist, sind einige IntelliJ-Updates vorgenommen worden. Zu Beginn war die Version 2019.2.3 aktuell, am Ende die Version 2019.3. Auf wesentliche Abhängigkeiten von der IntelliJ-Version wird hingewiesen.

Die Systemvoraussetzungen für IntelliJ unter Windows dürfte praktisch jeder Rechner erfüllen:

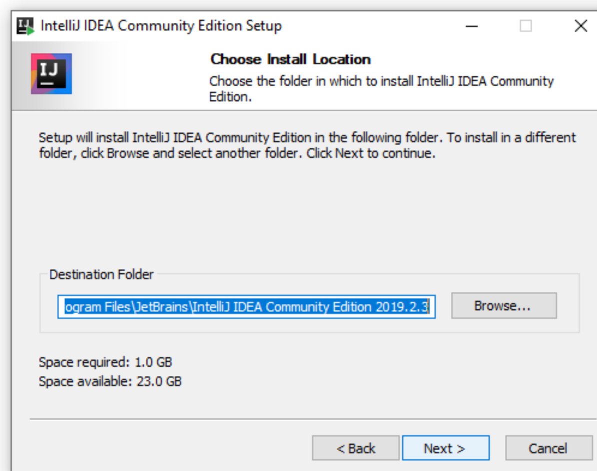
- 64-Bit-Version von Windows 10, 8, 7 (SP1)
- Mindestens 2 GB RAM
- 2,5 GB freier Festplattenspeicher für IntelliJ
- Minimale Display-Auflösung: 1024x768

Über einem Klick auf den **Download**-Schalter zur Community-Edition erhält man unter Windows einen Installationsassistenten als ausführbares Programm (am 26.10.2019: **ideaIC-2019.2.3.exe**). Nach einem Klick auf den daneben stehenden **EXE**-Schalter erlaubt ein Menü die Wahl zwischen einem ausführbaren Programm und einem ZIP-Archiv, wobei die erste Variante etwas mehr Bequemlichkeit und die zweite Variante etwas mehr Kontrolle bietet.

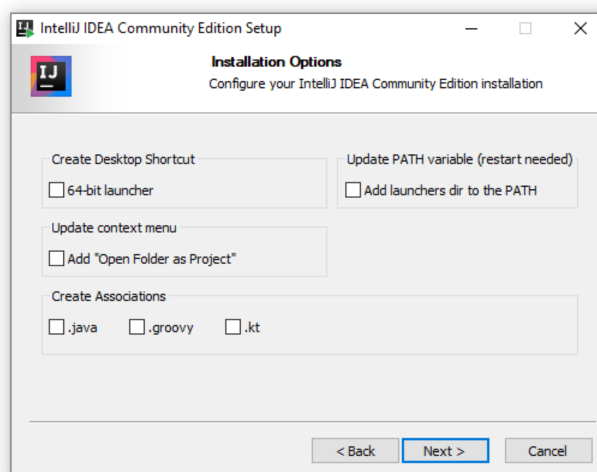
Nach dem Einstieg über einen Doppelklick auf die heruntergeladene Programmdatei **ideaIC-2019.2.3.exe** und einer positiven Antwort auf die UAC-Nachfrage (*User Account Control*) von Windows startet der Installationsassistent:



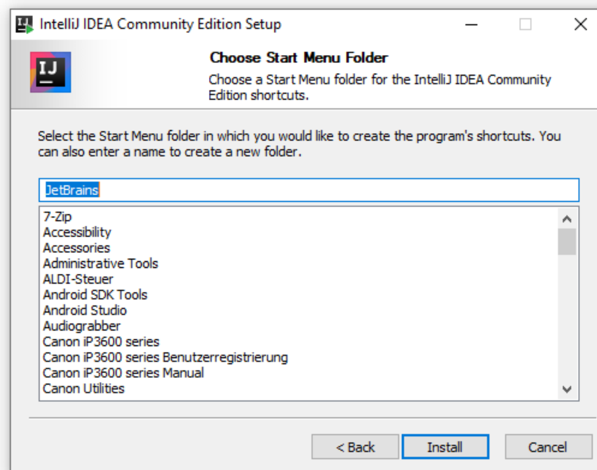
Es gibt kein starkes Argument dafür, den vorgeschlagenen Installationsordner zu ändern:



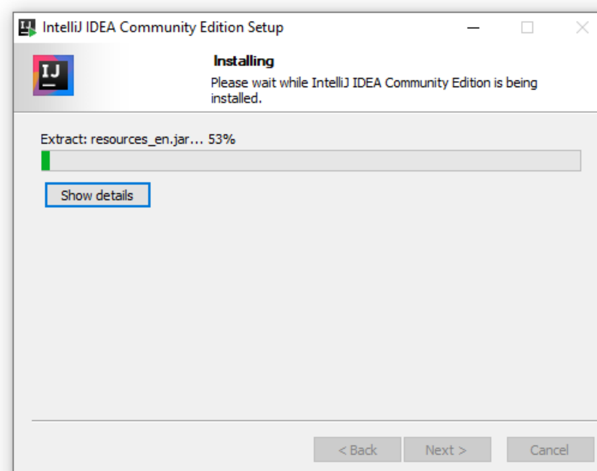
Auf die angebotenen Installationsoptionen kann man verzichten:



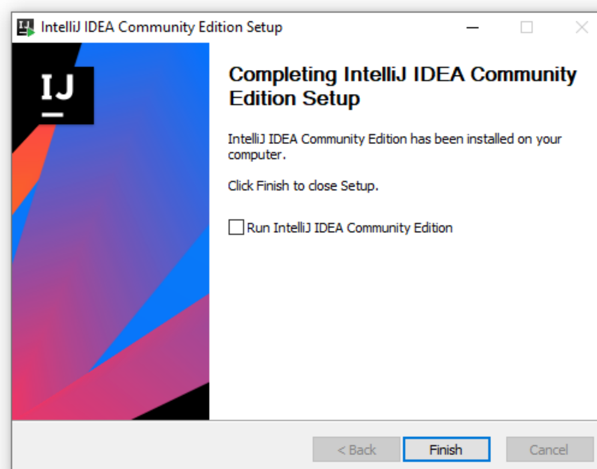
Nach der wenig relevanten Entscheidung über den Startmenüordner



legt das Installationsprogramm los



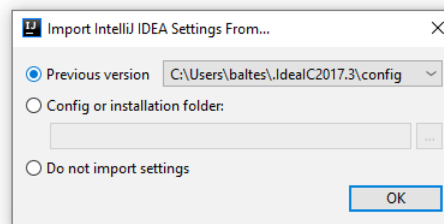
und endet nach wenigen Minuten mit der Erfolgsmeldung:



2.4 Java-Entwicklung mit IntelliJ IDEA

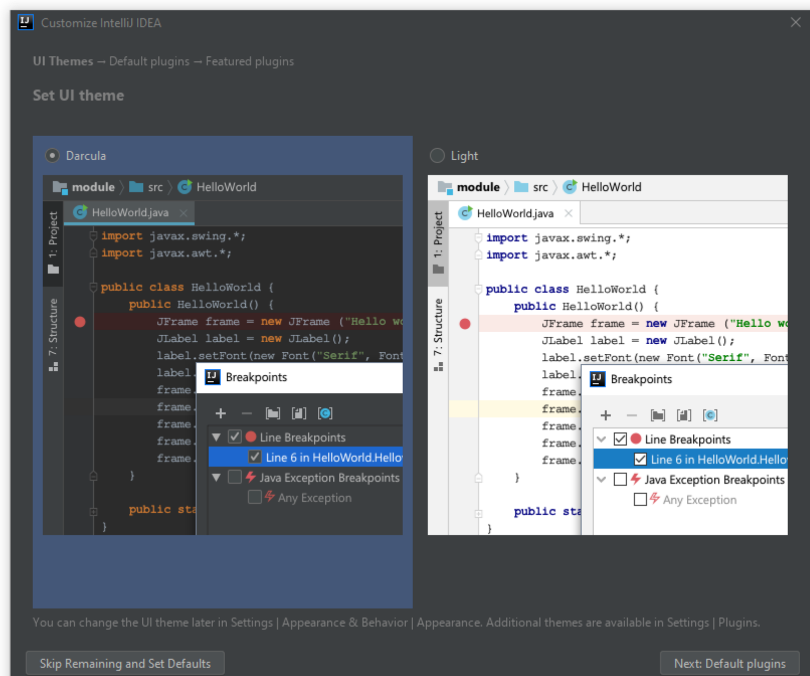
2.4.1 Erster Start

Beim ersten Start von IntelliJ IDEA können Einstellungen von einer vorhandenen Vorversion übernommen werden, z. B.:



Die Datenschutzrichtlinien von JetBrains muss man akzeptieren:

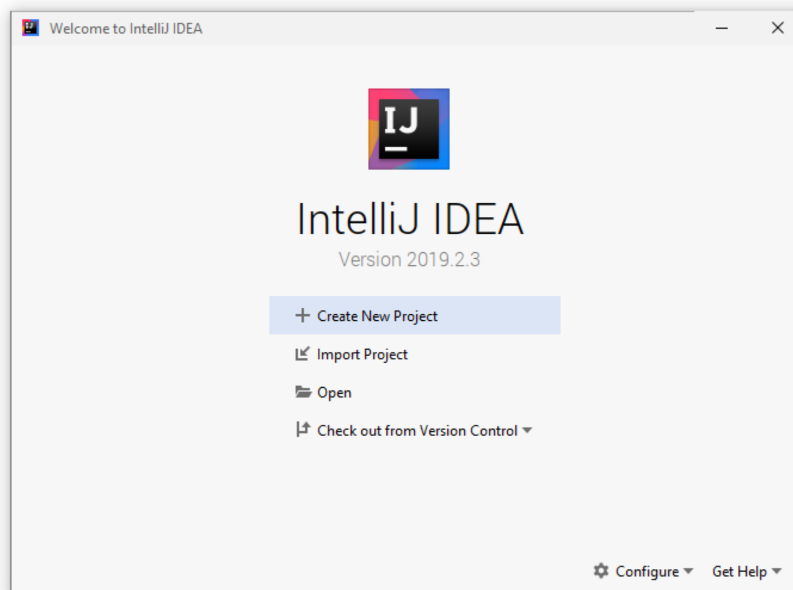
Wenn Sie keine Einstellungen von einer Vorversion übernommen haben, dürfen Sie ein Erscheinungsbild (**UI theme**) wählen, wobei Sie Ihre Meinung selbstverständlich später wieder ändern können:



Im Kurs wird das UI Theme **Light** verwendet.

Statt uns jetzt mit diversen Einstellungsfragen (zu Build Tools, Versionsverwaltung, Test Tools usw.) zu beschäftigen, klicken wir auf den Schalter **Skip All and Set Defaults**, um kurz einen Splash Screen zu bewundern

und danach mit der Erstellung eines neuen Projekts zu beginnen:

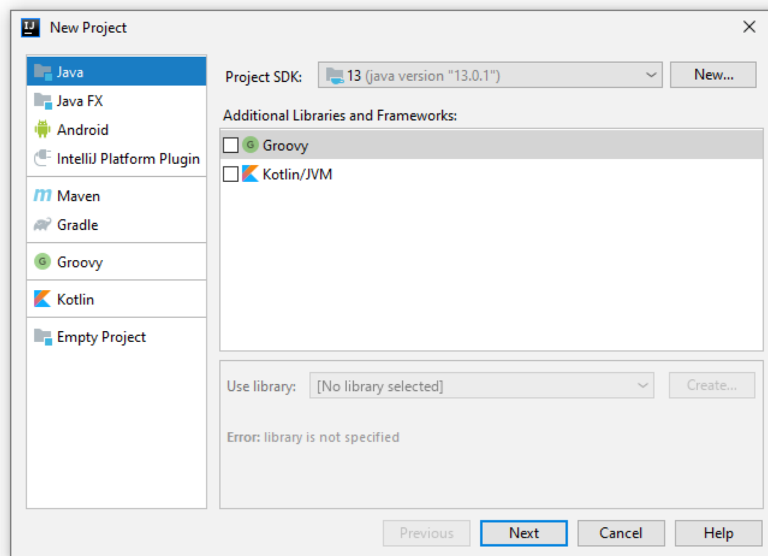


2.4.2 Projekt anlegen

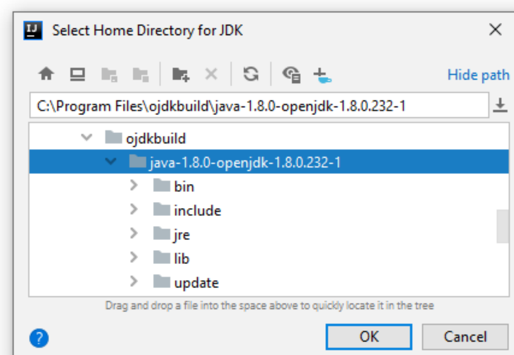
Ein IntelliJ-**Projekt** enthält eine vollständige Software-Lösung und kann (z. B. bei einer Server-Client - Lösung) aus mehreren **Modulen** bestehen (im Beispiel für die Server- und die Client-Komponente). Bei unseren IntelliJ-Projekten im Kurs werden wir stets mit *einem* IntelliJ-Modul auskommen.[1]

Wir legen ein neues Projekt aus der voreingestellten Kategorie **Java** an:

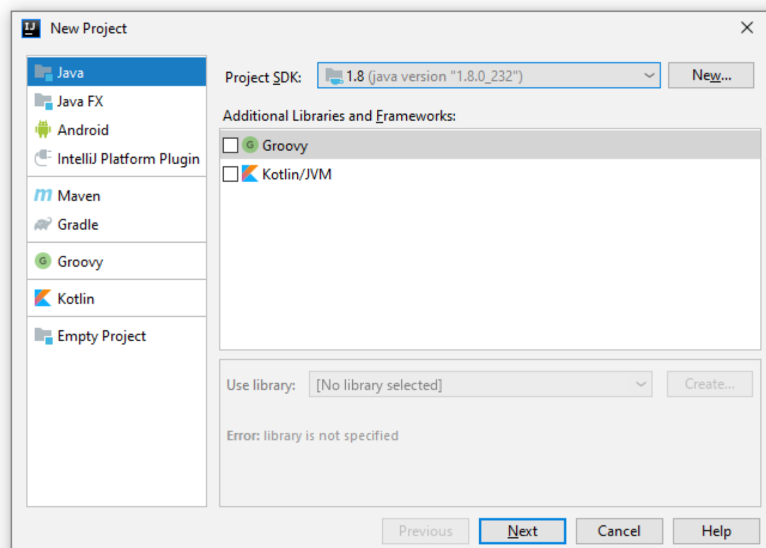
[1] In IntelliJ IDEA werden Projekte schon immer in *Module* unterteilt, wobei diese Module im Sinne der Entwicklungsumgebung nicht verwechselt werden dürfen mit den seit Java 9 vorhandenen Modulen der Programmiersprache (siehe Abschnitt 6.2). Letztere ergänzen die Pakete durch eine zusätzliche Ebene zur Zusammenfassung und Abschottung von Java-Typen.



Ein Projekt benötigt ein SDK (*Software Development Kit*), das die Standardbibliothek, den Compiler und die JVM zur Ausführung des Projekts innerhalb der Entwicklungsumgebung bereitstellt. Wenn noch kein SDK zur Auswahl bereitsteht, muss über den Schalter **New** ein JDK-Startverzeichnis benannt und so eine SDK-Definition erstellt werden. Wer gemäß Abschnitt 1.2.1 aus dem ojdkbuild-Projekt die OpenJDK 8 -Distribution installiert hat, kann z. B. deren Startordner angeben:



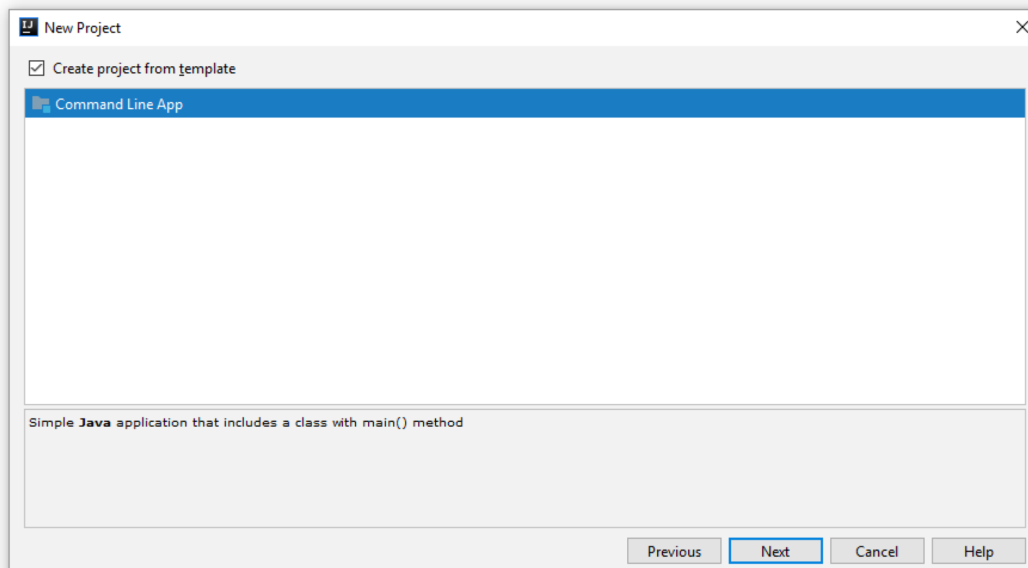
Für unser erstes IntelliJ-Projekt ist Java 8 mehr als ausreichend:



Dieses SDK kann später auch für andere Projekte verwendet werden.

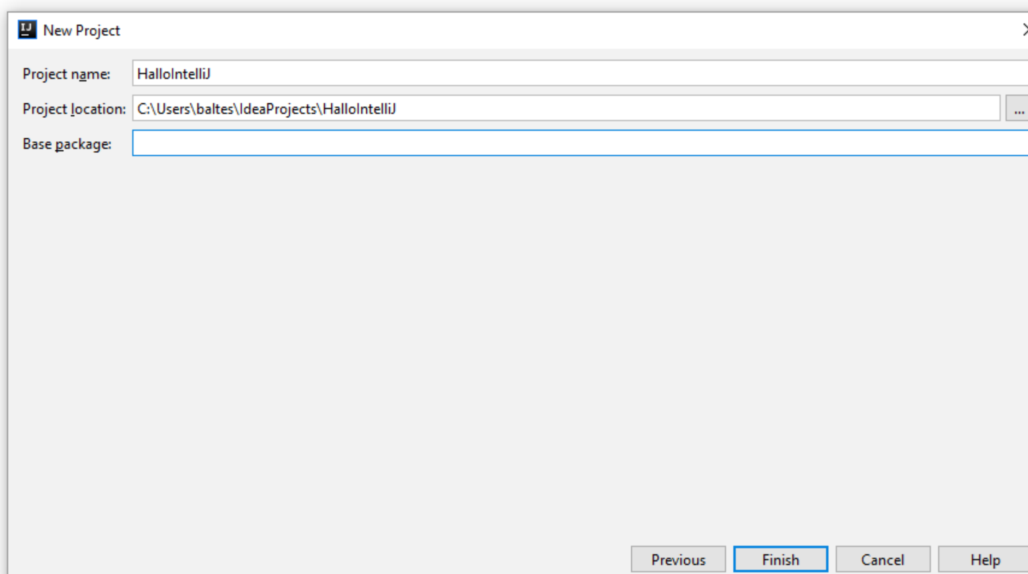
Bei Groovy und Kotlin handelt es sich um alternative Programmiersprachen, an denen wir in diesem Kurs nicht interessiert sind.

Im nächsten Dialog markieren wir das Kontrollkästchen **Create project from template** und akzeptieren die einzige Option (**Command Line App**) per **Next**:



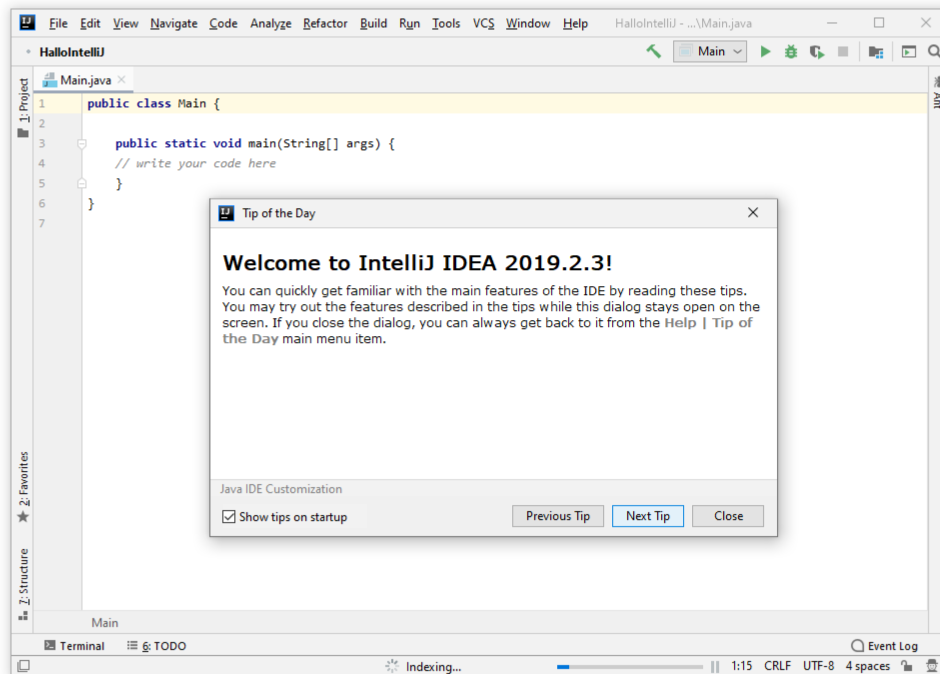
Die Wahl dieser Projektvorlage hat zur Folge, dass im entstehenden Projekt automatisch eine zu unserer Zielsetzung passende Java-Klasse samt **main()** - Methode angelegt wird, sodass wir anschließend etwas Aufwand sparen.

Wir wählen einen Projektnamen, übernehmen den resultierenden Projektordner und verzichten auf ein Basispaket, z. B.:[1]

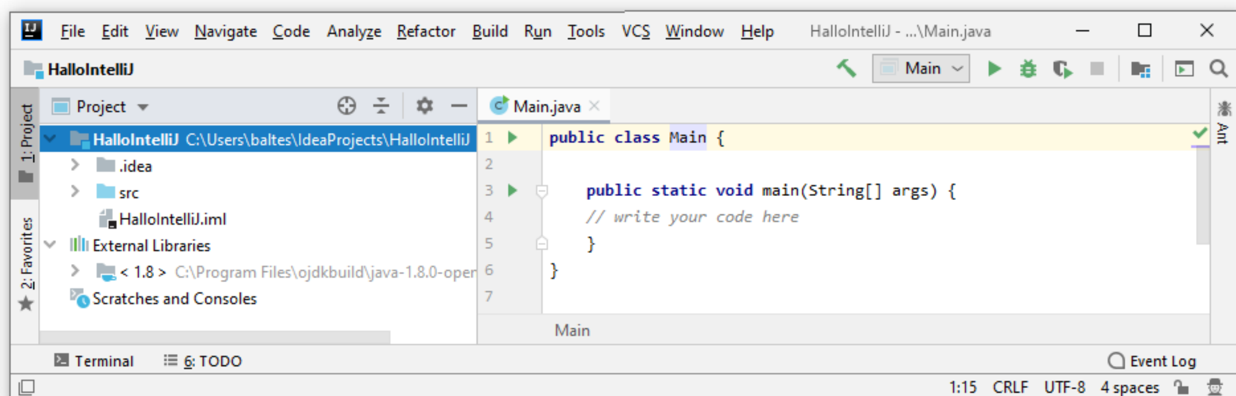


Nach einem Klick auf **Finish** erscheint die Entwicklungsumgebung, ist aber noch ein Weilchen mit Projektvorbereitungsarbeiten beschäftigt (siehe Fortschrittsbalken zum Indizieren in der Statuszeile):

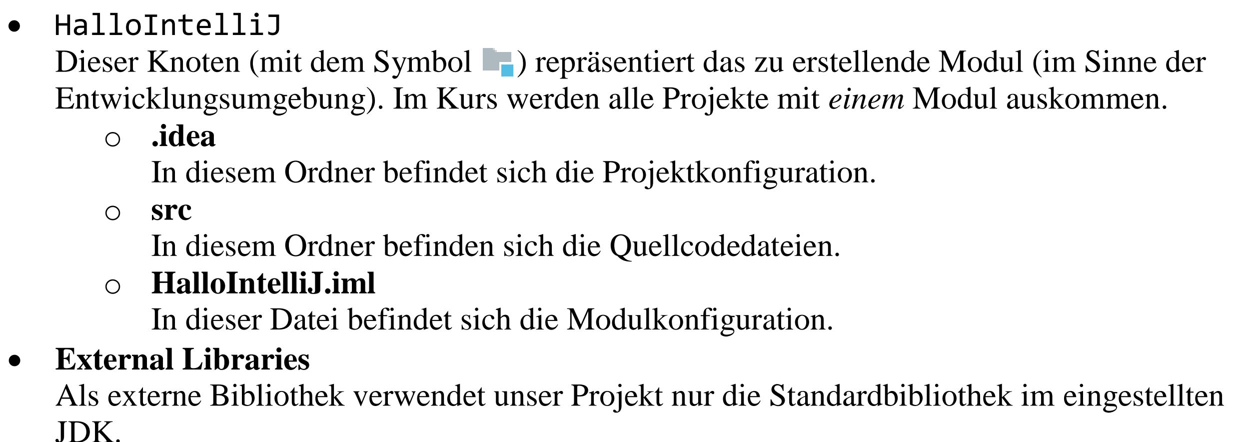
1 Durch den Verzicht auf ein Basispaket ergibt sich eine einfache Lernumgebung. Soll ein Programm veröffentlicht werden, ist ein Basispaket sehr zu empfehlen.



Schließlich ist IntelliJ einsatzbereit und präsentiert im Editor den basierend auf der Vorlagendefinition (**Command Line App**) erstellten Quellcode der Klasse `Main` mit bereits vorhandener Startmethode **main()**:



Im **Project**-Fenster sind zu sehen:



- `HalloIntelliJ`
  Dieser Knoten (mit dem Symbol) repräsentiert das zu erstellende Modul (im Sinne der Entwicklungsumgebung). Im Kurs werden alle Projekte mit *einem* Modul auskommen.
  - **.idea**
    In diesem Ordner befindet sich die Projektkonfiguration.
  - **src**
    In diesem Ordner befinden sich die Quellcodedateien.
  - **HalloIntelliJ.iml**
    In dieser Datei befindet sich die Modulkonfiguration.
- **External Libraries**
  Als externe Bibliothek verwendet unser Projekt nur die Standardbibliothek im eingestellten JDK.

Weitere Projekte lassen sich entweder über den IntelliJ-Startdialog oder über den folgenden Menübefehl anlegen:

Start > New > Project

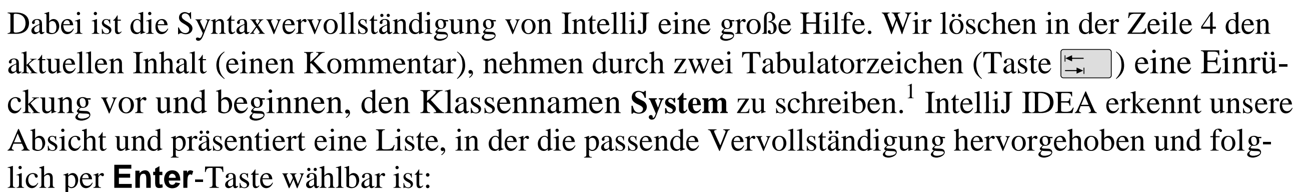
2.4.3 Quellcode-Editor

Um das von IntelliJ erstellte Programm zu vollenden, müssen wir noch im Editor die Ausgabeanweisung

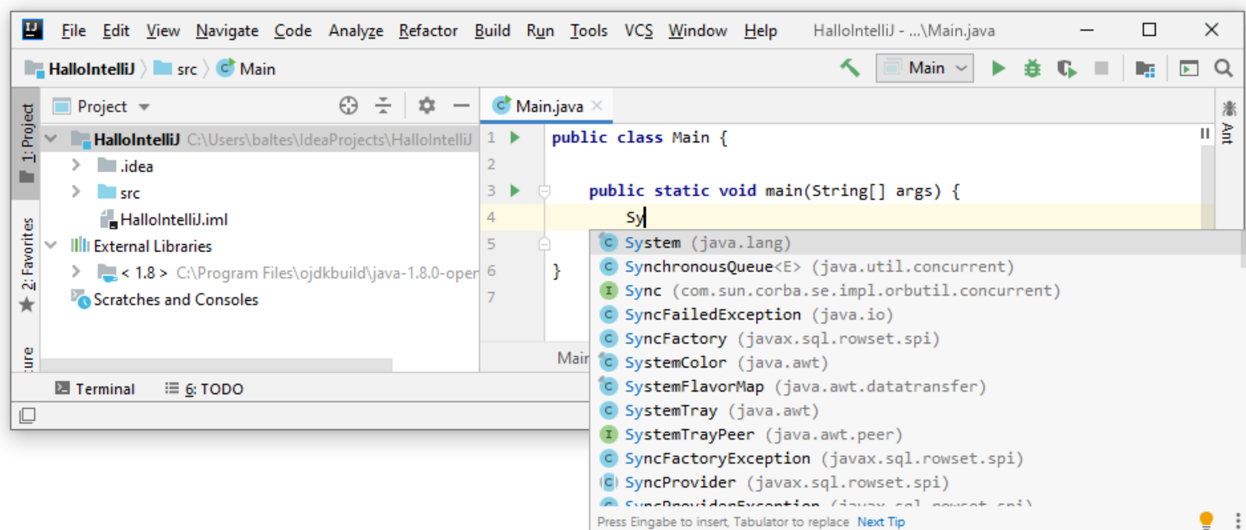
```
System.out.println("Hallo allerseits!");
```

verfassen (vgl. Abschnitt 2.2.1).

2.4.3.1 Syntaxvervollständigung



Dabei ist die Syntaxvervollständigung von IntelliJ eine große Hilfe. Wir löschen in der Zeile 4 den aktuellen Inhalt (einen Kommentar), nehmen durch zwei Tabulatorzeichen (Taste ⇆) eine Einrückung vor und beginnen, den Klassennamen **System** zu schreiben.[1] IntelliJ IDEA erkennt unsere Absicht und präsentiert eine Liste, in der die passende Vervollständigung hervorgehoben und folglich per **Enter**-Taste wählbar ist:

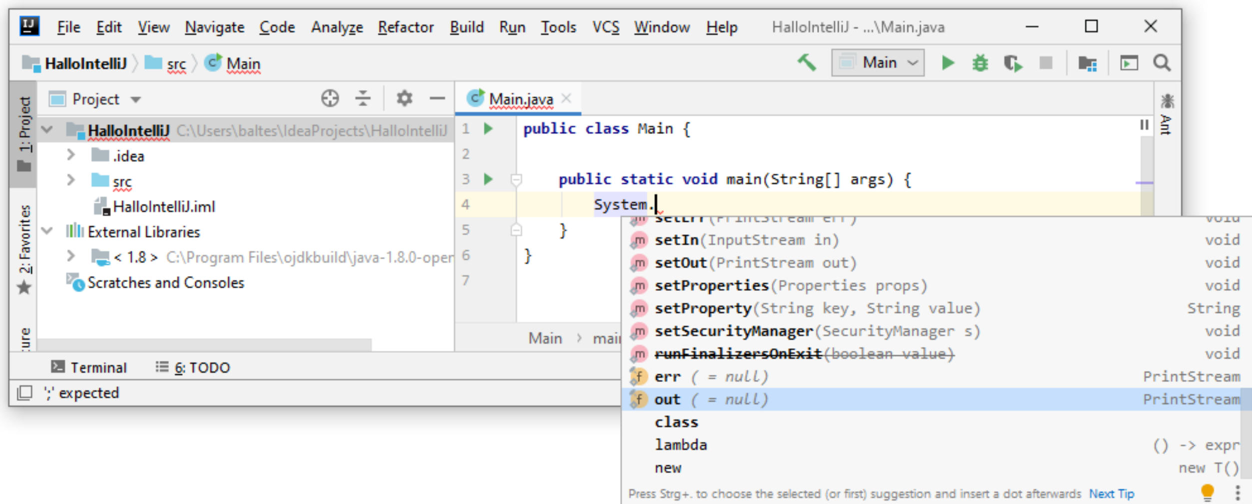


Sobald wir einen Punkt hinter den Klassennamen **System** setzen, erscheint eine neue Liste mit allen zulässigen Fortsetzungen, wobei wir uns im Beispiel für die Klassenvariable **out** entscheiden, die auf ein Objekt der Klasse **PrintStream** zeigt:[2]

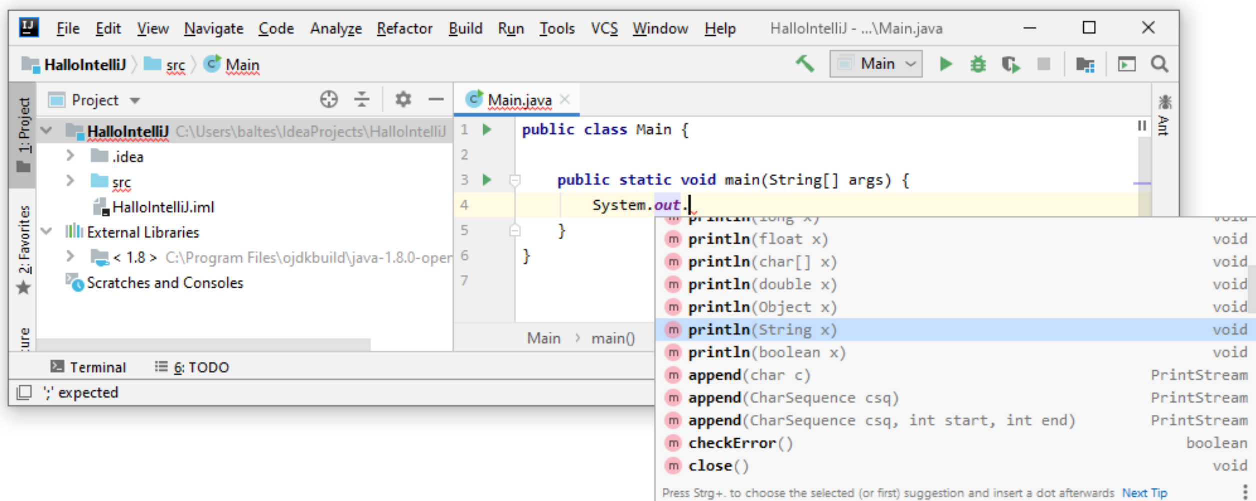
---

[1] Bei Bedarf lassen sich die Zeilennummern folgendermaßen einschalten:
File > Settings > Editor > General > Appearance > Show line numbers

[2] In der ersten Vorschlagsliste mit Bestandteilen der Klasse **System** erscheint die von uns häufig benötigte Klassenvariable **out** noch nicht an der bequemen ersten Position, doch passt sich IntelliJ schnell an unsere Gewohnheiten bzw. Bedürfnisse an.



Wir übernehmen das Ausgabeobjekt per **Enter**-Taste oder Doppelklick und setzen einen Punkt hinter seinen Namen (**out**). Jetzt werden u.a. die Instanzmethoden der Klasse **PrintStream** aufgelistet, und wir wählen die Variante (spätere Bezeichnung: *Überladung*) der Methode **println()** mit einem Parameter vom Typ **String**, die sich zur Ausgabe einer Zeichenfolge eignet:



Ein durch doppelte Hochkommata begrenzter Text komplettiert den **println()** - Methodenaufruf, den wir objektorientiert als Nachricht an das Objekt **System.out** auffassen.

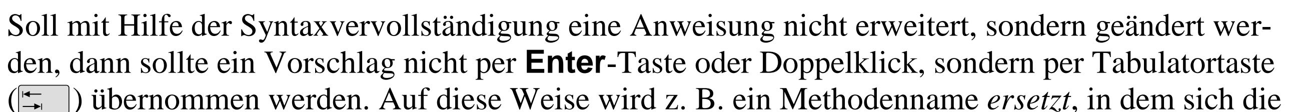
Die Syntaxvervollständigung von IntelliJ macht Vorschläge für Variablen, Typen, Methoden usw. Sollte sie nicht spontan tätig werden, kann sie mit der folgenden Tastenkombination angefordert werden:

Strg + Leertaste

Nach einer Anforderung mit

Strg + Umschalt + Leertaste

erscheint eine *smarte* Variante der Syntaxvervollständigung. Indem der erwartete Typ des ganzen Ausdrucks analysiert wird, können Variablen und Methoden kontextabhängig vorgeschlagen werden.

Soll mit Hilfe der Syntaxvervollständigung eine Anweisung nicht erweitert, sondern geändert werden, dann sollte ein Vorschlag nicht per **Enter**-Taste oder Doppelklick, sondern per Tabulatortaste () übernommen werden. Auf diese Weise wird z. B. ein Methodenname *ersetzt*, in dem sich die

Einfügemarke gerade befindet, statt durch Einfügen des neuen Namens ein fehlerhaftes Gebilde zu erzeugen.

Mit der Tastenkombination

Strg + Umschalt + Enter

fordert man die Vervollständigung einer Anweisung an, z. B.:

<table>
<tr><th>vorher</th><th>nachher</th></tr>
<tr><td><pre>public class Main {

    public static void main(String[] args) {
        System.out.println("Hallo Allerseits!
    }
}</pre></td><td><pre>public class Main {

    public static void main(String[] args) {
        System.out.println("Hallo Allerseits!");
    }
}</pre></td></tr>
</table>

Obwohl das vervollständigte Programm auf der rechten Seite fehlerfrei ist, unterschlängelt IntelliJ das Wort „allerseits". Kommentare, Klassennamen etc. werden per Voreinstellung von der Entwicklungsumgebung auf die Einhaltung der englischen Rechtschreibung überprüft werden. Wir schreiben *Denglisch* und kümmern uns nicht um die Kritik an unserer Orthographie.

2.4.3.2 Code-Inspektion und Quick-Fixes

IntelliJ führt Code-Inspektionen *on the fly* durch, macht auf potentielle Probleme aufmerksam und schlägt QuickFix-Korrekturen vor.

Wenn IntelliJ IDEA eine Code-Änderung vorschlagen möchte, erscheint eine gelbe Birne links neben der betroffenen Stelle, z. B.:

```
2
3     public static void Main(String[] args) {
4         System.out.println("Hallo Allerseits!");
5     }
```

Zeigt die Maus auf die Birne, kann ein Drop-Down - Menü mit Korrekturvorschlägen geöffnet werden, z. B.:

```
2
3     public static void Main(String[] args) {
4       Make method constructor
5       Rename
6       Safe delete 'Main(String[])'
```

Statt das Drop-Down - Menü zur gelben Birne zu öffnen, kann man auch die Einfügemarke auf den markierten Syntaxbestandteil setzen und die Tastenkombination **Alt + Enter** betätigen, um dieselbe Vorschlagsliste zu erhalten:

```
2
3     public static void Main(String[] args) {
4         System.out.prin  Make method constructor
5     }                    Rename
6                          Safe delete 'Main(String[])'
```

Im Beispiel muss der Name der Methode **main**() klein geschrieben werden, damit sie als Startmethode akzeptiert wird.

Wenn IntelliJ IDEA einen Syntaxfehler findet, erscheint eine *rote* Birne links neben der betroffenen, durch rote Schrift markierten Stelle, z. B.:

```
3     public static void main(String[] args) {
4         System.outsch.println("Hallo Allerseits!");
5     }
```

Zeigt die Maus auf die Birne, kann ein Drop-Down - Menü mit Korrekturvorschlägen geöffnet werden, z. B.:

```
3      public static void main(String[] args) {
4          System.outsch.println("Hallo Allerseits!");
5   Rename reference
```

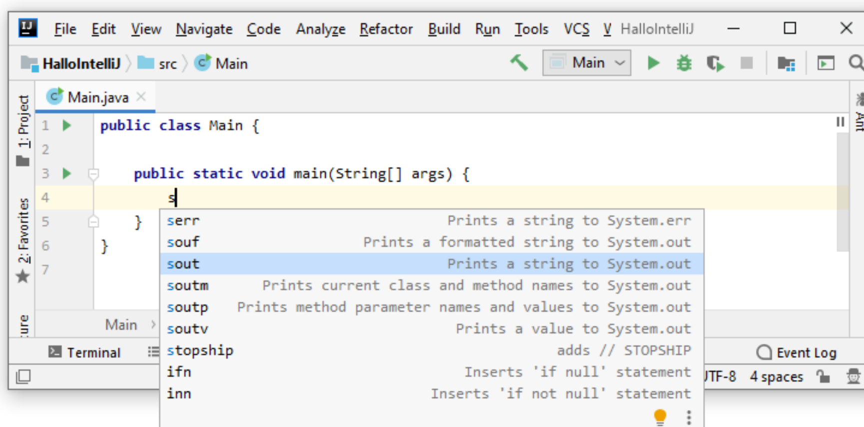
Statt das Drop-Down - Menü zur roten Birne zu öffnen, kann man auch die Einfügemarke auf rot gefärbten Syntaxbestandteil setzen und die Tastenkombination **Alt + Enter** betätigen, um dieselbe Vorschlagsliste zu erhalten:

```
3      public static void main(String[] args) {
4          System.outsch.println("Hallo Allerseits!");
5      }          Rename reference
```

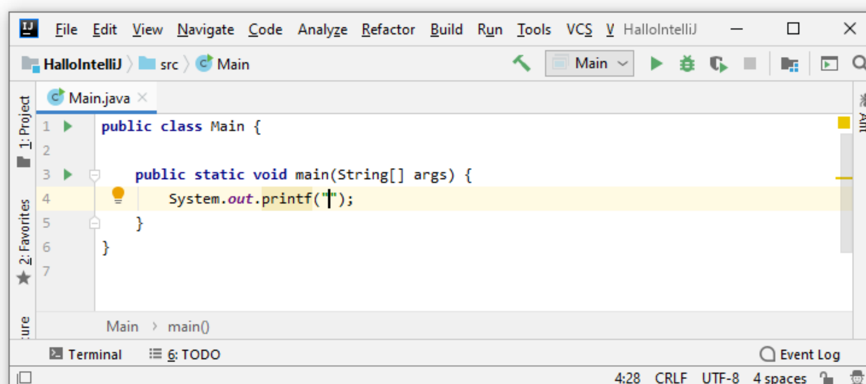
Im Beispiel muss der Name der Klassenvariablen **out** korrekt geschrieben werden.

2.4.3.3 *Live Templates*

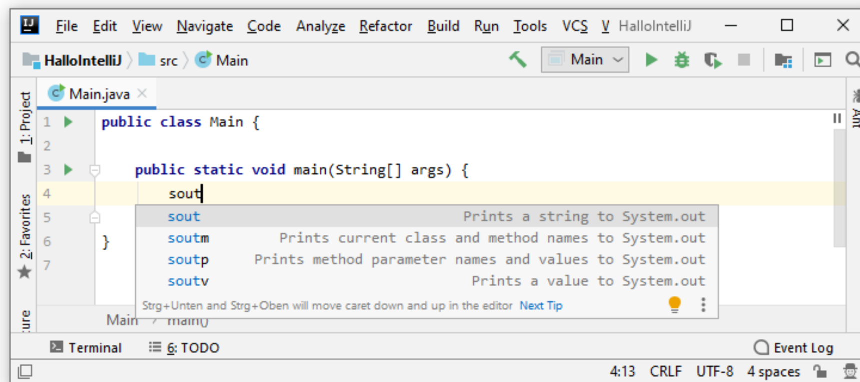
Häufig benötigte Code-Schnipsel (z. B. `System.out.println();`) kann IntelliJ über sogenannte *Live-Vorlagen* produzieren. Starten Sie zum Ausprobieren eine neue Zeile mit dem Buchstaben `s` und drücken Sie anschließend die Tastenkombination **Strg + J**, um alle durch Vervollständigung herstellbaren Live-Vorlagen auflisten zu lassen. Wählen Sie aus der Liste



die Alternative **sout** mit der Tabulatortaste (↹). Daraufhin erstellt IntelliJ einen Methodenaufruf, den Sie nur noch um die auszugebende Zeichenfolge erweitern müssen:



Wenn Sie die Vorlagenbezeichnung **sout** komplett eintippen, präsentiert IntelliJ spontan (ohne Anforderung durch **Strg + J**) eine Vorschlagsliste mit dem passenden Element in führender Position:



Um die Anweisung

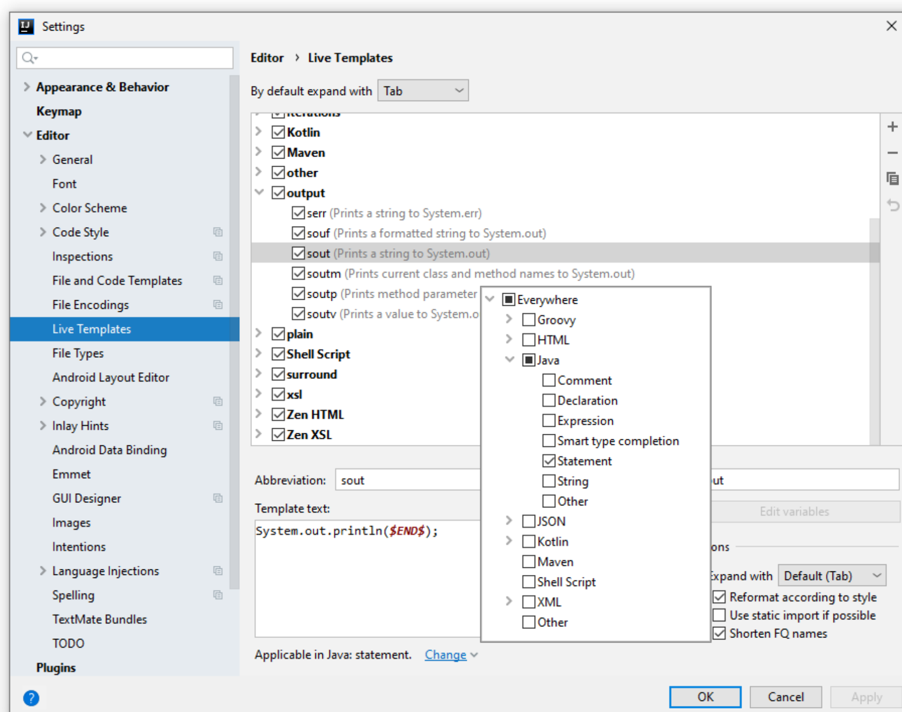
```
System.out.println();
```

zu erstellen, müssen Sie also nur **sout** schreiben und die Tabulatortaste drücken.

Nach

File > Settings > Editor > Live Templates

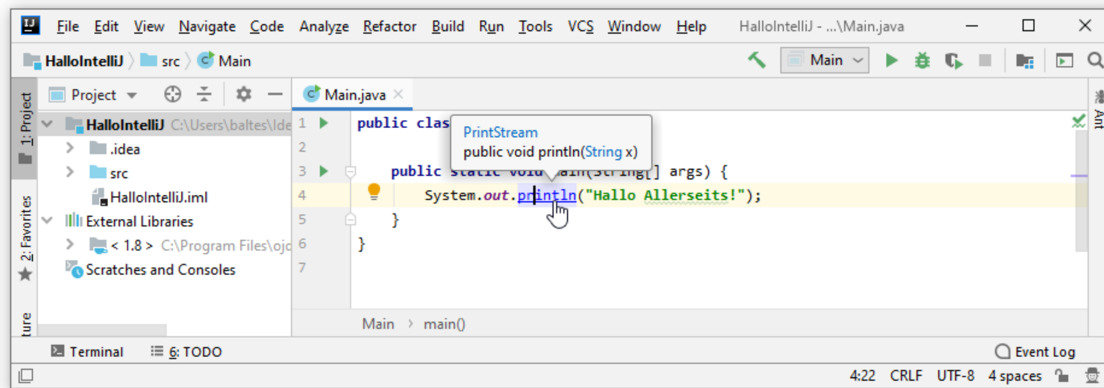
kann man im folgenden Dialog



die vorhandenen Vorlagen einsehen und konfigurieren und neue Vorlagen definieren.

2.4.3.4 Orientierungshilfen

Zeigt man bei gedrückter **Strg**-Taste mit dem Mauszeiger auf eine Methode, dann erscheint der Definitionskopf, z. B.:



Setzt man bei gedrückter **Strg**-Taste einen Mausklick auf einen Bezeichner (z. B. Klasse, Variable, Methode), dann springt IntelliJ zur Implementierung des angefragten Syntaxbestandteils. Nötigenfalls wird der Quellcode der zugehörigen Klasse in ein neues Registerblatt des Editors geladen, z. B.:

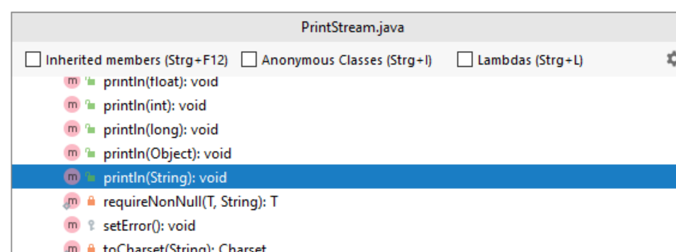
Zum selben Ziel kommt man auch ohne Mausbeteiligung:

- Einfügemarke auf den interessierenden Bezeichner setzen
- Tastenkombination **Strg + B**

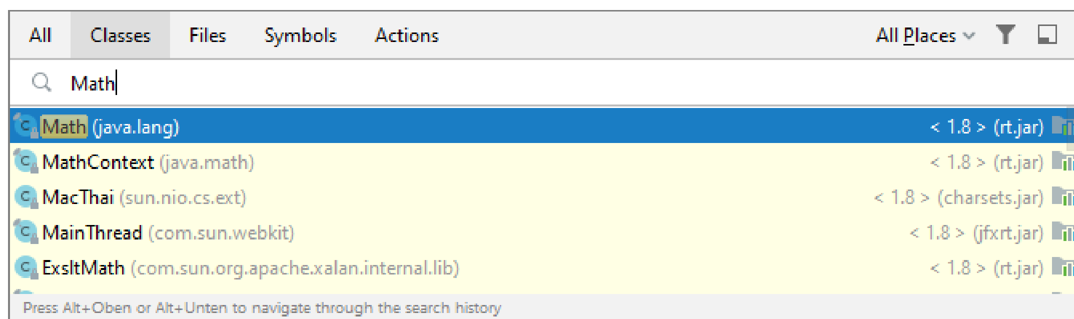
Über die Tastenkombination **Strg + F12** oder mit dem Menübefehl

Navigate > File Structure

erhält man einen Dialog mit der Struktur der aktuell im Editor bearbeiteten Datei und kann Bestandteile per Mausklick ansteuern:



Um den Quellcode einer beliebigen Klasse anzufordern, trägt man ihren Namen nach der Tastenkombination **Strg + N** in den folgenden Dialog ein:



2.4.3.5 Refaktorieren

Um z. B. einen Variablen- oder Klassennamen an allen Auftrittsstellen im Projekt zu ändern, setzt man die Einfügemarke auf ein Vorkommen, drückt die Tastenkombination **Umschalt + F6**, ändert die Bezeichnung und quittiert mit der Eingabetaste. Im Menüsystem ist die benutzte Refaktorierungsfunktion hier zu finden:

Refactor > Rename

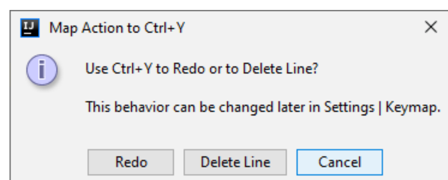
Die im **Refactor**-Menü zahlreich vorhandenen weiteren IntelliJ-Kompetenzen zur Quellcode-Umgestaltung werden wir im Kurs nicht benötigen.

2.4.3.6 Sonstige Hinweise

Im IntelliJ-Quellcodeeditor lassen sich die letzten Änderungen zurücknehmen mit der von vielen Programmen gewohnten Tastenkombination **Strg + Z**. Soll eine zurückgenommenen Änderung wiederhergestellt werden, ist offiziell die folgende Tastenkombination zu verwenden:

Strg + Umschalt + Z

Wer in dieser Situation die (z. B. aus Microsoft Office) gewohnte Tastenkombination **Strg + Y** verwendet, kann zwischen zwei Optionen wählen:



Mit **Redo** wählt man die von Microsoft Office gewohnte Funktion, und mit **Delete Line** wählt man die von IntelliJ gewohnte Funktion, die gesamte aktuelle Editorzeile zu löschen.

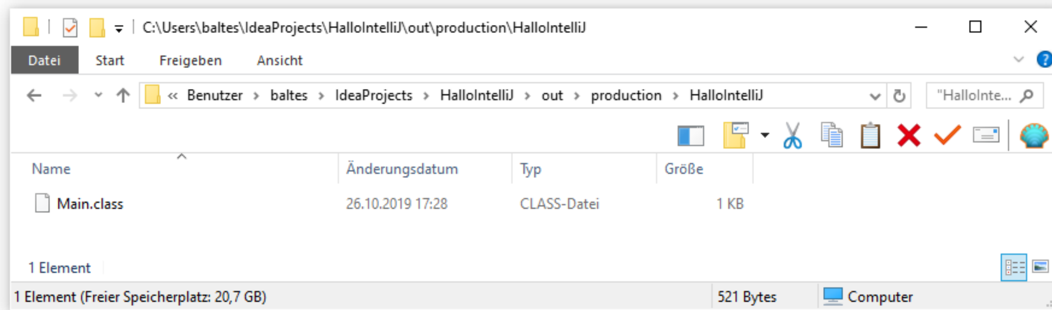
2.4.4 Übersetzen und Ausführen

Nun soll die im Editorfenster angezeigte Startklasse unseres Beispielprogramms übersetzt und ausgeführt werden. Genau genommen entscheidet die eingestellte **Configuration** über die zu verwendende Startklasse. Im Beispiel sollte die aktive Konfiguration den Namen `Main` haben und die gleichnamige Startklasse verwenden (siehe Symbolleiste über dem Editor):

Zum Starten klicken wir auf den grünen Play-Schalter neben der Konfiguration oder verwenden die Tastenkombination

Umschalt + F10

IntelliJ verwendet per Voreinstellung den Compiler im Projekt-SDK, um im Beispiel aus der Quellcodedatei **Main.java** die Bytecodedatei **Main.class** zu erstellen:

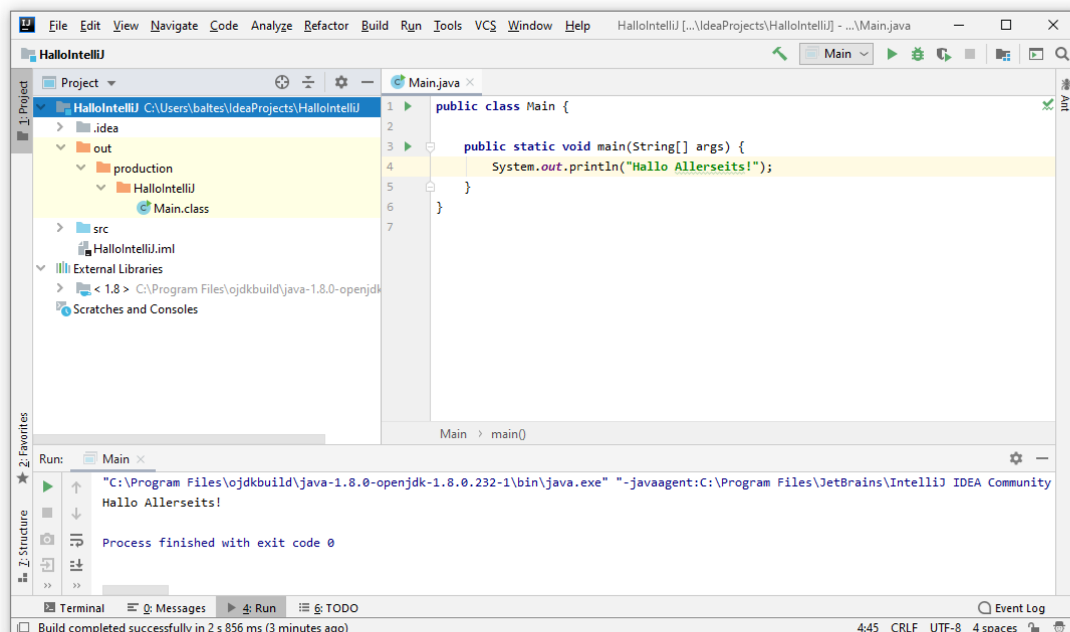


Nach der Übersetzung, die im Rahmen der Projekterstellung stattfindet, erscheint am unteren Fensterrand das **Run - Fenster** von IntelliJ:

Es zeigt in der Statuszeile das Erstellungsergebnis und im Inhaltsbereich ...

- die ausführende Laufzeitumgebung (JVM),[1]
- die Ausgabe des Programms
- und den Exit Code, wobei die 0 für eine fehlerfreie Ausführung steht.

Der beim Erstellen erzeugte Ausgabeordner wird im **Project**-Fenster angezeigt:

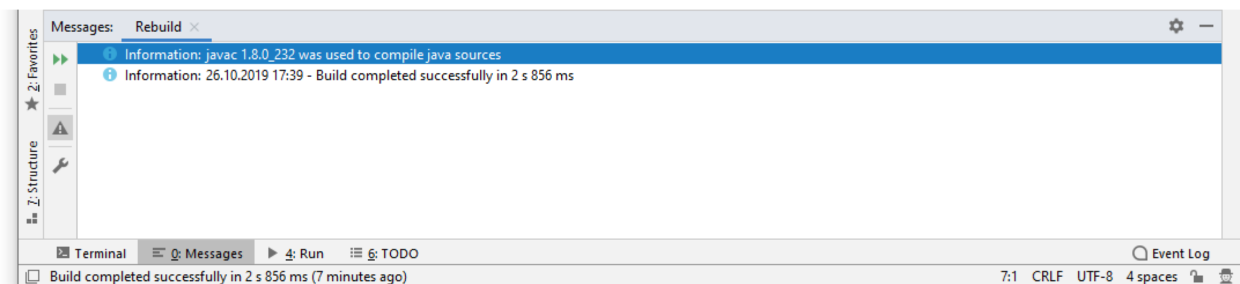


[1] Das zum Starten des Programms verwendete Kommando verrät, dass ein sogenannter *Java-Agent* im Spiel ist, wenn das Programm innerhalb der Entwicklungsumgebung ausgeführt wird. Er kann Daten über das Programm (z. B. Speicherbedarf von Objekten) zur Verwendung durch die Entwicklungsumgebung sammeln. Darum müssen wir uns im Augenblick nicht kümmern.

Öffnet man das **Messages**-Fenster über die gleichnamige Schaltfläche über der Statuszeile, mit dem Menübefehl

View > Tool Windows > Messages

oder durch die Tastenkombination **Alt + 0**, dann erfährt man, dass IntelliJ den Compiler **javac.exe** aus dem JDK benutzt hat:



Mit dem folgenden Menübefehl kann man die Übersetzung eines Programms *ohne* anschließende Ausführung anfordern:

Build > Recompile

2.4.5 Sichern und Wiederherstellen

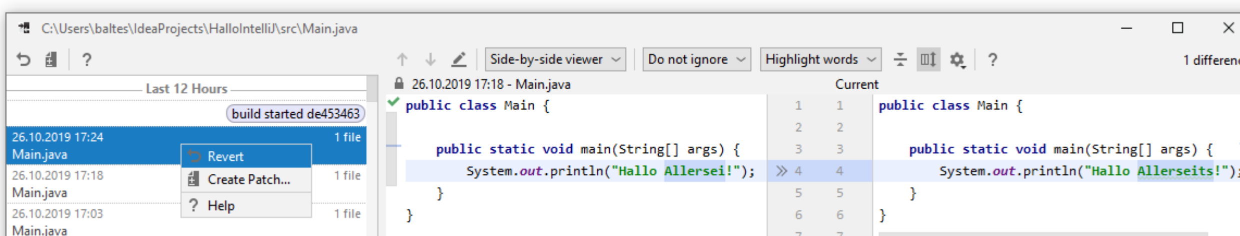
Bei der Arbeit mit IntelliJ IDEA muss man sich um das Sichern von Quellcode und anderen im Editor bearbeiteten Dateien kaum Gedanken machen, weil die Entwicklungsumgebung bei jeder passenden Gelegenheit (z. B. beim Erstellen des Projekts) automatisch sichert.

Außerdem ist ein **VCS** (Version Control System) integriert, das lokal arbeiten und mit Cloud-Diensten wie **GitHub** kooperieren kann. Wegen der zahlreichen Funktionen ist ein eigenes Hauptmenü namens **VCS** vorhanden.

Für eine Quellcodedatei sind über

VCS > Local History > Show History

alle Zwischenstände verfügbar, z. B.:



Um zu einem vorherigen Zustand zurückzukehren, wählt man aus seinem Kontextmenü das Item **Revert**.

2.4.6 Konfiguration

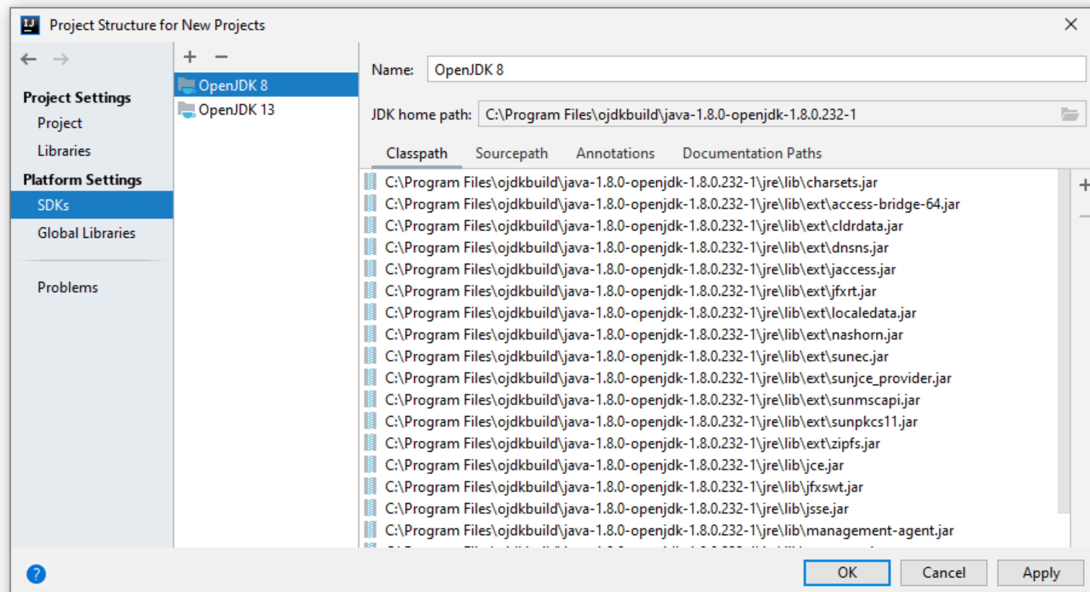
2.4.6.1 Verfügbare SDKs einrichten

Ein IntelliJ-Projekt benötigt ein SDK (*Software Development Kit*), das die Standardbibliothek, den Compiler und die JVM zur Ausführung des Programms innerhalb der Entwicklungsumgebung bereitstellt. Wir haben im Abschnitt 2.4.2 bei der Kreation des ersten Projekts ein SDK basierend auf dem im Abschnitt 1.2.1 installierten OpenJDK 8 angelegt. Eventuell hat IntelliJ das gemäß Ab-

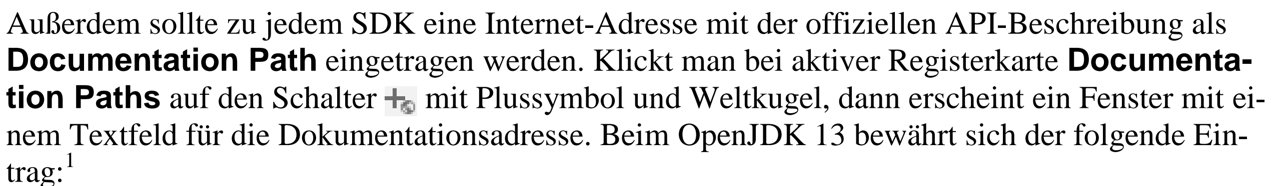
schnitt 2.1.1 installierte OpenJDK 13 entdeckt und als SDK eingerichtet. Eine Liste der bekannten und in Projekten wählbaren SDKs erhält man z. B. über

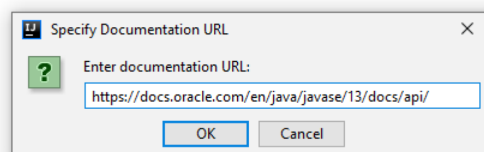
File > Other Settings > Structure for New Projects > SDKs

im folgenden Dialog, der primär dazu gedacht ist, für neue Projekte ein voreingestelltes SDK festzulegen:



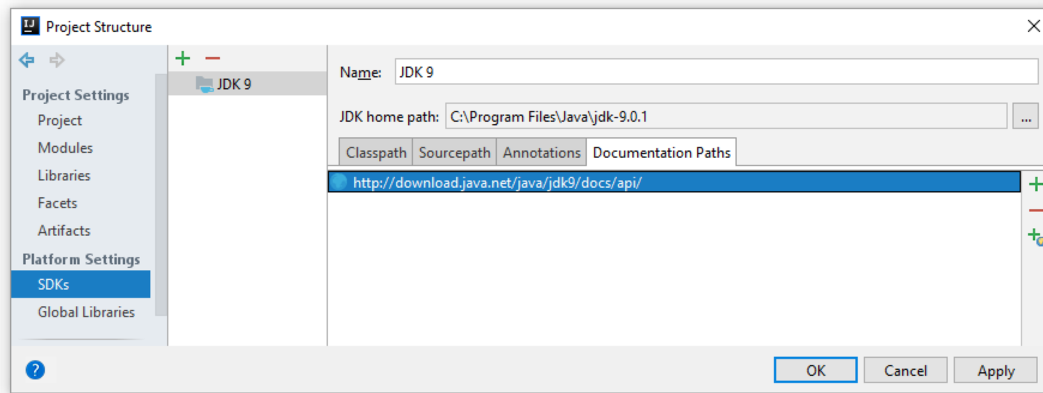
Über diesen Dialog können aber auch SDKs konfiguriert oder ergänzt werden. Es ist z. B. sinnvoll, die vorhandenen SDKs so zu benennen (siehe Bildschirmfoto), dass die Kursbeispiele direkt geöffnet werden können (vgl. Abschnitt 2.4.7).

Außerdem sollte zu jedem SDK eine Internet-Adresse mit der offiziellen API-Beschreibung als **Documentation Path** eingetragen werden. Klickt man bei aktiver Registerkarte **Documentation Paths** auf den Schalter  mit Plussymbol und Weltkugel, dann erscheint ein Fenster mit einem Textfeld für die Dokumentationsadresse. Beim OpenJDK 13 bewährt sich der folgende Eintrag:[1]



Aufgrund der resultierenden Einstellung

[1] Als Text für die Übernahme per Copy & Paste:
OpenJDK 8: https://docs.oracle.com/javase/8/docs/api/
OpenJDK 11: https://docs.oracle.com/en/java/javase/11/docs/api/
OpenJDK 13: https://docs.oracle.com/en/java/javase/13/docs/api/



kann IntelliJ bei der Arbeit mit dem Quellcode-Editor nach der Tastenkombination **Umschalt + F1** zu dem die Einfügemarke enthaltenen Java-Bezeichner die API-Dokumentation in einem externen Browser-Fenster liefern.

Wir verwenden im Kurs meist ...

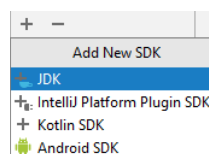
- das gemäß Abschnitt 1.2.1 installierte **OpenJDK 8**, wenn minimale Annahmen bzgl. der Laufzeitumgebung erwünscht sind,
- das gemäß Abschnitt 2.1.1 installierte **OpenJDK 13**, wenn alle aktuellen Java-Sprachmerkmale genutzt werden sollen.

Die in IntelliJ enthaltene OpenJDK-Version 11.0.4 mit dem Startverzeichnis

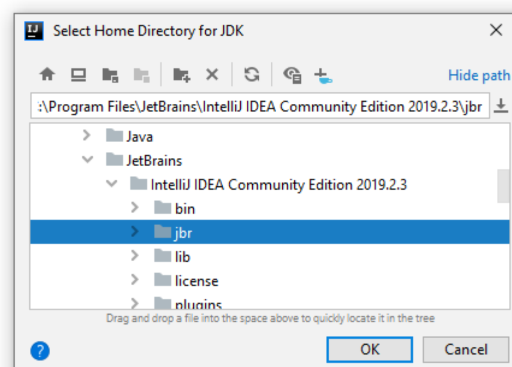
C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2019.2.3\jbr

kann ebenfalls als SDK für Projekte verwendet werden. Damit stehen uns die beiden momentan verfügbaren LTS-Version von Java (8 und 11) sowie die neueste Version 13 als SDK zur Verfügung. Wenn Sie in Ihrer IntelliJ-Installation SDKs mit diesen Hauptversionen und mit den Namen OpenJDK8, OpenJDK11 bzw. OpenJDK13 einrichten, dann sollten Sie alle im Kurs angebotenen Beispielprojekte problemlos in IntelliJ öffnen können.

Um das in IntelliJ enthaltene OpenJDK 11 als SDK - Option für neue Projekte zu vereinbaren, klicken wir auf das + - Symbol am oberen Fensterrand und wählen den Typ **JDK**:



Dann wählen wir den oben angegebenen SDK-Basisordner:



Wir vereinbaren den SDK-Namen **OpenJDK 11** und nötigenfalls den folgenden **Documentation Path**

https://docs.oracle.com/en/java/javase/11/docs/api/

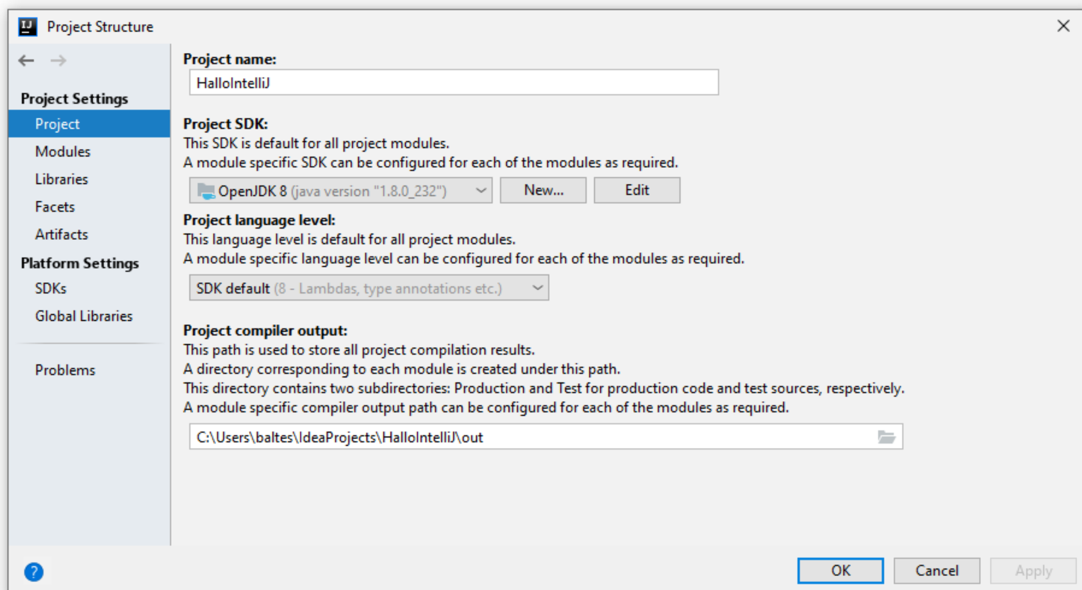
2.4.6.2 *Struktur des aktuellen Projekts*

Nach dem Menübefehl

File > Project Structure > Project

oder einem Mausklick auf das Symbol

sind im folgenden Dialog



unter **Project Settings** wichtige Konfigurationen möglich:

- **Project name**
  Hier lässt sich der Projektname ändern.
- **Project SDK**
  Hier wählt man das für die Ausführung des aktuellen Projekts zu verwendende SDK. Kommen z. B. in einem Projekt mit Java 13 übersetzte Bibliotheksklassen zum Einsatz, dann muss ein SDK mit dem entsprechenden Versionsniveau verwenden werden, damit die Ausführung der Anwendung gelingt.
- **Project language level**
  Hier legt man die Java-Version des Compilers und damit auch das Verhalten der IntelliJ-Syntaxvervollständigung fest. Dabei muss auf Kompatibilität geachtet werden. Es macht z. B. keinen Sinn, das OpenJDK 8 als SDK zu verwenden und gleichzeitig das Sprachniveau 11 zu verlangen. Es ist hingegen möglich, ein Sprachniveau unterhalb der eingestellten SDK-Version zu wählen. Dann kann man sich in eigenen, neu zu erstellenden Klassen auf die ältere Syntax beschränken, aber trotzdem Bibliotheksklassen einbinden, die mit einem höheren **language level** (also von einer entsprechenden Compiler-Version) übersetzt worden sind.

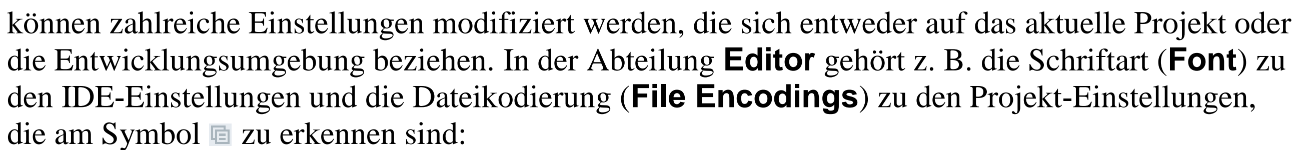
Damit ein ausgeliefertes Programm auf einem Kundenrechner von der dortigen JVM ausgeführt werden kann, dürfen alle ausgelieferten Klassen maximal das Compiler-Niveau der angetroffenen JVM haben. Bei einer mit dem Sprachniveau 11 erstellten Anwendung scheitert der Start auf einem Kundenrechner mit der JVM 8 verständlicherweise mit einer Fehlermeldung:

```
Exception in thread "main" java.lang.UnsupportedClassVersionError: Bruchaddition
has been compiled by a more recent version of the Java Runtime
```

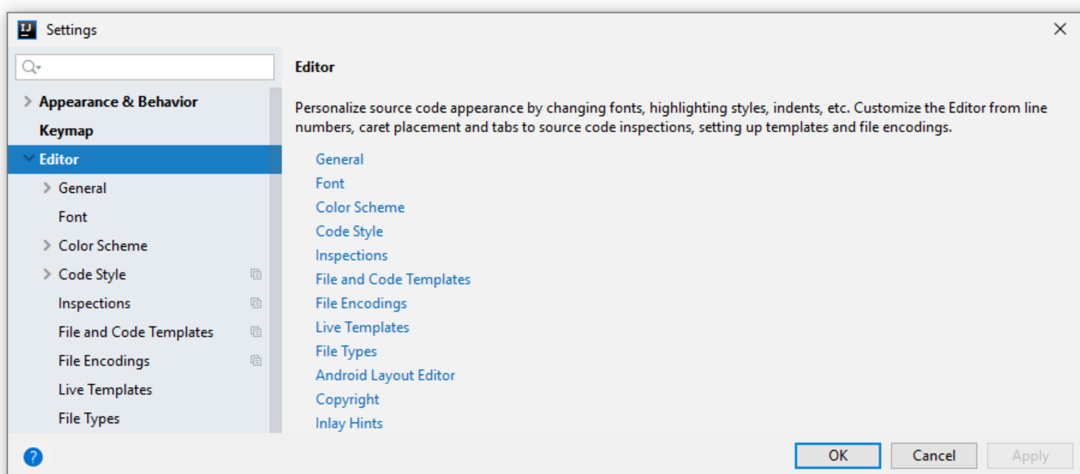
2.4.6.3 Einstellungen für IntelliJ oder das aktuelle Projekt

Über die Tastenkombination **Strg + Alt + S** oder den Menübefehl

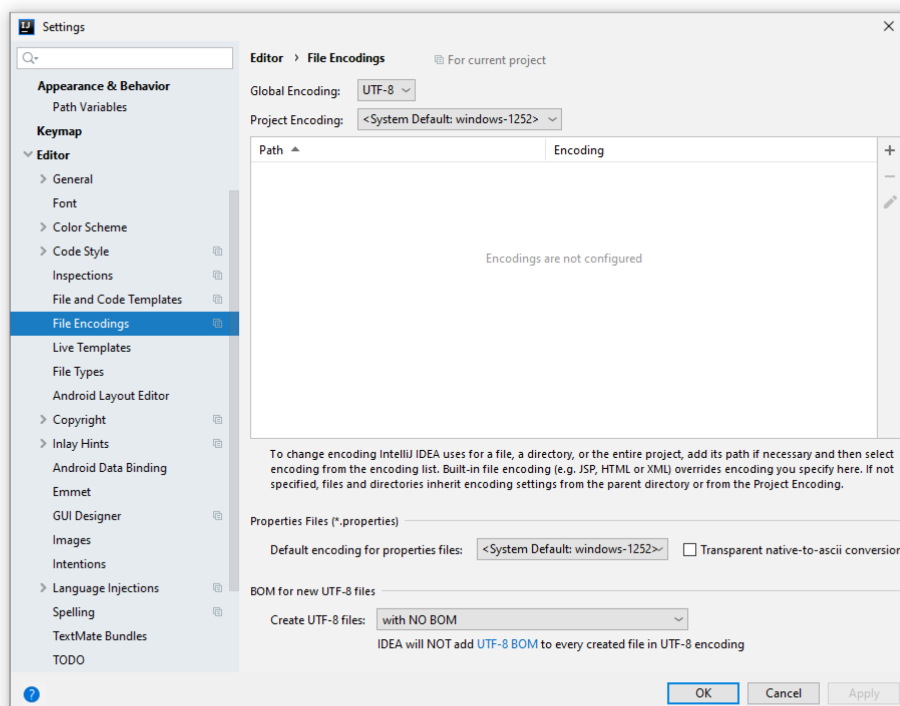
File > Settings



können zahlreiche Einstellungen modifiziert werden, die sich entweder auf das aktuelle Projekt oder die Entwicklungsumgebung beziehen. In der Abteilung **Editor** gehört z. B. die Schriftart (**Font**) zu den IDE-Einstellungen und die Dateikodierung (**File Encodings**) zu den Projekt-Einstellungen, die am Symbol ▤ zu erkennen sind:



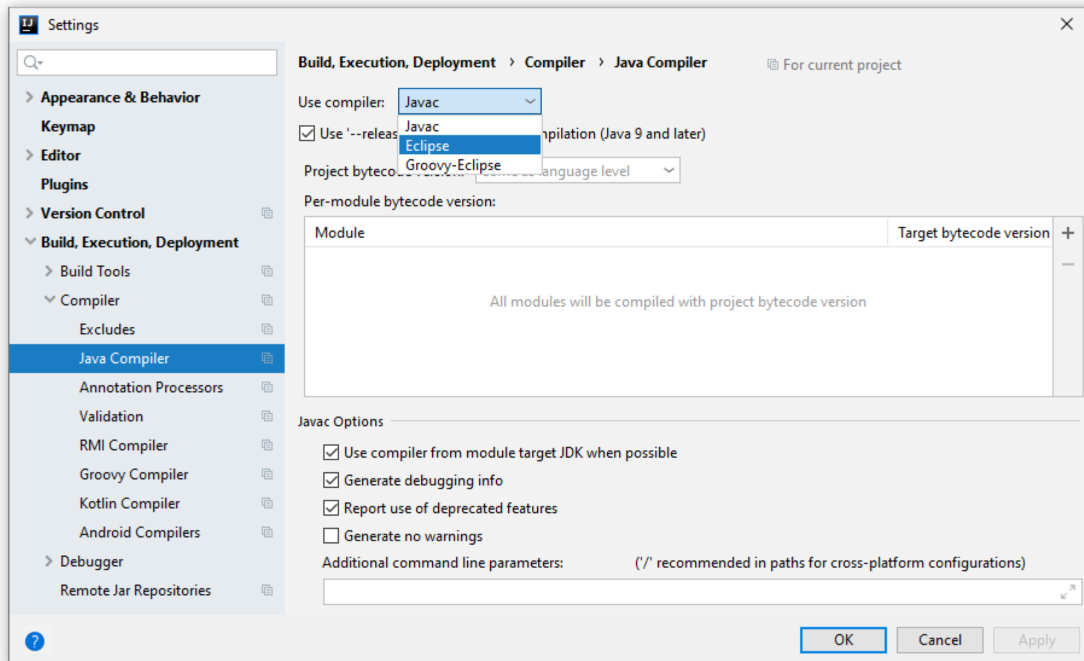
IntelliJ verwendet unter Windows für Quellcodedateien per Voreinstellung die UTF-8 - Kodierung (ohne *Byte Order Mark*, BOM), sodass bei der Übertragung der Dateien auf einen Entwicklungsrechner mit einem anderen Betriebssystem (MacOS, Linux oder UNIX) keine Kodierungsinkompatibilität stört. Es gibt also keinen Grund, unter Windows die folgenden Kodierungsvoreinstellungen:



Nach

File > Settings > Build, Execution, Deployment > Compiler > Java-Compiler

kann im folgenden Dialog z. B. für das aktuelle Projekt der voreingestellte Compiler **javac.exe** aus dem Projekt-SDK durch den Compiler aus der Open Source - Entwicklungsumgebung Eclipse ersetzt werden:



Die IDE-Konfiguration zu IntelliJ 2019.2.x für den Windows-Benutzer `otto` landet im folgenden Ordner:

C:\Users\otto\.IdeaIC2019.2\config

Weitere Einstellungen befinden sich in:

C:\Users\otto\AppData\Roaming\JetBrains

Sollte die IDE-Konfiguration einmal außer Kontrolle geraten, kann man durch Löschen der Einstellungsordner (bei inaktiver Entwicklungsumgebung) den Ausgangszustand wiederherstellen.

Die Einstellungen zu einem Projekt befinden sich in einer Serie von XML-Dateien im **.idea** - Unterordner des Projekts, also z. B. in:

C:\Users\otto\IdeaProjects\HalloEclipse\.idea

2.4.6.4 Einstellungen für neue Projekte

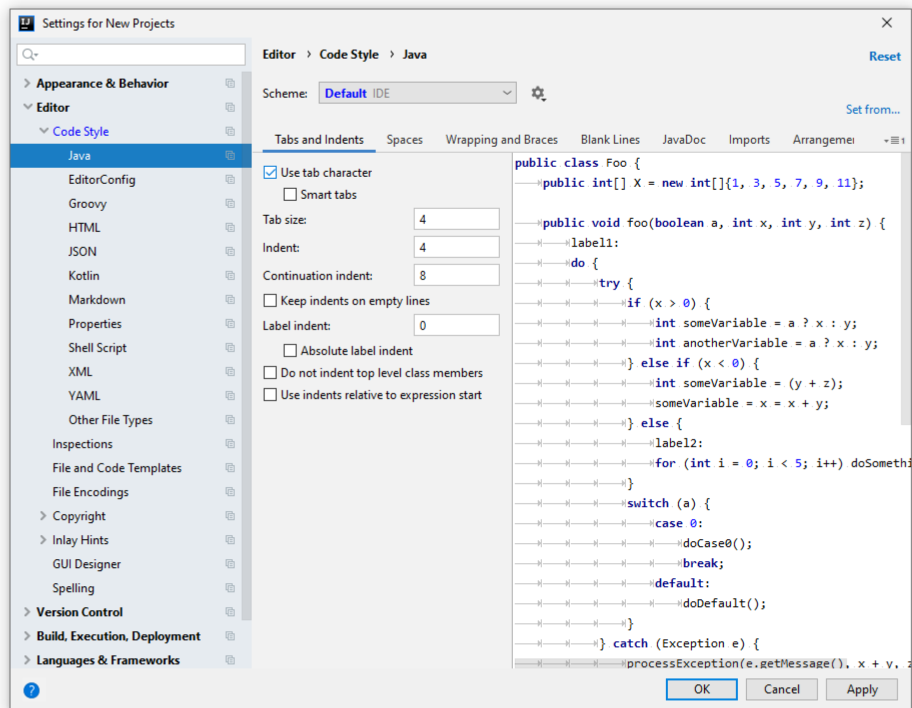
Über

File > Other Settings > Settings for New Projects

legt man die Einstellungen für *neue* Projekte fest. Man kann z. B. im Abschnitt

Editor > Code Style > Java

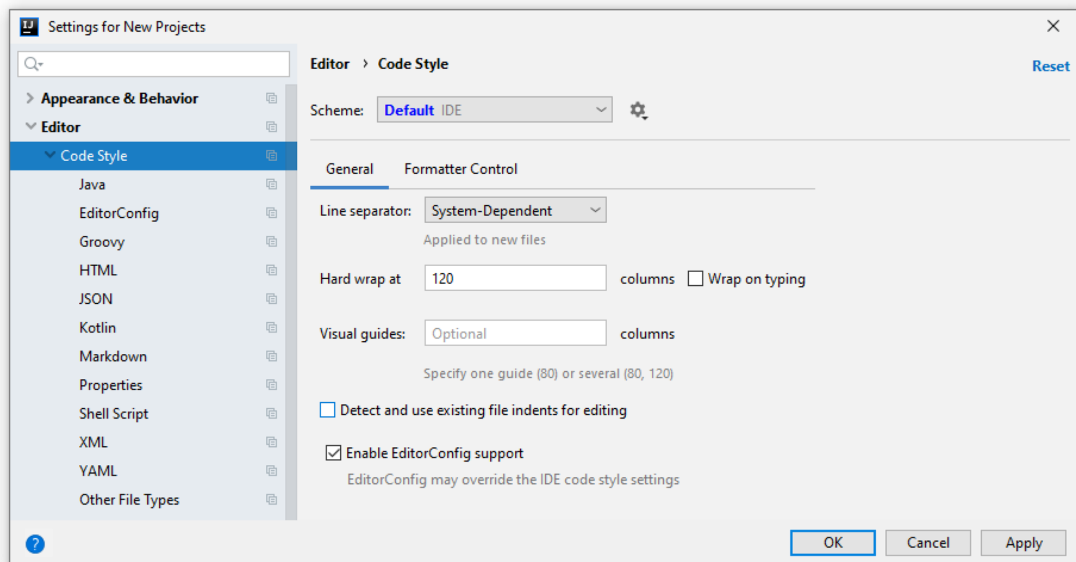
dafür sorgen, dass im Editor für Java-Quellcode das Tabulatorzeichen *nicht* durch mehrere Leerzeichen ersetzt wird:



Diese Einstellung bleibt allerdings ohne Effekt, wenn IntelliJ in einer bereits vorhandenen Datei durch Leerzeichen realisierte Einrückungen antrifft. Soll die Tabulatortaste auch dort ein Tabulatorzeichen produzieren, muss im Abschnitt

Editor > Code Style

das Kontrollkästchen bei **Detect and use existing file indents for editing** entfernt werden:



2.4.7 Übungsprojekte zum Kurs verwenden

Die im Kurs angebotenen IntelliJ-Übungsprojekte lassen sich auf Ihrem Rechner mit der dortigen IntelliJ-Installation aus den kopierten Projektordnern öffnen, wenn auf Ihrem Rechner ...

- das OpenJDK 8 (zum Bezug und zur Installation siehe Abschnitt 1.2.1), das OpenJDK 11 (siehe Abschnitt 2.4.6.1) sowie das OpenJDK 13 (zum Bezug und zur Installation siehe Abschnitt 2.1.1) installiert sind,
- und in IntelliJ für diese SDKs die Namen **OpenJDK 8**, **OpenJDK 11** bzw. **OpenJDK 13** vereinbart sind.

Zur Einrichtung von SDKs in IntelliJ siehe Abschnitt 2.4.6.1.

Einige Übungsprojekte verwenden die selbst erstellte, nicht zum Java-API gehörige Klasse `Simput` zur Vereinfachung der Konsolenausgabe. Im Abschnitt 3.4.2 werden Sie erfahren, wie die Java-Archivdatei **Simput.jar** mit der Klasse `Simput` als IntelliJ-globale Bibliothek eingerichtet wird, sodass die Klasse `Simput` in allen Projekten verfügbar ist, die eine Bibliothek mit dem Namen `Simput` einbinden.

2.5 OpenJFX und Scene Builder installieren

Wir wollen bei der Entwicklung von Programmen mit grafischer Bedienoberfläche die JavaFX-Bibliothek einsetzen. Seit Java 11 ist diese Bibliothek kein JDK-Bestandteil mehr, wird aber unter dem Namen *OpenJFX* als Open Source (unter derselben Lizenz wie das OpenJDK) aktiv weiterentwickelt, wobei sich besonders die Firma **Gluon** engagiert.[1] Hier

https://gluonhq.com/products/javafx/

steht (im Oktober 2019) die OpenJFX-Version 13.0.1 zur Verfügung. Wir beziehen die Variante **JavaFX Windows SDK** in der Datei **openjfx-13.0.1_windows-x64_bin-sdk.zip** und packen diese im folgenden Ordner aus:

C:\Program Files\Java\OpenJFX-SDK-13

Wie es der Bestandteil **windows** im Namen der heruntergeladenen Datei vermuten lässt, sind die mit diesem SDK erstellten JavaFX-Programme wegen der enthaltenen nativen Bibliotheken (DLL-Dateien) nur unter Windows zu verwenden. Um eine Multi-Plattform - JavaFX-Anwendung zu erstellen, muss man auch die SDK-Varianten für Linux und MacOS X herunterladen.[2]

Weil die gemäß Abschnitt 1.2.1 installierte OpenJDK 8 - Distribution aus dem Open Source - Projekt **ojdkbuild** ein OpenJFX-SDK enthält, und außerdem das in IntelliJ IDEA integrierte OpenJDK 11 ebenfalls mit einer OpenJFX-Unterstützung ausgestattet ist, können wir JavaFX-Anwendungen für die LTS-Versionen Java 8 und 11 sowie für aktuelle Java-Version 13 erstellen.

Beim Einsatz der JavaFX-Technik wird die Bedienoberfläche in der Regel in einer FXML-Datei deklariert. Deren Gestaltung wird erheblich erleichtert durch das unter der BSD-Lizenz stehende Programm *Scene Builder*, das von der Firma Oracle entwickelt wurde und mittlerweile von der Firma Gluon gepflegte wird. Es steht auf der folgenden Webseite zur Verfügung:

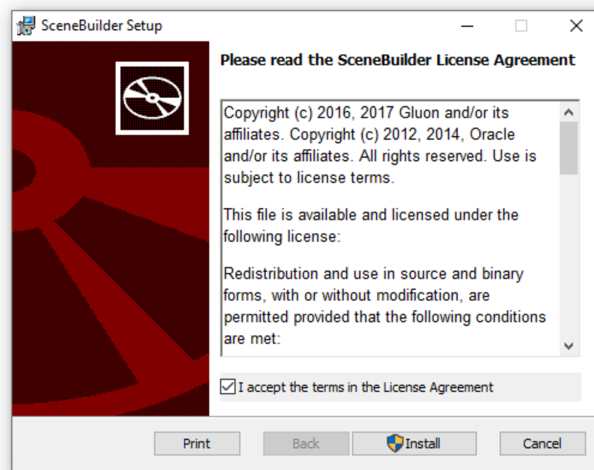
https://gluonhq.com/products/scene-builder/

Aktuell (im Oktober 2019) werden die Version 8.5.0 (für Java 8) sowie die Version 11.0.0 (für Java 11) angeboten. Wir beschränken uns auf die Version 11.0.0, die auch zusammen mit JavaFX 13 verwendbar ist, wählen das Format **Windows Installer** und erhalten somit die Datei **SceneBuilder-11.0.0.msi**.

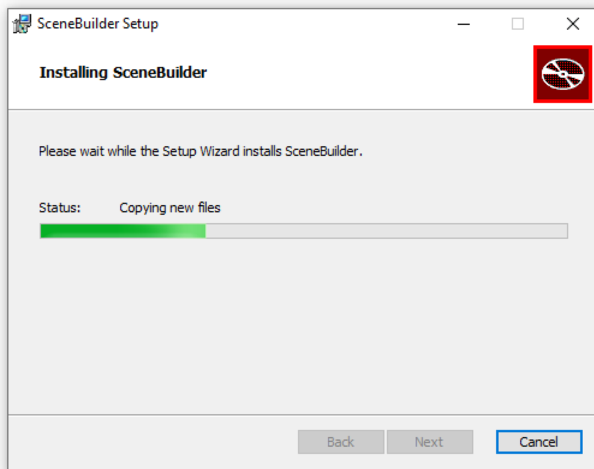
[1] Im Manuskript werden die Bezeichnungen *JavaFX* und *OpenJFX* synonym verwendet.

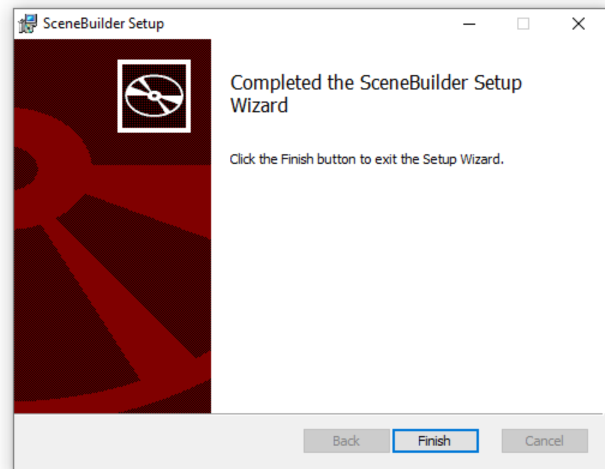
[2] https://openjfx.io/openjfx-docs/#modular

Wir starten die Installation per Doppelklick auf diese MSI-Datei und akzeptieren die Lizenzbedingungen:



Das Installationsprogramm legt ohne Fragen bzw. Optionen los und vollendet zügig die Installation:



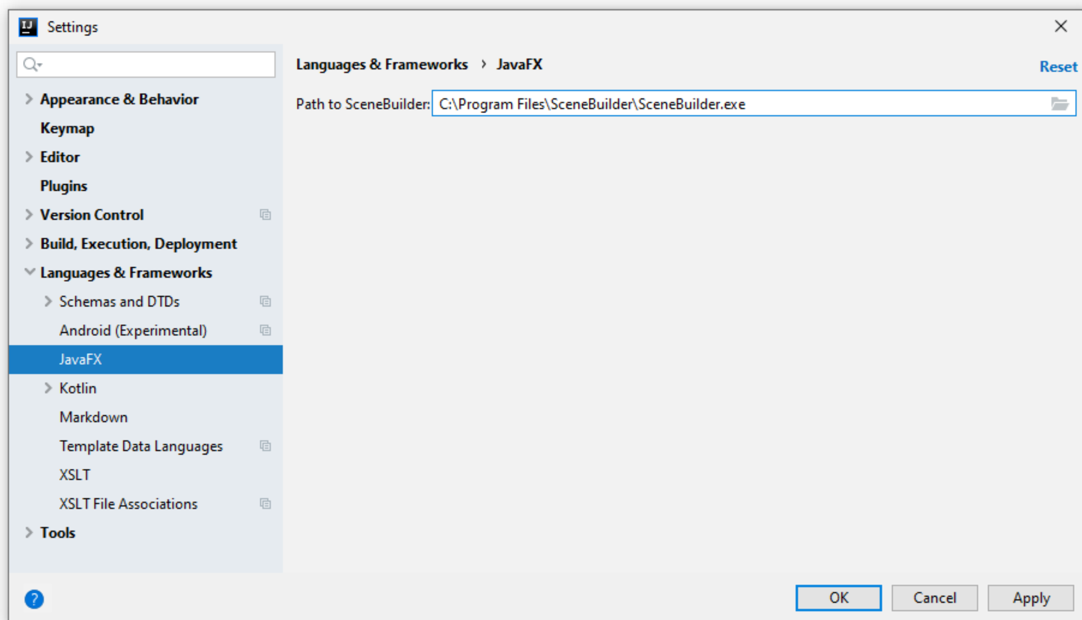


Man findet den Scene Builder schließlich im Ordner **C:\Program Files\SceneBuilder**.

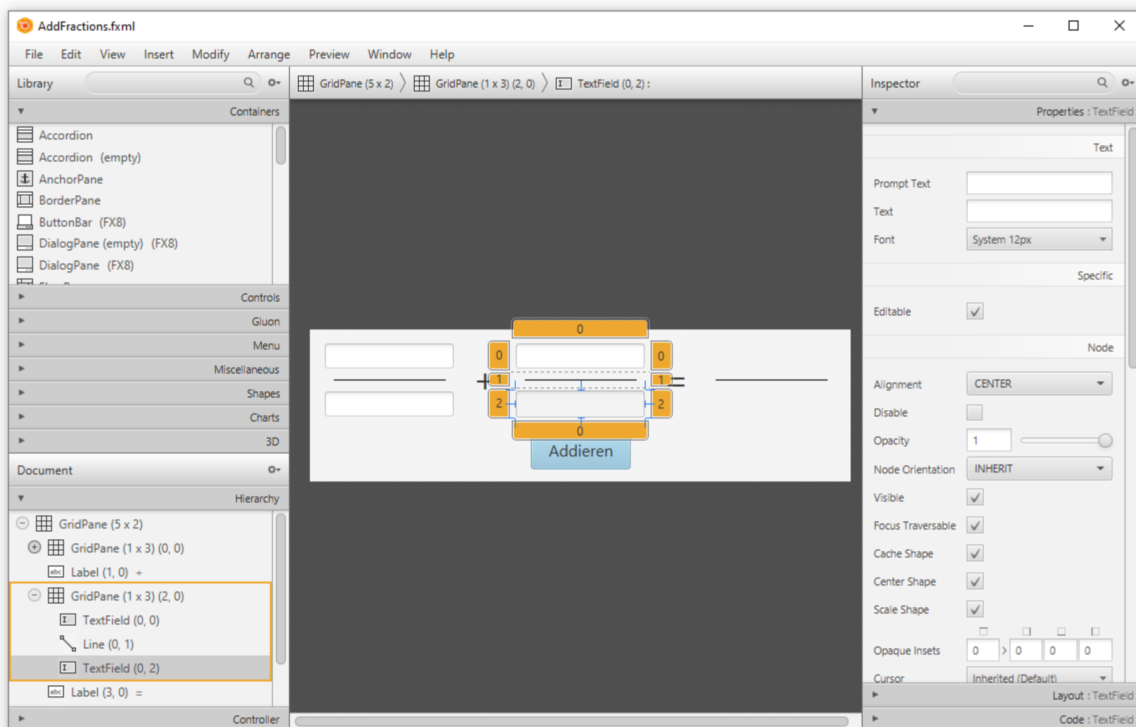
Damit unsere Entwicklungsumgebung IntelliJ IDEA mit dem Scene Builder kooperieren kann, muss nach dem Menübefehl

File > Settings > Languages & Frameworks > JavaFX

der Pfad zum ausführbaren Programm bekanntgegeben werden, z. B.:



Eine erste Verwendung des Scene Builders werden Sie im Abschnitt 4.9 erleben. Ein Blick auf die Arbeitsoberfläche des GUI-Designers mit dem geöffneten Fenster des im Abschnitt 1.2.3 vorgestellten Bruchadditionsprogramms lässt erkennen, dass wir zur Entwicklung attraktiver Programme ein modernes Werkzeug zur Verfügung haben:



Hinweis von Rolf Schwung zu JavaFX unter Linux (per Mail erhalten am 3.3.21):
Hallo,

vielen Dank für das Skript "Einführung in das Programmieren mit Java13".
Ich nutze das Dokument zum Selbststudium und komme damit gut zurecht.

Nur bei der Installation/Konfiguration von JavaFX in Intellij hatte ich meine Probleme und bin im Internet fündig geworden.
Ich nutze ein Linux System, die Infos habe ich hier gefunden:

https://kofler.info/java-11-javafx-intellij-idea-und-linux/

Evtl. können diese Hinweise entsprechend übernommen werden.

Mit freundlichem Gruß
Rolf Schwung

Herr Schwung ist bereits in die Liste der Beiträger im Vorwort aufgenommen worden.

2.6 Übungsaufgaben zum Kapitel 2

1) Experimentieren Sie mit dem `Hallo`-Beispielprogramm aus Abschnitt 2.2.1, z. B. indem Sie weitere Ausgabeanweisungen ergänzen.

2) Beseitigen Sie die Fehler in folgender Variante des `Hallo`-Programms:

```
class Hallo {
    static void mein(String[] args) {
        System.out.println("Hallo allerseits!);
    }
```

3) Welche der folgenden Aussagen sind richtig bzw. falsch?

1. Beim Übersetzen einer Java-Quellcodedatei mit dem OpenJDK-Compiler **javac.exe** muss man den Dateinamen samt Erweiterung (**.java**) angeben.
2. Beim Starten eines Java-Programms muss man den Namen der auszuführenden Klasse samt Extension (**.class**) angeben.
3. Damit der Aufruf des OpenJDK-Compilers **javac.exe** (ohne Pfadangabe) von jedem Verzeichnis aus klappt, muss unter Windows das **bin**-Unterverzeichnis der OpenJDK-Installation vorrangig in die Definition der Umgebungsvariablen PATH aufgenommen werden.
4. Die **main()** - Methode der Startklasse eines Java-Programms muss einen Parameter mit dem Datentyp **String[]** und dem Namen `args` besitzen, damit sie von der JVM erkannt wird.

4) Führen Sie nach Möglichkeit auf Ihrem eigenen PC mindestens die in den Abschnitten 1.2.1 (OpenJDK 8) und 2.3 (IntelliJ 2019.2.3, inkl. OpenJDK 11) beschriebenen Installationen aus. Richten Sie in IntelliJ basierend auf den installierten OpenJDK-Versionen jeweils ein SDK ein (siehe Abschnitt 2.4.6.1).

OpenJFX (im OpenJDK 8 aus dem ojdkbuild-Projekt enthalten, im OpenJDK11 und im OpenJDK 13 aber nicht) sowie den Scene Builder benötigen wir erst zur Erstellung von Programmen mit grafischer Bedienoberfläche.

5) Kopieren Sie die Klasse

…**\BspUeb\Simput\Standardpaket\Simput.class**

auf Ihren heimischen PC, und tragen Sie das Zielverzeichnis in den CLASSPATH ein (siehe Abschnitt 2.2.4). Testen Sie den Zugriff auf die **class**-Datei z. B. mit der Konsolenvariante des Bruchadditionsprogramms (siehe Abschnitt 1.2.2). Alternativ können Sie auch die Java-Archivdatei

…**\BspUeb\Simput\Standardpaket\Simput.jar**

kopieren und in den Klassenpfad aufnehmen. Mit Java-Archivdateien werden wir uns später noch ausführlich beschäftigen.

3 Elementare Sprachelemente

Im Kapitel 1 wurde anhand eines halbwegs realistischen Beispiels ein erster Eindruck von der objektorientierten Software-Entwicklung mit Java vermittelt. Nun erarbeiten wir uns die Details der Programmiersprache Java und beginnen dabei mit elementaren Sprachelementen. Diese dienen zur Realisation von Algorithmen innerhalb von Methoden und sehen bei Java nicht wesentlich anders aus als bei älteren, *nicht* objektorientierten Sprachen (z. B. C).

3.1 Einstieg

3.1.1 Aufbau eines Java-Programms

Zunächst soll unser bisheriges Wissen über die Struktur von Java-Programmen zusammengefasst werden:

- Ein Java-Programm besteht aus **Klassen**.
  Für das Bruchrechnungsbeispiel im Abschnitt 1.1 wurden die Klassen `Bruch` und `Bruchaddition` definiert. In den Methoden der beiden Klassen kommen weitere Klassen zum Einsatz:
  - Klassen aus der Standardbibliothek (z. B. **System**, **Math**)
  - Die zur Erleichterung von Benutzereingaben in Konsolenprogrammen selbst erstellte Klasse `Simput`

  Meist verwendet man für den Quellcode einer Klasse jeweils eine eigene Textdatei mit der Namenserweiterung **.java**.[1] Der Compiler erzeugt grundsätzlich für jede Klasse eine eigene Bytecode-Datei mit der Namenserweiterung **.class**.
- Eine **Klassendefinition** besteht aus …
  - dem **Kopf**
    Er enthält nach dem Schlüsselwort **class** den Namen der Klasse. Soll eine Klasse für beliebige andere Klassen (aus fremden Paketen, siehe unten) nutzbar sein, muss dem Schlüsselwort **class** der Zugriffsmodifikator **public** vorangestellt werden, z. B.:
    ```
    public class Bruch {
      . . .
    }
    ```
  - und dem **Rumpf**
    Begrenzt durch ein Paar geschweifter Klammern befinden sich hier …
    - die Deklarationen der **Instanz-** und **Klassenvariablen** (Eigenschaften)
    - und die Definitionen der **Methoden** (Handlungskompetenzen).
- Auch eine **Methodendefinition** besteht aus …
  - dem **Kopf**
    Hier werden vereinbart: Modifikatoren, Rückgabetyp, Name der Methode, Parameterliste. All diese Bestandteile werden noch ausführlich erläutert.

---

[1] Unsere Entwicklungsumgebung IntelliJ verwendet unter Windows für Quellcodedateien per Voreinstellung die UTF-8 - Kodierung (ohne *Byte Order Mark*, BOM), sodass bei der Übertragung der Dateien auf einen Entwicklungsrechner mit einem anderen Betriebssystem (MacOS, Linux oder UNIX) keine Kodierungsinkompatibilität stört. Eine Änderung der Kodierung ist möglich über:
File > Settings > Editor > File Encodings

- und dem **Rumpf**
  Begrenzt durch ein Paar geschweifte Klammern befinden sich hier **Anweisungen**, mit denen zur Realisation von Algorithmen z. B. lokale Variablen deklariert oder verändert werden. Der Unterschied zwischen Instanzvariablen (Eigenschaften von Objekten), statischen Variablen (Eigenschaften von Klassen) und lokalen Variablen von Methoden wird im Abschnitt 3.3 erläutert.

- Eine **Anweisung** ist die kleinste ausführbare Einheit eines Programms.
  In Java sind bis auf wenige Ausnahmen alle Anweisungen mit einem **Semikolon** abzuschließen.
- Von den Klassen eines Programms muss eine **startfähig** sein.
  Dazu benötigt sie eine **Methode** mit dem Namen **main()**, dem Rückgabetyp **void**, einer bestimmten Parameterliste (**String[] args**) sowie den Modifikatoren **public** und **static**. Im Bruchrechnungsbeispiel im Abschnitt 1.1 ist die Klasse `Bruchaddition` startfähig.

3.1.2 Projektrahmen zum Üben von elementaren Sprachelementen

Während der Beschäftigung mit elementaren Java-Sprachelementen werden wir der Einfachheit halber mit einer relativ untypischen, jedenfalls nicht sonderlich objektorientierten Programmstruktur arbeiten, die Sie schon aus dem `Hallo`-Beispiel kennen (siehe Abschnitt 2.2.1). Es wird nur *eine* Klasse definiert, und diese erhält nur eine einzige Methodendefinition. Weil die Klasse startfähig sein muss, liegt der einzige Methodenkopf nach den im letzten Abschnitt wiederholten Regeln fest. Weil die Klasse *nicht* für andere Klassen ansprechbar sein soll, ist der Zugriffsmodifikator **public** *für die Klasse* überflüssig, und wir erhalten die folgende Programmstruktur:

```
class Prog {
   public static void main(String[] args) {
      //Platz für elementare Sprachelemente
   }
}
```

Damit die pseudo-objektorientierten (POO-) Programme Ihren Programmierstil nicht prägen, wurde an den Beginn des Manuskripts ein Beispiel gestellt (Bruchrechnung), das bereits etliche OOP-Prinzipien realisiert.

Für die meist kurzzeitige Beschäftigung mit bestimmten elementaren Sprachelementen lohnt sich selten ein spezielles IntelliJ-Projekt. Legen Sie daher für solche Zwecke mit dem Menübefehl

File > New > Project

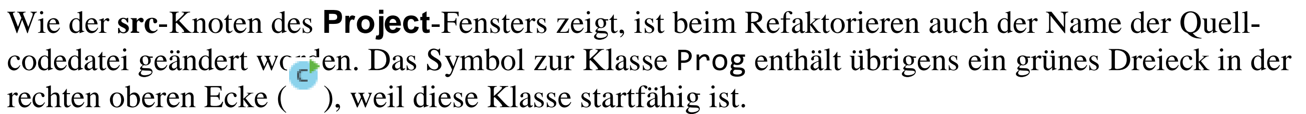
analog zu Abschnitt 2.4.2 ...

- ein **Java**-Projekt
- basierend auf dem OpenJDK 8
- unter Verwendung des Templates **Command Line App**
- mit dem Namen `Prog`
- ohne **Base package**

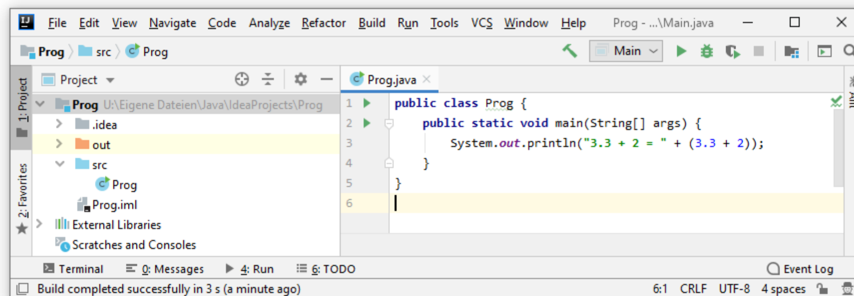
an. Den überflüssigen **class**-Modifikator **public** im automatisch erstellten Klassendefinitionskopf können Sie löschen oder belassen, weil er keinen Nachteil bringt.

Ändern Sie mit der im Abschnitt 2.4.3.5 beschriebenen Refaktorierung den Namen der vordefinierten Klasse von `Main` in `Prog`, z. B. so:

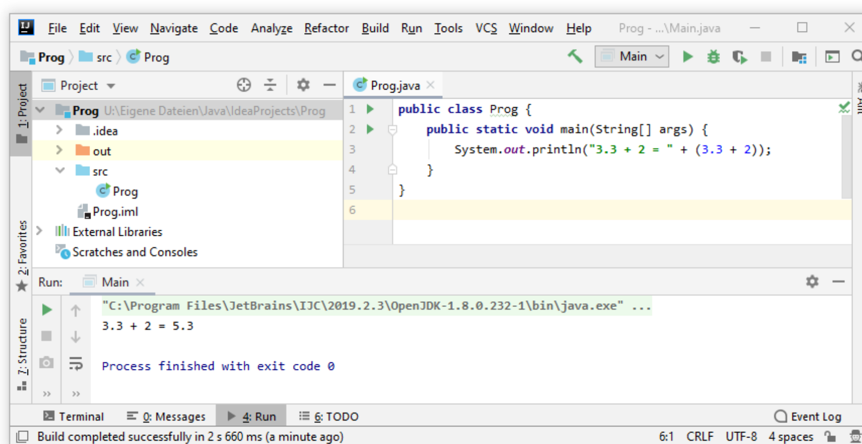
- Einfügemarke im Editor auf den alten Namen setzen
- Tastenkombination **Umschalt + F6**
- Neuen Namen eintragen und mit **Enter** quittieren

Wie der **src**-Knoten des **Project**-Fensters zeigt, ist beim Refaktorieren auch der Name der Quellcodedatei geändert worden. Das Symbol zur Klasse `Prog` enthält übrigens ein grünes Dreieck in der rechten oberen Ecke (), weil diese Klasse startfähig ist.

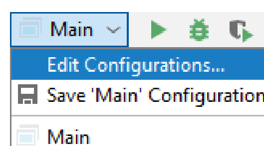
Zum Üben elementarer Sprachelemente werden wir im Rumpf der **main()** - Methode passende Anweisungen einfügen, z. B.:



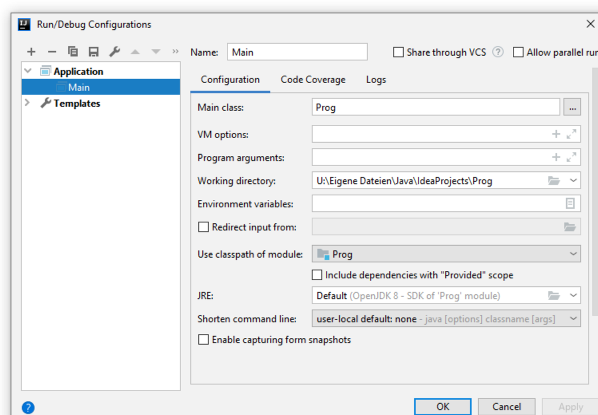
Über das Symbol oder die Tastenkombination **Umschalt + F10** lassen wir das Programm übersetzen und ausführen:



Dabei wird die vorgegebene Ausführungskonfiguration verwendet. Wenn wir das Drop-Down - Menü zur Ausführungskonfiguration öffnen und das Item **Edit Configuration**



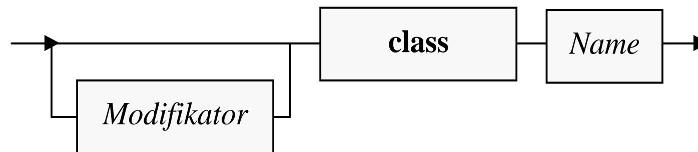
wählen, stellt sich heraus, dass IntelliJ beim Refaktorieren auch die **Main class** angepasst hat. Man könnte die Konfiguration umbenennen, was sich aber nicht lohnt:



3.1.3 Syntaxdiagramme

Um für Java-Sprachbestandteile (z. B. Definitionen oder Anweisungen) die Bildungsvorschriften kompakt und genau zu beschreiben, werden wir im Manuskript u.a. sogenannte **Syntaxdiagramme** einsetzen, für die folgende Vereinbarungen gelten:

- Man bewegt sich vorwärts in Pfeilrichtung durch das Syntaxdiagramm und gelangt dabei zu Rechtecken, welche die an der jeweiligen Stelle zulässigen Sprachbestandteile angeben, wie z. B. im folgenden Syntaxdiagramm zum Kopf einer Klassendefinition:



- Bei einer Verzweigung kann man sich für eine Richtung entscheiden, wenn nicht per Pfeil eine Bewegungsrichtung vorgeschrieben ist. Zulässige Realisationen zum obigen Segment sind also z. B.:
    - `class Bruchaddition`
    - `public class Bruch`

  Verboten sind hingegen z. B. folgende Sequenzen:
    - `class public Bruchaddition`
    - `Bruchaddition public class`
- Für **konstante** (**terminale**) Sprachbestandteile, die aus einem Rechteck exakt in der angegebenen Form in konkreten Quellcode zu übernehmen sind, wird **fette** Schrift verwendet.
- *Platzhalter* sind an *kursiver* Schrift zu erkennen. Im konkreten Quellcode muss anstelle des Platzhalters eine zulässige Realisation stehen, und die zugehörigen Bildungsregeln sind an anderer Stelle (z. B. in einem anderen Syntaxdiagramm) erklärt.
- Als Klassenmodifikator ist uns bisher nur der Zugriffsmodifikator **public** begegnet, der für die allgemeine Verfügbarkeit einer Klasse sorgt. Später werden Sie noch weitere Klassenmodifikatoren kennenlernen. Sicher kommt niemand auf die Idee, z. B. den Modifikator **public** mehrfach zu vergeben und damit gegen eine Java-Syntaxregel zu verstoßen. Das obige (möglichst einfach gehaltene) Syntaxdiagrammsegment lässt diese offenbar sinnlose Praxis zu. Es bieten sich zwei Lösungen an:
    - Das Syntaxdiagramm mit einem gesteigerten Aufwand präzisieren
    - Durch eine generelle Regel die Mehrfachverwendung eines Modifikators verbieten

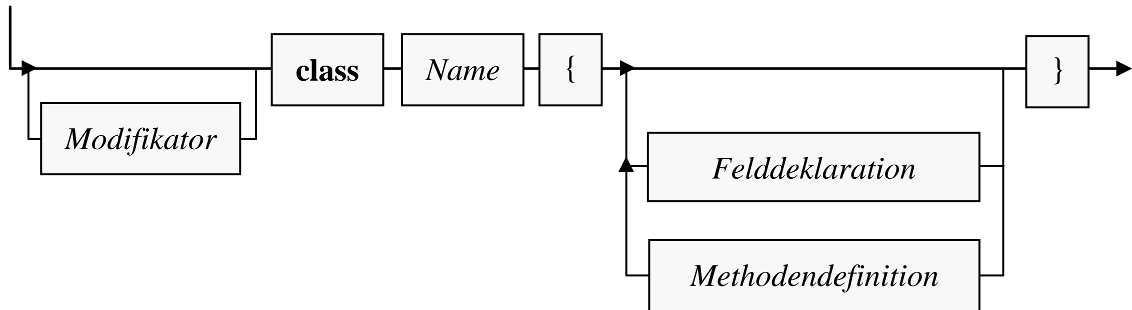
  Im Manuskript wird die zweite Lösung verwendet.

Als Beispiele betrachten wir anschließend die Syntaxdiagramme zur Definition von Klassen und Methoden. Aus didaktischen Gründen zeigen die Diagramme nur solche Sprachbestandteile, die im Beispielprogramm von Abschnitt 1.1 (mit der Klasse `Bruch`) verwendet wurden. Durch den engen Bezug zum Beispiel sollte es in diesem Abschnitt gelingen, …

- Syntaxdiagramme als metasprachliche Hilfsmittel einzuführen
- und gleichzeitig zur allmählichen Klärung der wichtigen Begriffe *Klasse* und *Methode* beizutragen.

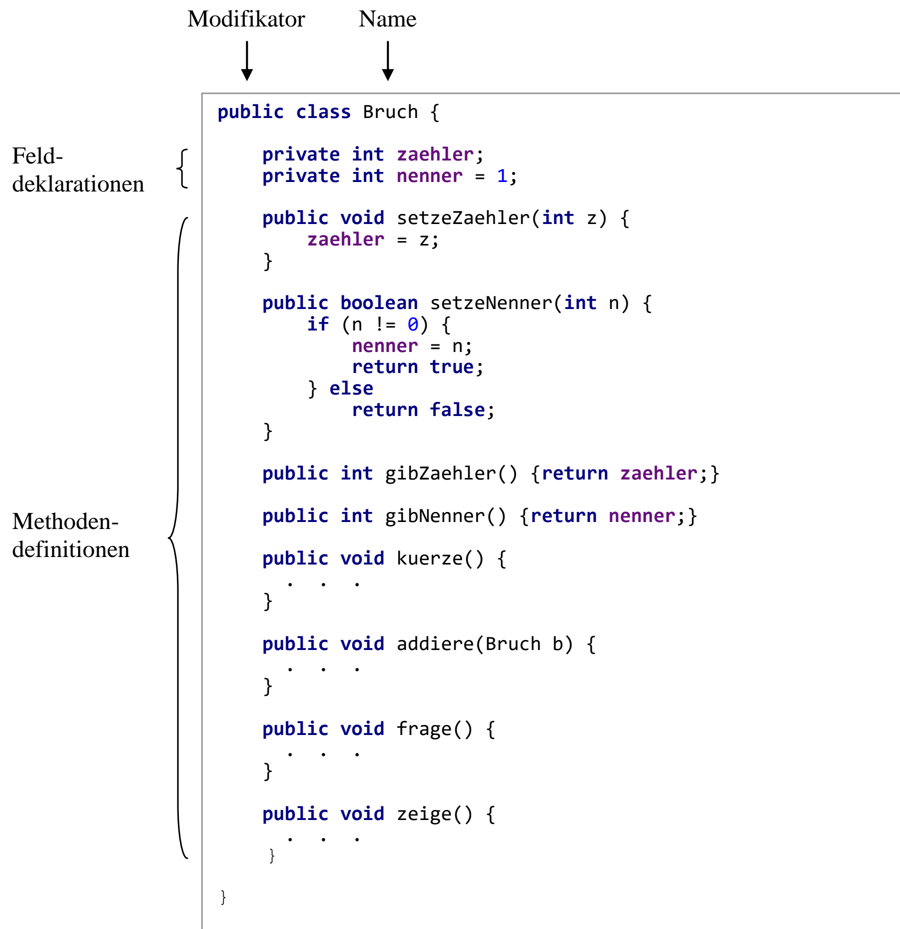
3.1.3.1 Klassendefinition

Wir arbeiten vorerst mit dem folgenden, leicht vereinfachten Klassenbegriff:



Solange man sich vorwärts bewegt (immer in Pfeilrichtung, eventuell auch in Schleifen), an den Stationen (Rechtecken) entweder den terminalen Sprachbestandteil exakt übernimmt oder den Platzhalter auf zulässige (an anderer Stelle erläuterte) Weise ersetzt, entsteht eine syntaktisch korrekte Klassendefinition.

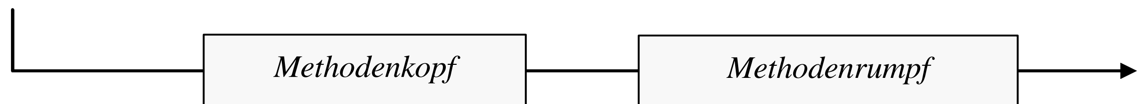
Als Beispiel betrachten wir die Klasse `Bruch` aus Abschnitt 1.1:



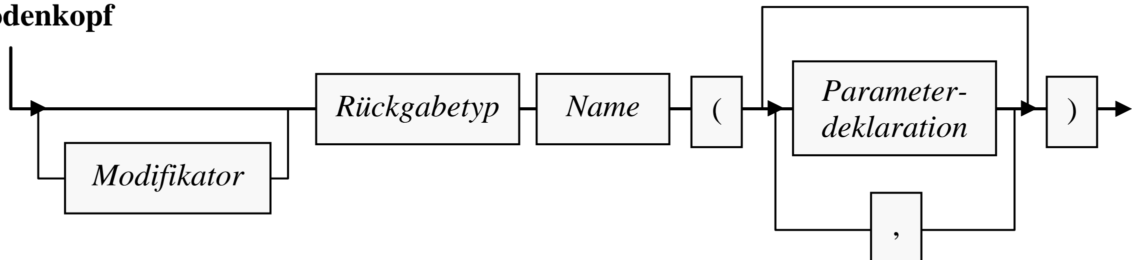
3.1.3.2 Methodendefinition

Weil *ein* Syntaxdiagramm für die komplette Methodendefinition etwas unübersichtlich wäre, betrachten wir separate Diagramme für die Begriffe *Methodenkopf* und *Methodenrumpf*:

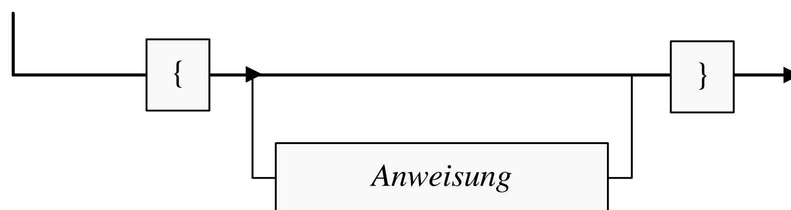
Methodendefinition



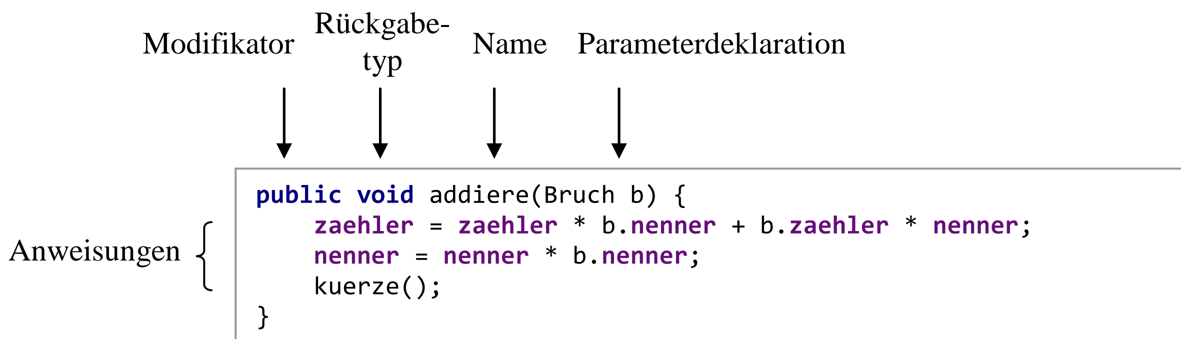
Methodenkopf



Methodenrumpf



Als Beispiel betrachten wir die Definition der `Bruch`-Methode `addiere()`:



Zur Erläuterung des Begriffs *Parameterdeklaration* beschränken wir uns vorläufig auf das Beispiel in der `addiere()` - Definition. Es enthält einen Datentyp (Klasse `Bruch`) und einen Parameternamen (`b`).

In vielen Methoden werden sogenannte *lokale Variablen* (vgl. Abschnitt 3.3.4) deklariert, z. B. in der `Bruch`-Methode `kuerze()`:

```
public void kuerze() {
    if (zaehler != 0) {
        int az = Math.abs(zaehler);
        int an = Math.abs(nenner);
         .  .  .
        zaehler = zaehler / ggt;
        nenner = nenner / ggt;
    } else
        nenner = 1;
}
```

Weil wir bald u.a. die *Variablendeklarationsanweisung* kennenlernen werden, benötigt das Syntaxdiagramm zum Methodenrumpf jedoch (im Unterschied zum Klassendefinitionsdiagramm) *kein* separates Rechteck für die Variablendeklaration.

3.1.4 Hinweise zur Gestaltung des Quellcodes

Der Compiler ist hinsichtlich der Formatierung des Quellcodes sehr tolerant und beschränkt sich auf folgende Regeln:

- Die einzelnen Bestandteile einer Definition oder Anweisung müssen in der richtigen **Reihenfolge** stehen.
- Zwischen zwei Sprachbestandteilen muss im Prinzip ein **Trennzeichen** stehen, wobei das Leerzeichen, das Tabulatorzeichen und der Zeilenumbruch erlaubt sind. Diese Trennzeichen dürfen sogar in beliebigen Anzahlen und Kombinationen auftreten. *Innerhalb* eines Sprachbestandteils (z. B. Namens) sind Trennzeichen (z. B. Zeilenumbruch) natürlich verboten.
- Zeichen mit festgelegter Bedeutung wie z. B. "{", ";", "(", "+", ">" sind **selbstisolierend**, d.h. davor und danach sind keine Trennzeichen nötig (aber erlaubt).

Um die Verarbeitung des Quellcodes durch Menschen zu erleichtern, haben sich Formatierungskonventionen entwickelt, die wir bei passender Gelegenheit besprechen werden. Ein erster Hinweis aus dieser Kategorie betrifft die Position der öffnenden geschweiften Klammer zum Rumpf einer Klassen- oder Methodendefinition. Manche Autoren setzen diese Klammer ans Ende der Kopfzeile (siehe linkes Beispiel), andere bevorzugen den Anfang der Folgezeile (siehe rechtes Beispiel):

```
class Hallo {
   public static void main(String[] par) {
      System.out.print("Hallo");
   }
}
```

```
class Hallo
{
   public static void main(String[] par)
   {
      System.out.print("Hallo");
   }
}
```

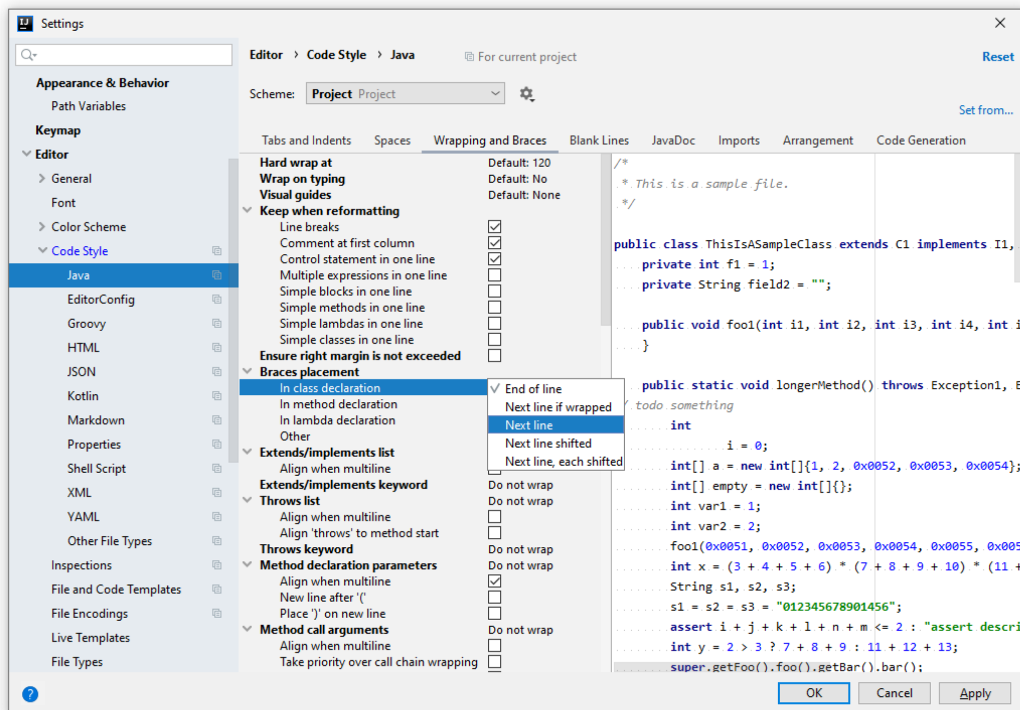
Unsere Entwicklungsumgebung verwendet per Voreinstellung die linke Variante, kann aber mit Gültigkeit für das aktuelle Projekt nach

File > Settings > Editor > Code Style > Java> Scheme=Project

bzw. mit Gültigkeit für neue Projekte nach

File > Settings > Editor > Code Style > Java > Scheme=Default

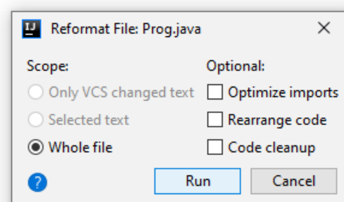
umgestimmt werden, z. B.:



Damit die (geänderten) Projekteinstellungen für eine *vorhandene* Quellcodedatei realisiert werden, muss bei aktivem Editorfenster die folgende akrobatische Tastenkombination

Umschalt + Strg + Alt + L

betätigt und anschließend die folgende Dialogbox mit **Run** quittiert werden:



IntelliJ unterstützt die Einhaltung der Layout-Regeln nicht dadurch, dass beim Editieren Abweichungen verhindert werden, sondern ...

- durch die beschriebene Möglichkeit zur automatisierten Layout-Anpassung
- und durch das Verhalten von Assistenten, die Quellcode erstellen.

Wer dieses Manuskript liest, profitiert hoffentlich von der Syntaxgestaltung durch Farben und Textattribute, die von IntelliJ stammen.

3.1.5 Kommentare

Kommentare unterstützen die spätere Verwendung (z. B. Weiterentwicklung) des Quellcodes und werden vom Compiler ignoriert. Java bietet drei Möglichkeiten, den Quellcode zu kommentieren:

- **Zeilenrestkommentar**
  Alle Zeichen vom doppelten Schrägstrich (//) bis zum Ende der Zeile gelten als Kommentar, z. B.:
  ```
  private int zaehler;     // wird automatisch mit 0 initialisiert
  ```
  Im Beispiel wird eine Variablendeklarationsanweisung in derselben Zeile kommentiert.

- **Explizit terminierter Kommentar**
  Ein durch `/*` eingeleiteter Kommentar muss explizit durch `*/` terminiert werden. In der Regel wird diese Syntax für einen ausführlichen Kommentar verwendet, der sich über mehrere Zeilen erstreckt, z. B.:

```
/*
Ein Bruch-Objekt verhindert, dass sein Nenner auf 0
gesetzt wird, und hat daher stets einen definierten Wert.
*/
public boolean setzeNenner(int n) {
   if (n != 0) {
      nenner = n;
      return true;
   } else
      return false;
}
```

  Ein mehrzeiliger Kommentar eignet sich auch dazu, einen Programmteil (vorübergehend) zu deaktivieren, ohne ihn löschen zu müssen.

  Weil der explizit terminierte Kommentar (jedenfalls ohne farbliche Hervorhebung der auskommentierten Passage) unübersichtlich ist, wird er selten verwendet.

- **Zeilenblock in IntelliJ kommentieren**
  Um in IntelliJ einen markierten Zeilenblock als Kommentar zu deklarieren, wählt man den Menübefehl

  Code > Comment with Line Comment

  oder drückt die **Strg**-Taste zusammen mit der Divisionstaste im numerischen Ziffernblock:[1]

  Strg + ÷

  Anschließend werden doppelte Schrägstriche vor jede Zeile des Blocks gesetzt. Bei Anwendung des Menü- bzw. Tastenbefehls auf einen zuvor mit Doppelschrägstrichen auskommentierten Block entfernt IntelliJ die Kommentar-Schrägstriche.

- **Dokumentationskommentar**
  Vor der Definition bzw. Deklaration von Klassen, Interfaces (siehe unten), Methoden oder Variablen darf ein Dokumentationskommentar stehen, eingeleitet mit **`/**`** und beendet mit **`*/`**. Er kann mit dem JDK-Werkzeug **javadoc** in eine HTML-Datei extrahiert werden. Die systematische Dokumentation wird über Tags für Methodenparameter, Rückgabewerte usw. unterstützt. Nähere Informationen finden Sie in der Oracle-Dokumentation JDK-Werkzeug **javadoc**:[2]

  Im Quellcode der wichtigen API-Klasse **System** befindet sich z. B. der folgende Dokumentationskommentar zum Ausgabeobjekt **out**, das Sie schon kennengelernt haben:[3]

```
/**
 * The "standard" output stream. This stream is already
 * open and ready to accept output data. Typically this stream
 * corresponds to display output or another output destination
 * specified by the host environment or user.
 * <p>
 * For simple stand-alone Java applications, a typical way to write
 * a line of output data is:
 * <blockquote><pre>
 *     System.out.println(data)
 * </pre></blockquote>
```

---

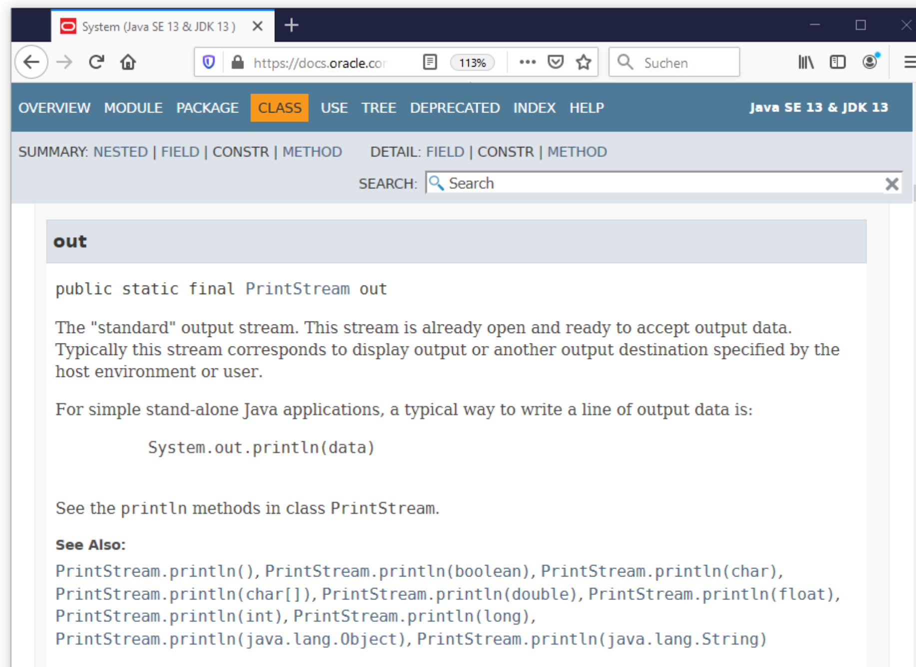
1 Die offiziell beschriebene Tastenkombination klappt nur mit einem US-Tastaturlayout.

2 Siehe z. B.: https://docs.oracle.com/en/java/javase/13/docs/specs/man/javadoc.html

3 Die Quellcodedatei **System.java** steckt im API-Quellcodearchiv **src.zip**. Wo diese Archivdatei bei der Installation landet, wird gleich beschrieben. Die Klasse **System.java** befindet sich im Paket **java.lang**. Ab Java 9 muss man zusätzlich wissen, dass das Paket **java.lang** zum Modul **java.base** gehört. Der im Text wiedergegebene Dokumentationskommentar stammt aus dem OpenJDK 13.

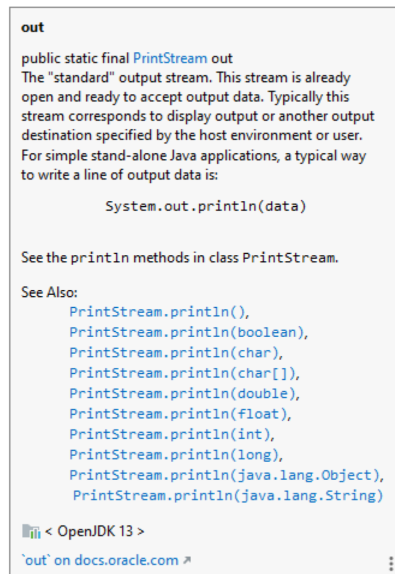
```
     * <p>
     * See the {@code println} methods in class {@code PrintStream}.
     *
     * @see     java.io.PrintStream#println()
     * @see     java.io.PrintStream#println(boolean)
     * @see     java.io.PrintStream#println(char)
     * @see     java.io.PrintStream#println(char[])
     * @see     java.io.PrintStream#println(double)
     * @see     java.io.PrintStream#println(float)
     * @see     java.io.PrintStream#println(int)
     * @see     java.io.PrintStream#println(long)
     * @see     java.io.PrintStream#println(java.lang.Object)
     * @see     java.io.PrintStream#println(java.lang.String)
     */
    public static final PrintStream out = null;
```

So sieht die vom JDK-Werkzeug **javadoc** daraus erstellte HTML-Dokumentation aus:



Diese Dokumentation kann aus IntelliJ aufgerufen werden, sofern sich die Einfügemarke des Editors in dem interessierenden Bezeichner (im Beispiel: **out**) befindet, und dann eine von den folgenden Tastenkombinationen gedrückt wird:

- **Umschalt + F1**
  Nach dieser Tastenkombination (oder nach dem Menübefehl **View > External Documentation**) versucht IntelliJ, die HTML-Datei mit der Dokumentation in einem externen Browser-Fenster zu öffnen und den Fokus passend zu setzen (siehe obiges Bildschirmfoto). Damit dies gelingt, muss ein **Documentation Path** in der SDK-Konfiguration gesetzt sein (siehe Abschnitt 2.4.6.2).
- **Strg + Q**
  Über diese Tastenkombination (oder den Menübefehl **View > Quick Documentation**) erhält man in IntelliJ ein PopUp-Fenster, das sich auf die Information zum angefragten Begriff beschränkt, z. B.:



IntelliJ kann die Informationen über den **Documentation Path** in der SDK-Konfiguration beschaffen (siehe Abschnitt 2.4.6.2) oder die per Voreinstellung als JDK-Bestandteil installierte Datei **src.zip** mit dem Quellcode der Standardbibliothek auswerten. Bei der im Abschnitt 1.2.1 beschriebenen OpenJDK 8 - Installation landet die Datei **src.zip** im Installationsordner. Bei der im Abschnitt 2.1.1 beschriebenen OpenJDK 13 - Installation landet sie im Unterordner **lib**.

Für die Nutzung der externen Dokumentation spricht, dass auch Kontextinformationen geliefert werden.

Während vielleicht noch einige Zeit vergeht, bis Sie den ersten Dokumentationskommentar zu einer eigenen Klasse schreiben, sind die in diesem Abschnitt vermittelten Techniken zum Zugriff auf die Dokumentation der Bibliotheksklassen im Alltag der Software-Entwicklung unverzichtbar.

3.1.6 Namen

Für Klassen, Methoden, Felder, Parameter und sonstige Elemente eines Java-Programms benötigen wir Namen, wobei folgende Regeln zu beachten sind:

- Die Länge eines Namens ist *nicht* begrenzt.
  Zwar fördern kurze Namen die Übersicht im Quellcode, doch ist die Verstehbarkeit eines Namens noch wichtiger als die Kürze.
- Das erste Zeichen muss ein Buchstabe, Unterstrich oder Dollar-Zeichen sein, danach dürfen außerdem auch Ziffern auftreten.
- Damit sind insbesondere das Leerzeichen sowie Zeichen mit spezieller syntaktischer Bedeutung (z. B. -, (, *) als Namensbestandteile verboten.
- Java-Programme werden intern im **Unicode**-Zeichensatz dargestellt. Daher erlaubt Java in Namen auch Umlaute oder sonstige nationale Sonderzeichen, die als Buchstaben gelten.
- Die **Groß-/Kleinschreibung** ist signifikant. Für den Java-Compiler sind also z. B.
  `Anz    anz    ANZ`
  grundverschiedene Namen.
- Die folgenden **reservierten Wörter** dürfen nicht als Namen verwendet werden:

abstract	assert	boolean	break	byte	case	catch
char	class	const	continue	default	do	double
else	enum	extends	false	final	finally	float
for	goto	if	implements	import	instanceof	int
interface	long	native	new	null	package	private
protected	public	return	short	static	strictfp	super
switch	synchronized	this	throw	throws	transient	true
try	void	volatile	while			

Die Schlüsselwörter **const** und **goto** sind reserviert, werden aber derzeit nicht verwendet.

- Seit Java 9 ist ein isolierter Unterstrich ("_") nicht mehr als Name erlaubt.
- Namen müssen innerhalb ihres Kontexts (siehe unten) eindeutig sein.

3.1.7 Vollständige Klassennamen und Import-Deklaration

Jede Java-Klasse gehört zu einem **Paket** (siehe Kapitel 6), und dem Namen der Klasse ist grundsätzlich der Paketname voranzustellen. Dies gilt natürlich auch für die API-Klassen, also z. B. für die im folgenden Beispielprogramm verwendete Klasse **Random** aus dem Paket **java.util**.[1] Objekte dieser Klasse beherrschen u.a. die Methode **nextInt()**, die eine Pseudozufallszahl mit dem Datentyp **int** liefert, z. B.:

Quellcode	Ausgabe

`class Prog {`
`   public static void main(String[] args) {`
`      java.util.Random zuf = new java.util.Random();`
`      System.out.println(zuf.nextInt());`
`   }`
`}`
	`1053985008`

Keine Mühe mit Paketnamen hat man bei …

- den Klassen des sogenannten **Standardpakets**
  Zu diesem Paket gehören alle Klassen, die keinem Paket (per **package**-Deklaration, siehe unten) zugeordnet wurden. Das Standardpaket hat keinen Namen.
- den Klassen aus dem API-Paket **java.lang** (z. B. **Math**)
  Die Klassen dieses Pakets werden automatisch in jede Quellcodedatei importiert (siehe unten).

Um in einer Quellcodedatei bei Klassen aus anderen (API-)Paketen die lästige Angabe von Paketnamen zu vermeiden, kann man einzelne Klassen und/oder komplette Pakete *importieren*. Die zuständigen **import**-Deklarationen sind an den Anfang der Quellcodedatei zu setzen, z. B. zum Importieren der Klasse **java.util.Random**:

[1] Ab Java 9 befindet sich das Paket **java.util** im Modul **java.base**. Das gilt bis auf wenige Ausnahmen für alle im Manuskript verwendeten Pakete, sodass der Hinweis auf die Modulzugehörigkeit bald nur noch in den Ausnahmefällen erscheint.

Quellcode	Ausgabe

`import java.util.Random;`
`class Prog {`
`   public static void main(String[] args) {`
`      Random zuf = new Random();`
`      System.out.println(zuf.nextInt());`
`   }`
`}`
	1053985008

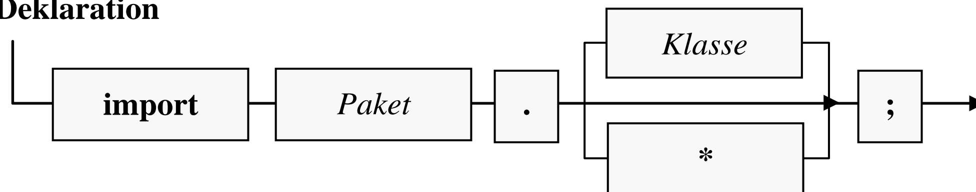
Um *alle* Klassen eines Pakets zu importieren, gibt man einen Stern an Stelle des Klassennamens an, z. B.:

```
import java.util.*;
```

Unterpakete (siehe Kapitel 6) sind dabei *nicht* einbezogen.

Zur Erläuterung der Import-Deklaration hätten die beiden Beispiele eigentlich genügt, und das folgende Syntaxdiagramm ist ziemlich überflüssig:

Import-Deklaration



In vergleichbaren Fällen werden wir zukünftig auf ein Syntaxdiagramm verzichten.

Mit der Anzahl importierter Bezeichner steigt das Risiko für eine Namenskollision. Die Entwicklungsumgebung und der Compiler meckern, ...

- wenn zwei namensgleiche Klassen aus verschiedenen Paketen *explizit* importiert werden,
- wenn eine in der Quellcodedatei definierte und eine explizit importierte Klasse denselben Namen besitzen.

Wenn aufgrund der Platzhaltersyntax aus mehreren Paketen namensgleiche Klassen importiert worden sind, dann muss bei der Verwendung einer Klasse durch Voranstellen des Paketnamens die Eindeutigkeit hergestellt werden.

Das Importieren von Klassen bzw. kompletten Paketen kann nur dann zum gewünschten Ergebnis führen, wenn die zugehörigen Bytecode-Dateien von der Entwicklungsumgebung bzw. vom Compiler (beim Übersetzen) und von der JVM (bei der Ausführung des Programms) gefunden werden. Bei den Klassen aus dem Java-API ist dies garantiert. Damit das Importieren anderer Klassen klappt, müssen die Entwicklungsumgebung bzw. der Compiler und die Runtime darüber informiert werden, an welchen Orten gesucht werden soll (siehe Abschnitte 2.2.4 bzw. 3.4.2).

3.2 Ausgabe bei Konsolenanwendungen

In diesem Abschnitt beschäftigen wir uns mit der Ausgabe von Zeichen in einem Konsolenfenster. Eine einfache Möglichkeit zur Konsolen*ein*gabe wird im Abschnitt 3.4 vorgestellt.

3.2.1 Ausgabe einer (zusammengesetzten) Zeichenfolge

Um eine einfache Konsolenausgabe in Java zu bewerkstelligen, bittet man das Objekt **System.out** (aus der Klasse **PrintStream**) seine **print()** - oder seine **println()** - Methode auszuführen.[1] Im Unterschied zu **print()** schließt **println()** die Ausgabe mit einem Zeilenwechsel ab, sodass die nächsten Aus- oder Eingabe in einer neuen Zeile erfolgt. Folglich ist **print()** zu bevorzugen, …

- wenn eine Benutzereingabe unmittelbar hinter einer Ausgabe in derselben Zeile ermöglicht werden soll,
- wenn die von mehreren Methodenaufrufen veranlassten Ausgaben in *einer* Zeile erscheinen sollen.

Beide Methoden erwarten ein einziges Argument, wobei erlaubt sind:

- eine Zeichenfolge, in doppelte Anführungszeichen eingeschlossen
  Beispiel: `System.out.print("Hallo allerseits!");`
- ein sonstiger Ausdruck (siehe Abschnitt 3.5)
  Dessen Wert wird automatisch in eine Zeichenfolge gewandelt.
  Beispiele:
  - `System.out.println(ivar);`
    Hier wird der Wert der Variablen `ivar` ausgegeben.
  - `System.out.println(i==13);`
    An die Möglichkeit, als **print()** - bzw. **println()** - Parameter, nahezu beliebige Ausdrücke anzugeben, müssen sich Einsteiger erst gewöhnen. Hier wird der Wert eines *Vergleichs* (der Variablen `i` mit der Zahl 13) ausgegeben. Bei Identität erscheint auf der Konsole das Wort **true**, ansonsten **false**.

Besonders angenehm ist die Möglichkeit, mehrere Teilausgaben mit dem Plusoperator zu verketten, z. B.:

```
System.out.println("Ergebnis: " + netto*MWST);
```

Im Beispiel wird der numerische Wert von `netto*MWST` (Produkt aus zwei Variablen) in eine Zeichenfolge gewandelt und dann mit `"Ergebnis: "` verknüpft.

3.2.2 Formatierte Ausgabe

Gelegentlich sind bei einer Konsolenausgabe die Gestaltungsmöglichkeiten der **PrintStream**-Methoden **print()** und **println()** unzureichend, weil sich z. B. die Anzahl der bei einer Zahl ausgegebenen Dezimalstellen nicht beeinflussen lässt. Dann bietet sich die **PrintStream**-Methode **printf()** an, die eine *formatierte* Ausgabe von *mehreren* Ausdrücken erlaubt.[2] Weil **System.out** ein Objekt der Klasse **PrintStream** ist, beherrscht es auch die Methode **printf()**, z. B.:

[1] Für eine genauere Erläuterung reichen unsere bisherigen OOP-Kenntnisse noch nicht ganz aus. Wer aus anderen Quellen Vorkenntnisse besitzt, kann die folgenden Sätze vielleicht jetzt schon verdauen: Wir benutzen bei der Konsolenausgabe die im Paket **java.lang** definierte und damit automatisch in jedem Java-Programm verfügbare Klasse **System**. Unter den Membern dieser Klasse befindet sich das statische (klassenbezogene) Feld **out**, das als Referenzvariable auf ein Objekt aus der Klasse **PrintStream** zeigt. Dieses Objekt beherrscht u.a. die Methoden **print()** und **println()**, die jeweils ein einziges Argument von beliebigem Datentyp erwarten und zur Standardausgabe befördern.

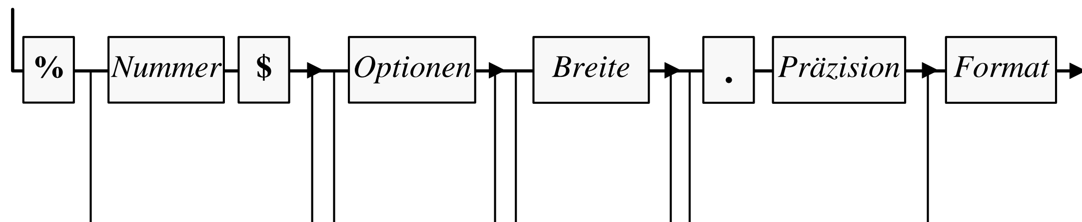
[2] Alternativ kann die äquivalente **PrintStream**-Methode **format()** benutzt werden.

Quellcode	Ausgabe

`class Prog {`
`  public static void main(String[] args) {`
`    System.out.printf("Pi = %9.3f%n", Math.PI);`
`    System.out.printf("Pi = %9.7f, e = %9.7f",`
`                       Math.PI, Math.E);`
`  }`
`}`
	`Pi =     3,142`
	`Pi = 3,1415927, e = 2,7182818`

Als erster Parameter wird an **printf()** eine Zeichenfolge übergeben, die Formatierungsangaben für die restlichen Parameter enthält. Für die Formatierungsangabe zu einem Ausgabeparameter ist die folgende Syntax zu verwenden, wobei Leerzeichen zwischen ihren Bestandteilen verboten sind:

Platzhalter für die formatierte Ausgabe



Darin bedeuten:

Nummer	Nummer des auszugebenden Arguments (mit 1 beginnend) Die Angabe einer Nummer ist z. B. dann von Nutzen, wenn ein Argument *mehrfach* ausgegeben werden soll.
Optionen	Formatierungsoptionen, u.a. sind erlaubt: - bewirkt eine linksbündige Ausgabe , ist nur für Zahlen erlaubt und bewirkt eine Zifferngruppierung (z. B. Ausgabe von 12.123,33 statt 12123,33)
Breite	Ausgabebreite für das zugehörige Argument
Präzision	Anzahl der Nachkommastellen oder sonstige Präzisionsangabe (abhängig vom Format)
Format	Formatspezifikation gemäß anschließender Tabelle

Es werden u.a. folgende Formate unterstützt:

Format	Beschreibung	Beispiele	
		printf() - Parameterliste	**Ausgabe**

d	ganze Zahl	`("%7d", 4711)`
		`("%-7d", 4711)`
		`("%1$d %1$,d", 4711)`
			`   4711`
			`4711`
			`4711 4.711`

f	Rationale Zahl mit fester Anzahl von Nachkommastellen Präzision: Anzahl der Nachkommastellen (Voreinstellung: 6)	`("%5.2f", 4.711)`	` 4,71`

e	Rationale Zahl in wissenschaftlicher Notation
	Präzision: Anzahl Stellen in der Mantisse (Voreinstellung: 6)
		`("%e", 47.11)`
		`("%.2e", 47.11)`
		`("%12.2e", 47.11)`
			`4,711000e+01`
			`4,71e+01`
			`    4.71e+01`

c	ein einzelnes Zeichen	`// x ist eine char-`
		`// Variable`
		`("Inhalt von x: %c", x)`
			`Inhalt von x: h`

Format	Beschreibung	Beispiele	
		printf() - Parameterliste	**Ausgabe**
n	plattformspezifische Zeilentrennung[1]	`("Inhalt von x:%n%c",x)`	`Inhalt von x:` `h`

s	eine Zeichenfolge	`// str ist eine String-`
		`// Variable`
		`("Text: %-7s", str)`
			`Text:     abc`

Wie **print()** produziert auch **printf()** *keinen* automatischen Zeilenwechsel nach der Ausgabe. Im obigen Beispielprogramm wird daher in der Formatierungszeichenfolge des ersten **printf()** - Aufrufs durch die Formatspezifikation **%n** für einen Zeilenwechsel gesorgt.

Im Unterschied zu **print()** und **println()** gibt **printf()** das landesübliche Dezimaltrennzeichen aus, z. B.:

Quellcode

```
class Prog {
   public static void main(String[] args) {
      System.out.println(Math.PI);
      System.out.printf("%-12.7f", Math.PI);
   }
}
```

Ausgabe

```
3.141592653589793
3,1415927
```

Eben wurde eine kleine Teilmenge der Syntax einer Java-Formatierungszeichenfolge vorgestellt. Die komplette Information findet sich in der API-Dokumentation zur Klasse **Formatter** (Paket **java.util**, ab Java 9 im Modul **java.base**).[2] Zur Online-Version dieser Dokumentation gelangen Sie z. B. auf dem folgenden Weg:

- Öffnen Sie z. B. die HTML-Startseite der API-Dokumentation zu Java 13 über die Adresse
  https://docs.oracle.com/en/java/javase/13/docs/api/index.html
- Tragen Sie den Klassennamen **Formatter** in das **SEARCH**-Feld ein (oben rechts), und wählen Sie aus der Trefferliste den Typ **java.util.Formatter**.

Noch bequemer klappt es mit Hilfe von IntelliJ z. B. so:

- Im Quellcodeeditor die Einfügemarke auf den Namen der Methode **printf()** setzen
- Tastenbefehl **Umschalt + F1**
- Im auftauchenden Browser-Fenster Klick auf den Link **Format string syntax**

---

[1] Im Zusammenhang mit den **char**-Literalen (siehe Abschnitt 3.3.11.4) werden wir die sogenannte Escape-Sequenz **\n** kennenlernen, die in ihrer Funktionalität der Formatspezifikation **%n** ähnelt. Durch **\n** wird auf allen Plattformen dasselbe Byte (Bedeutung: Line Feed) in den Ausgabestrom befördert. In Textdateien wird unter den Betriebssystemen Linux, Unix und MacOS X durch **\n** eine Zeilentrennung signalisiert; unter Windows wird dieser Zweck hingegen durch die Sequenz **\r\n** erreicht. Verwendet man statt **\n** die Formatspezifikation **%n**, landet auf jeder Plattform in einer Textausgabedatei die plattformspezifische Zeilentrennung.

[2] Mit den Modulen und Paketen der Standardklassenbibliothek werden wir uns später ausführlich beschäftigen. An dieser Stelle dient die Angabe der Modul- und Paketzugehörigkeit dazu, eine Klasse eindeutig zu identifizieren und die Standardbibliothek allmählich kennenzulernen.

3.3 Variablen und Datentypen

Während ein Programm läuft, müssen zahlreiche Daten im Arbeitsspeicher des Rechners abgelegt werden und anschließend mehr oder weniger lange für lesende und schreibende Zugriffe verfügbar sein, z. B.:

- Die Eigenschaftsausprägungen eines Objekts werden aufbewahrt, solange das Objekt existiert.
- Die zur Ausführung einer Methode benötigten Daten werden bis zum Ende der Methodenausführung gespeichert.

Zum Speichern eines Werts (z. B. einer ganzen Zahl) wird eine sogenannte **Variable** verwendet, worunter Sie sich einen **benannten Speicherplatz für einen Wert mit einem bestimmten Datentyp** (z. B. Ganzzahl) vorstellen können.

Eine Variable erlaubt (bei bestehender Zugriffsberechtigung) über ihren Namen den lesenden und/oder schreibenden Zugriff auf die zugehörige Stelle im Arbeitsspeicher, z. B.:

```
class Prog {
   public static void main(String[] args) {
      int ivar = 4711;            //schreibender Zugriff auf ivar
      System.out.println(ivar);   //lesender Zugriff auf ivar
   }
}
```

3.3.1 Strenge Compiler-Überwachung bei Java-Variablen

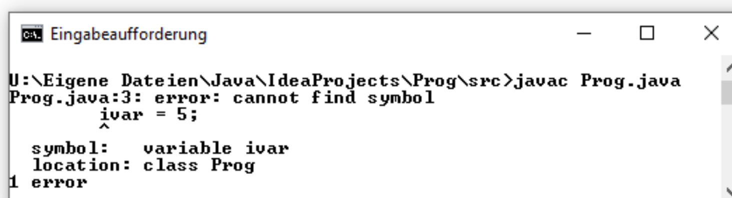
Um die Details bei der Verwaltung der Variablen im Arbeitsspeicher müssen wir uns nicht kümmern, da wir schließlich mit einer problemorientierten, „höheren" Programmiersprache arbeiten. Allerdings verlangt Java beim Umgang mit Variablen im Vergleich zu anderen Programmier- oder Skriptsprachen einige Sorgfalt, letztlich mit dem Ziel, Fehler zu vermeiden:

- Variablen müssen **explizit deklariert** werden, z. B.:

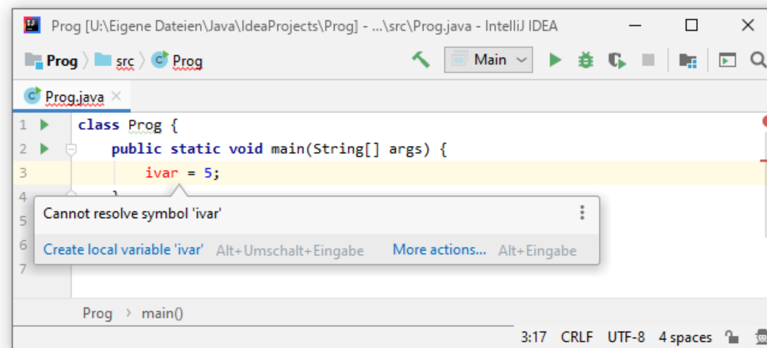
  ```
  int ivar = 5;
  ```

  Durch den Deklarationszwang werden z. B. Programmfehler wegen falsch geschriebener Variablennamen verhindert.

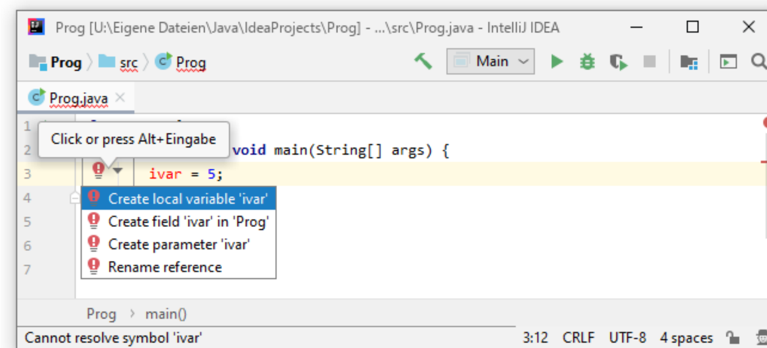
  Wenn Sie versuchen, eine nicht deklarierte Variable zu verwenden, wird beim Übersetzungsversuch ein Fehler gemeldet, z. B. vom Compiler **javac.exe** aus dem OpenJDK 8:

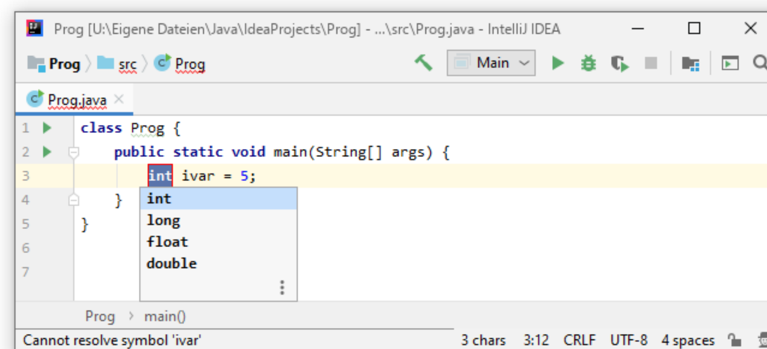
  Unsere Entwicklungsumgebung IntelliJ erkennt und dokumentiert das Problem unmittelbar nach der Eingabe im Editor. Wenn sich der Mauszeiger im Umfeld des Fehlers befindet, erscheint eine Erläuterung:



Nach einem Mausklick an derselben Stelle geht Ihnen ein rotes Licht auf. Durch einen Klick auf die rote Glühbirne oder mit der Tastenkombination **Alt + Enter** erhalten Sie in dieser Situation eine Liste mit Reparaturvorschlägen:



Nach der Wahl des ersten Vorschlags nimmt IntelliJ im Beispiel die fehlende Variablendeklaration vor und empfiehlt dabei einen Datentyp:



- Java ist **streng und statisch typisiert**.[1]
  Für jede Variable ist bei der Deklaration ein fester (später nicht mehr änderbarer) **Datentyp** anzugeben. Er legt fest, …
  - welche Informationen (z. B. ganze Zahlen, Zeichen, Adressen von `Bruch`-Objekten) in der Variablen gespeichert werden können,
  - welche Operationen auf die Variable angewendet werden dürfen.

  Der Compiler kennt zu jeder Variablen den Datentyp und kann daher **Typsicherheit** garantieren, d.h. die Zuweisung von Werten mit ungeeignetem Datentyp verhindern. Außerdem

[1] Halten Sie bitte die eben erläuterte *statische Typisierung* (im Sinn von *unveränderlicher* Typfestlegung) in begrifflicher Distanz zu den bereits erwähnten *statischen Variablen* (im Sinn von *klassenbezogenen* Variablen). Das Wort *statisch* ist eingeführter Bestandteil bei beiden Begriffen, sodass es mir nicht sinnvoll erschien, eine andere Bezeichnung vorzunehmen, um die Doppelbedeutung zu vermeiden.

kann auf (zeitaufwändige) Typprüfungen *zur Laufzeit* verzichtet werden. In der folgenden Anweisung

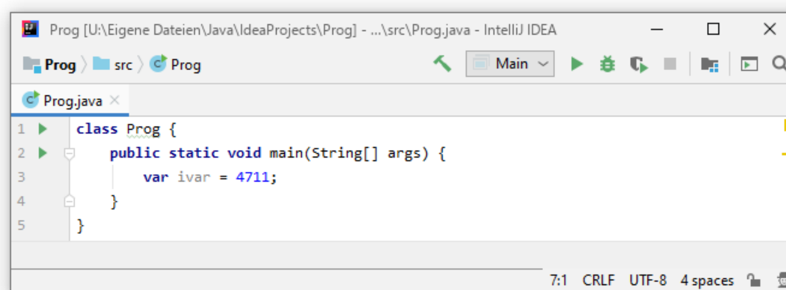
```
int ivar = 4711;
```

wird die Variable `ivar` vom Typ **int** deklariert, der sich für ganze Zahlen im Bereich von -2147483648 bis 2147483647 eignet.

Im Unterschied zu vielen Skriptsprachen arbeitet Java mit einer *statischen* Typisierung, sodass der einer Variablen zugewiesene Typ nicht mehr geändert werden kann.

In der obigen Anweisung erhält die Variable `ivar` beim Deklarieren gleich den **Initialisierungswert** 4711. Auf diese oder andere Weise müssen Sie jeder *lokalen*, d.h. innerhalb einer Methode deklarierten, Variablen einen Wert zuweisen, bevor Sie zum ersten Mal lesend darauf zugreifen (vgl. Abschnitt 3.3.8). Weil die zu einem Objekt oder zu einer Klasse gehörigen Variablen (siehe unten) automatisch initialisiert werden, hat in Java *jede* Variable beim Lesezugriff stets einen definierten Wert.

Seit der Version 10 können Java-Compiler den Typ von lokalen (in Methoden definierten) Variablen aus einem zugewiesenen Initialisierungswert erschließen (Typinferenz), und man darf in der Variablendeklaration die Typangabe durch das Schlüsselwort **var** ersetzen, z. B.:



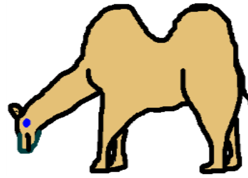
Im Beispiel gelingt die Inferenz, weil die zugewiesene Zahl 4711, die wir später als *Ganzzahlliteral* bezeichnen werden, den Typ **int** besitzt. Wie man bei gedrückter **Strg**-Taste für die in der Nähe des Mauszeigers befindliche Variable erfährt, kennt der Compiler (bzw. die Entwicklungsumgebung) den Datentyp:

Gelegentlich ermöglicht das Schlüsselwort **var** eine Arbeitserleichterung. Wenn dabei die Lesbarkeit des Quellcodes leidet, wird langfristig jedoch eher Zeit (anderer Programmierer) vergeudet.

3.3.2 Variablennamen

Es sind beliebige Bezeichner gemäß Abschnitt 3.1.6 erlaubt. Eine Beachtung der folgenden Konventionen verbessert aber die Lesbarkeit des Quellcodes, insbesondere auch für andere Programmierer (vgl. Bloch 2018, S. 289ff; Gosling et al. 2019, Abschnitt 6.1):

- Variablennamen beginnen mit einem Kleinbuchstaben.
- Besteht ein Name aus mehreren Wörtern (z. B. `numberOfObjects`), schreibt man ab dem zweiten Wort die Anfangsbuchstaben groß (*Camel Casing*). Das zur Vermeidung von Urheberrechtsproblemen handgemalte Tier kann hoffentlich trotz ästhetischer Mängel zur Begriffsklärung beitragen:



- Variablennamen mit einem *einzigen* Buchstaben sollten nur in speziellen Fällen verwendet werden (z. B. als Indexvariable von Wiederholungsanweisungen, siehe unten).

3.3.3 Primitive Typen und Referenztypen

In der objektorientierten Programmierung werden neben den traditionellen (elementaren, primitiven) Variablen zur Aufbewahrung von Zahlen, Zeichen oder Wahrheitswerten auch Variablen benötigt, welche die Adresse eines Objekts aufnehmen und so die Kommunikation mit dem Objekt ermöglichen. Wir unterscheiden also in Java bei den Datentypen von Variablen zwei übergeordnete Kategorien:

- **Primitive Datentypen**
  Die Variablen mit primitivem Datentyp sind auch in Java unverzichtbar (z. B. als Felder von Klassen oder als lokale Variablen in Methoden), obwohl sie „nur" zur Verwaltung ihres Inhalts dienen und keine Rolle bei Kommunikation mit Objekten spielen.

  In der `Bruch`-Klassendefinition (siehe Abschnitt 1.1.2) haben die Felder für Zähler und Nenner eines Objekts den primitiven Typ **int**, können also eine Ganzzahl im Bereich von -2147483648 bis 2147483647 (-2^{31} bis 2^{31} - 1) aufnehmen. Diese Felder werden in den folgenden Anweisungen deklariert, wobei das Feld `nenner` auch noch einen expliziten Initialisierungswert erhält:[1]

```
private int zaehler;
private int nenner = 1;
```

  Beim Feld `zaehler` wird auf die explizite Initialisierung verzichtet, sodass die automatische Null-Initialisierung von Feldern greift. Für ein frisch erzeugtes `Bruch`-Objekt befinden sich also im Arbeitsspeicher die folgenden Instanzvariablen (Felder):

zaehler	nenner
0	1

  Zur Realisation der Datenkapselung erhalten die beiden Felder den Zugriffsmodifikator **private**.

  In der `Bruch`-Methode `kuerze()` tritt u.a. die lokale Variable `ggt` auf, die ebenfalls den primitiven Typ **int** besitzt:

```
int ggt = 0;
```

  Im Abschnitt 3.3.6 werden zahlreiche weitere primitive Datentypen vorgestellt.

---

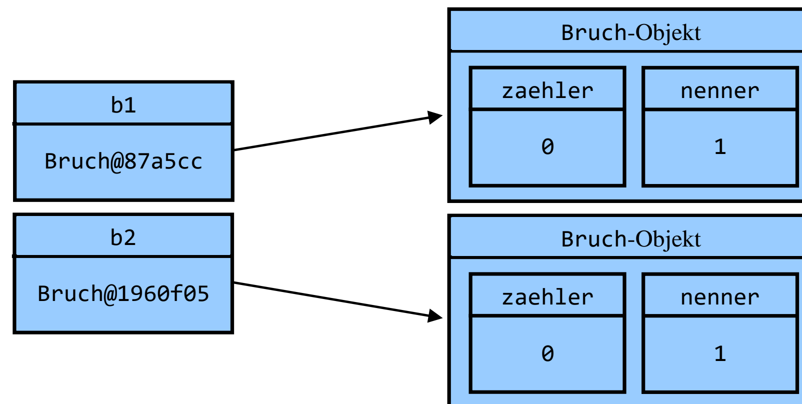
[1] Um die bei objektorientierter Programmierung oft empfehlenswerte Datenkapselung zu realisieren, also die Felder vor dem direkten Zugriff durch fremde Klassen zu schützen, wird der Modifikator **private** gesetzt. Bei den lokalen Variablen einer Methode ist dies weder erforderlich, noch möglich.

- **Referenztypen**
  Besitzt eine Variable einen Referenztyp, dann kann ihr Speicherplatz die **Adresse eines Objekts** aus einer bestimmten Klasse aufnehmen. Sobald ein solches Objekt erzeugt und seine Adresse der Referenzvariablen zugewiesen worden ist, kann das Objekt über die Referenzvariable angesprochen werden. Von den Variablen mit primitivem Typ unterscheidet sich eine Referenzvariable also …
  - durch ihren speziellen Inhalt (eine Objektadresse)
  - und durch ihre Rolle bei der Kommunikation mit Objekten.

  Man kann jede Klasse (aus der Java-Standardbibliothek oder selbst definiert) als Referenzdatentyp verwenden, also Referenzvariablen dieses Typs deklarieren. In der **main()** - Methode der Klasse `Bruchaddition` (siehe Abschnitt 1.1.4) werden z. B. die Referenzvariablen `b1` und `b2` vom Datentyp `Bruch` deklariert:

  ```
  Bruch b1 = new Bruch(), b2 = new Bruch();
  ```

  Sie erhalten als Initialisierungswert jeweils eine Referenz auf ein (per **new**-Operator, siehe unten) neu erzeugtes `Bruch`-Objekt. Daraus resultiert im Arbeitsspeicher die folgende Situation:



  Das von `b1` referenzierte `Bruch`-Objekt wurde bei einem konkreten Programmlauf von der JVM an der Speicheradresse 0x87a5cc (ganze Zahl, ausgedrückt im Hexadezimalsystem) untergebracht. Wir müssen diese Adresse nicht kennen, sondern sprechen das dort abgelegte Objekt über die Referenzvariable an, z. B. in der folgenden Anweisung aus der **main()** - Methode der Klasse `Bruchaddition`:

  ```
  b1.frage();
  ```

  Jedes `Bruch`-Objekt besitzt die Felder (Instanzvariablen) `zaehler` und `nenner` vom primitiven Typ **int**.

Zur Beziehung der Begriffe *Objekt* und *Variable* halten wir fest:

- Ein Objekt enthält im Allgemeinen mehrere Instanzvariablen (Felder) von beliebigem Datentyp. So enthält z. B. ein `Bruch`-Objekt die Felder `zaehler` und `nenner` vom primitiven Typ **int** (zur Aufnahme einer Ganzzahl). Bei einer späteren Erweiterung der `Bruch`-Klassendefinition werden ihre Objekte auch eine Instanzvariable mit Referenztyp erhalten.
- Eine Referenzvariable dient zur Aufnahme einer Objektadresse. So kann z. B. eine Variable vom Datentyp `Bruch` die Adresse eines `Bruch`-Objekts aufnehmen und zur Kommunikation mit diesem Objekt dienen. Es ist ohne weiteres möglich und oft sinnvoll, dass mehrere Referenzvariablen die Adresse *desselben* Objekts enthalten. Das Objekt existiert unabhängig vom Schicksal einer konkreten Referenzvariablen, wird jedoch überflüssig (und damit zum potentiellen Opfer des Garbage Collectors der JVM), wenn im gesamten Programm keine einzige Referenz (Kommunikationsmöglichkeit) mehr vorhanden ist.

Wir werden im Kapitel 3 überwiegend mit Variablen von primitivem Typ arbeiten, können und wollen dabei aber den Referenzvariablen (z. B. zur Ansprache des Objekts **System.out** aus der Klasse **PrintStream** bei der Konsolenausgabe, siehe Abschnitt 3.2) nicht aus dem Weg gehen.

3.3.4 Klassifikation der Variablen nach Zuordnung

In Java unterscheiden sich Variablen nicht nur hinsichtlich des Datentyps (Inhalts), sondern auch hinsichtlich der Zuordnung zu einer *Methode*, zu einem *Objekt* oder zu einer *Klasse*:

- **Lokale Variablen**
  Sie werden innerhalb einer Methode deklariert. Ihre Gültigkeit (Verwendbarkeit) beschränkt sich auf die Methode bzw. auf einen Anweisungsblock (siehe Abschnitt 3.3.9) innerhalb der Methode.

  Solange eine Methode ausgeführt wird, befinden sich ihre Variablen in einem Bereich des programmeigenen Arbeitsspeichers, den man als **Stack** (deutsch: *Stapel*) bezeichnet.
- **Instanzvariablen (nicht-statische Felder)**
  Instanzvariablen werden außerhalb jeder Methode deklariert. Jedes Objekt (synonym: jede *Instanz*) einer Klasse verfügt über einen vollständigen Satz der Instanzvariablen der Klasse. So besitzt z. B. jedes Objekt der Klasse `Bruch` einen `zaehler` und einen `nenner`.

  Solange ein Objekt existiert, befinden es sich mit all seinen Instanzvariablen in einem Bereich des programmeigenen Arbeitsspeichers, den man als **Heap** (deutsch: *Haufen*) bezeichnet.
- **Klassenvariablen (statische Felder)**
  Klassenvariablen werden außerhalb jeder Methode deklariert und erhalten dabei den Modifikator **static**. Diese Variablen beziehen sich auf eine Klasse insgesamt, nicht auf einzelne Instanzen der Klasse. Z. B. kann man in einer Klassenvariablen festhalten, wie viele Objekte der Klasse bereits bei einem Programmeinsatz erzeugt worden sind. In unserem Bruchrechnungsbeispiel haben wir der Einfachheit halber bisher auf statische Felder verzichtet. Allerdings sind uns schon statische Felder aus anderen Klassen begegnet:
  - Aus der Klasse **System** kennen wir die statische Variable **out**. Sie zeigt auf ein Objekt der Klasse **PrintStream**, das wir häufig mit Konsolenausgaben beauftragen.
  - In einem Beispielprogramm von Abschnitt 3.2.2 über die formatierte Ausgabe haben wir die Zahl π aus der statischen Variablen **PI** der Klasse **Math** gelesen.[1]

  Während jedes Objekt einer Klasse über einen eigenen Satz mit allen Instanzvariablen verfügt, die beim Erzeugen des Objekts auf dem Heap angelegt werden, existieren Klassenvariablen nur *einmal*. Sie werden beim Laden der Klasse in der sogenannten **Method Area** des Arbeitsspeichers abgelegt.

Die im Wesentlichen schon aus Abschnitt 3.3.3 bekannte Abbildung zur Lage im Arbeitsspeicher bei Ausführung der **main()** - Methode der Klasse `Bruchaddition` aus unserem OOP-Standardbeispiel (vgl. Abschnitt 1.1) wird anschließend ein wenig präzisiert. Durch Farben und Ortsangaben wird für die beteiligten lokalen Variablen bzw. Instanzvariablen die Zuordnung zu einer Methode bzw. zu einem Objekt und die damit verbundene Speicherablage verdeutlicht:

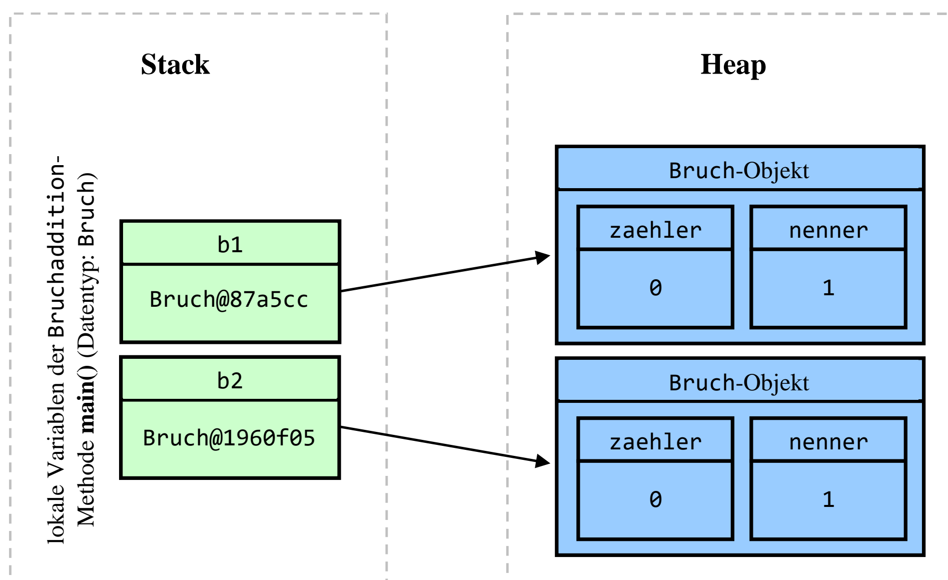
---

[1] Die statischen Felder **out** (aus der API-Klasse **System**) und **PI** (aus der API-Klasse **Math**) sind *finalisiert* (siehe Abschnitt 4.5.1), können also nicht geändert werden. Außerdem kommt keine Änderung der Repräsentation in Betracht. In dieser Situation ist eine Ausnahme vom Prinzip der Datenkapselung sinnvoll, um den Zugriff zu vereinfachen. Die Deklaration der Referenzvariablen **out** kennen Sie schon aus Abschnitt 3.1.5

```
public static final PrintStream out = null;
```

Hier ist die Deklaration der Variablen **PI** (mit dem primitiven Typ **double**) zu sehen:

```
public static final double PI = 3.14159265358979323846;
```



Die lokalen Referenzvariablen `b1` und `b2` der Methode **main()** befinden sich im Stack-Bereich des programmeigenen Arbeitsspeichers und enthalten jeweils die Adresse eines `Bruch`-Objekts. Jedes `Bruch`-Objekt besitzt die Felder (Instanzvariablen) `zaehler` und `nenner` vom primitiven Typ **int** und befindet sich im Heap-Bereich des programmeigenen Arbeitsspeichers.

Auf Instanz- und Klassenvariablen kann in allen Methoden der eigenen Klasse zugegriffen werden. Wenn (als gut begründete Ausnahme vom Prinzip der Datenkapselung) entsprechende Rechte eingeräumt wurden, ist dies auch in Methoden fremder Klassen möglich.

Im Kapitel 3 werden wir überwiegend mit *lokalen* Variablen arbeiten, aber z. B. auch das statische Feld **out** der Klasse **System** benutzen, das auf ein Objekt der Klasse **PrintStream** zeigt. Im Zusammenhang mit der systematischen Behandlung der objektorientierten Programmierung werden die Instanz- und Klassenvariablen ausführlich erläutert.

Im Unterschied zu anderen Programmiersprachen (z. B. C++) ist es in Java *nicht* möglich, sogenannte *globale* Variablen außerhalb von Klassen zu definieren.

3.3.5 Eigenschaften einer Variablen

Als Eigenschaften einer Java-Variablen haben Sie nun kennengelernt:

- **Zuordnung**
  Eine Variable gehört entweder zu einer Methode, zu einem Objekt oder zu einer Klasse. Daraus resultiert ihr **Ablageort im Arbeitsspeicher**. Als wichtige Speicherregionen, in denen die Variablen bzw. Objekte eines Java-Programms von der JVM abgelegt werden, unterscheiden wir Stack, Heap und Method Area. Dieses Hintergrundwissen hilft z. B., wenn ein **StackOverflowError** gemeldet wird.
- **Datentyp**
  Damit sind festgelegt: Zulässige Werte (hinsichtlich Typ und Größe), Speicherplatzbedarf und zulässige Operationen. Besonders wichtig ist die Unterscheidung zwischen primitiven Datentypen und Referenztypen.
- **Name**
  Es sind beliebige Bezeichner gemäß Abschnitt 3.1.6 erlaubt, wobei die Empfehlungen aus Abschnitt 3.3.2 beachtet werden sollten.

- **Aktueller Wert**
  Im folgenden Beispiel taucht eine lokale Variable namens `ivar` auf, die zur Methode **main**() gehört, vom primitiven Typ **int** ist und den Wert 5 besitzt:

```
public class Prog {
   public static void main(String[] args) {
      int ivar = 5;
   }
}
```

3.3.6 Primitive Datentypen in Java

Als *primitiv* bezeichnet man in Java die (auch in älteren Programmiersprachen bekannten) Datentypen zur Aufnahme von einzelnen Zahlen, Zeichen oder Wahrheitswerten. Speziell für Zahlen existieren diverse Datentypen, die sich hinsichtlich Speichertechnik, Wertebereich und Platzbedarf unterscheiden. Von der folgenden Tabelle sollte man sich vor allem merken, wo sie im Bedarfsfall zu finden ist. Eventuell sind Sie aber auch jetzt schon neugierig auf einige Details:

Typ	Beschreibung	Werte	Speicher-bedarf in Bit
byte	Diese Variablentypen speichern ganze Zahlen. Beispiel: `int alter = 31;`	-128 … 127	8
short		-32768 … 32767	16
int		-2147483648 ... 2147483647	32
long		-9223372036854775808 … 9223372036854775807	64
float	Variablen vom Typ **float** speichern Gleitkommazahlen nach der Norm IEEE-754 mit einer Genauigkeit von mindestens 7 Dezimalstellen in der Mantisse. Beispiel: `float pi = 3.141593f;` **float**-Literale (siehe Beispiel) benötigen das Suffix **f** (oder **F**).	Minimum: $-3{,}4028235 \cdot 10^{38}$ Maximum: $3{,}4028235 \cdot 10^{38}$ Kleinster positiver Betrag: $1.40129846 \cdot 10^{-45}$	32 1 für das Vorz., 8 für den Expon., 23 für die Mantisse
double	Variablen vom Typ **double** speichern Gleitkommazahlen nach der Norm IEEE-754 (64 Bit) mit einer Genauigkeit von mind. 15 signifikanten Dezimalstellen in der Mantisse. Beispiel: `double ph = 1.57079632679490;`	Minimum: $-1{,}7976931348623157 \cdot 10^{308}$ Maximum: $1{,}7976931348623157 \cdot 10^{308}$ Kleinster positiver Betrag: $4{,}9406564584124654 \cdot 10^{-324}$	64 1 für das Vorz., 11 für den Expon., 52 für die Mantisse

Typ	Beschreibung	Werte	Speicher-bedarf in Bit
char	Variablen vom Typ **char** dienen zum Speichern *eines* Unicode-Zeichens. Im Speicher landet aber nicht die Gestalt des Zeichens, sondern seine Nummer im Unicode-Zeichensatz. Daher zählt **char** zu den *integralen* (ganzzahligen) Datentypen. Beispiel: `char zeichen = 'j';` **char**-Literale sind durch *einfache* Anführungszeichen zu begrenzen (siehe Beispiel).	Unicode-Zeichen Tabellen mit allen Unicode-Zeichen sind z. B. auf der Webseite des Unicode-Konsortiums verfügbar: http://www.unicode.org/charts/	16
boolean	Variablen vom Typ **boolean** können Wahrheitswerte aufnehmen. Beispiel: `boolean cond = true;`	**true**, **false**	8

Eine Variable mit einem integralen Datentyp (z. B. **int** oder **byte**) speichert eine ganze Zahl (z. B. 4711) *exakt*, sofern es nicht durch eine Wertebereichsüberschreitung zu einem Überlauf und damit zu einem sinnlosen Speicherinhalt kommt (siehe Abschnitt 3.6).

Eine Variable zur Aufnahme einer *Gleitkommazahl* (synonym: *Gleitpunkt*- oder *Fließkommazahl*, englisch: *floating point number* dient zur *approximativen* Darstellung einer reellen Zahl. Dabei werden drei Bestandteile separat gespeichert: Vorzeichen, Mantisse und Exponent. Diese ergeben nach folgender Formel den dargestellten Wert, wobei ***b*** für die Basis eines Zahlensystems steht (meist verwendet: 2 oder 10):

$$\textbf{Wert} = \textbf{Vorzeichen} \cdot \textbf{Mantisse} \cdot b^{\textbf{Exponent}}$$

Bei dieser von Konrad Zuse entwickelten Darstellungstechnik[1] resultiert im Vergleich zur Festkommadarstellung bei gleichem Speicherplatzbedarf ein erheblich größerer Wertebereich. Während die Mantisse für die Genauigkeit sorgt, speichert der Exponentialfaktor die Größenordnung, z. B.:

$$-0{,}0000001252612 = (-1) \cdot 1{,}252612 \cdot 10^{-7}$$
$$1252612000000000 = (1) \cdot 1{,}252612 \cdot 10^{15}$$

Durch eine Änderung des Exponenten könnte man das Dezimalkomma durch die Mantisse „gleiten“ lassen. Allerdings wird in der Regel durch eine Restriktion der Mantisse (z. B. auf das Intervall [1; 2)) für eine eindeutige Darstellung gesorgt.

Weil der verfügbare Speicher für Mantisse und Exponent begrenzt ist (siehe obige Tabelle), bilden die Gleitkommazahlen nur eine endliche (aber für die meisten praktischen Zwecke ausreichende) Teilmenge der reellen Zahlen. Nähere Informationen über die Darstellung von Gleitkommazahlen im Arbeitsspeicher eines Computers folgen für speziell interessierte Leser gleich im Abschnitt 3.3.7.

Im Vergleich zu den Programmiersprachen C, C++ und C# fällt auf, dass in Java auf *vorzeichenfreie* Ganzzahltypen verzichtet wurde.

[1] Quelle: http://de.wikipedia.org/wiki/Konrad_Zuse

Die abwertend klingende Bezeichnung *primitiv* darf keinesfalls so verstanden werden, dass elementare Datentypen nach Möglichkeit in Java-Programmen zu vermeiden wären. Sie sind bei den Feldern von Klassen und bei den lokalen Variablen von Methoden unverzichtbar.

3.3.7 Vertiefung: Darstellung von Gleitkommazahlen im Arbeitsspeicher des Computers

Die als *Vertiefung* bezeichneten Abschnitte können beim ersten Lesen des Manuskripts gefahrlos übersprungen werden. Sie enthalten interessante Details, über die man sich irgendwann im Verlauf der Programmierkarriere informieren sollte.

3.3.7.1 Binäre Gleitkommadarstellung

Bei den binären Gleitkommatypen **float** und **double** werden auch „relativ glatte" Zahlen im Allgemeinen nur approximativ gespeichert, wie das folgende Programm zeigt:

Quellcode	Ausgabe

`class Prog {`
`   public static void main(String[] args) {`
`      float f130 = 1.3f;`
`      float f125 = 1.25f;`
`      System.out.printf("%9.7f", f130);`
`      System.out.println();`
`      System.out.printf("%10.8f", f130);`
`      System.out.println();`
`      System.out.printf("%20.18f", f125);`
`   }`
`}`
	`1,3000000`
	`1,29999995`
	`1,250000000000000000`

Bei einer Ausgabe mit mehr als sieben Nachkommastellen zeigt sich, dass die **float**-Zahl 1,3 nicht exakt abgespeichert worden ist. Demgegenüber tritt bei der **float**-Zahl 1,25 *keine* Ungenauigkeit auf.

Diese Ergebnisse sind durch das Speichern der Zahlen im **binären Gleitkommaformat** nach der vom ***I****nstitute of* ***E****lectrical and* ***E****lectronics* ***E****ngineers* (IEEE) veröffentlichten Norm **IEEE-754** zu erklären, wobei jede Zahl als Produkt aus drei getrennt zu speichernden Faktoren dargestellt wird:[1]

$$\textbf{Vorzeichen} \cdot \textbf{Mantisse} \cdot \textbf{2}^{\textbf{Exponent}}$$

Im ersten Bit einer **float**- oder **double** - Variablen wird das Vorzeichen gespeichert (0: positiv, 1: negativ).

Für die Ablage des Exponenten (zur Basis 2) als Ganzzahl stehen 8 (**float**) bzw. 11 (**double**) Bits zur Verfügung, die jeweils die Werte 0 oder 1 repräsentieren. Das i-te Exponenten-Bit (von *rechts* nach *links* mit 0 beginnend nummeriert) hat die Wertigkeit 2^i, sodass ein Wertebereich von 0 bis 255 (= 2^8-1) bzw. von 0 bis 2047 (= 2^{11}-1) resultiert:

$$\sum_{i=0}^{7 \text{ bzw. } 10} b_i 2^i, \;\; b_i \in \{0, 1\}$$

Allerdings sind im Exponenten die Werte 0 und 255 (**float**) bzw. 0 und 2047 (**double**) für Spezialfälle (z. B. denormalisierte Darstellung, +/- Unendlich) reserviert (siehe unten). Um auch die für Zahlen mit einem Betrag kleiner 1 benötigten *negativen* Exponenten darstellen zu können, werden die Exponenten mit einer Verschiebung (*Bias*) um den Wert 127 (**float**) bzw. 1023 (**double**) abgespeichert und interpretiert. Bei einer **float**-Variablen wird z. B. für den Exponenten 0 der Wert 127 und für den Exponenten -2 der Wert 125 im Speicher abgelegt.

---

[1] https://de.wikipedia.org/wiki/IEEE_754

Abgesehen von betragsmäßig sehr kleinen Zahlen (siehe unten) werden die **float**- und **double**-Werte **normalisiert**, d.h. auf eine Mantisse im Intervall [1; 2) gebracht, z. B.:

$$24{,}48 = 1{,}53 \cdot 2^{4}$$

$$0{,}2448 = 1{,}9584 \cdot 2^{-3}$$

Zur Speicherung der Mantisse werden 23 (**float**) bzw. 52 (**double**) Bits verwendet. Das *i*-te Mantissen-Bit (von links nach rechts mit 1 beginnend nummeriert) hat die Wertigkeit 2^{-i}, sodass sich der *dezimale* Mantissenwert folgendermaßen ergibt:

$$1+m \quad \text{mit} \quad m = \sum_{i=1}^{23 \text{ bzw. } 52} b_i 2^{-i},\ \ b_i \in \{0,1\}$$

Der Summenindex *i* startet mit 1, weil die führende 1 (= 2^0) der normalisierten Mantisse *nicht* abgespeichert wird (*hidden bit*). Daher stehen alle Bits für die Restmantisse (die Nachkommastellen) zur Verfügung mit dem Effekt einer verbesserten Genauigkeit. Oft wird daher die Anzahl der Mantissen-Bits mit 24 (**float**) bzw. 53 (**double**) angegeben.

Eine **float**- bzw. **double**-Variable mit den Speicherinhalten

- *v* (0 oder 1) für das Vorzeichen
- *e* für den Exponenten
- *m* für die Mantisse

repräsentiert also bei normalisierter Darstellung den Wert:

$$\mathbf{(-1)^{\mathit{v}} \cdot (1 + \mathit{m}) \cdot 2^{\mathit{e}-127}} \quad \text{bzw.} \quad \mathbf{(-1)^{\mathit{v}} \cdot (1 + \mathit{m}) \cdot 2^{\mathit{e}-1023}}$$

In der folgenden Tabelle finden Sie einige normalisierte **float**-Werte:

Wert	float-Darstellung (normalisiert)		
	Vorz.	Exponent	Mantisse
0,75 = $(-1)^0 \cdot 2^{(126-127)} \cdot (1+0{,}5)$	0	01111110	10000000000000000000000
1,0 = $(-1)^0 \cdot 2^{(127-127)} \cdot (1+0{,}0)$	0	01111111	00000000000000000000000
1,25 = $(-1)^0 \cdot 2^{(127-127)} \cdot (1+0{,}25)$	0	01111111	01000000000000000000000
-2,0 = $(-1)^1 \cdot 2^{(128-127)} \cdot (1+0{,}0)$	1	10000000	00000000000000000000000
2,75 = $(-1)^0 \cdot 2^{(128-127)} \cdot (1+0{,}25+0{,}125)$	0	10000000	01100000000000000000000
-3,5 = $(-1)^1 \cdot 2^{(128-127)} \cdot (1+0{,}5+0{,}25)$	1	10000000	11000000000000000000000

Nun kommen wir endlich zur Erklärung der eingangs dargestellten Genauigkeitsunterschiede beim Speichern der Zahlen 1,25 und 1,3. Während die Restmantisse

$$0{,}25 = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4}$$

perfekt dargestellt werden kann, gelingt dies bei der Restmantisse 0,3 nur approximativ:

$$0{,}3 = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + \ldots = 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 0 \cdot \frac{1}{16} + 1 \cdot \frac{1}{32} + \ldots$$

Sehr aufmerksame Leser werden sich darüber wundern, wieso die Tabelle mit den elementaren Datentypen im Abschnitt 3.3.6 z. B.

$$1{,}40129846 \cdot 10^{-45}$$

als betragsmäßig kleinsten positiven **float**-Wert nennt, obwohl der minimale Exponent nach obigen Überlegungen -126 (= 1 - 127) beträgt, was zum (gerundeten) dezimalen Exponentialfaktor

$$1{,}175 \cdot 10^{-38}$$

führt. Dahinter steckt die *denormalisierte* (synonym: *subnormale*) Gleitkommadarstellung, die als Ergänzung zur bisher beschriebenen normalisierten Darstellung eingeführt wurde, um eine bessere Annäherung an die Zahl 0 zu erreichen. Alle Exponenten-Bits sind auf 0 gesetzt, und dem Exponentialfaktor wird der feste Wert 2^{-126} (**float**) bzw. 2^{-1022} (**double**) zugeordnet. Die Mantissen-Bits haben dieselbe Wertigkeiten (2^{-i}) wie bei der normalisierten Darstellung (siehe oben). Weil es *kein hidden bit* gibt, stellen sie aber nun einen dezimalen Wert im Intervall [0, 1) dar. Eine **float**- bzw. **double**-Variable mit dem Vorzeichen v (0 oder 1), mit komplett auf 0 gesetzten Exponenten-Bits und dem gespeicherten Mantissenwert m repräsentiert also bei denormalisierter Darstellung die Zahl:

$$(-1)^v \cdot 2^{-126} \cdot m \quad \text{bzw.} \quad (-1)^v \cdot 2^{-1022} \cdot m$$

In der folgenden Tabelle finden Sie einige denormalisierte **float**-Werte:

Wert			float-Darstellung (denormalisiert)		
			Vorz.	Exponent	Mantisse
0,0	=	$(-1)^0 \cdot 2^{-126} \cdot 0$	0	00000000	00000000000000000000000
$-5{,}877472 \cdot 10^{-39}$	≈	$(-1)^1 \cdot 2^{-126} \cdot 2^{-1}$	1	00000000	10000000000000000000000
$1{,}401298 \cdot 10^{-45}$	≈	$(-1)^0 \cdot 2^{-126} \cdot 2^{-23}$	0	00000000	00000000000000000000001

Weil die Mantissen-Bits auch zur Darstellung der Größenordnung verwendet werden, schwindet die Genauigkeit mit der Annäherung an die Null.[1]

IntelliJ-Projekte mit Java-Programmen zur Anzeige der Bits einer (de)normalisierten **float**- bzw. **double**-Zahl finden Sie in den Ordnern

…\BspUeb\Elementare Sprachelemente\Bits\FloatBits
…\BspUeb\Elementare Sprachelemente\Bits\DoubleBits

Weil im Quellcode der Programme mehrere noch unbekannte Sprachelemente auftreten, wird hier auf eine Wiedergabe verzichtet. Einer Nutzung der Programme steht aber nichts im Wege. Hier wird z. B. mit dem Programm **FloatBits** das Speicherabbild der **float**-Zahl -3,5 ermittelt (vgl. obige Tabelle):

```
float: -3,5

Bits:
1 76543210 12345678901234567890123
1 10000000 11000000000000000000000
```

Zur Verarbeitung von binären Gleitkommazahlen wurde die *binäre Gleitkommaarithmetik* entwickelt, normiert und zur Verbesserung der Verarbeitungsgeschwindigkeit sogar teilweise in Computer-Hardware realisiert.

3.3.7.2 Dezimale Gleitkommadarstellung

Wenn die Speicher- und Rechengenauigkeit der binären Gleitkommatypen für eine Anwendung nicht reicht, kommt in Java die Klasse **BigDecimal** aus dem Paket **java.math** in Frage. Objekte dieser Klasse können Dezimalzahlen mit beliebiger Genauigkeit speichern und verwenden eine dezimale Gleitkommaarithmetik mit einstellbarer Rechengenauigkeit.

Gespeichert werden:

---

[1] Bei einer formatierten Ausgaben in wissenschaftlicher Notation (vgl. Abschnitt 3.2.2) liegt die Anzahl der signifikanten Dezimalstellen in der Mantisse deutlich unter 7.

- Eine Ganzzahl beliebiger Größe für den unskalierten Wert (*uv*)
- Eine Ganzzahl mit 32 Bit für die Anzahl der Nachkommastellen (*scale*)

Bei der Zahl

$$1{,}3 = 13 \cdot 10^{-1}$$

gelingt eine verlustfreie Speicherung mit:

$$uv = 13,\ scale = 1$$

Die Ausgabe des folgenden Programms

```
import java.math.*;
class Prog {
   public static void main(String[] args) {
      BigDecimal bdd = new BigDecimal(1.3);
      System.out.println(bdd);
      BigDecimal bds = new BigDecimal("1.3");
      System.out.println(bds);
   }
}
```

belegt zunächst als Nachtrag zum Abschnitt 3.3.7.1, dass auch eine **double**-Variable den Wert 1,3 nur approximativ speichern kann:

```
1.3000000000000000444089209850062616169452667236328125
1.3
```

Zwar zeigt die Variable `bdd` auf ein Objekt vom Typ **BigDecimal**, doch wird zur Erstellung dieses Objekts ein **double**-Wert verwendet, der im Speicher nicht exakt abgelegt werden kann.

Erfolgt die Kreation des **BigDecimal**-Objekts über eine Zeichenfolge, kann die Zahl 1,3 exakt gespeichert werden, wie die zweite Ausgabezeile belegt.

Allerdings hat der Typ **BigDecimal** auch Nachteile im Vergleich zu den binären Gleitkommatypen **float** und **double**:

- Höherer Speicherbedarf
- Höherer Zeitaufwand bei arithmetischen Operationen
- Aufwändigere Syntax

Bei der Aufgabe,

$$1700000000 \ - \ \sum_{i=1}^{1000000000} 1{,}7$$

zu berechnen, ergeben sich für die Datentypen **double** und **BigDecimal** folgende Genauigkeits- und Laufzeitunterschiede (gemessen auf einem PC mit der Intel-CPU Core i3 mit 3,2 GHz):[1]

---

[1] Ein IntelliJ-Projekt mit dem Java-Programm, das die Berechnungen angestellt hat, ist hier zu finden:
...\BspUeb\Elementare Sprachelemente\BigDecimalDouble

```
double:
 Abweichung:             -29.96745276451111
 Zeit in Millisekunden: 1121

BigDecimal:
 Abweichung:             0.0
 Zeit in Millisekunden: 10152
```

Die gut bezahlten Verantwortlichen vieler Banken, die sich gerne als „Global Player" betätigen und dabei den vollen Sinn der beiden Worte ausschöpfen (mit Niederlassungen in Schanghai, New York, Mumbai etc. und einem Verhalten wie im Spielcasino) wären heilfroh, wenn nach einem Spiel mit 1,7 Milliarden Euro Einsatz nur 30 Euro in der Kasse fehlen würden. Generell sind im Finanzsektor solche Fehlbeträge aber unerwünscht, sodass man bei finanzmathematischen Aufgaben trotz des erhöhten Zeitaufwands (im Beispiel: Faktor nahe 10) die Klasse **BigDecimal** verwenden sollte.

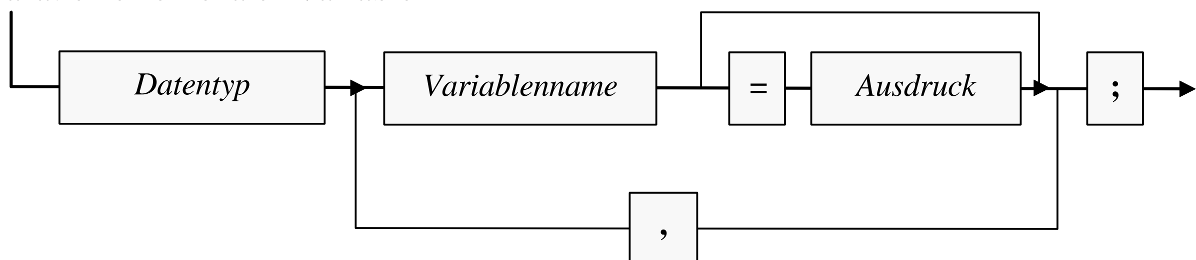
Für manche monetäre Berechnungen taugen auch die Ganzzahltypen **int** oder **long**, die bei Addition und Subtraktion mit (z. B. mit Cent-Werten) bei geringem Zeitaufwand perfekte Genauigkeit bieten, sofern ihr Wertebereich nicht verlassen wird.

3.3.8 Variablendeklaration, Initialisierung und Wertzuweisung

In einem Java-Programm muss jede Variable vor ihrer ersten Verwendung deklariert werden, wobei auf jeden Fall ein Datentyp und ein Name anzugeben sind. Wir betrachten vorläufig nur *lokale* Variablen, die innerhalb einer Methode existieren. Ihre Deklaration darf im Methodenquellcode an beliebiger Stelle *vor* der ersten Verwendung erscheinen. Um den (im Abschnitt 3.3.9 behandelten) Gültigkeitsbereich einer lokalen Variablen zur Vermeidung von Fehlern zu minimieren, sollte sie unmittelbar vor der ersten Verwendung deklariert werden (Bloch 2018, S. 261).

Es folgt das Syntaxdiagramm zur Deklaration einer lokalen Variablen, wobei zunächst der Übersichtlichkeit halber die mit Java 10 eingeführte Typinferenz (siehe unten) ignoriert wird:

Deklaration einer lokalen Variablen



Als Datentypen kommen in Frage (vgl. Abschnitt 3.3.3):

- Primitive Datentypen, z. B.
  ```
  int wasser;
  ```
- Referenztypen, also Klassen (aus dem Java-API oder selbst definiert), z. B.
  ```
  Bruch b1;
  ```

Neu deklarierte lokale Variablen kann man optional gleich **initialisieren**, also auf einen gewünschten Wert bringen, und das ist in vielen Fällen auch empfehlenswert, z. B.:

```
int wasser = 4711;
Bruch b1 = new Bruch();
```

Im zweiten Beispiel wird per **new**-Operator ein `Bruch`-Objekt erzeugt und dessen Adresse in die neue Referenzvariable `b1` geschrieben. Mit der Objektkreation und auch mit der Konstruktion von gültigen *Ausdrücken*, die einen Wert von passendem Datentyp liefern müssen, werden wir uns noch ausführlich beschäftigen.

Es ist üblich, Variablennamen mit einem Kleinbuchstaben beginnen zu lassen (vgl. Abschnitt 3.3.2), sodass man sie im Quelltext z. B. gut von Klassennamen unterscheiden kann, die per Konvention mit einem Großbuchstaben beginnen.

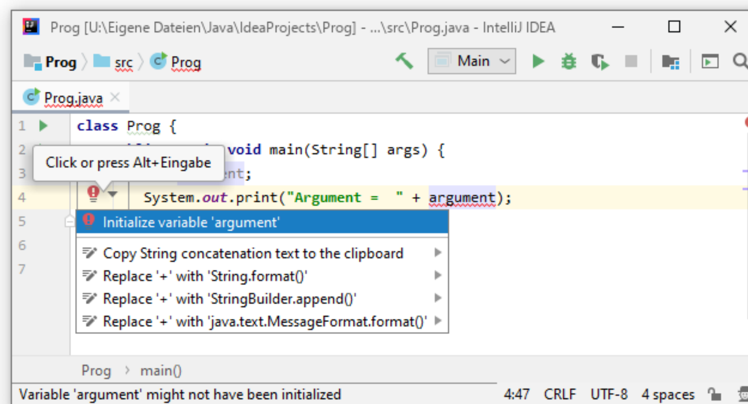
Weil *lokale* Variablen *nicht* automatisch initialisiert werden, muss man ihnen vor dem ersten lesenden Zugriff einen Wert zuweisen. Auch im Umgang mit uninitialisierten lokalen Variablen zeigt sich das Bemühen der Java-Designer um robuste Programme. Während C++ - Compiler in der Regel nur warnen, produzieren Java-Compiler eine Fehlermeldung und erstellen *keinen* Bytecode.[1] Dieses Verhalten wird durch das folgende Programm demonstriert:

```
class Prog {
   public static void main(String[] args) {
      int argument;
      System.out.print("Argument =  " + argument);
   }
}
```

Der OpenJDK 8 - Compiler meint dazu:

```
Prog.java:4: error: variable argument might not have been initialized
        System.out.print("Argument =  " + argument);
                                          ^
1 error
```

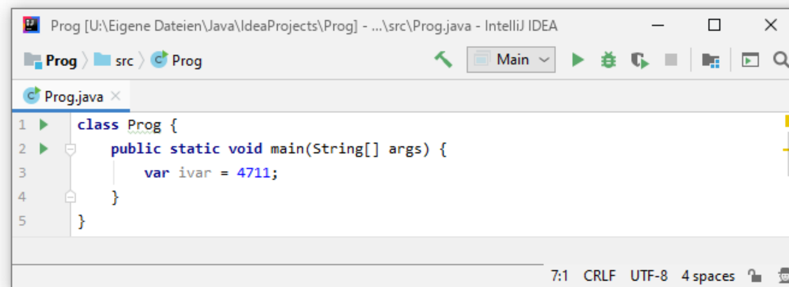
IntelliJ markiert den Fehler und schlägt eine sinnvolle Reparaturmaßnahme vor:



Weil Instanz- und Klassenvariablen automatisch mit dem typspezifischen Nullwert initialisiert werden (siehe unten), kann in einem Java-Programm kein Zugriff auf undefinierte Werte stattfinden.

Wie bereits erwähnt, können Java-Compiler seit der Version 10 den Typ von lokalen Variablen aus einem zugewiesenen Initialisierungswert erschließen (Typinferenz), und man darf in der Variablendeklaration die Typangabe durch das Schlüsselwort **var** ersetzen, z. B.:

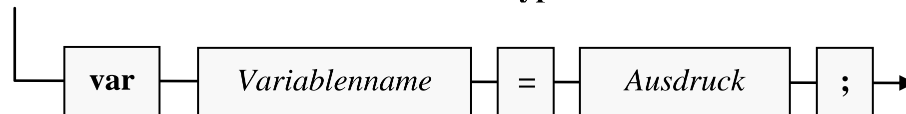
[1] Der im Visual Studio 2019 enthaltene C++ - Compiler der Firma Microsoft produziert beim Lesezugriff auf eine nicht-initialisierte lokale Variable die Warnung C4700, siehe https://docs.microsoft.com/de-de/cpp/error-messages/compiler-warnings/compiler-warning-level-1-and-level-4-c4700?view=vs-2019.



Im Beispiel gelingt die Inferenz, weil die zugewiesene Zahl 4711, die wir im Abschnitt 3.3.11.1 als *Ganzzahlliteral* bezeichnen werden, den Typ **int** besitzt.

Wie das Syntaxdiagramm zur Deklaration einer lokalen Variablen mit Typinferenz zeigt,

Deklaration einer lokalen Variablen mit Typinferenz



sind folgende Regeln einzuhalten:

- Es kann nur *eine* Variable deklariert werden.
- Es muss eine Initialisierung erfolgen.

Bei komplexen Typangaben, die im Kurs bisher noch nicht aufgetaucht sind, ermöglicht das Schlüsselwort **var** eine Arbeitserleichterung. Wenn sich dabei die Lesbarkeit des Quellcodes verschlechtert, wird die Nutzung des Codes (durch andere Programmierer) erschwert.

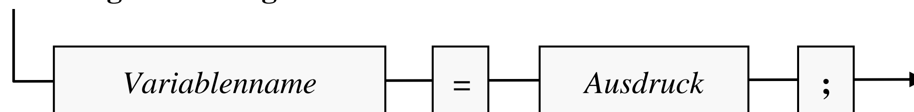
Durch lokale Variablen werden namensgleiche Instanz- bzw. Klassenvariablen überdeckt. Diese bleiben jedoch über ein geeignetes Präfix weiter ansprechbar:

- **this** bei Instanzvariablen
- Klassenname bei statischen Variablen

Weil eine solche Benennungspraxis kaum sinnvoll ist, verzichten wir auf Beispiele.

Um den Wert einer Variablen im weiteren Programmablauf zu verändern, verwendet man eine **Wertzuweisung**, die zu den einfachsten Anweisungen gehört:

Wertzuweisungsanweisung



Beispiel: `ggt = az;`
Durch diese Wertzuweisungsanweisung aus der `kuerze()` - Methode unserer Klasse `Bruch` (siehe Abschnitt 1.1.2) erhält die **int**-Variable `ggt` den Wert der **int**-Variablen `az`.

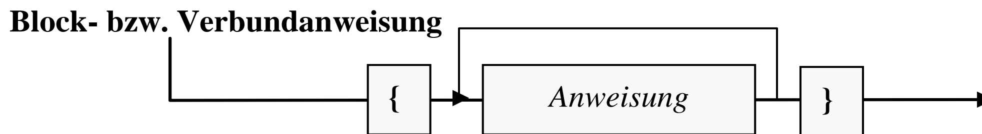
Es wird sich bald herausstellen, dass auch ein Ausdruck stets einen Datentyp besitzt. Bei der Wertzuweisung muss dieser Typ kompatibel zum Datentyp der Variablen sein.

Mittlerweile haben Sie Java-Anweisungen für folgende Zwecke kennengelernt:

- Deklaration einer lokalen Variablen
- Wertzuweisung

3.3.9 Blöcke und Sichtbarkeitsbereiche für lokale Variablen

Wie Sie bereits wissen, besteht der Rumpf einer Methodendefinition aus einem Block mit beliebig vielen Anweisungen, abgegrenzt durch geschweifte Klammern. Innerhalb des Methodenrumpfes können untergeordnete Anweisungsblöcke gebildet werden, wiederum durch geschweifte Klammen begrenzt:



Man spricht hier auch von einer **Block- bzw. Verbundanweisung**, und diese kann überall stehen, wo eine einzelne Anweisung erlaubt ist.

Unter den Anweisungen innerhalb eines Blocks dürfen sich selbstverständlich auch wiederum Verbundanweisungen befinden. Einfacher ausgedrückt: Blöcke dürfen geschachtelt werden.

In der Regel verwendet man die Blockanweisung als Bestandteil einer bedingten Anweisung oder einer Wiederholungsanweisung (siehe unten). Bei diesen Kontrollstrukturen wird *eine* Anweisung unter einer Bedingung bzw. wiederholt ausgeführt. Sollen z. B. unter einer Bedingung mehrere Anweisungen ausgeführt werden, wäre die Wiederholung der Bedingung für jede einzelne Anweisung außerordentlich lästig. Stattdessen fasst man die Anweisungen zu einem *Block* zusammen, der als *eine* Anweisung gilt, sodass die Bedingung nur einmal formuliert werden muss. Dieses sehr oft benötigte Muster ist z. B. in der Methode `setzeNenner()` der Klasse `Bruch` zu sehen:

```
public boolean setzeNenner(int n) {
    if (n != 0) {
        nenner = n;
        return true;
    } else
        return false;
}
```

Anweisungsblöcke haben einen wichtigen Effekt auf die Sichtbarkeit (alias: Gültigkeit) der darin deklarierten Variablen: Eine lokale Variable ist verfügbar von der deklarierenden Anweisung bis zur schließenden Klammer des Blocks, in dem sich die Deklaration befindet. Nur in diesem **Sichtbarkeitsbereich** (alias: *Gültigkeitsbereich*, engl. *scope*) kann sie über ihren Namen angesprochen werden. Im folgenden (weitgehend sinnfreien) Beispielprogramm wird versucht, auf die Variable `wert2` außerhalb ihres Sichtbarkeitsbereichs zuzugreifen:

```
class Prog {
   public static void main(String[] args) {
      int wert1 = 1;
      System.out.println("Wert1 = " + wert1);
      if (wert1 == 1) {
         int wert2 = 2;
         System.out.println("Gesamtwert = " + (wert1 + wert2));
      }
      System.out.println("Wert2 = " + wert2);
   }
}
```

Das veranlasst den OpenJDK 8 - Compiler zu der folgenden Fehlermeldung:

```
Prog.java:9: error: cannot find symbol
                System.out.println("Gesamtwert = " + (wert1 + wert2));
                                                              ^
  symbol:   variable wert2
  location: class Prog
1 error
```

Wird die fehlerhafte Zeile auskommentiert, lässt sich das Programm übersetzen. In dem zur **if**-Anweisung gehörenden Block ist die im übergeordneten Block der **Main()** - Methode deklarierte Variable `wert1` also gültig.

Bei hierarchisch geschachtelten Blöcken ist es in Java *nicht* erlaubt, auf mehreren Stufen Variablen mit identischem Namen zu deklarieren. Diese kaum sinnvolle Option ist z. B. in der Programmiersprache C++ vorhanden und erlaubt dort Fehler, die schwer aufzuspüren sind. In Java gehört ein eingeschachtelter Block zum Gültigkeitsbereich des umgebenden Blocks.

Der Sichtbarkeitsbereich einer lokalen Variablen sollte möglichst klein gehalten werden, um die Lesbarkeit und die Wartungsfreundlichkeit des Quellcodes zu verbessern. Vor allem wird auf diese Weise das Risiko von Programmierfehlern reduziert. Wird eine Variable zu früh deklariert, bestehen viele Gelegenheiten für schädliche Wertzuweisungen. Aus einer längst überwundenen Verpflichtung alter Programmiersprachen ist bei manchen Programmierern die Gewohnheit entstanden, alle lokale Variablen am Blockbeginn zu deklarieren. Stattdessen sollten lokale Variablen zur Minimierung ihres Sichtbarkeitsbereichs unmittelbar vor der ersten Verwendung deklariert werden (Bloch 2018, S. 261).

Zur übersichtlichen Gestaltung von Java-Programmen ist das Einrücken von Anweisungsblöcken sehr zu empfehlen, wobei Sie die Position der einleitenden Blockklammer und die Einrücktiefe nach persönlichem Geschmack wählen können, z. B.:

```
if (wert1 == 1) {
   int wert2 = 2;
   System.out.println("Wert2 = " + wert2);
}
```

```
if (wert1 == 1)
{
   int wert2 = 2;
   System.out.println("Wert2 = " + wert2);
}
```

Im Abschnitt 3.1.4 wurde erläutert, wie man in IntelliJ Regeln zum Quellcode-Layout definiert und auf eine Quellcodedatei anwendet. Wie man regelt, ob IntelliJ zum Einrücken ein Tabulatorzeichen oder eine (wählbare) Anzahl von Leerzeichen verwenden soll, wurde im Abschnitt 2.4.6.4 beschrieben.

Ein markierter Block aus mehreren Zeilen kann in IntelliJ mit

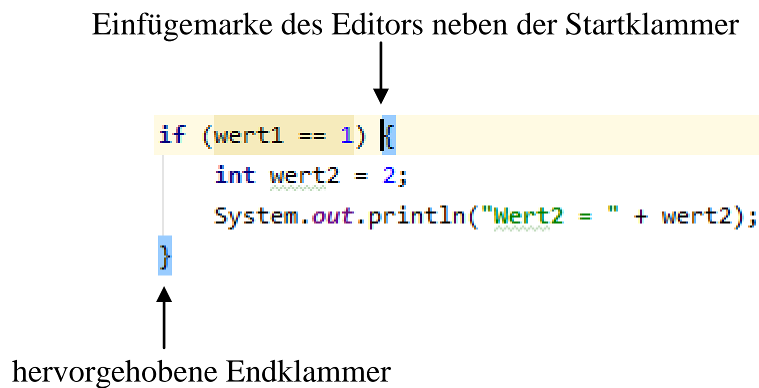
Tab komplett nach rechts eingerückt

und mit

Umschalt + Tab komplett nach links ausgerückt

werden.

Außerdem kann man sich zu einer Blockklammer das Gegenstück anzeigen lassen:



3.3.10 Finalisierte lokale Variablen

In der Regel sollten auch die im Programm benötigten konstanten Werte (z. B. für den Mehrwertsteuersatz) in einer Variablen abgelegt und im Quellcode über ihren Variablennamen angesprochen werden, denn:

- Bei einer späteren Änderung des Wertes ist nur die Quellcodezeile mit der Variablendeklaration und -initialisierung betroffen.
- Der Quellcode ist leichter zu lesen, wenn Variablennamen an Stelle von „magischen Zahlen" stehen.
- Irrtümliche Wertveränderungen werden verhindert.

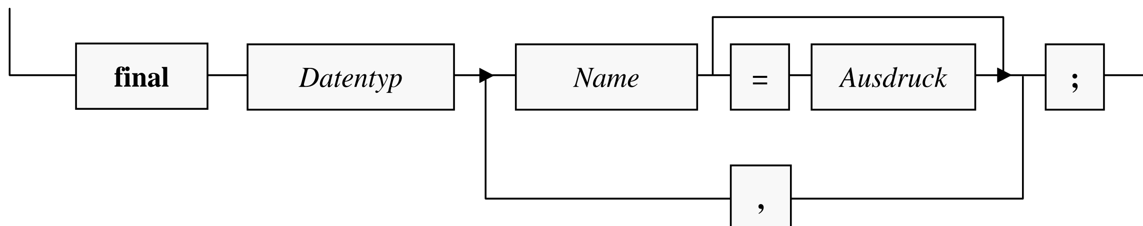
Beispiel:

Quellcode	Ausgabe

`class Prog {`
`   public static void main(String[] args) {`
`      final double mwst = 1.19;`
`      double netto = 100.0, brutto;`
`      brutto = netto * mwst;`
`      System.out.println("Brutto: " + brutto);`
`   }`
`}`
	`Brutto: 119.0`

Lokale Variablen, die nach ihrer Initialisierung auf denselben Wert fixiert bleiben sollen, deklariert man als **final**. Für finalisierte lokale (in einer Methode deklarierte) Variablen erhalten wir folgendes Syntaxdiagramm:

Deklaration einer finalisierten lokalen Variablen



Im Unterschied zur gewöhnlichen Variablendeklaration ist einleitend der Modifikator **final** zu setzen. Das Initialisieren einer finalisierten Variablen kann bei der Deklaration oder in einer späteren Wertzuweisung erfolgen. Danach ist keine weitere Wertveränderung mehr erlaubt.

Seit Java kann auch für eine finalisierte lokale Variable bei der Deklaration über das Schlüsselwert **var** die Typinferenz genutzt werden, z. B.:

```
final var mwst = 1.19;
```

Durch Verwendung des Modifikators **final** schützen wir uns davor, einen als fixiert geplanten Wert versehentlich doch zu ändern. In manchen Fällen wird auf diese Weise ein unangenehmer und nur mit großem Aufwand aufzuklärender Logikfehler zu einem harmlosen Syntaxfehler, der vom Compiler aufgedeckt, vom Entwickler ohne nennenswerten Aufwand beseitigt und vom Benutzer nie erlebt wird (Simons 2004, S. 51).

Weitere Argumente für das Finalisieren:

- Andere Programmierer, die später ebenfalls mit einer Methode arbeiten, erhalten durch die **final**-Deklaration eine wichtige Information zur intendierten Verwendung der betroffenen Variablen.
- Im funktionalen Programmierstil werden finalisierte (unveränderliche) Variablen strikt bevorzugt.

Durch den systematischen Gebrauch des **final**-Modifikators für lokale Variablen wirken Beispielprogramme etwas komplizierter, sodass im Manuskript meist der Einfachheit halber darauf verzichtet wird.

Neben lokalen Variablen können auch (statische) Felder einer Klasse als **final** deklariert werden (siehe Abschnitte 4.2.5 und 4.5.1).

Die empfohlene Camel Casing - Namenskonvention (vgl. Abschnitt 3.3.2) gilt bei *lokalen* Variablen trotz **final**-Deklaration. Nur bei **static**-Feldern mit **final**-Modifikator ist es üblich, den Namen komplett in Großbuchstaben zu schreiben (siehe Bloch 2018, S. 290).

3.3.11 Literale

Die im Quellcode auftauchenden expliziten Werte bezeichnet man als *Literale*. Wie Sie aus dem Abschnitt 3.3.10 wissen, sollten Literale vorzugsweise bei der Initialisierung von finalen Variablen verwendet werden, z. B.:

```
final double mwst = 1.19;
```

Auch die Literale besitzen in Java stets einen **Datentyp**, wobei einige Regeln zu beachten sind, die gleich erläutert werden. Im aktuellen Abschnitt 3.3.11 haben manche Passagen Nachschlagecharakter, sodass man beim ersten Lesen nicht jedes Detail aufnehmen muss bzw. kann.

3.3.11.1 Ganzzahlliterale

Für ein Ganzzahlliteral wird meist das **dezimale** Zahlensystem verwendet, z. B.:

```
final int wasser = 4711;
```

Java unterstützt aber auch alternative Zahlensysteme:

- das **binäre** (mit der Basis 2 und den Ziffern 0, 1),
- das **oktale** (mit der Basis 8 und den Ziffern 0, 1, 2, …, 7)
- und das **hexadezimale** (mit der Basis 16 und den Ziffern 0, 1, …, 9, A, B, C, D, E, F)

Wenn ein Ganzzahlliteral in einem nicht-dezimalen Zahlensystem interpretiert werden soll, muss ein Präfix (mit einleitender Null) vorangestellt werden:

Zahlensystem	**Präfix**	**Beispiele**	
		println() - Aufruf	**Ausgabe**
binär	0b, 0B	`System.out.println(0b11);`	3
oktal	0	`System.out.println(011);`	9
hexadezimal	0x, 0X	`System.out.println(0x11);`	17

Für das Ganzzahlliteral `0x11` ergibt sich der dezimale Wert 17 aufgrund der Stellenwertigkeiten im Hexadezimalsystem folgendermaßen:

$$11_{\text{Hex}} \;=\; 1 \cdot 16^1 + 1 \cdot 16^0 \;=\; 1 \cdot 16 + 1 \cdot 1 = 17$$

Vermutlich fragen Sie sich, wozu man sich mit dem Hexadezimalsystem plagen sollte. Gelegentlich ist ein ganzzahliger Wert (z. B. als Methodenparameter) anzugeben, den man (z. B. aus einer Tabelle) nur in hexadezimaler Darstellung kennt. In diesem Fall spart man sich durch Verwendung dieser Darstellung die Wandlung in das Dezimalsystem.

Tückisch ist der Präfix für die (selten benötigten) Literale im Oktalsystem. Die führende Null im Ganzzahlliteral `011` ist keinesfalls irrelevant, sondern bewirkt eine oktale Interpretation:

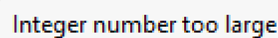
$$11_{\text{Oktal}} = 1 \cdot 8 + 1 \cdot 1 = 9$$

Unabhängig vom verwendeten Zahlensystem haben Ganzzahlliterale in Java den Datentyp **int**, wenn nicht durch das Suffix **L** oder **l** der Datentyp **long** erzwungen wird. Das ist im folgenden Beispiel

```
final long betrag = 2147483648L;
```

erforderlich, weil anderenfalls ein **int**- Literal mit Wert außerhalb des zulässigen Bereichs resultiert:

```
final long betrag = 2147483648;
```

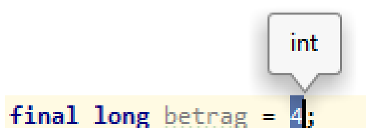


Der Kleinbuchstabe **l** ist leicht mit der Ziffer **1** zu verwechseln und daher als Suffix wenig geeignet.

Dass ein Ganzzahlliteral tatsächlich per Voreinstellung den Datentyp **int** besitzt, können Sie mit Hilfe unserer Entwicklungsumgebung überprüfen. Befindet sich die Einfügemarke neben einem (oder in einem) Ganzzahlliteral, dann liefert die Tastenkombination

Umschalt + Strg + P

den Datentyp, z. B.:



Seit Java 7 dürfen bei Ganzzahlliteralen zwischen zwei Ziffern Unterstriche zur optischen Gruppierung gesetzt werden, z. B.:

```
final int wasser = 4_711;
```

Weil **int**-Literale als Bestandteile der im nächsten Abschnitt behandelten Gleitkommaliterale auftreten, lässt sich die Ziffergruppierung durch Unterstriche auch dort verwenden.

3.3.11.2 Gleitkommaliterale

Zahlen mit Dezimalpunkt oder Exponent sind in Java vom Typ **double**, wenn nicht durch das Suffix **F** oder **f** der Datentyp **float** erzwungen wird, z. B.:

```
final double mwst = 1.19;
final float ff = 9.78f;
```

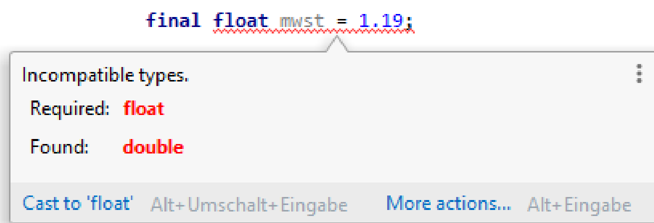
Mit dem Suffix **D** oder **d** wird auch bei einer Zahl *ohne* Dezimalpunkt oder Exponent der Datentyp **double** erzwungen. Warum das Suffix **d** im folgenden Beispiel für das korrekte Rechenergebnis sorgt, erfahren Sie im Zusammenhang mit dem Unterschied zwischen Ganzzahl- und Gleitkommaarithmetik:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
    public static void main(String[] args) {
        System.out.println(5/2);
        System.out.println(5d/2);
    }
}</pre></td><td><pre>2
2.5</pre></td></tr>
</table>

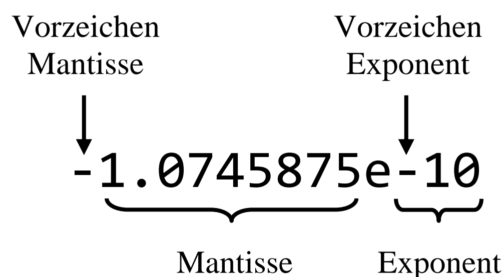
Die Java-Compiler achten bei Wertzuweisungen unter Verwendung von Gleitkommaliteralen streng auf die Typkompatibilität. Z. B. führt die folgende Deklaration mit Initialisierung:

```
final float mwst = 1.19;
```

zu einer Fehlermeldung, weil das Gleitkommaliteral (und damit der Ausdruck rechts vom Gleichheitszeichen) den Typ **double** besitzt, die Variable `mwst` hingegen den Typ **float**:

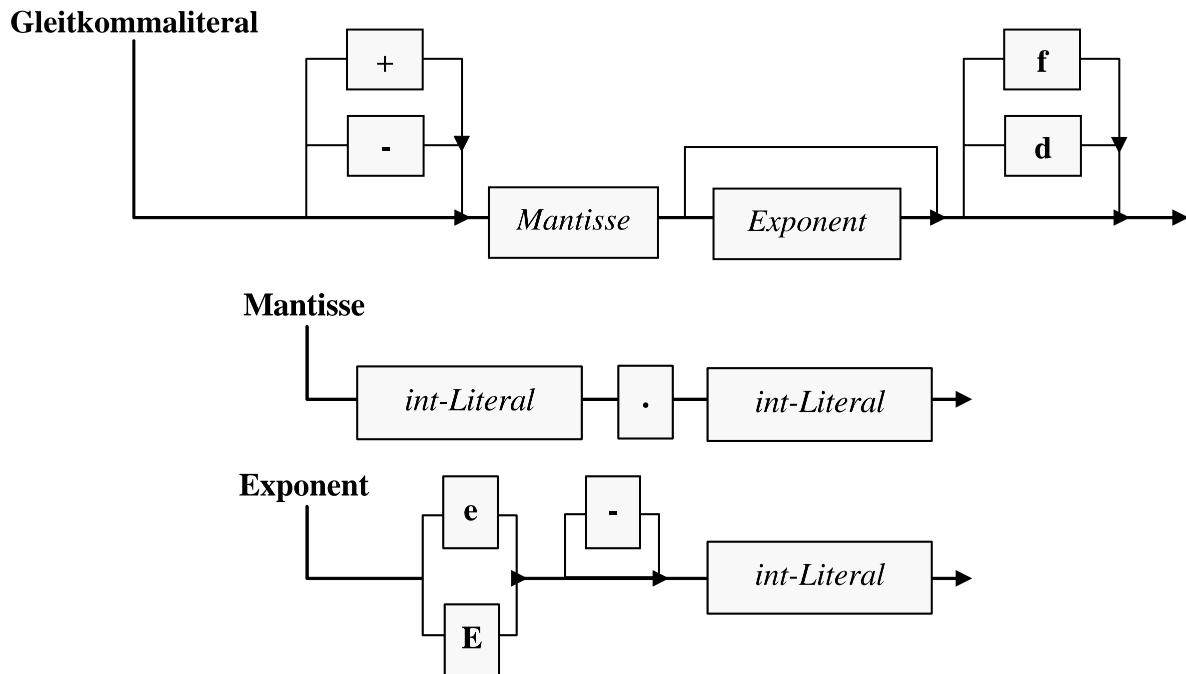


Neben der alltagsüblichen Schreibweise (mit dem *Punkt* als Dezimaltrennzeichen) erlaubt Java bei Gleitkommaliteralen auch die wissenschaftliche Exponentialnotation (mit der Basis 10), z. B. bei der Zahl -0,00000000010745875):



Eine Veränderung des Exponenten lässt das Dezimaltrennzeichen gleiten und macht somit die Bezeichnung *Gleitkommaliteral* (engl.: *floating-point literal*) plausibel.

In den folgenden Syntaxdiagrammen werden die wichtigsten Regeln für Gleitkommaliterale beschrieben:



Die in der Mantisse und im Exponenten auftretenden Ganzzahlliterale müssen das dezimale Zahlensystem verwenden und den Datentyp **int** besitzen, sodass die im Abschnitt 3.3.11.1 beschriebenen Präfixe (0, 0b, 0B, 0x, 0X) und Suffixe (L, l) verboten sind. Die Exponenten werden zur Basis 10 verstanden.

Der Einfachheit halber unterschlagen die Syntaxdiagramme die folgende, im letzten Beispielprogramm benutzte Konstruktion eines Gleitkommaliterals über das Suffix **d**:

```
System.out.println(5d/2);
```

3.3.11.3 boolean-Literale

Als Literale vom Typ **boolean** sind nur die beiden reservierten Wörter **true** und **false** erlaubt, z. B.:

```
boolean cond = true;
```

3.3.11.4 char-Literale

char-Literale werden in Java durch *einfache* Hochkommata begrenzt. Es sind erlaubt:

- **Einfache Zeichen**
  Beispiel:

  ```
  final char bst = 'b';
  ```

  Das einfache Hochkomma kann allerdings auf diese Weise ebenso wenig zum **char**-Literal werden wie der Rückwärts-Schrägstrich (\). In diesen Fällen benötigt man eine sogenannte *Escape-Sequenz*:

- **Escape-Sequenzen**
  Indem man ein Zeichen hinter einen einleitenden Rückwärts-Schrägstrich setzt (z. B. `\'`, `\n`) und damit eine sogenannte *Escape-Sequenz* bildet, kann man …

  - ein Zeichen von seiner besonderen Bedeutung befreien (z. B. das zur Begrenzung von **char**-Literalen dienende Hochkomma) und wie ein einfaches Zeichen behandeln, z. B.:

    ```
    \'
    \"
    ```

`\\`

- ein Steuerzeichen für die Textausgabe im Konsolenfenster ansprechen, z. B.:

Neue Zeile	`\n`
Horizontaler Tabulator	`\t`

  Wir werden die Escape-Sequenz `\n` oft in einem Zeichenfolgenliteral (siehe Abschnitt 3.3.11.5) unter normale Zeichen mischen, um bei der Konsolenausgabe einen Zeilenwechsel anzuordnen. [1]

Beispiel:

```
final char rs = '\\';
```

- **Unicode-Escape-Sequenzen**
  Eine Unicode-Escape-Sequenz enthält eine Unicode-Zeichennummer (vorzeichenlose Ganzzahl mit 16 Bit, also im Bereich von 0 bis 2^{16}-1 = 65535) in hexadezimaler, vierstelliger Schreibweise (ggf. links mit Nullen aufgefüllt) *ohne* Hexadezimal-Präfix) nach der Einleitung durch **\u** (kleines u!). So lassen sich Zeichen ansprechen, die per Tastatur nicht einzugeben sind.
  Beispiel:

```
final char alpha = '\u03b1';
```

  Im Konsolenfenster werden die Unicode-Zeichen oberhalb von `\u00ff` in der Regel als Fragezeichen dargestellt. In einem GUI-Fenster erscheinen sie jedoch in voller Pracht (siehe nächsten Abschnitt).

3.3.11.5 Zeichenfolgenliterale

Zeichenfolgenliterale werden (im Unterschied zu **char**-Literalen) durch *doppelte* Hochkommata begrenzt. Ein Zeichenfolgenliteral kann einfache Zeichen, Escape-Sequenzen und Unicode-Escape-Sequenzen enthalten (vgl. Abschnitt 3.3.11.4). Das einfache Hochkomma hat innerhalb eines Zeichenfolgenliterals keine Sonderrolle, z. B.:

```
System.out.println("Otto's Welt");
```

Um ein doppeltes Hochkomma in eine Zeichenfolge aufzunehmen, ist die Escape-Sequenz **\"** zu verwenden.

Zeichenkettenliterale sind vom Datentyp **String**, und später wird sich herausstellen, dass es sich bei diesem Typ um eine Klasse aus dem Java-API handelt.

Während ein **char**-Literal stets *genau ein* Zeichen enthält, kann ein Zeichenkettenliteral aus beliebig vielen Zeichen bestehen oder auch leer sein, z. B.:

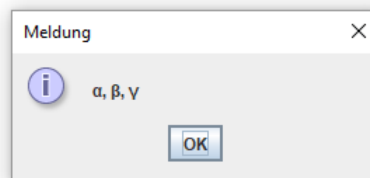
```
String name = "";
```

[1] Bei der Ausgabe in eine Textdatei sollte die Escape-Sequenz `\n` *nicht* verwendet werden, weil sie nicht auf allen Plattformen bzw. von allen Editoren als Zeilenwechsel interpretiert wird (siehe Abschnitte 3.2.2 und 14.8.1).

Das folgende Programm verwendet einen Aufruf der statischen Methode **showMessageDialog()** der Klasse **JOptionPane** aus dem Paket **javax.swing** (seit Java 9 im Modul **java.desktop**) zur Anzeige eines Zeichenkettenliterals, das drei Unicode-Escape-Sequenzen enthält:[1]

```
class Prog {
   public static void main(String[] args) {
      javax.swing.JOptionPane.showMessageDialog(null, "\u03b1, \u03b2, \u03b3");
   }
}
```

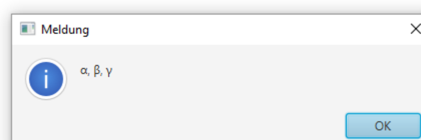
Beim Programmstart erscheint das folgende Meldungsfenster:



[1] Im Manuskript wird überwiegend an Stelle des betagten GUI-Frameworks Swing die moderne Alternative JavaFX (alias OpenJFX) verwendet. Beim aktuellen Beispiel verursacht die JavaFX-Variante aber erheblich mehr Aufwand und Vorgriffe auf noch unbehandelte Kursthemen (z. B. Vererbung):

```
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.application.Application;
import javafx.stage.Stage;

public class Prog extends Application {
   @Override
   public void start(Stage primaryStage) {
      Alert alert = new Alert(AlertType.INFORMATION, "\u03b1, \u03b2, \u03b3");
      alert.setHeaderText("");
      alert.showAndWait();
   }
   public static void main(String[] args) {
      launch(args);
   }
}
```

Das Ergebnis:



3.3.11.6 Referenzliteral null

Einer Referenzvariablen kann das Referenzliteral **null** zugewiesen werden, z. B.:[1]

```
Bruch b1 = null;
```

Damit ist sie nicht undefiniert, sondern zeigt explizit auf nichts.

Zeigt eine Referenzvariable aktuell auf ein existentes Objekt, kann man diese Referenz per **null**-Zuweisung aufheben. Sofern im Programm keine andere Referenz auf dasselbe Objekt vorliegt, ist es zum Abräumen durch den Garbage Collector der JVM freigegeben.

3.4 Eingabe bei Konsolenprogrammen

Konsolenprogramme sind ein geeignetes Umfeld, um die Programmiersprache Java zu erlernen und mit der Standardbibliothek vertraut zu werden. Bald werden wir selbstverständlich auch die Erstellung von Anwendungen mit grafischer Bedienoberfläche behandeln. Um mit Konsolenanwendungen unsere didaktischen Ziele zu erreichen, benötigen wir eine Möglichkeit, Benutzereingaben entgegenzunehmen. Im aktuellen Abschnitt wird eine Lösung vorgestellt, die sich mit geringem Aufwand in unseren Demonstrations- und Übungsprogrammen verwenden lässt.

3.4.1 Die Klassen Scanner und Simput

Für die Übernahme von Tastatureingaben in Konsolenprogrammen kann die API-Klasse **Scanner** (Paket **java.util**, ab Java 9 im Modul **java.base**) verwendet werden.[2] Im folgenden Beispielprogramm zur Berechnung der Fakultät zu einer ganzen Zahl wird ein **Scanner**-Objekt per **nextInt()** - Methodenaufruf gebeten, vom Benutzer eine **int**-Ganzzahl entgegenzunehmen:

```
import java.util.Scanner;
class Prog {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Argument:  ");
        int argument = input.nextInt();
        double fakul = 1.0;
        for (int i = 2; i <= argument; i++)
            fakul = fakul * i;
        System.out.println("Fakultät: " + fakul);
    }
}
```

Zwei Hinweise zum Quellcode:

---

[1] Da Java eine streng typisierte Programmiersprache ist, und das Literal **null** einen Ausdruck darstellt (vgl. Abschnitt **3.5**), muss es einen Datentyp besitzen. Es ist der **Nulltyp** (engl.: *null type*). Weil es in Java keinen Bezeichner für den Nulltyp gibt, kann man keine Variable von diesem Typ deklarieren. Wie das folgende Zitat aus der aktuellen Java-Sprachspezifikation (Gosling et al. 2019, S. 44) belegt, müssen Sie sich um den Nulltyp keine großen Gedanken machen:

> In practice, the programmer can ignore the null type and just pretend that null is merely a special literal that can be of any reference type.

[2] Mit den Paketen und Modulen der Standardbibliothek werden wir uns später ausführlich beschäftigen. An dieser Stelle dient die Angabe der Paket- und Modulzugehörigkeit dazu, eine Klasse eindeutig zu identifizieren und die Standardbibliothek allmählich kennenzulernen.

- Weil sich die Klasse **Scanner** im API-Paket **java.util** befindet, muss sie importiert werden, damit sie im Quellcode ohne Paket-Präfix angesprochen werden kann.
- Die im Programm verwendete **for**-Wiederholungsanweisung wird im Abschnitt 3.7.3 behandelt.

Bei einer gültigen Eingabe arbeitet das Programm wunschgemäß, z. B.:

```
Argument:  4
Fakultät: 24.0
```

Auf ungültige Benutzereingaben reagiert die Methode **nextInt()** mit einer sogenannten Ausnahme, und das Programm „stürzt ab“, z. B.:

```
Argument:  vier
Exception in thread "main" java.util.InputMismatchException
  at java.base/java.util.Scanner.throwFor(Scanner.java:864)
  at java.base/java.util.Scanner.next(Scanner.java:1485)
  at java.base/java.util.Scanner.nextInt(Scanner.java:2117)
  at java.base/java.util.Scanner.nextInt(Scanner.java:2076)
  at Prog.main(Prog.java:6)
```

Es wäre nicht allzu aufwändig, in der Fakultätsanwendung ungültige Eingaben abzufangen. Allerdings stehen uns die erforderlichen Programmiertechniken (der Ausnahmebehandlung) noch nicht zur Verfügung, und außerdem ist bei den möglichst kurzen Demonstrations- und Übungsprogrammen jeder Zusatzaufwand störend.

Um Tastatureingaben in Konsolenprogrammen bequem und sicher bewerkstelligen zu können, wurde für den Kurs eine Klasse namens **Simput** erstellt.[1] Mit Hilfe der Klassenmethode Simput.gint() lässt sich das Fakultätsprogramm einfacher und zugleich robust gegenüber Eingabefehlern realisieren:

```
class Prog {
    public static void main(String[] args) {
        System.out.print("Argument:  ");
        int argument = Simput.gint();
        double fakul = 1.0;
        for (int i = 2; i <= argument; i++)
            fakul = fakul * i;
        System.out.println("Fakultät: " + fakul);
    }
}
```

Weil die Klasse Simput keinem Paket zugeordnet wurde, gehört sie zum Standardpaket und kann daher *in anderen Klassen des Standardpakets* bequem ohne Paket-Präfix bzw. Paket-Import angesprochen werden (vgl. Abschnitt 3.1.7). In Klassen anderer Pakete steht Simput (wie alle anderen Klassen des Standardpakets) jedoch *nicht* zur Verfügung. Im Kurs erstellen wir meist kleine Demonstrationsprogramme und verwenden dabei der Einfachheit halber das Standardpaket, sodass die Klasse Simput als bequemes Hilfsmittel genutzt werden kann. Für ernsthafte Projekte werden Sie jedoch eigene Pakete definieren (siehe Kapitel 6), sodass die (kompilierte) Klasse Simput dort *nicht* verwendbar ist. Diese Einschränkung ist aber leicht durch eine Änderung des Quellcodes in der Datei **Simput.java** zu beheben.

---

[1] Die Datei **Simput.java** ist im folgenden Verzeichnis zu finden (weitere Ortsangaben siehe Vorwort):
…\BspUeb\Simput\Standardpaket\IntelliJ-Projekt\src
Die zugehörige Bytecode-Datei **Simput.class** steckt im Ordner
...\BspUeb\Simput\Standardpaket\IntelliJ-Projekt\out\production\Simput
und der Bequemlichkeit auch im Ordner
…\BspUeb\Simput\Standardpaket

Die statische `Simput`-Methode `gint()` erwartet vom Benutzer eine per **Enter**-Taste quittierte Eingabe und versucht, diese als **int**-Wert zu interpretieren. Im Erfolgsfall erhält die aufrufende Methode das Ergebnis als `gint()` - Rückgabewert. Anderenfalls sieht der Benutzer eine Fehlermeldung, und die aufrufende Methode erhält den (Verlegenheits-)Rückgabewert 0, z. B.

```
Argument:  vier
Falsche Eingabe!

Fakultät: 1.0
```

Die `Simput`-Klassenmethode `gint()` liefert eine Rückgabe vom Typ **int** und besitzt keinen Parameter. Diese und andere syntaktische Aspekte ihrer Verwendung werden durch den Methodenkopf dokumentiert:

```
public static int gint()
```

Auch in der API-Dokumentation wird zur Beschreibung einer Methode deren Definitionskopf angegeben, z. B. bei der statischen Methode **exp()** der Klasse **Math** im Paket **java.lang**:

exp

`public static double exp(double a)`

Returns Euler's number *e* raised to the power of a `double` value. Special cases:

- If the argument is NaN, the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is negative infinity, then the result is positive zero.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:

`a` - the exponent to raise *e* to.

Returns:

the value e^a, where *e* is the base of the natural logarithms.

Ergänzend liefert die API-Dokumentation aber auch Informationen zur Arbeitsweise der Methode, zur Rolle der Parameter, zum Inhalt der Rückgabe und ggf. zu den möglichen Ausnahmefehlern.

Bei `gint()` oder anderen `Simput`-Methoden, die auf Eingabefehler *nicht* mit einer Ausnahme reagieren (vgl. Abschnitt 11), kann man sich durch einen Aufruf der `Simput`-Klassenmethode `checkError()` mit Rückgabetyp **boolean** darüber informieren, ob ein Fehler aufgetreten ist (Rückgabewert **true**) oder nicht (Rückgabewert **false**). Die `Simput`-Klassenmethode `getErrorDescription()` hält im Fehlerfall darüber hinaus eine Erläuterung bereit. In obigem Beispielprogramm ignoriert die aufrufende Methode **main()** allerdings die diagnostischen Informationen und liefert ggf. eine unpassende Ausgabe. Wir werden in vielen weiteren Beispielprogrammen den `gint()` - Rückgabewert der Kürze halber ohne Fehlerstatuskontrolle benutzen. Bei Anwendungen für den praktischen Einsatz sollte aber wie in folgender Variante des Fakultätsprogramms eine Überprüfung stattfinden. Die dazu erforderliche **if**-Anweisung wird im Abschnitt 3.7.2 behandelt.

Quellcode	Ein- und Ausgabe

`class Prog {`
`  public static void main(String args[]) {`
`    System.out.print("Argument:  ");`
`    int argument = Simput.gint();`
`    double fakul = 1.0;`
`    if (!Simput.checkError()) {`
`      for (int i = 2; i <= argument; i += 1)`
`        fakul = fakul * i;`
`      System.out.println("Fakultät: "+fakul);`
`    } else`
`      System.out.println(`
`          Simput.getErrorDescription());`
`    }`
`}`
	`Argument:  vier`
	`Falsche Eingabe!`
	
	`Eingabe konnte nicht konvertiert werden.`

Neben `gint()` besitzt die Klasse `Simput` noch analoge Methoden für andere Datentypen, u.a.:

- `public static long glong()`
  Liest eine ganze Zahl vom Typ **long** von der Konsole
- `public static double gdouble()`
  Liest eine Gleitkommazahl vom Typ **double** von der Konsole, wobei das erwartete Dezimaltrennzeichen vom eingestellten Gebietsschema des Benutzers abhängt. Bei der Einstellung `de_DE` wird ein Dezimalkomma erwartet.
- `public static char gchar()`
  Liest ein Zeichen von der Konsole

3.4.2 Eine globale Bibliothek mit Simput in IntelliJ einrichten

Benutzt ein Programm die Klasse `Simput`, dann muss ...

- beim Übersetzen des Programms durch den OpenJDK-Compiler (**javac.exe**) entweder die Quellcodedatei **Simput.java** im aktuellen Verzeichnis liegen, oder die Bytecode-Datei **Simput.class** über den Klassenpfad auffindbar sein,
- bei der Ausführung des Programms durch die JVM (**java.exe**) die Bytecode-Datei **Simput.class** über den Klassenpfad auffindbar sein.

Der Klassenpfad kann über die CLASSPATH-Umgebungsvariable oder durch die beim Compiler- bzw. Interpreter-Aufruf verwendete **classpath** - Option definiert werden (vgl. Abschnitt 2.2.4).

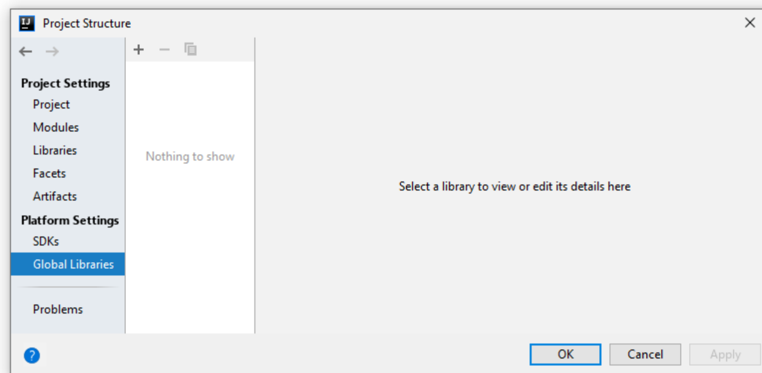
Unsere Entwicklungsumgebung IntelliJ IDEA ignoriert die CLASSPATH-Umgebungsvariable, bietet aber die äquivalente Möglichkeit zur Definition von Bibliotheken auf Modul-, Projekt- oder IDE-Ebene. Beim Aufruf der Werkzeuge zum Übersetzen oder Starten von Java-Programmen (z. B. **javac.exe** oder **java.exe**) erstellt IntelliJ jeweils eine **-classpath** - Option aus den Bibliotheksdefinitionen.

Es hat sich als günstig erwiesen, wenn benötigte Bibliotheksklassen in einer Java-Archivdatei vorliegen. Im Ordner **...\BspUeb\Simput\Standardpaket** (weitere Ortsangaben im Vorwort) finden Sie daher neben der Bytecode-Datei **Simput.class** auch die Archivdatei **Simput.jar**. Wir werden uns im Kapitel 6 mit Java-Archivdateien beschäftigen.

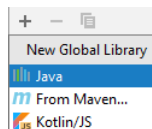
Wir definieren nun die Datei **Simput.jar** als IDE-globale Bibliothek, die in beliebigen Projekten genutzt werden kann. Nach

File > Project Structure > Global Libraries

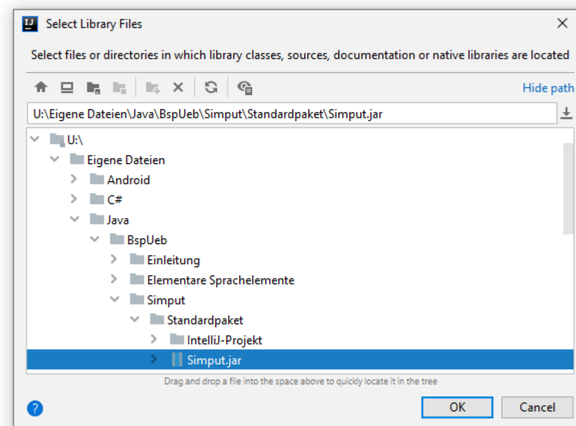
klicken wir im folgenden Dialog



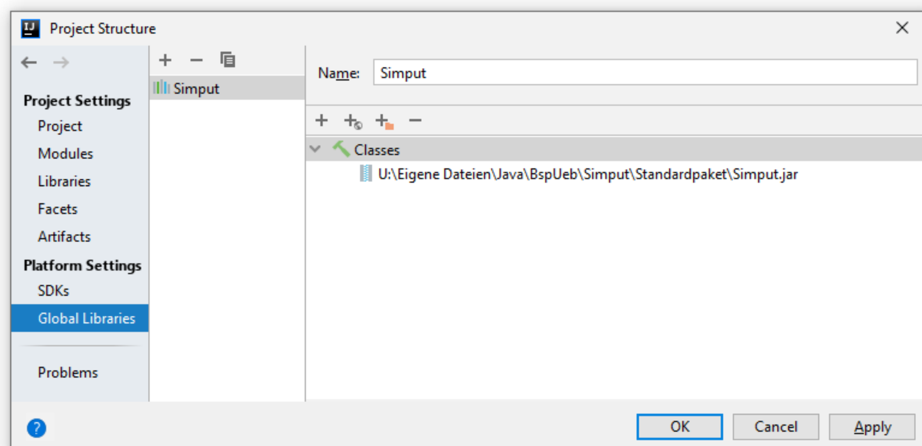
auf den Schalter **+**, entscheiden uns für die Kategorie **Java**



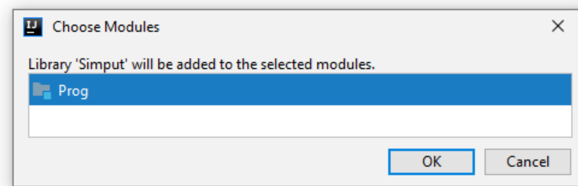
und wählen anschließend die Datei **Simput.jar**:



Damit die nun definierte globale Bibliothek `Simput`



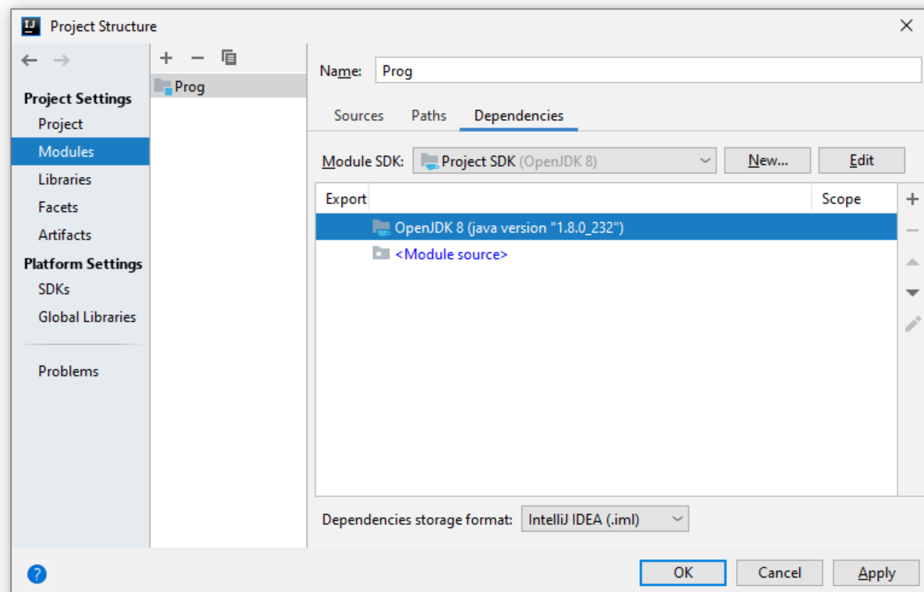
in einem konkreten Projekt bzw. Modul benutzt werden kann, muss sie in die Liste der Abhängigkeiten des Moduls aufgenommen werden. Für das aktuell geöffnete Projekt kann dies bequem per Mausklick geschehen, z. B.:



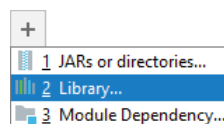
Bei einem anderen Projekt öffnet man nach

File > Project Structure > Modules

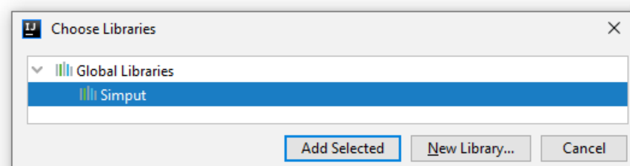
im folgenden Fenster für das meist einzige vorhandene IntelliJ-Modul die Registerkarte **Dependencies**:



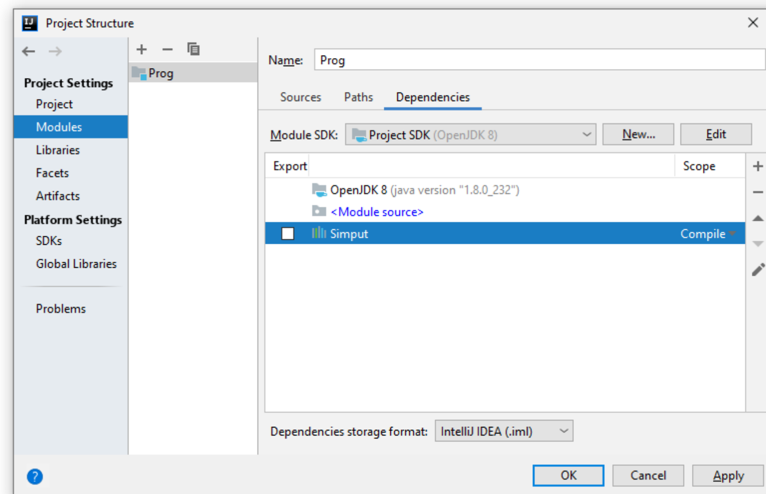
Nach einem Klick auf den Schalter + am rechten Fensterrand (!) entscheidet man sich für die Kategorie **Library**



und wählt dann die globale Bibliothek `Simput`,



die anschließend in der Liste der Abhängigkeiten erscheint:



Nun können aus der Klasse `Simput` z. B. die statischen Methoden im Projekt genutzt werden.

3.5 Operatoren und Ausdrücke

Im Zusammenhang mit der Variablendeklaration und der Wertzuweisung haben wir das Sprachelement *Ausdruck* ohne Erklärung benutzt, und die soll nun nachgeliefert werden. Im aktuellen Abschnitt 3.5 werden wir Ausdrücke als wichtige Bestandteile von Java-Anweisungen recht detailliert betrachten. Dabei lernen Sie elementare Datenverarbeitungsmöglichkeiten kennen, die von sogenannten Operatoren mit ihren Argumenten realisiert werden, z. B. von den arithmetischen Operatoren (+, -, *, /) für die Grundrechenarten. Am Ende des Abschnitts kann immerhin schon das Programmieren eines Währungskonverters als Übungsaufgabe gestellt werden. Allzu große Begeisterung wird wohl trotzdem nicht aufkommen, doch ein sicherer Umgang mit Operatoren und Ausdrücken ist unabdingbare Voraussetzung für das erfolgreiche Implementieren von Methoden. Dort werden Algorithmen bzw. die Handlungskompetenzen von Klassen bzw. Objekten realisiert.

Während die Variablen zur *Speicherung* von Werten dienen, geht es bei den **Operatoren** darum, aus vorhandenen Variableninhalten und/oder anderen Argumenten neue Werte zu berechnen. Den zur Berechnung eines Werts geeigneten, aus Operatoren und zugehörigen Argumenten aufgebauten Teil einer Anweisung bezeichnet man als **Ausdruck**, z. B. in der folgenden Wertzuweisung:[1]

Operator

```
az = az - an;
```

Ausdruck

Durch diese Anweisung aus der `kuerze()` - Methode unserer Klasse `Bruch` (siehe Abschnitt 1.1) wird der lokalen **int**-Variablen `az` der Wert des Ausdrucks `az - an` zugewiesen. Wie in diesem Beispiel landen die Werte von Ausdrücken oft in Variablen, wobei Ausdruck und Variable typkompatibel sein müssen. Aus den Operatoren eines Ausdrucks und den zugehörigen Argumenten ergibt sich nicht nur ein **Wert**, sondern auch ein **Datentyp**.

Schon bei einem Literal, einer Variablen oder einem Methodenaufruf haben wir es mit einem Ausdruck zu tun.[2]

[1] Im Abschnitt 3.5.8 werden Sie eine Möglichkeit kennenlernen, diese Anweisung etwas kompakter zu formulieren.

[2] Besteht ein Ausdruck aus einem Methodenaufruf mit dem Pseudorückgabetyp **void**, dann liegt allerdings *kein* Wert vor.

Beispiele:

- `1.5`
  Dieses Gleitkomma-Literal ist ein Ausdruck mit dem Typ **double** und dem Wert 1,5.
- `Simput.gint()`
  Dieser Methodenaufruf ist ein Ausdruck mit dem Typ **int** (= Rückgabetyp der Methode), wobei die Eingabe des Benutzers über den Wert entscheidet (siehe Abschnitt 3.4.1 zur Beschreibung der Klassenmethode `Simput.gint()`, die *nicht* zum Java-API gehört).

Mit Hilfe diverser Operatoren entsteht ein komplexerer Ausdruck, dessen Typ und Wert von den Argumenten und den Operatoren abhängen.

Beispiele:

- `2 * 1.5`
  Hier resultiert der **double**-Wert 3,0.
- `2 * 3`
  Hier resultiert der **int**-Wert 6.
- `2 > 1.5`
  Hier resultiert der **boolean**-Wert **true**.

In der Regel beschränken sich die Operatoren darauf, aus ihren Argumenten (Operanden) einen Wert zu ermitteln und für die weitere Verarbeitung zur Verfügung zu stellen. Einige Operatoren haben jedoch zusätzlich einen **Nebeneffekt** auf eine als Argument fungierende *Variable*, z. B.:

```
int i = 12;
int j = i++;
```

In der zweiten Anweisung des Beispiels tritt der **Postinkrementoperator** ++ mit der **int**-Variablen `i` als Argument auf. Der Ausdruck `i++` hat den Typ **int** und den Wert 12, welcher in der Zielvariablen `j` landet. Außerdem wird die Argumentvariable `i` beim Auswerten des Ausdrucks durch den Postinkrementoperator auf den neuen Wert 13 gebracht.

Die meisten Operatoren verarbeiten *zwei* Operanden (Argumente) und heißen daher **zweistellig** oder **binär**. Im folgenden Beispiel ist der **Additionsoperator** zu sehen, der zwei numerische Argumente erwartet:

```
a + b
```

Manche Operatoren begnügen sich mit *einem* Argument und heißen daher **einstellig** oder **unär**. Als Beispiel haben wir eben schon den Postinkrementoperator kennengelernt. Ein weiteres ist der **Negationsoperator**, der durch ein Ausrufezeichen („**!**“) dargestellt wird, ein Argument vom Typ **boolean** erwartet und dessen Wahrheitswert umdreht (**true** und **false** vertauscht), z. B.:

```
!cond
```

Wir werden auch noch einen *dreistelligen* (*ternären*) Operator kennenlernen.

Weil Ausdrücke von passendem Ergebnistyp als Argumente einer Operation erlaubt sind, können beliebig komplexe Ausdrücke aufgebaut werden. Unübersichtliche Exemplare sollten jedoch als potentielle Fehlerquellen vermieden werden.

3.5.1 Arithmetische Operatoren

Die arithmetischen Operatoren sind für die Grundrechenarten zuständig, und ihre Operanden (Argumente) müssen einen primitiven Ganzzahl- oder Gleitkommatyp haben (**byte**, **short**, **int**, **long**, **char**, **float** oder **double**). Die resultieren Ausdrücke haben wiederum einen numerischen Ergebnistyp und werden oft als **arithmetische Ausdrücke** bezeichnet.

Es hängt von den Datentypen der Operanden ab, ob bei den Berechnungen die **Ganzzahl**- oder die **Gleitkommaarithmetik** zum Einsatz kommt. Besonders auffällig sind die Unterschiede im Verhalten des Divisionsoperators (dargestellt durch einen Schrägstrich), z. B.:

Quellcode

```
class Prog {
    public static void main(String[] args) {
        int i = 2, j = 3;
        double a = 2.0;
        System.out.printf("%10d\n", i / j);
        System.out.printf("%10.5f", a / j);
    }
}
```

Ausgabe

```
         0
   0,66667
```

Bei der Ganzzahldivision werden die Nachkommastellen *abgeschnitten*, was gelegentlich durchaus erwünscht ist. Im Zusammenhang mit dem Über- bzw. Unterlauf (siehe Abschnitt 3.6) werden Sie noch weitere Unterschiede zwischen Ganzzahl- und Gleitkommaarithmetik kennenlernen.

Trifft ein arithmetischer Operator auf Argumente mit *unterschiedlichen* Datentypen, dann findet vor der Berechnung automatisch eine **erweiternde Typanpassung** statt, bei der z. B. ein ganzzahliges Argument in einen Gleitkommatyp gewandelt wird (siehe Abschnitt 3.5.7.1 zur automatischen Typanpassung). Im obigen Beispielprogramm trifft der Divisionsoperator im Ausdruck

```
a / j
```

auf ein **double**- und ein **int**-Argument. In dieser Situation wird der **int**-Wert in den „größeren“ Typ **double** gewandet, bevor schließlich die Gleitkommaarithmetik zum Einsatz kommt.

Wie der vom Compiler gewählte Arithmetiktyp und der Ergebnisdatentyp von den Datentypen der Argumente abhängen, ist der folgenden Tabelle zu entnehmen:

<table>
<tr><th>Datentypen der Operanden</th><th>Verwendete Arithmetik</th><th>Datentyp des Ergebniswertes</th></tr>
<tr><td>Beide Operanden haben den Typ byte, short, char oder int (nicht unbedingt denselben).</td><td rowspan="2">Ganzzahlarithmetik</td><td>int</td></tr>
<tr><td>Beide Operanden haben einen integralen Typ, und mind. ein Operand hat den Datentyp long.</td><td>long</td></tr>
<tr><td>Mindestens ein Operand hat den Typ float, keiner hat den Typ double.</td><td rowspan="2">Gleitkomma-arithmetik</td><td>float</td></tr>
<tr><td>Mindestens ein Operand hat den Datentyp double.</td><td>double</td></tr>
</table>

In der nächsten Tabelle werden alle arithmetischen Operatoren beschrieben, wobei die Platzhalter *Num*, *Num1* und *Num2* für Ausdrücke mit einem numerischen Typ stehen, und *Var* für eine numerische *Variable*:

<table>
<tr><th rowspan="2">Operator</th><th rowspan="2">Bedeutung</th><th colspan="2">Beispiel</th></tr>
<tr><th>Programmfragment</th><th>Ausgabe</th></tr>
<tr><td>-Num</td><td>Vorzeichenumkehr</td><td><code>int i = 2, j = -3;
System.out.printf("%d %d",-i,-j);</code></td><td><code>-2 3</code></td></tr>
<tr><td>Num1 + Num2</td><td>Addition</td><td><code>System.out.println(2 + 3);</code></td><td><code>5</code></td></tr>
<tr><td>Num1 – Num2</td><td>Subtraktion</td><td><code>System.out.println(2.6 - 1.1);</code></td><td><code>1.5</code></td></tr>
<tr><td>Num1 * Num2</td><td>Multiplikation</td><td><code>System.out.println(4 * 5);</code></td><td><code>20</code></td></tr>
<tr><td>Num1 / Num2</td><td>Division</td><td><code>System.out.println(8.0 / 5);
System.out.println(8 / 5);</code></td><td><code>1.6
1</code></td></tr>
<tr><td>Num1 % Num2</td><td>Modulo (Divisionsrest)
Sei GAD der ganzzahlige Anteil aus dem Ergebnis der Division (Num1 / Num2). Dann ist Num1 % Num2 def. durch
Num1 - GAD · Num2</td><td><code>System.out.println(19 % 5);
System.out.println(-19 % 5.25);</code></td><td><code>4
-3.25</code></td></tr>
<tr><td>++Var
--Var</td><td>Präinkrement bzw. -dekrement
Als Argumente sind hier nur Variablen erlaubt.
++Var erhöht Var um 1 und liefert Var + 1
--Var reduz. Var um 1 und liefert Var – 1</td><td><code>int i = 4;
double a = 0.2;
System.out.println(++i + "\n" +
--a);</code></td><td><code>5
-0.8</code></td></tr>
<tr><td>Var++
Var--</td><td>Postinkrement bzw. -dekrement
Als Argumente sind hier nur Variablen erlaubt.
Var++ liefert Var und erhöht Var um 1
Var-- liefert Var und reduziert Var um 1</td><td><code>int i = 4;
System.out.println(i++ + "\n" +
i);</code></td><td><code>4
5</code></td></tr>
</table>

Bei den Inkrement- bzw. Dekrementoperatoren ist zu beachten, dass sie *zwei* Effekte haben:

- Der Wert des Ausdrucks wird ermittelt, wozu das Argument auszulesen ist.
- Die als Argument fungierende numerische Variable wird vor oder nach dem Auslesen verändert. Wegen dieses **Nebeneffekts** sind Inkrement- bzw. Dekrementausdrücke im Unterschied zu den sonstigen arithmetischen Ausdrücken bereits vollständige *Anweisungen* (vgl. Abschnitt 3.7.1), wenn man ein Semikolon dahinter setzt, z. B.:

Quellcode	Ausgabe

<pre>class Prog {
    public static void main(String[] args) {
        int i = 12;
        i++;
        System.out.println(i);
    }
}</pre>
	13

Ein In- bzw. Dekrementoperator erhöht bzw. vermindert durch seinen Nebeneffekt den Wert einer Variablen um 1 und bietet für diese oft benötigte Operation eine vereinfachte Schreibweise. So ist z. B. die folgende Anweisung

```
j = ++i;
```

mit den beiden **int**-Variablen `i` und `j` äquivalent zu:

```
i = i + 1;
j = i;
```

Für den eventuell bei manchen Lesern noch wenig bekannten Modulo-Operator gibt es viele sinnvolle Anwendungen, z. B.:

- Man kann für eine ganze Zahl bequem feststellen, ob sie gerade (durch 2 teilbar) ist. Dazu prüft man, ob der Rest aus der Division durch 2 gleich 0 ist:

Quellcode-Fragment	Ausgabe
`int i = 19;` `System.out.println(i % 2 == 0);`	`false`

- Man kann bei einer Gleitkommazahl den gebrochenen Anteil ermitteln bzw. abspalten:

Quellcode-Fragment	Ausgabe

`double a = 7.124824;`
`double rest = a % 1.0;`
`double ganz = a - rest;`
`System.out.printf("%f = %1.0f + %f", a, ganz, rest);`
	`7,124824 = 7 + 0,124824`

Der Modulo-Operator wird meist auf zwei ganzzahlige Argumente angewendet, sodass nach der Tabelle auf Seite 138 auch das Ergebnis einen ganzzahligen Typ besitzt. Wie der zweite Punkt in der letzten Aufzählung zeigt, kann die Modulo-Operation aber auch auf Gleitkommaargumente angewendet werden, wobei ein Ergebnis mit Gleitkommatyp resultiert.

3.5.2 Methodenaufrufe

Obwohl Ihnen eine gründliche Behandlung der Methoden noch bevorsteht, haben Sie doch schon einige Erfahrungen mit diesen Handlungskompetenzen von Klassen bzw. Objekten gesammelt:

- Die Arbeitsweise einer Methode kann von Argumenten (Parametern) abhängen.
- Viele Methoden liefern ein Ergebnis an den Aufrufer. Die im Abschnitt 3.4.1 vorgestellte Methode `Simput.gint()` liefert z. B. einen **int**-Wert. Bei der Methodendefinition ist der Datentyp der Rückgabe anzugeben (siehe Syntaxdiagramm im Abschnitt 3.1.3.2). Liefert eine Methode dem Aufrufer *kein* Ergebnis, ist in der Definition der Pseudo-Rückgabetyp **void** anzugeben.
- Neben der Wertrückgabe hat ein Methodenaufruf oft weitere Effekte, z. B. auf die Merkmalsausprägungen des handelnden Objekts oder auf die Konsolenausgabe.

In syntaktischer Hinsicht ist festzuhalten, dass ein Methodenaufruf einen **Ausdruck** darstellt, wobei seine Rückgabe den Datentyp und den Wert des Ausdrucks bestimmt.

Bei passendem Rückgabetyp darf ein Methodenaufruf auch als Argument für komplexere Ausdrücke oder für Methodenaufrufe verwendet werden (siehe Abschnitt 4.3.1.2). Bei einer Methode ohne Rückgabewert resultiert ein Ausdruck vom Typ **void**, der nicht als Argument für Operatoren oder andere Methoden taugt.

Ein Methodenaufruf mit angehängtem Semikolon stellt eine **Anweisung** dar (vgl. Abschnitt 3.7), was Sie z. B. bei den zahlreichen Einsätzen der statischen Methode **println()** in unseren Beispielprogrammen beobachten konnten.

Mit den im Abschnitt 3.5.1 beschriebenen arithmetischen Operatoren lassen sich nur elementare mathematische Probleme lösen. Darüber hinaus stellt Java eine große Zahl mathematischer Standardfunktionen (z. B. Potenzfunktion, Logarithmus, Wurzel, trigonometrische Funktionen) über

statische Methoden der Klasse **Math** im API-Paket **java.lang** (ab Java 9 im Modul **java.base**) zur Verfügung.[1] Im folgenden Programm wird die Methode **pow()** zur Potenzberechnung genutzt:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        System.out.println(Math.pow(2, 3));`
`    }`
`}`
	`8.0`

3.5.3 Vergleichsoperatoren

Durch Anwendung eines *Vergleichsoperators* auf zwei komparable (miteinander vergleichbare) Argumentausdrücke entsteht ein **Vergleich**. Dies ist ein einfacher **logischer Ausdruck** (vgl. Abschnitt 3.5.5), kann dementsprechend die booleschen Werte **true** (wahr) und **false** (falsch) annehmen und eignet sich dazu, eine *Bedingung* zu formulieren, z. B.:

```
if (arg > 0)
    System.out.println(Math.log(arg));
```

In der folgenden Tabelle mit den von Java unterstützten Vergleichsoperatoren stehen

- *Expr1* und *Expr2* für komparable Ausdrücke
- *Num1* und *Num2* für numerische Ausdrücke (mit Datentyp **byte**, **short**, **int**, **long**, **char**, **float** oder **double**)

Operator	Bedeutung	Beispiel	
		Programmfragment	Ausgabe
Expr1 == *Expr2*	**Gleichheit**	`System.out.println(2 == 3);`	`false`
Expr1 != *Expr2*	**Ungleichheit**	`System.out.println(2 != 3);`	`true`
Num1 > *Num2*	**größer**	`System.out.println(3 > 2);`	`true`
Num1 < *Num2*	**kleiner**	`System.out.println(3 < 2);`	`false`
Num1 >= *Num2*	**größer oder gleich**	`System.out.println(3 >= 3);`	`true`
Num1 <= *Num2*	**kleiner oder gleich**	`System.out.println(3 <= 2);`	`false`

Achten Sie unbedingt darauf, dass der Identitätsoperator durch **zwei** „=“ - Zeichen ausgedrückt wird. Ein nicht ganz seltener Java-Programmierfehler besteht darin, beim Identitätsoperator nur *ein* Gleichheitszeichen zu schreiben. Dabei muss nicht unbedingt ein harmloser Syntaxfehler entstehen, der nach dem Studium einer Compiler-Meldung leicht zu beseitigen ist, sondern es kann auch ein unangenehmer Logikfehler resultieren, also ein irreguläres Verhalten des Programms (vgl. Abschnitt 2.2.5 zur Unterscheidung von Syntax- und Logikfehlern). Im ersten **println() -** Aufruf des folgenden Beispielprogramms wird das Ergebnis eines Vergleichs auf die Konsole geschrieben:[2]

[1] Mit den Paketen und Modulen der Standardbibliothek werden wir uns später ausführlich beschäftigen. An dieser Stelle dient die Angabe der Paket- und Modulzugehörigkeit dazu, eine Klasse eindeutig zu identifizieren und die Standardbibliothek allmählich kennenzulernen. Das Paket **java.lang** wird im Unterschied zu allen anderen API-Paketen automatisch in jede Quellcodedatei importiert.

[2] Wir wissen schon aus Abschnitt 3.2, dass **println**() einen beliebigen Ausdruck verarbeiten kann, wobei automatisch eine Zeichenfolgen-Repräsentation erstellt wird.

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        int i = 1;`
`        System.out.println(i == 2);`
`        System.out.println(i);`
`    }`
`}`
	`false`
	`1`

Nach dem Entfernen eines Gleichheitszeichens wird aus dem logischen Ausdruck ein *Wertzuweisungsausdruck* (siehe Abschnitt 3.5.8) mit dem Datentyp **int** und dem Wert 2:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        int i = 1;`
`        System.out.println(i = 2);`
`        System.out.println(i);`
`    }`
`}`
	`2`
	`2`

Der Fehler verändert nicht nur den Typ des Ausdrucks, sondern auch den Wert der Variablen `i`, was im weiteren Verlauf eines Programms unangenehme Folgen haben kann.

3.5.4 Identitätsprüfung bei Gleitkommawerten

Bei den *binären* Gleitkommatypen (**float** und **double**) sind simple Identitätstests wegen technisch bedingter Abweichungen von der reinen Mathematik unbedingt zu unterlassen, z. B.:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        final double epsilon = 1.0e-14;`
`        double d1 = 10.0 - 9.9;`
`        double d2 = 0.1;`
`        System.out.println(d1 == d2);`
`        System.out.println(Math.abs((d1 - d2)/d1) < epsilon);`
`    }`
`}`
	`false`
	`true`

Der Vergleich

```
10.0 - 9.9 == 0.1
```

führt trotz Datentyp **double** (mit mindestens 15 signifikanten Dezimalstellen) zum Ergebnis **false**. Wenn man die im Abschnitt 3.3.7.1 beschriebenen Genauigkeitsprobleme bei der Speicherung von binären Gleitkommazahlen berücksichtigt, ist das Vergleichsergebnis durchaus *nicht* überraschend. Im Kern besteht das Problem darin, dass mit der binären Gleitkommatechnik auch relativ „glatte" rationale Zahlen (wie z. B. 9,9) nicht exakt gespeichert werden können. Im zwischengespeicherten Berechnungsergebnis 10,0 - 9,9 steckt ein anderer Fehler als im Speicherabbild der Zahl 0,1. Weil die Vergleichspartner nicht Bit für Bit identisch sind, meldet der Identitätsoperator das Ergebnis **false**.

Mit den Objekten der im Abschnitt 3.3.7 vorgestellten und insbesondere für Anwendungen im Bereich der Finanzmathematik empfohlenen Klasse **BigDecimal** gibt es *keine* Probleme bei der Speichergenauigkeit und bei Identitätsvergleichen (vgl. Mitran et al. 2008), z. B.:

Quellcode	Ausgabe

`import java.math.*;`
`class Prog {`
`    public static void main(String[] args) {`
`        BigDecimal bd1 = new BigDecimal("10.0");`
`        BigDecimal bd2 = new BigDecimal("9.9");`
`        BigDecimal bd3 = new BigDecimal("0.1");`
`        System.out.println(bd3.equals(bd1.subtract(bd2)));`
`    }`
`}`
	`true`

Allerdings ist ein erhöhter Speicher- und Zeitaufwand in Kauf zu nehmen. Damit nicht doch ungenaue Werte in die Berechnungen einfließen, sollte bei der Klasse **BigDecimal** der Konstruktor mit **String**-Parameter (siehe Beispiel) gegenüber dem Konstruktor mit **double**-Parameter bevorzugt werden (vgl. Abschnitt 3.3.7.2).

Den etwas anstrengenden Rest des Abschnitts kann überspringen, wer aktuell keinen Algorithmus mit auf Identität zu prüfenden **double**-Werten zu implementieren hat.

Um eine praxistaugliche Identitätsbeurteilung von **double**-Werten zu erhalten, sollte eine an der Rechen- bzw. Speichergenauigkeit orientierte **Unterschiedlichkeitsschwelle** verwendet werden. Nach diesem Vorschlag werden zwei **normalisierte** (also insbesondere von null verschiedene) **double**-Werte d_1 und d_2 (vgl. Abschnitt 3.3.7.1) dann als numerisch identisch betrachtet, wenn der relative Abweichungsbetrag kleiner als $1{,}0 \cdot 10^{-14}$ ist:

$$\left| \frac{d_1 - d_2}{d_1} \right| < 1{,}0 \cdot 10^{-14}$$

Die Vergabe der d_1-Rolle, also die Wahl des Nenners, ist beliebig. Um das Verfahren vollständig festzulegen, wird die Verwendung der betragsmäßig größeren Zahl vorgeschlagen.

Ein Vorschlag zur Definition der *numerischen Identität* von zwei **double**-Werten muss die *relative* Differenz zugrunde legen, weil die technisch bedingten Mantissen-Fehler bei zwei **double**-Variablen mit eigentlich identischem Wert in Abhängigkeit vom Exponenten zu sehr unterschiedlichen Gesamtfehlern führen können. Vom gelegentlich anzutreffenden Vorschlag, die betragsmäßige Differenz

$$|d_1 - d_2|$$

mit einer Schwelle zu vergleichen, ist daher abzuraten. Dieses Verfahren ist (bei geeignet gewählter Schwelle) nur tauglich für Zahlen in einem engen Größenbereich. Bei einer Änderung der Größenordnung muss die Schwelle angepasst werden.

Zu einer Schwelle für die relative Abweichung $\left| \frac{d_1 - d_2}{d_1} \right|$ gelangt man durch Betrachtung von zwei normalisierten **double**-Variablen d_1 und d_2, die bis auf ihre durch begrenzte Speicher- und Rechengenauigkeit bedingten Mantissenfehler e_1 bzw. e_2 denselben Wert $(1 + m)\, 2^k$ enthalten:

$$d_1 = (1 + m + e_1)\, 2^k \quad \text{und} \quad d_2 = (1 + m + e_2)\, 2^k$$

Bei einem normalisierten **double**-Wert (mit 52 Mantissen-Bits) kann aufgrund der begrenzten Speichergenauigkeit als maximaler absoluter Mantissenfehler ε der halbe Abstand zwischen zwei benachbarten Mantissenwerten auftreten:

$$\varepsilon = 2^{-53} \approx 1{,}1 \cdot 10^{-16}$$

Für den Betrag der technisch bedingten relativen Abweichung von zwei eigentlich identischen normalisierten Werten (mit einer Mantisse im Intervall [1, 2)) gilt die Abschätzung:

$$\left|\frac{d_1 - d_2}{d_1}\right| = \left|\frac{e_1 - e_2}{1 + m + e_1}\right| \leq \frac{|e_1| + |e_2|}{|1 + m + e_1|} \leq \frac{2 \cdot \varepsilon}{|1 + m + e_1|} \leq 2 \cdot \varepsilon \quad (\text{wegen}\,(1 + m + e_1) \in [1, 2))$$

Die oben vorgeschlagene Schwelle $1{,}0 \cdot 10^{-14}$ berücksichtigt über den Speicherfehler hinaus auch noch eingeflossene Rechnungsungenauigkeiten. Mit welcher Fehlerkumulation bzw. -verstärkung zu rechnen ist, hängt vom konkreten Algorithmus ab, sodass die Unterschiedlichkeitsschwelle eventuell angehoben werden muss. Immerhin hängt sie (anders als bei einem Kriterium auf Basis der einfachen Differenz $|d_1 - d_2|$) nicht von der Größenordnung der Zahlen ab.

An der vorgeschlagenen Identitätsbeurteilung mit Hilfe einer Schwelle für den relativen Abweichungsbetrag ist u.a. zu bemängeln, dass eine Verallgemeinerung für die mit einer geringeren relativen Genauigkeit gespeicherten *denormalisierten* Werte (Betrag kleiner als 2^{-1022} beim Typ **double**, siehe Abschnitt 3.3.7.1) benötigt wird.

Dass die definierte numerische Identität nicht transitiv ist, muss hingenommen werden. Für drei **double**-Werte *a*, *b* und *c* kann also das folgende Ergebnismuster auftreten:

- *a* numerisch identisch mit *b*
- *b* numerisch identisch mit *c*
- *a* **nicht** numerisch identisch mit *c*

Für den Vergleich einer **double**-Zahl *a* mit dem Wert 0.0 ist eine Schwelle für die *absolute* Abweichung (statt der relativen) sinnvoll, z. B.:

$$|a| < 1{,}0 \cdot 10^{-14}$$

Die besprochenen Genauigkeitsprobleme sind auch bei den gerichteten Vergleichen (<, <=, >, >=) relevant.

Bei vielen naturwissenschaftlichen oder technischen Problemen ist es generell wenig sinnvoll, zwei Größen auf exakte Übereinstimmung zu testen, weil z. B. schon aufgrund von Messungenauigkeiten eine Abweichung von der theoretischen Identität zu erwarten ist. Bei Verwendung einer anwendungslogisch gebotenen Unterschiedsschwelle dürften die technischen Beschränkungen der Gleitkommatypen keine große Rolle mehr spielen. Präzisere Aussagen zur Computer-Arithmetik finden sich z. B. bei Strey (2005).

3.5.5 Logische Operatoren

Aus dem Abschnitt 3.5.3 wissen wir, dass jeder Vergleich (z. B. `arg > 0`) bereits ein logischer Ausdruck ist, also die Werte **true** und **false** annehmen kann. Durch Anwendung von logischen Operatoren (Negation, UND, ODER) auf bereits vorhandene logische Ausdrücke kann man neue, komplexere logische Ausdrücke erstellen. Die Wirkungsweise der in Java unterstützten logischen Operatoren wird anschließend in **Wahrheitstafeln** beschrieben, wobei die Platzhalter *LA*, *LA1* und *LA2* für logische Ausdrücke stehen.

Um einen logischen Ausdruck *LA* zu *negieren*, also die Wahrheitswerte **true** und **false** zu vertauschen), wendet man den unären logischen Operator **!** auf *LA* an:

Argument *LA*	**Negation** **!***LA*
true	**false**
false	**true**

Mit den anschließend beschriebenen *binären* logischen Operatoren erstellt man aus zwei Argumentausdrücken einen komplexeren logischen Ausdruck:

<table>
<tr><th>Argument 1
LA1</th><th>Argument 2
LA2</th><th>Logisches UND
LA1 && LA2
LA1 & LA2</th><th>Logisches ODER
LA1 || LA2
LA1 | LA2</th><th>Exklusives ODER
LA1 ^ LA2</th></tr>
<tr><td>true</td><td>true</td><td>true</td><td>true</td><td>false</td></tr>
<tr><td>true</td><td>false</td><td>false</td><td>true</td><td>true</td></tr>
<tr><td>false</td><td>true</td><td>false</td><td>true</td><td>true</td></tr>
<tr><td>false</td><td>false</td><td>false</td><td>false</td><td>false</td></tr>
</table>

Es folgt eine Tabelle mit Erläuterungen und Beispielen zu den logischen Operatoren:

<table>
<tr><th rowspan="2">Operator</th><th rowspan="2">Bedeutung</th><th colspan="2">Beispiel</th></tr>
<tr><th>Programmfragment</th><th>Ausgabe</th></tr>
<tr><td>!LA</td><td>Negation
Der Wahrheitswert wird durch sein Gegenteil ersetzt.</td><td>boolean erg = true;
System.out.println(!erg);</td><td>false</td></tr>
<tr><td>LA1 && LA2</td><td>Logisches UND mit bedingter Auswertung
LA1 && LA2 ist genau dann wahr, wenn beide Argumente wahr sind. Ist LA1 falsch, wird LA2 nicht ausgewertet.</td><td>int i = 3;
boolean erg = false && i++ > 3;
System.out.println(erg + "\n"+i);

erg = true && i++ > 3;
System.out.println(erg + "\n"+i);</td><td>false
3

false
4</td></tr>
<tr><td>LA1 & LA2</td><td>Logisches UND mit unbedingter Auswertung
LA1 & LA2 ist genau dann wahr, wenn beide Argumente wahr sind. Es werden auf jeden Fall beide Ausdrücke ausgewertet.</td><td>int i = 3;
boolean erg = false & i++ > 3;
System.out.println(erg + "\n"+i);</td><td>false
4</td></tr>
<tr><td>LA1 || LA2</td><td>Logisches ODER mit bedingter Auswertung
LA1 || LA2 ist genau dann wahr, wenn mindestens ein Argument wahr ist. Ist LA1 wahr, wird LA2 nicht ausgewertet.</td><td>int i = 3;
boolean erg = true || i++ == 3;
System.out.println(erg + "\n"+i);

erg = false || i++ == 3;
System.out.println(erg + "\n"+i);</td><td>true
3

true
4</td></tr>
</table>

<table>
<tr><th rowspan="2">Operator</th><th rowspan="2">Bedeutung</th><th colspan="2">Beispiel</th></tr>
<tr><th>Programmfragment</th><th>Ausgabe</th></tr>
<tr><td>LA1 | LA2</td><td>Logisches ODER mit unbedingter Auswertung
LA1 | LA2 ist genau dann wahr, wenn mindestens ein Argument wahr ist. Es werden auf jeden Fall beide Ausdrücke ausgewertet.</td><td><code>int i = 3;</code>
<code>boolean erg = true | i++ == 3;</code>
<code>System.out.println(erg + "\n"+i);</code></td><td><code>true</code>
<code>4</code></td></tr>
<tr><td>LA1 ^ LA2</td><td>Exklusives logisches ODER
LA1 ^ LA2 ist genau dann wahr, wenn genau ein Argument wahr ist, wenn also die Argumente verschiedene Wahrheitswerte haben.</td><td><code>boolean erg = true ^ true;</code>
<code>System.out.println(erg);</code></td><td><code>false</code></td></tr>
</table>

Der Unterschied zwischen den beiden logischen UND-Operatoren **&&** und **&** bzw. zwischen den beiden logischen ODER-Operatoren || und | ist für Einsteiger vielleicht wenig beeindruckend, weil man spontan den nicht ausgewerteten logischen Ausdrücken keine Bedeutung beimisst. Allerdings ist es in Java nicht unüblich, „Nebeneffekte“ in einen logischen Ausdruck einzubauen, z. B.

```
bv & i++ > 3
```

Hier erhöht der Postinkrementoperator beim Auswerten des rechten **&**-Arguments den Wert der Variablen `i`. Eine solche Auswertung wird jedoch in der folgenden Variante des Beispiels (mit **&&**-Operator) unterlassen, wenn bereits nach Auswertung des linken **&&**-Arguments das Gesamtergebnis **false** feststeht:

```
bv && i++ > 3
```

Das vom Programmierer nicht erwartete Ausbleiben einer Auswertung (z. B. bei `i++`) kann erhebliche Auswirkungen auf die Programmausführung haben.

Dank der beim Operator **&&** realisierten bedingten Auswertung kann man sich im rechten Operanden darauf verlassen, dass der linke Operand den Wert **true** besitzt, was im folgenden Beispiel ausgenutzt wird. Dort prüft der linke Operand die Existenz und der rechte Operand die Länge einer Zeichenfolge:

```
if(str != null  &&  str.length() < 10) {...}
```

Wenn die Referenzvariable `str` vom Typ der Klasse **String** keine Objektadresse enthält, darf der rechte Ausdruck nicht ausgewertet werden, weil eine Längenanfrage an ein nicht existentes Objekt zu einem Laufzeitfehler führen würde.

Mit der Entscheidung, grundsätzlich die unbedingte Operatorvariante zu verwenden, verzichtet man auf die eben beschriebene Option, im rechten Ausdruck den Wert **true** des linken Ausdrucks voraussetzen zu können, und man nimmt (mehr oder weniger relevante) Leistungseinbußen durch überflüssige Auswertungen des rechten Ausdrucks in Kauf. Eher empfehlenswert ist der Verzicht auf Nebeneffekt-Konstruktionen im Zusammenhang mit bedingt arbeitenden Operatoren.

Wie der Tabelle auf Seite 157 zu entnehmen ist, unterscheiden sich die beiden UND-Operatoren **&&** und **&** bzw. die beiden ODER-Operatoren || und | auch hinsichtlich der Bindungskraft auf Operanden (Auswertungspriorität).

Die bedingte Auswertung wird gelegentlich als *Kurzschlussauswertung* bezeichnet (engl.: *short-circuiting*).

Um die Verwirrung noch ein wenig zu steigern, werden die Zeichen **&** und | auch für *bitorientierte* Operatoren verwendet (siehe Abschnitt 3.5.6). Diese Operatoren erwarten zwei *integrale* Argumente (z. B. mit dem Datentyp **int**), während die logischen Operatoren den Datentyp **boolean** voraus-

setzen. Folglich kann der Compiler erkennen, ob ein logischer oder ein bitorientierter Operator gemeint ist.

3.5.6 Vertiefung: Bitorientierte Operatoren

Über unseren momentanen Bedarf hinausgehend bietet Java einige Operatoren zur bitweisen Analyse und Manipulation von Variableninhalten. Statt einer systematischen Darstellung der verschiedenen Operatoren (siehe z. B. den Trail *Learning the Java Language* in den *Java Tutorials*, Oracle 2019) beschränken wir uns auf ein Beispielprogramm, das zudem nützliche Einblicke in die Speicherung von **char**-Werten im Arbeitsspeicher eines Computers erlaubt. Allerdings sind Beispiel und zugehörige Erläuterungen mit einigen technischen Details belastet. Wenn Ihnen der Sinn momentan nicht danach steht, können Sie den aktuellen Abschnitt ohne Sorge um den weiteren Kurserfolg an dieser Stelle verlassen.

Das folgende Programm `CharBits` liefert die Unicode-Kodierung zu einem vom Benutzer erfragten Zeichen Bit für Bit. Dabei kommt die statische Methode `gchar()` aus der im Abschnitt 3.4 beschriebenen Klasse `Simput` zum Einsatz, welche das erste Element einer vom Benutzer eingetippten und mit **Enter** quittierten Zeichenfolge abliefert. Außerdem wird mit der **for**-Schleife eine Wiederholungsanweisung verwendet, die erst im Abschnitt 3.7.3.1 offiziell vorgestellt wird. Im Beispiel startet die Indexvariable `i` mit dem Wert 15, der am Ende jedes Schleifendurchgangs um eins dekrementiert wird (`i--`). Ob es zum nächsten Schleifendurchgang kommt, hängt von der Fortsetzungsbedingung ab (`i >= 0`):

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class CharBits {
   public static void main(String[] args) {
      char cbit;
      System.out.print("Zeichen:  ");
      cbit = Simput.gchar();
      System.out.print("Unicode:  ");
      for(int i = 15; i >= 0; i--) {
         if ((1 << i & cbit) != 0)
            System.out.print("1");
         else
            System.out.print("0");
      }
      System.out.println("\nint-Wert: " + (int)cbit);
   }
}
```

</td><td>

```
Zeichen:  x
Unicode:  0000000001111000
int-Wert: 120
```

</td></tr>
</table>

Der **Links-Shift-Operator** << im Ausdruck

```
1 << i
```

verschiebt die Bits in der binären Repräsentation der Ganzzahl Eins um `i` Stellen nach links, wobei am linken Rand `i` Stellen verworfen werden, und auf der rechten Seite `i` Nullen nachrücken. Von den 32 Bits, die ein **int**-Wert insgesamt belegt (siehe Abschnitt 3.3.6), interessieren im Augenblick nur die rechten 16. Bei der 1 erhalten wir:

```
0000000000000001
```

Im 10. Schleifendurchgang (`i = 6`) geht dieses Muster z. B. über in:

```
0000000001000000
```

Nach dem Links-Shift - kommt der **bitweise UND-Operator** zum Einsatz:

```
1 << i & cbit
```

Das Operatorzeichen **&** wird in Java leider in doppelter Bedeutung verwendet: Wenn beide Argumente vom Typ **boolean** sind, wird **&** als *logischer* Operator interpretiert (siehe Abschnitt 3.5.5). Sind jedoch (wie im vorliegenden Fall) beide Argumente von integralem Typ, was auch für den Typ **char** zutrifft, dann wird **&** als UND-Operator für Bits aufgefasst. Er erzeugt dann ein Bitmuster, das genau dann an der Stelle *k* eine 1 enthält, wenn *beide* Argumentmuster an dieser Stelle eine 1 besitzen und anderenfalls eine 0. Hat in einem Programmablauf die **char**-Variable `cbit` z. B. den Wert `'x'` erhalten (dezimale Unicode-Zeichensatznummer 120), dann ist dieses Bitmuster

```
0000000001111000
```

im Spiel, und `1 << i & cbit` liefert z. B. bei `i = 6` das Muster:

```
0000000001000000
```

Der von `1 << i & cbit` erzeugte Wert hat den Typ **int** und kann daher mit dem **int**-Literal `0` verglichen werden:[1]

```
(1 << i & cbit) != 0
```

Dieser logische Ausdruck wird bei einem Schleifendurchgang genau dann wahr, wenn das zum aktuellen `i`-Wert korrespondierende Bit in der Binärdarstellung des untersuchten Zeichens den Wert 1 hat.

3.5.7 Typumwandlung (Casting) bei primitiven Datentypen

Wie Sie aus Abschnitt 3.3.1 wissen, ist in Java der Datentyp einer Variablen unveränderlich, und dieses Prinzip wird im aktuellen Abschnitt keineswegs aufgeweicht. Es gibt aber gelegentlich einen Grund dafür, z. B. den Inhalt einer **int**-Variablen in eine **double**-Variable zu übertragen. Aufgrund der abweichenden Speichertechniken ist dann eine Typanpassung fällig. Das geschieht manchmal automatisch durch eine Initiative des Compilers, kann aber auch vom Programmierer explizit angefordert werden.

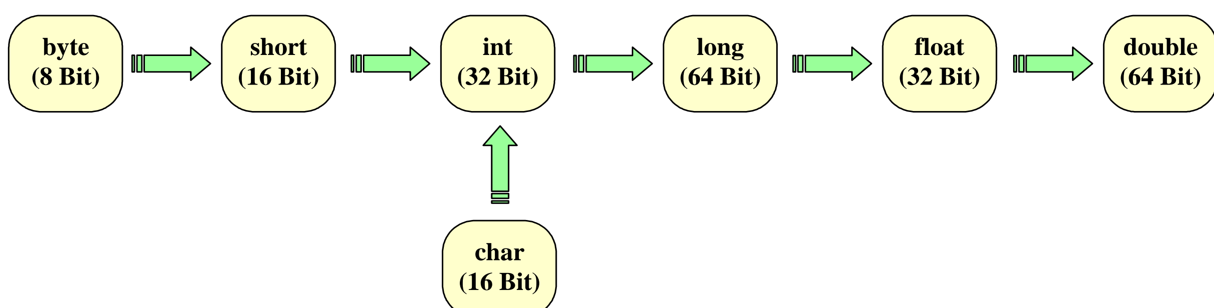
3.5.7.1 Automatische erweiternde Typanpassung

Beim der Auswertung des Ausdrucks

```
2.0 / 7
```

trifft der Divisionsoperator auf ein **double**- und ein **int**-Argument, sodass nach der Tabelle im Abschnitt 3.5.1 (Seite 138) die Gleitkommaarithmetik zum Einsatz kommt. Dazu wird für das **int**-Argument eine automatische (implizite) Wandlung in den Datentyp **double** vorgenommen.

Java nimmt bei Bedarf für primitive Datentypen die folgenden **erweiternden Typanpassungen** automatisch vor:



Weil eine **char**-Variable die Unicode-Nummer eines Zeichens speichert, macht die Konvertierung in numerische Typen kein Problem, z. B.:

[1] Die runden Klammern sind erforderlich, um die korrekte Auswertungsreihenfolge zu erreichen (siehe Abschnitt 3.5.10).

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        System.out.printf("x/2    = %5d", 'x'/2);`
`    }`
`}`
	`x/2    =    60`

Noch eine Randnotiz zur impliziten Typanpassung bei numerischen Literalen (vgl. Fußnote auf Seite **Fehler! Textmarke nicht definiert.**): Während sich Java-Compiler weigern, ein **double**-Literal in einer **float**-Variablen zu speichern, erlauben sie z. B. das Speichern eines **int**-Literals in einer Variablen vom Typ **byte** (Ganzzahltyp mit 8 Bits), sofern der Wertebereich dieses Typs nicht verlassen wird, z. B.:

```
float f = 3.14;
byte b = 13;
```

3.5.7.2 Explizite Typumwandlung

Gelegentlich gibt es gute Gründe dafür, über den sogenannten **Casting-Operator** eine *explizite* Typumwandlung zu erzwingen. Im nächsten Beispielprogramm wird mit

```
(int)'x'
```

die **int**-erpretation des (aus Abschnitt 3.5.6 bekannten) Bitmusters zum kleinen „x" vorgenommen, damit Sie nachvollziehen können, warum das Beispielprogramm im vorigen Abschnitt beim „Halbieren" dieses Zeichens auf den Wert 60 kam:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        System.out.println((int)'x');`
 	
`        double a = 3.7615926;`
`        System.out.println((int)a);`
`        System.out.println((int)(a + 0.5));`
 	
`        a = 7294452388.13;`
`        System.out.println((int)a);`
`    }`
`}`
	`120`
	`3`
	`4`
	`2147483647`

Manchmal ist es erforderlich, einen Gleitkommawert in eine Ganzzahl zu wandeln, z. B. weil bei einem Methodenaufruf für einen Parameter ein ganzzahliger Datentyp benötigt wird. Dabei werden die Nachkommastellen abgeschnitten. Soll auf die nächstgelegene ganze Zahl gerundet werden, addiert man vor der Typumwandlung 0,5 zum Gleitkommawert.

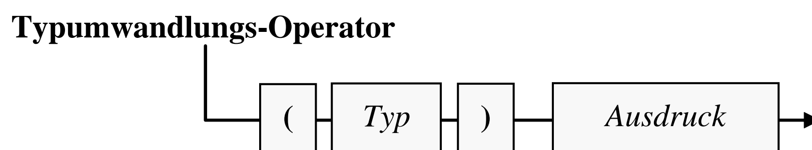
Es ist auf jeden Fall zu beachten, dass dabei eine **einschränkende Konvertierung** stattfindet, und dass die zu erwartende Gleitkommazahl im Wertebereich des Ganzzahltyps liegen muss. Wie die letzte Ausgabe zeigt, sind kapitale Programmierfehler möglich, wenn die Wertebereiche der beteiligten Datentypen nicht beachtet werden, und bei der Zielvariablen ein Überlauf auftritt (vgl. Abschnitt 3.6.1). So soll die Explosion der europäischen Rakete Ariane 5 am 4. Juni 1996 (Schaden: ca. 500 Millionen Dollar)

durch die Konvertierung eines **double**-Werts (mögliches Maximum: $1{,}7976931348623157 \cdot 10^{308}$) in einen **short**-Wert (mögliches Maximum: 2^{15}-1 = 32767) verursacht worden sein. Die kritische Typumwandlung hatte bei der langsameren Rakete Ariane 4 noch keine Probleme gemacht. Es zeigt sich, dass profunde Kenntnisse über elementare Sprachelemente unverzichtbar sind für eine erfolgreiche Raketenforschung und -entwicklung.

Später wird sich zeigen, dass auch zwischen Referenztypen gelegentlich eine explizite Wandlung erforderlich ist.

Welche expliziten Typkonvertierungen in Java erlaubt sind, ist der Sprachspezifikation zu entnehmen (Gosling et al. 2019, Abschnitt 5.1).

Die Java-Syntax zur expliziten Typumwandlung:



Die Wandlung eines Gleitkommawerts in einen Ganzzahlwert ist übrigens *keine* sinnvolle Technik, um die Nachkommastellen zu entfernen. Dazu kann man den Modulo-Operator verwenden (vgl. Abschnitt 3.5.1), ohne ein Wertebereichsproblem befürchten zu müssen, z. B.:

Quellcode	Ausgabe

<pre>class Prog {
    public static void main(String[] args) {
        double a = 85347483648.13, b;
        int i = (int) a;
        b = a - a%1;
        System.out.printf("%15.2f\n%12d\n%15.2f", a, i, b);
    }
}</pre>
	<pre>85347483648,13
	 2147483647
	85347483648,00</pre>

3.5.8 Zuweisungsoperatoren

Bei den ersten Erläuterungen zu Wertzuweisungen (vgl. Abschnitt 3.3.8) blieb aus didaktischen Gründen unerwähnt, dass in Java eine Wertzuweisung als *Ausdruck* aufgefasst wird, dass wir es also mit dem binären (zweistelligen) Operator „=“ zu tun haben, für den folgende Regeln gelten:

- Auf der linken Seite muss eine Variable stehen.
- Auf der rechten Seite muss ein Ausdruck mit kompatiblem Typ stehen.
- Der zugewiesene Wert stellt auch den Ergebniswert des Ausdrucks dar.

Wie beim Inkrement- bzw. Dekrementoperator sind auch beim Zuweisungsoperator *zwei* Effekte zu unterscheiden:

- Die als linkes Argument fungierende Variable erhält einen neuen Wert.
- Es wird ein Wert für den Ausdruck produziert.

Im folgenden Beispiel fungiert ein Zuweisungsausdruck als Parameter für einen **println() -** Methodenaufruf:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        int ivar = 13;`
`        System.out.println(ivar = 4711);`
`        System.out.println(ivar);`
`    }`
`}`
	4711
	4711

Beim Auswerten des Ausdrucks `ivar = 4711` entsteht der an **println()** zu übergebende Wert (identisch mit dem zugewiesenen Wert), *und* die Variable `ivar` wird verändert.

Selbstverständlich kann eine Zuweisung auch als Operand in einen übergeordneten Ausdruck integriert werden, z. B.:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        int i = 2, j = 4;`
`        i = j = j * i;`
`        System.out.println(i + "\n" + j);`
`    }`
`}`
	8
	8

Beim mehrfachen Auftreten des Zuweisungsoperators erfolgt eine Abarbeitung von **rechts nach links** (vgl. Tabelle im Abschnitt 3.5.10), sodass die Anweisung

```
i = j = j * i;
```

folgendermaßen ausgeführt wird:

- Weil der Multiplikationsoperator eine höhere Bindungskraft besitzt als der Zuweisungsoperator (siehe Abschnitt 3.5.10.1), wird zuerst der Ausdruck `j * i` ausgewertet, was zum Zwischenergebnis 8 (mit Datentyp **int**) führt.
- Nun wird die *rechte* Zuweisung ausgeführt. Der folgende Ausdruck mit Wert 8 und Typ **int**

  ```
  j = 8
  ```

  verschafft der Variablen `j` einen neuen Wert.
- In der zweiten Zuweisung (bei Betrachtung von rechts nach links) wird der Wert des Ausdrucks `j = 8` an die Variable `i` übergeben.

Anweisungen der Art

```
i = j = k;
```

sind in Java - Programmen gelegentlich anzutreffen, weil Schreibaufwand gespart wird im Vergleich zur Alternative

```
j = k;
i = k;
```

Wie wir seit Abschnitt 3.3.8 wissen, stellt ein Zuweisungsausdruck bereits eine vollständige **Anweisung** dar, sobald man ein Semikolon dahinter setzt. Dies gilt auch für die Prä- und Postinkre-

mentausdrücke (vgl. Abschnitt 3.5.1) sowie für Methodenaufrufe, jedoch *nicht* für die anderen Ausdrücke, die im Abschnitt 3.5 vorgestellt werden.

Für die häufig benötigten Zuweisungen nach dem Muster

```
j = j * i;
```

(eine Variable erhält einen neuen Wert, an dessen Konstruktion sie selbst mitwirkt) bietet Java spezielle Zuweisungsoperatoren für Schreibfaule, die gelegentlich auch als **Aktualisierungsoperatoren** oder als *zusammengesetzte Zuweisungsoperatoren* (engl.: *compound assignment operators*) bezeichnet werden. In der folgenden Tabelle steht *Var* für eine numerische Variable (mit Datentyp **byte**, **short**, **int**, **long**, **char**, **float** oder **double**) und *Expr* für einen numerischen Ausdruck:

Operator	Bedeutung	Beispiel	
		Programmfragment	**Neuer Wert von `i`**
Var += *Expr*	*Var* erhält den neuen Wert *Var* + *Expr*.	`int i = 2;` `i += 3;`	`5`
Var -= *Expr*	*Var* erhält den neuen Wert *Var* – *Expr*.	`int i = 10, j = 3;` `i -= j * j;`	`1`
Var *= *Expr*	*Var* erhält den neuen Wert *Var* * *Expr*.	`int i = 2;` `i *= 5;`	`10`
Var /= *Expr*	*Var* erhält den neuen Wert *Var* / *Expr*.	`int i = 10;` `i /= 5;`	`2`
Var %= *Expr*	*Var* erhält den neuen Wert *Var* % *Expr*.	`int i = 10;` `i %= 5;`	`0`

Es ist keine schlechte Idee, der Klarheit halber auf die Aktualisierungsoperatoren zu verzichten. In fremden Programmen (erstellt von schreibfaulen Kollegen) muss man aber mit diesen Operatoren rechnen.

Ein weiteres Argument gegen die Aktualisierungsoperatoren ist die implizit darin enthaltene Typwandlung. Während z. B. für die beiden Variablen

```
int ivar = 1;
double dvar = 3_000_000_000.0;
```

die folgende Zuweisung

```
ivar = ivar + dvar;
```

vom Compiler verhindert wird, weil der Ausdruck `(ivar + dvar)` den Typ **double** besitzt (vgl. Tabelle mit den Ergebnistypen der arithmetischen Operationen im Abschnitt 3.5.1), akzeptiert der Compiler die folgende Anweisung mit Aktualisierungsoperator:

```
ivar += dvar;
```

Es kommt zum Ganzzahlüberlauf (vgl. Abschnitt 3.6.1), und man erhält für `ivar` den ebenso sinnlosen wie gefährlichen Wert `2147483647`.

In der Java-Sprachdefinition (Gosling et al. 2019, Abschnitt 15.26.2) findet sich die folgende Erläuterung zum Verhalten des Java-Compilers, der bei Aktualisierungsoperatoren eine untypische und gefährliche Laxheit zeigt:

> A compound assignment expression of the form *`E1 op= E2`* is equivalent to *`E1 = (T) ((E1) op (E2))`*, where *`T`* is the type of *`E1`*, except that *`E1`* is evaluated only once.

Der Ausdruck `ivar += dvar` steht also für

```
ivar = (int) (ivar + dvar)
```

und enthält eine riskante einschränkende Typanpassung.

Beim Einsatz eines Aktualisierungsoperators sollte der Wertebereich des rechten Operanden keinesfalls größer sein als der Wertebereich des linken Operanden, und es ist zu bedauern, dass keine entsprechende Compiler-Regel existiert.

3.5.9 Konditionaloperator

Der **Konditionaloperator** erlaubt eine sehr kompakte Schreibweise, wenn beim neuen Wert für eine Zielvariable bedingungsabhängig zwischen zwei Ausdrücken zu entscheiden ist, z. B.

$$i = \begin{cases} i + j & \text{falls } k > 0 \\ i - j & \text{sonst} \end{cases}$$

In Java ist für diese Zuweisung mit Fallunterscheidung nur eine einzige Zeile erforderlich:

Quellcode	Ausgabe

<pre>class Prog {
    public static void main(String[] args) {
        int i = 2, j = 1, k = 7;
        i = k>0 ? i+j : i-j;
        System.out.println(i);
    }
}</pre>
	3

Eine Besonderheit des Konditionaloperators besteht darin, dass er *drei* Argumente verarbeitet, welche durch die Zeichen **?** und **:** getrennt werden:

Konditionaloperator

Ist der logische Ausdruck *wahr*, liefert der Konditionaloperator den Wert von *Ausdruck 1*, anderenfalls den Wert von *Ausdruck 2*.

Die Frage nach dem Datentyp eines Konditionalausdrucks ist etwas knifflig, und in der Java 13 - Sprachspezifikation werden zahlreiche Fälle unterschieden (Gosling et al. 2019, Abschnitt 15.25). Es liegt an Ihnen, sich auf den einfachsten und wichtigsten Fall zu beschränken: Wenn der zweite und der dritte Operand denselben Datentyp haben, ist dies auch der Datentyp des Konditionalausdrucks.

3.5.10 Auswertungsreihenfolge

Bisher haben wir zusammengesetzte Ausdrücke mit *mehreren* Operatoren und das damit verbundene Problem der Auswertungs*reihenfolge* nach Möglichkeit gemieden. Wie gleich deutlich wird, sind für Schwierigkeiten und Fehler bei der Verwendung zusammengesetzter Ausdrücke hauptverantwortlich:

- Komplexität des Ausdrucks (Anzahl der Operatoren, Schachtelungstiefe)
- Operatoren mit Nebeneffekten

Um Problemen aus dem Weg zu gehen, sollte man also eine übertriebene Komplexität vermeiden und auf Nebeneffekte weitgehend verzichten.

3.5.10.1 Regeln

In diesem Abschnitt werden die Regeln vorgestellt, nach denen der Java-Compiler einen Ausdruck mit mehreren Operatoren auswertet.

1) Runde Klammern

Wenn aus den anschließend erläuterten Regeln zur Bindungskraft und Assoziativität der beteiligten Operatoren nicht die gewünschte Operandenzuordnung bzw. Auswertungsreihenfolge resultiert, greift man mit runden Klammern steuernd ein, wobei auch eine Schachtelung erlaubt ist. Durch Klammern werden Terme zu *einem* Operanden zusammengefasst, sodass die *internen* Operationen ausgeführt sind, bevor der Klammerausdruck von einem *externen* Operator verarbeitet wird.

2) Bindungskraft (Priorität)

Steht ein Operand (ein Ausdruck) zwischen zwei Operatoren, dann wird er dem Operator mit der stärkeren Bindungskraft (siehe Tabelle im Abschnitt 3.5.10.2) zugeordnet. Mit den numerischen Variablen `a`, `b` und `c` als Operanden wird z. B. der Ausdruck

```
a + b * c
```

nach der Regel „*Punktrechnung geht vor Strichrechnung*“ interpretiert als

```
a + (b * c)
```

In der Konkurrenz um die Zuständigkeit für den Operanden `b` hat der Multiplikationsoperator Vorrang gegenüber dem Additionsoperator.

Die implizite Klammerung kann durch eine explizite Klammerung geändert werden:

```
(a + b) * c
```

3) Assoziativität (Orientierung)

Steht ein Operand zwischen zwei Operatoren mit *gleicher* Bindungskraft, dann entscheidet deren Assoziativität (Orientierung) über die Zuordnung des Operanden:

- Mit Ausnahme der Zuweisungsoperatoren sind alle binären Operatoren *links-assoziativ*. Z. B. wird

  ```
  x – y – z
  ```

  ausgewertet als

  ```
  (x – y) – z
  ```

  Diese implizite Klammerung kann durch eine explizite Klammerung geändert werden:

  ```
  x – (y – z)
  ```

- Die Zuweisungsoperatoren sind *rechts-assoziativ*. Z. B. wird

  ```
  a += b -= c = d
  ```

  ausgewertet als

  ```
  a += (b -= (c = d))
  ```

  Diese implizite Klammerung kann *nicht* durch eine explizite Klammerung geändert werden, weil der linke Operand einer Zuweisung eine Variable oder eine Eigenschaft sein muss.

In Java ist dafür gesorgt, dass Operatoren mit gleicher Bindungskraft stets auch die gleiche Assoziativität besitzen, z. B. die im letzten Beispiel enthaltenen Operatoren +=, -= und =.

Für manche Operationen gilt das mathematische Assoziativitätsgesetz, sodass die Reihenfolge der Auswertung irrelevant ist, z. B.:

$$(3 + 2) + 1 = 6 \quad = \quad 3 + (2 + 1)$$

Anderen Operationen fehlt diese Eigenschaft, z. B.:

$$(3-2)-1=0 \quad \neq \quad 3-(2-1)=2$$

Während sich die Addition und die Multiplikation von *Ganzzahltypen* in Java tatsächlich assoziativ verhalten, gilt das aus technischen Gründen *nicht* für die Addition und die Multiplikation von *Gleitkommatypen* (Gosling et al 2019, Abschnitt 15.7.3).

4) Links vor rechts bei der Auswertung der Argumente eines binären Operators

Bevor ein Operator ausgeführt werden kann, müssen erst seine Argumente (Operanden) ausgewertet sein. Als Operand eines binären Operators kann z. B. der Ausdruck `++ivar` auftreten (siehe nächstes Beispielprogramm). Bei jedem binären Operator ist in Java sichergestellt, dass erst der linke Operand ausgewertet wird, dann der rechte. Bei den logischen Operatoren mit *bedingter* Ausführung (**&&**, **||**) verhindert ein bestimmter Wert des linken Operanden die Auswertung des rechten Operanden (siehe Abschnitt 3.5.5).[1]

Das folgende Beispiel zeigt, dass die hohe Priorität (Bindungskraft) des Präinkrementoperators (siehe Tabelle im Abschnitt 3.5.10.2) *nicht* dazu führt, dass sich der Nebeneffekt des Ausdrucks `++ivar` auf den linken Operanden der Multiplikation auswirkt:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        int ivar = 2;`
`        int erg = ivar * ++ivar;`
`        System.out.printf("%d %d", erg, ivar);`
`    }`
`}`
	`6 3`

Die Auswertung des Ausdrucks `ivar * ++ivar` verläuft so:

- Zuerst wird der linke Operand der Multiplikation ausgewertet (Ergebnis: 2)
- Dann wird der rechte Operand der Multiplikation ausgewertet:
  - Die Präinkrementoperation hat einen Nebeneffekt auf die Variable `ivar`.
  - Der Ausdruck `++ivar` hat den Wert 3.
- Die Ausführung der Multiplikationsoperation liefert schließlich das Endergebnis 6.

Wie eine leichte Variation des letzten Beispiels zeigt, kann sich ein Nebeneffekt im *linken* Operanden einer binären Operation auf den rechten Operanden auswirken:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        int ivar = 2;`
`        int erg = ivar++ * ivar;`
`        System.out.printf("%d %d", erg, ivar);`
`    }`
`}`
	`6 3`

Im folgenden Beispiel stehen *a*, *b* und *c* für beliebige numerische Operanden (z. B. `++ivar`). Für den Ausdruck

$$a + b * c$$

resultiert aus der Bindungskraftregel die folgende Zuordnung der Operanden:

[1] Kommt es bei der Auswertung des linken Operanden zu einem Ausnahmefehler (siehe unten), dann unterbleibt die Auswertung des rechten Operanden.

$$a + (b * c)$$

Zusammen mit der Links-vor-rechts - Regel ergibt sich für die Auswertung der Operanden bzw. Ausführung der Operatoren die folgende Reihenfolge:

$$a, b, c, *, +$$

Wenn als Operanden numerische Literale oder Variablen auftreten, wird bei der „Auswertung" eines Operanden lediglich sein Wert ermittelt, und die Reihenfolge der Operandenauswertungen ist belanglos. Im letzten Beispiel eine falsche Auswertungsreihenfolge zu unterstellen (z. B. *b*, *c*, *, *a*, +), bleibt ungestraft. Wenn Operanden *Nebeneffekte* enthalten (Zuweisungen, In- bzw. Dekrementoperationen oder Methodenaufrufe), dann ist die Reihenfolge der Operandenauswertungen jedoch relevant, und eine falsche Vermutung kann gravierende Fehler verursachen. Der Übersichtlichkeit halber sollte ein Ausdruck maximal *einen* Nebeneffekt enthalten.

Auch bei einem rechts-assoziativen Operator wird der linke Operand vor dem rechten ausgewertet, sodass im folgenden Beispiel mit der **int**-Variablen `alfa`

```
alfa += ++alfa
```

diese Auswertungs- bzw. Ausführungsreihenfolge resultiert:

```
alfa, ++alfa, +=
```

Als neuer Wert von `alfa` entsteht:

```
alfa + (alfa + 1)
```

Die oft anzutreffende Behauptung, Klammerausdrücke würden generell zuerst ausgewertet, ist falsch, wie das folgende Beispiel zeigt:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        int ivar = 2;`
`        int erg = ivar * (++ivar + 5);`
`        System.out.println(erg);`
`    }`
`}`
	16

Die Auswertung des Ausdrucks `ivar * (++ivar + 5)` verläuft so:

- Wegen Regel 4 (links-vor-rechts bei der Auswertung der Operanden eines binären Operators) wird zuerst der linke Operand der Multiplikation ausgewertet (Ergebnis: 2)
- Dann wird der rechte Operand der Multiplikation ausgewertet (also der Klammerausdruck).
- Hier ist mit der Addition eine weitere binäre Operation vorhanden, und nach der Links-vor-rechts - Regel wird zunächst deren linker Operand ausgewertet (Ergebnis: 3, Nebeneffekt auf die Variable `ivar`). Dann wird der rechte Operand der Addition ausgewertet (Ergebnis: 5). Die Ausführung der Additionsoperation liefert für den Klammerausdruck den Wert 8.
- Schließlich führt die Multiplikation zum Endergebnis 16.

3.5.10.2 Operatorentabelle

In der folgenden Tabelle sind die bisher behandelten Operatoren mit absteigender Bindungskraft (Priorität) aufgelistet. Gruppen von Operatoren mit gleicher Bindungskraft sind durch eine horizontale Linie voneinander getrennt. In der **Operanden**-Spalte werden die zulässigen Datentypen der Argumentausdrücke mit Hilfe der folgenden Platzhalter beschrieben:

N Ausdruck mit numerischem Datentyp (**byte**, **short**, **int**, **long**, **char**, **float**, **double**)
I Ausdruck mit integralem (ganzzahligem) Datentyp (**byte**, **short**, **int**, **long**, **char**)
L logischer Ausdruck (Typ **boolean**)

K Ausdruck mit kompatiblem Datentyp
S **String** (Zeichenfolge)
V Variable mit kompatiblem Datentyp
V_n Variable mit numerischem Datentyp (**byte**, **short**, **int**, **long**, **char**, **float**, **double**)

Operator	Bedeutung	Operanden
!	Negation	*L*
++, --	Prä- oder Postinkrement bzw. -dekrement	V_n
-	Vorzeichenumkehr	*N*
(*Typ*)	Typumwandlung	*K*
*, /	Punktrechnung	*N*, *N*
%	Modulo	*N*, *N*
+, -	Strichrechnung	*N*, *N*
+	**String**-Verkettung	*S*, *K* oder *K*, *S*
<<, >>	Links- bzw. Rechts-Verschiebung	*I*, *I*
>, <, >=, <=	Vergleichsoperatoren	*N*, *N*
==, !=	Gleichheit, Ungleichheit	*K*, *K*
&	Bitweises UND	*I*, *I*
&	Logisches UND (mit unbedingter Auswertung)	*L*, *L*
^	Exklusives logisches ODER	*L*, *L*
|	Bitweises ODER	*I*, *I*
|	Logisches ODER (mit unbedingter Auswertung)	*L*, *L*
&&	Logisches UND (mit bedingter Auswertung)	*L*, *L*
||	Logisches ODER (mit bedingter Auswertung)	*L*, *L*
? :	Konditionaloperator	*L*, *K*, *K*
=	Wertzuweisung	*V*, *K*
+=, -=, *=, /=, %=	Wertzuweisung mit Aktualisierung	V_n, *N*

Im Anhang A finden Sie eine erweiterte Version dieser Tabelle, die zusätzlich alle Operatoren enthält, die im weiteren Verlauf des Manuskripts noch behandelt werden.

3.6 Über- und Unterlauf bei numerischen Variablen

Wie Sie inzwischen wissen, haben die primitiven Datentypen für Zahlen jeweils einen bestimmten Wertebereich (siehe Tabelle im Abschnitt 3.3.6). Dank strenger Typisierung kann der Compiler

verhindern, dass einer Variablen ein Ausdruck mit „zu großem Typ“ zugewiesen wird. So kann z. B. einer **int**-Variablen kein Wert vom Typ **long** zugewiesen werden. Bei der Auswertung eines Ausdrucks kann jedoch „unterwegs“ ein Wertebereichsproblem (z. B. ein Überlauf) auftreten. Im betroffenen Programm ist mit einem mehr oder weniger gravierenden Fehlverhalten zu rechnen, sodass Wertebereichsprobleme unbedingt vermieden bzw. rechtzeitig diagnostiziert werden müssen.

Im Zusammenhang mit Wertebereichsproblemen bieten sich gelegentlich die Klassen **BigDecimal** und **BigInteger** aus dem Paket **java.math** als Alternativen zu den primitiven Datentypen an. Wenn wir gleich auf einen solchen Fall stoßen, verzichten wir nicht auf eine kurze Beschreibung der jeweiligen Vor- und Nachteile, obwohl die beiden Klassen nicht zu den elementaren Sprachelementen gehören. Analog wurde schon im Abschnitt 3.3.7.2 demonstriert, dass die Klasse **BigDecimal** bei finanzmathematischen Anwendungen wegen ihrer praktisch unbeschränkten Genauigkeit zu bevorzugen ist.

3.6.1 Überlauf bei Ganzzahltypen

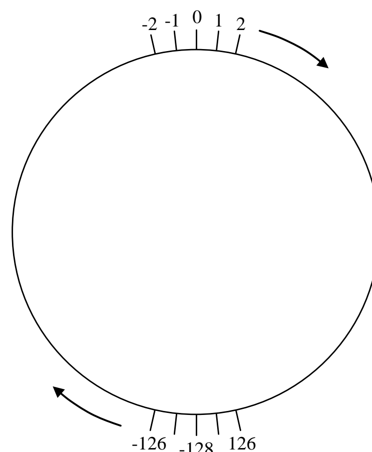
Wird z. B. zu einer ganzzahligen Variablen, die bereits den maximalen Wert ihres Typs besitzt, eine positive Zahl addiert, kann das Ergebnis nicht mehr korrekt abgespeichert werden. Ohne besondere Vorkehrungen stellt ein Java-Programm im Falle eines solchen Ganzzahlüberlaufs keinesfalls seine Tätigkeit (z. B. mit einem Ausnahmefehler) ein, sondern arbeitet munter weiter. Das folgende Programm

```
class Prog {
    public static void main(String[] args) {
        int i = 2_147_483_647, j = 5, k;
        k = i + j; // Überlauf!
        System.out.println(i + " + " + j + " = " + k);
    }
}
```

liefert ohne jede Warnung das sinnlose Ergebnis:

```
2147483647 + 5 = -2147483644
```

Um das Auftreten eines negativen „Ergebniswerts“ zu verstehen, machen wir einen kurzen Ausflug in die Informatik. Die Werte der Ganzzahltypen sind nach dem **Zweierkomplementprinzip** auf einem Zahlenkreis angeordnet, und nach der größten positiven Zahl beginnt der Bereich der negativen Zahlen (mit abnehmendem Betrag), z. B. beim Typ **byte**:



Speziell bei der Steuerung von Raketenmotoren (vgl. Abschnitt 3.5.7) ist also Vorsicht geboten, weil ansonsten das Kommando „Mr. Spock, please push the engine.“ zum heftigen Rückwärtsschub

führen könnte.[1] Es zeigt sich erneut, dass eine erfolgreiche Raketenforschung und -entwicklung ohne die sichere Beherrschung der elementaren Sprachelemente kaum möglich ist.

Natürlich kann nicht nur den positiven Rand eines Ganzzahlwertebereichs überschreiten, sondern auch den negativen Rand, indem z. B. vom kleinstmöglichen Wert eine positive Zahl subtrahiert wird, z. B.:

Quellcode	Ausgabe

`class Prog {`
`  public static void main(String[] args) {`
`    int i = -2_147_483_648, j = 5, k;`
`    k = i - j;`
`    System.out.println(i+" - "+j+" = "+k);`
`  }`
`}`
	`-2147483648 - 5 = 2147483643`

Bei Wertebereichsproblemen durch eine betragsmäßig zu große Zahl wird im Manuskript generell von einem *Überlauf* gesprochen. Unter einem *Unterlauf* soll später das Verlassen eines Gleitkommawertebereichs in Richtung null durch eine betragsmäßig zu kleine Zahl verstanden werden (vgl. Abschnitt 3.6.3).

Oft lässt sich ein Überlauf durch die Wahl eines geeigneten Datentyps verhindern. Mit den Deklarationen

```
long i = 2_147_483_647, j = 5, k;
```

kommt es in der Anweisung

```
k = i + j;
```

nicht zum Überlauf, weil neben `i`, `j` und `k` nun auch der Ausdruck `i+j` den Typ **long** besitzt. Die Anweisung

```
System.out.println(i + " + " + j + " = " + k);
```

liefert das korrekte Ergebnis:

```
2147483647 + 5 = 2147483652
```

Im Beispiel genügt es *nicht*, für die Zielvariable `k` den beschränkten Typ **int** durch **long** zu ersetzen, weil der Überlauf beim Berechnen des Ausdrucks („unterwegs") auftritt. Mit den Deklarationen

```
int i = 2_147_483_647, j = 5;
long k;
```

bleibt das Ergebnis falsch, denn …

- In der Anweisung
  ```
  k = i + j;
  ```
  wird der Ausdruck `i + j` berechnet, bevor die Zuweisung ausgeführt wird.
- Weil beide Operanden vom Typ **int** sind, erhält auch der Ausdruck diesen Typ (vgl. Tabelle im Abschnitt 3.5.1), und die Summe kann nicht korrekt berechnet bzw. zwischenspeichert werden.
- Schließlich wird der **long**-Variablen `k` das falsche Ergebnis zugewiesen.

Wenn auch der **long**-Wertebereich nicht ausreicht, und weiterhin mit ganzen Zahlen gerechnet werden soll, bietet sich die Klasse **BigInteger** aus dem Paket **java.math** an.[1] Das folgende Programm

---

[1] Mr. Spock arbeitete jahrelang als erster Offizier auf dem Raumschiff Enterprise.

```
import java.math.*;
class Prog {
    public static void main(String[] args) {
        BigInteger bigi = new BigInteger("9223372036854775808");
        bigi = bigi.multiply(bigi);
        System.out.println("2 hoch 126 = " + bigi);
    }
}
```

speichert im **BigInteger**-Objekt `bigi` die knapp außerhalb des **long**-Wertebereichs liegende Zahl 2^{63}, quadriert diese auch noch mutig und findet selbstverständlich das korrekte Ergebnis:

```
2 hoch 126 = 85070591730234615865843651857942052864
```

Im Vergleich zu den primitiven Ganzzahltypen verursacht die Klasse **BigInteger** allerdings einen höheren Speicher- und Rechenzeitaufwand.

Seit Java 8 bietet die Klasse **Math** im Paket **java.lang** statische Methoden für arithmetische Operationen mit Ganzzahltypen, die auf einen Überlauf mit einem Ausnahmefehler reagieren. Neben den anschließend aufgelisteten Methoden für **int**-Argumente sind analog arbeitende Methoden für long-Argumente vorhanden:

- **public static int addExact(int** *x*, **int** *y*)
- **public static int subtractExact(int** *x*, **int** *y*)
- **public static int multiplyExact(int** *x*, **int** *y*)
- **public static int incrementExact(int** *a*)
- **public static int decrementExact(int** *a*)
- **public static int negateExact(int** *a*)

Falls ein Ausnahmefehler nicht abgefangen wird, endet das betroffene Programm, statt mit sinnlosen Zwischenergebnissen weiterzurechnen, z. B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
  public static void main(String[] args) {
    int i = 2147483647, j = 5, k;
    k = Math.addExact(i, j);
    System.out.println(i+" + "+j+" = "+k);
  }
}</pre></td><td><pre>Exception in thread "main" java.lang.ArithmeticException:
 integer overflow
 at java.base/java.lang.Math.addExact(Math.java:825)
 at Prog.main(Prog.java:4)</pre></td></tr>
</table>

3.6.2 Unendliche und undefinierte Werte bei den Typen float und double

Auch bei den binären Gleitkommatypen **float** und **double** kann ein Überlauf auftreten, obwohl die unterstützten Wertebereiche hier weit größer sind. Dabei kommt es aber weder zu einem sinnlosen Zufallswert, sondern zu den speziellen Gleitkommawerten **+/- Unendlich**, mit denen anschließend sogar weitergerechnet werden kann. Das folgende Programm

```
class Prog {
    public static void main(String[] args) {
        double bigd = Double.MAX_VALUE;
        System.out.printf("Double.MAX_VALUE      = %15e%n", bigd);
        bigd = Double.MAX_VALUE * 10.0;
        System.out.printf("Double.MaxValue * 10 = %15e%n", bigd);
        System.out.printf("Unendlich + 10        = %15e%n", bigd + 10);
```

[1] Ab Java 9 befindet sich das Paket **java.util** im Modul **java.base**. Das gilt bis auf wenige Ausnahmen für alle im Manuskript verwendeten Pakete, sodass der Hinweis auf die Modulzugehörigkeit nur noch in den Ausnahmefällen erscheint.

```
        System.out.printf("Unendlich * (-1)     = %15e%n",bigd * -1);
        System.out.printf("13.0/0.0             = %15e", 13.0 / 0.0);
    }
}
```

liefert die Ausgabe:

```
Double.MAX_VALUE     =   1,797693e+308
Double.MaxValue * 10 =        Infinity
Unendlich + 10       =        Infinity
Unendlich * (-1)     =       -Infinity
13.0/0.0             =        Infinity
```

Im Programm erhält die **double**-Variable `bigd` den größtmöglichen Wert ihres Typs. Anschließend wird `bigd` mit dem Faktor 10 multipliziert, was zum Ergebnis +Unendlich führt. Mit diesem Zwischenergebnis kann Java durchaus rechnen:

- Addiert man die Zahl 10, bleibt es beim Wert +Unendlich.
- Eine Multiplikation von +Unendlich mit (-1) führt zum Wert -Unendlich.

Mit Hilfe der Unendlich-Werte „gelingt“ offenbar bei der Gleitkommaarithmetik sogar die Division durch null, während bei der Ganzzahlarithmetik ein solcher Versuch zu einem Laufzeitfehler (aus der Klasse **ArithmeticException**) führt.

Bei den folgenden „Berechnungen“

$$\text{Unendlich} - \text{Unendlich}$$

$$\frac{\text{Unendlich}}{\text{Unendlich}}$$

$$\text{Unendlich} \cdot 0$$

$$\frac{0}{0}$$

resultiert der spezielle Gleitkommawert **NaN** (*Not a Number*), wie das nächste Beispielprogramm zeigt:

```
class Prog {
    public static void main(String[] args) {
        double bigd = Double.MAX_VALUE * 10.0;
        System.out.printf("Unendlich – Unendlich = %3f%n", bigd-bigd);
        System.out.printf("Unendlich / Unendlich = %3f%n", bigd/bigd);
        System.out.printf("Unendlich * 0.0       = %3f%n", bigd * 0.0);
        System.out.printf("0.0 / 0.0             = %3f", 0.0/0.0);
    }
}
```

Es liefert die Ausgaben:

```
Unendlich – Unendlich = NaN
Unendlich / Unendlich = NaN
Unendlich * 0.0       = NaN
0.0 / 0.0             = NaN
```

Zu den letzten Beispielprogrammen ist noch anzumerken, dass man über das öffentliche, statische und finalisierte Feld **MAX_VALUE** der Klasse **Double** aus dem Paket **java.lang** den größten Wert in Erfahrung bringt, der in einer **double**-Variablen gespeichert werden kann.

Über die statischen **Double**-Methoden

- **public static boolean isInfinite(double** *arg***)**
- **public static boolean isNaN(double** *arg***)**

mit Rückgabetyp **boolean** lässt sich für eine **double**-Variable prüfen, ob sie einen unendlichen oder undefinierten Wert besitzt, z. B.:

Quellcode	Ausgabe

<pre>class Prog {
    public static void main(String[] args) {
        System.out.println(Double.isInfinite(1.0/0.0));
        System.out.print(Double.isNaN(0.0/0.0));
    }
}</pre>
	<pre>true
	true</pre>

Für besonders neugierige Leser sollen abschließend noch die **float**-Darstellungen der speziellen Gleitkommawerte angegeben werden (vgl. Abschnitt 3.3.7.1):

Wert	float-Darstellung		
	Vorz.	Exponent	Mantisse
+unendlich	0	11111111	00000000000000000000000
-unendlich	1	11111111	00000000000000000000000
NaN	0	11111111	10000000000000000000000

Wenn der **double**-Wertebereich längst in Richtung **Infinity** überschritten ist, kann man mit Objekten der Klasse **BigDecimal** aus dem Paket **java.math** noch rechnen:

Quellcode	Ausgabe

<pre>import java.math.*;
class Prog {
    public static void main(String[] args) {
        BigDecimal bigd = new BigDecimal("1000111");
        bigd = bigd.pow(500);
        System.out.printf("Very Big: %e", bigd);
    }
}</pre>
	<pre>Very Big: 1.057066e+3000</pre>

Ein Überlauf ist bei **BigDecimal**-Objekten nicht zu befürchten, solange das Programm genügend Hauptspeicher zur Verfügung hat.

3.6.3 Unterlauf bei den Gleitkommatypen

Bei den Gleitkommatypen **float** und **double** ist auch ein Unterlauf möglich, wobei eine Zahl mit sehr kleinem Betrag nicht mehr dargestellt werden kann. In diesem Fall rechnet ein Java-Programm mit dem Wert 0,0 weiter, was in der Regel akzeptabel ist, z. B.:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        double smalld = Double.MIN_VALUE;`
`        System.out.println(smalld);`
`        smalld /= 2.0;`
`        System.out.println(smalld);`
`    }`
`}`
	`4.9E-324`
	`0.0`

Das statische, öffentliche und finalisierte Feld **MIN_VALUE** der Klasse **Double** im Paket **java.lang** enthält den betragsmäßig kleinsten Wert, der in einer **double**-Variablen gespeichert werden kann (vgl. Abschnitt 3.3.6).

In unglücklichen Fällen wird aber ein deutlich von null verschiedenes Endergebnis grob falsch berechnet, weil unterwegs ein Zwischenergebnis der Null zu nahe gekommen ist, z. B.

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        double a = 1E-323;`
`        double b = 1E308;`
`        double c = 1E16;`
`        System.out.println(a * b * c);`
`        System.out.print(a * 0.1 * b * 10.0 * c);`
`    }`
`}`
	`9.881312916824932`
	`0.0`

Das Ergebnis des Ausdrucks

```
a * b * c
```

wird halbwegs korrekt ermittelt (vgl. Abschnitt 3.3.7.1 zu den Genauigkeitsproblemen der Gleitkommatypen). Bei der Berechnung des Ausdrucks

```
a * 0.1 * b * 10.0 * c
```

wird jedoch das Zwischenergebnis

```
a * 0.1 = 1E-324 < 4.9E-324
```

aufgrund eines Unterlaufs auf null gesetzt, und das korrekte Endergebnis 10 kann nicht mehr erreicht werden.

Mit Objekten der Klasse **BigDecimal** aus dem Paket **java.math** an Stelle von **double**-Variablen kann ein Unterlauf zuverlässig verhindert werden:

```
import java.math.*;
class Prog {
    public static void main(String[] args) {
        BigDecimal a = new BigDecimal("1E-323");
        BigDecimal b = new BigDecimal("1E308");
        BigDecimal c = new BigDecimal("1E16");
        BigDecimal nk1 = new BigDecimal("0.1");
        BigDecimal zehn = new BigDecimal("10.0");
        System.out.println(a.multiply(nk1).multiply(b).multiply(zehn).multiply(c));
    }
}
```

Weil **BigDecimal**-Objekte als Argumente der arithmetischen Operatoren nicht zugelassen sind, muss das Multiplizieren per Methodenaufruf erledigt werden. Als Gegenleistung für den Aufwand erhält man das korrekte Ergebnis 10,0 ohne Unterlauf und ohne Genauigkeitsproblem (siehe oben). Neben dem leicht zu verschmerzenden Schreibaufwand entsteht durch die Verwendung von **Big-**

Decimal-Objekten aber auch ein erhöhter Speicher- und Rechenaufwand (siehe Abschnitt 3.3.7.2), sodass die binären Gleitkommatypen in vielen Situationen die erste Wahl bleiben.

3.6.4 Vertiefung: Der Modifikator strictfp

In der Norm IEEE-754 für die binären Gleitkommatypen ist neben der strikten Gleitkommaarithmetik auch eine erweiterte Variante erlaubt, die bei Zwischenergebnissen einen größeren Wertebereich und eine höhere Genauigkeit bietet. Eine Nutzung dieser möglicherweise nur auf manchen CPUs verfügbaren Variante durch die JVM kann Über- bzw. Unterlaufprobleme reduzieren. Andererseits geht aber die Plattformunabhängigkeit der Rechenergebnisse verloren.

Nach Gosling et al (2019, Abschnitt 15.4) ist einer JVM bei einem Ausdruck vom Typ **float** oder **double** die Nutzung der optimierten Gleitkommaarithmetik der lokalen Plattform mit folgenden Ausnahmen erlaubt:

- Der Wert des Ausdrucks kann bereits zur Übersetzungszeit berechnet werden.
- Es ist für die betroffene Klasse, für ein implementiertes Interface (siehe unten) oder für die betroffene Methode der Modifikator **strictfp** deklariert, um eine an der strikten IEEE-754 - Norm orientierte und damit plattformunabhängige Gleitkommaarithmetik anzuordnen.

Mit der JVM 8 oder 13 ist es mir auf einem Rechner mit Intel-CPU (Core i3) unter Windows 10 (64 Bit) *nicht* gelungen, einen Effekt des **strictfp**-Modifikators zu beobachten. Das folgende Beispielprogramm aus Gosling et al (2019, S. 499)

```
strictfp class Prog  {
    public static void main(String[] args) {
        double d = 8e+307;
        System.out.println(4.0 * d * 0.5);
        System.out.println(2.0 * d);
    }
}
```

produziert *mit und ohne* den **class**-Modifikator **strictfp** dieselbe Ausgabe:

```
Infinity
1.6E308
```

Offenbar wird die Abwesenheit des Modifikators *nicht* dazu genutzt, durch Verwendung eines größeren Wertebereichs für den Exponenten von Zwischenergebnissen den Überlauf beim Zwischenergebnis

```
4.0 * d
```

zu verhindern. Es ist davon auszugehen, dass der Modifikator **strictfp** derzeit bei modernen x86-CPUs keinen Effekt hat.

Wenn sich die meisten aktuellen CPUs grundsätzlich an die Norm IEEE 754 halten, hat die Verwendung des Modifikators **strictfp** dort keine negativen Konsequenzen. Die mögliche Existenz von anders arbeitenden Systemen spricht dafür, den Modifikator generell zu verwenden, um die Plattformunabhängigkeit der Software sicherzustellen.[1] Wo ein Über- oder Unterlauf verhindert werden muss, sollte an Stelle des binären Gleitkommatyps **double** ein Objekt der Klasse **BigDecimal** verwendet werden (siehe Abschnitt 3.6.2).

Für die API-Klasse **StrictMath** im Paket **java.lang** wird (im Unterschied zur Klasse **Math** im selben Paket) die strikte IEEE-754 - Gleitkommaarithmetik garantiert. Im Quellcode dieser Klasse findet sich das folgende Beispiel für die Verwendung des Methoden-Modifikators **strictfp**:

---

[1] Diese überzeugende Schlussfolgerung stammt von der folgenden Webseite:
http://stackoverflow.com/questions/22562510/does-java-strictfp-modifier-have-any-effect-on-modern-cpus

```
public static strictfp double toRadians(double angdeg) {
    return angdeg / 180.0 * StrictMath.PI;
}
```

3.7 Anweisungen (zur Ablaufsteuerung)

Wir haben uns im Kapitel 3 über elementare Sprachelemente zunächst mit (lokalen) **Variablen** und primitiven **Datentypen** vertraut gemacht. Dann haben wir gelernt, aus Variablen, Literalen und Methodenaufrufen mit Hilfe von **Operatoren** mehr oder weniger komplexe **Ausdrücke** zu bilden. Diese wurden entweder mit Hilfe des Objekts **System.out** auf der Konsole ausgegeben oder in Wertzuweisungen verwendet.

In den meisten Beispielprogrammen traten nur wenige Sorten von *Anweisungen* auf (Variablendeklarationen, Wertzuweisungen und Methodenaufrufe). Nun werden wir uns systematisch mit dem allgemeinen Begriff einer Java-Anweisung befassen und vor allem die wichtigen Anweisungen zur Ablaufsteuerung (Fallunterscheidungen und Schleifen) kennenlernen.

3.7.1 Überblick

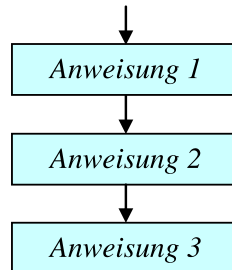
Ein ausführbarer Programmteil, also der Rumpf einer Methode, besteht aus *Anweisungen* (engl. *statements*).

Am Ende von Abschnitt 3.7 werden Sie die folgenden Sorten von Anweisungen kennen:

- **Variablendeklarationsanweisung**
  Die Anweisung zur Deklaration von lokalen Variablen wurde schon im Abschnitt 3.3.8 eingeführt.
  Beispiel: `int i = 1, j = 2, k;`
- **Ausdrucksanweisungen**
  Folgende Ausdrücke werden zu Anweisungen, sobald man ein Semikolon dahinter setzt:
  - **Wertzuweisung** (vgl. Abschnitte 3.3.8 und 3.5.8)
    Beispiel: `k = i + j;`
  - **Prä- bzw. Postinkrement- oder -dekrementoperation**
    Beispiel: `i++;`
    Im Beispiel ist nur der „Nebeneffekt“ des Ausdrucks `i++` von Bedeutung (vgl. Abschnitt 3.5.1). Sein Wert bleibt ungenutzt.
  - **Methodenaufruf**
    Beispiel: `System.out.println(k);`
    Besitzt die im Rahmen einer eigenständigen Anweisung aufgerufene Methode einen Rückgabewert, wird dieser ignoriert.
- **Leere Anweisung**
  Beispiel: `;`
  Die durch ein einsames (nicht anderweitig eingebundenes) Semikolon ausgedrückte *leere* Anweisung hat keinerlei Effekte und kommt gelegentlich zum Einsatz, wenn die Syntax eine Anweisung verlangt, aber nichts geschehen soll.
- **Block- bzw. Verbundanweisung**
  Eine Folge von Anweisungen, die durch geschweifte Klammern zusammengefasst bzw. abgegrenzt werden, bildet eine **Block- bzw. Verbundanweisung**. Wir haben uns bereits im Abschnitt 3.3.9 im Zusammenhang mit dem Gültigkeitsbereich für lokale Variablen mit der Blockanweisung beschäftigt. Wie gleich näher erläutert wird, fasst man z. B. *dann* mehrere Abweisungen zu einem Block zusammen, wenn diese Anweisungen unter einer gemeinsa-

men Bedingung ausgeführt werden sollen. Es wäre sehr unpraktisch, dieselbe Bedingung für jede betroffene Anweisung wiederholen zu müssen.

- **Anweisungen zur Ablaufsteuerung**
  Die **main()** - Methoden der bisherigen Beispielprogramme im Kapitel 3 bestanden meist aus einer *Sequenz* von Anweisungen, die bei jedem Programmlauf komplett und linear durchlaufen wurde:



  Oft möchte man jedoch ...
  - die Ausführung einer Anweisung (eines Anweisungsblocks) von einer *Bedingung* abhängig machen
  - oder eine Anweisung (einen Anweisungsblock) *wiederholt* ausführen lassen.

  Für solche Zwecke enthält Java etliche Anweisungen zur Ablaufsteuerung, die bald ausführlich behandelt werden (**bedingte Anweisung**, **Fallunterscheidung**, **Schleifen**).

Blockanweisungen sowie Anweisungen zur Ablaufsteuerung enthalten andere Anweisungen und werden daher auch als **zusammengesetzte Anweisungen** bezeichnet.

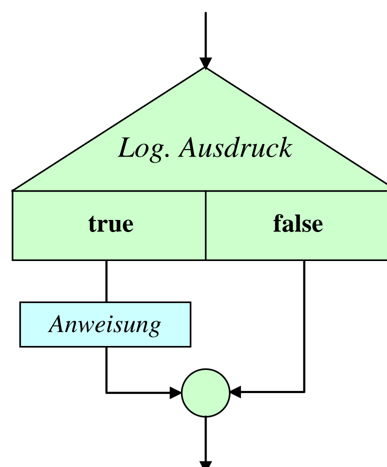
Anweisungen werden durch ein **Semikolon** abgeschlossen, sofern sie nicht mit einer schließenden Blockklammer enden.

3.7.2 Bedingte Anweisung und Fallunterscheidung

Oft ist es erforderlich, dass eine Anweisung nur unter einer bestimmten Bedingung ausgeführt wird. Etwas allgemeiner formuliert geht es darum, dass viele Algorithmen *Fallunterscheidungen* benötigen, also an bestimmten Stellen in Abhängigkeit vom Wert eines steuernden Ausdrucks in unterschiedliche Pfade verzweigen müssen.

3.7.2.1 if-Anweisung

Nach dem folgenden **Programmablaufplan** (**PAP**) bzw. **Flussdiagramm** soll eine Anweisung nur dann ausgeführt werden, wenn ein logischer Ausdruck den Wert **true** besitzt:



Wir werden diese Darstellungstechnik ab jetzt verwenden, um einen Algorithmus bzw. Programmablauf zu beschreiben. Die verwendeten Symbole sind hoffentlich anschaulich, entsprechen aber keiner strengen Normierung.

Während der Programmablaufplan den Zweck (die Semantik) eines Sprachbestandteils erläutert, beschreibt das vertraute Syntaxdiagramm, wie zulässige Exemplare des Sprachbestandteils zu bilden sind. Das folgende Syntaxdiagramm beschreibt die zur Realisation einer bedingten Ausführung dienende **if**-Anweisung:

Um genau zu sein, muss zu diesem Syntaxdiagramm noch angemerkt werden, dass als bedingt auszuführende Anweisung keine Variablendeklaration erlaubt ist. Es ist übrigens nicht vergessen worden, ein Semikolon ans Ende des **if**-Syntaxdiagramms zu setzen. Dort wird eine *Anweisung* verlangt, wobei konkrete Beispiele oft mit einem Semikolon enden, manchmal aber auch mit einer schließenden geschweiften Klammer.

Im folgenden Beispiel wird eine Meldung ausgegeben, wenn die Variable `anz` den Wert 0 besitzt:

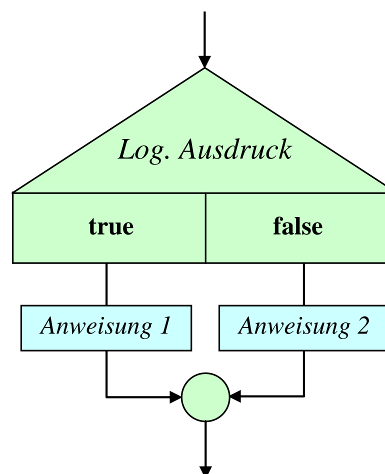
```
if (anz == 0)
    System.out.println("Die Anzahl muss > 0 sein!");
```

Der Zeilenumbruch zwischen dem logischen Ausdruck und der Anweisung dient nur der Übersichtlichkeit und ist für den Compiler irrelevant.

Selbstverständlich ist als Anweisung auch ein Block erlaubt.

3.7.2.2 if-else - Anweisung

Soll auch etwas passieren, wenn der steuernde logische Ausdruck den Wert **false** besitzt,



dann erweitert man die **if**-Anweisung um eine **else**-Klausel.

Zur Beschreibung der **if-else** - Anweisung wird an Stelle eines Syntaxdiagramms eine alternative Darstellungsform gewählt, die sich am typischen Java - Quellcode-Layout orientiert:

if (*Logischer Ausdruck***)**
  Anweisung 1
else
  Anweisung 2

Wie bei den Syntaxdiagrammen gilt auch für diese Form der Syntaxbeschreibung:

- Für **terminale Sprachbestandteile**, die exakt in der angegebenen Form in konkreten Quellcode zu übernehmen sind, wird **fette** Schrift verwendet.
- *Platzhalter* sind an *kursiver* Schrift zu erkennen.

Während die Syntaxbeschreibung im Quellcode-Layout relativ einfache Bildungsregeln (mit einer einzigen zulässigen Sequenz) sehr anschaulich beschreibt, bietet das Syntaxdiagramm den Vorteil, bei komplizierter, variantenreicher Syntax alle zulässigen Sequenzen kompakt und präzise als Pfade durch das Diagramm zu dokumentieren.

Wie schon bei der einfachen **if**-Anweisung gilt auch bei der **if-else** - Anweisung, dass Variablendeklarationen nicht als eingebettete Anweisungen erlaubt sind.

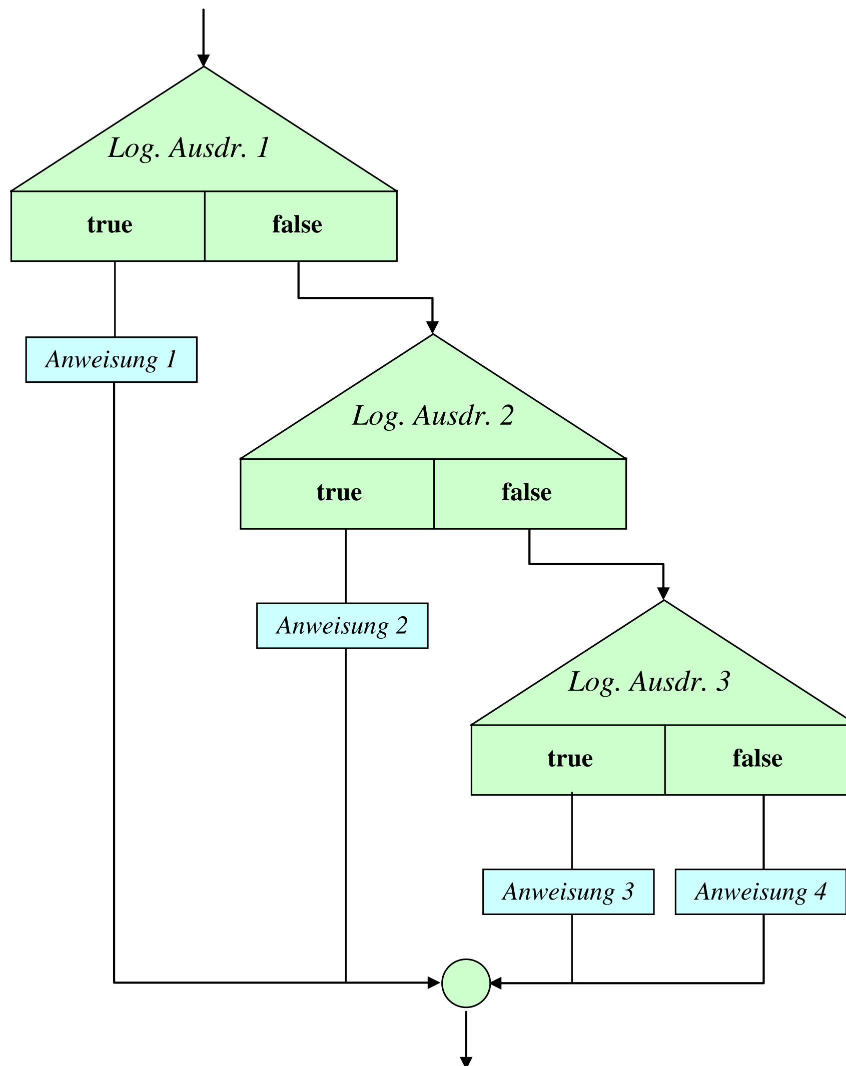
Im folgenden **if-else** - Beispiel wird der natürliche Logarithmus zu einer Zahl berechnet, falls diese positiv ist. Anderenfalls erscheint eine Fehlermeldung. Das Argument wird vom Benutzer über die `Simput`-Methode `gdouble()` erfragt (vgl. Abschnitt 3.4).[1]

Quellcode	Ein- und Ausgabe

`class Prog {`
`  public static void main(String[] args) {`
`    System.out.print("Argument: ");`
`    double arg = Simput.gdouble();`
`    if (arg > 0)`
`      System.out.printf("ln(%.3f) = %.3f", arg, Math.log(arg));`
`    else`
`      System.out.println("Argument ungültig oder <= 0!");`
`  }`
`}`
	`Argument: 2,4`
	`ln(2,400) = 0,875`

Eine bedingt auszuführende Anweisung darf durchaus wiederum vom **if-** bzw. **if-else** - Typ sein, sodass sich mehrere, *hierarchisch geschachtelte* Fälle unterscheiden lassen. Den folgenden Programmablauf mit „sukzessiver Restaufspaltung“

1 Bei einer irregulären Eingabe liefert `gdouble()` den (Verlegenheits-)Rückgabewert 0.0.



realisiert z. B. eine **if-else** - Konstruktion nach diesem Muster:

if (*Logischer Ausdruck 1***)**
  Anweisung 1
else if (*Logischer Ausdruck 2***)**
  Anweisung 2
    . . .
    . . .
else if (*Logischer Ausdruck k***)**
  Anweisung k
else
  Default-Anweisung

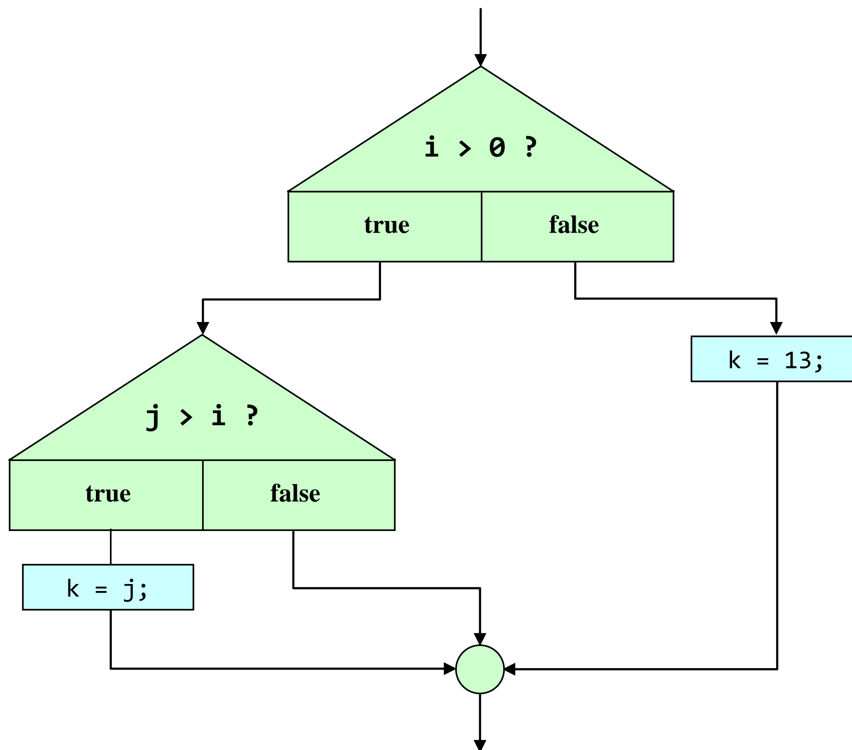
Wenn alle logischen Ausdrücke den Wert **false** annehmen, dann wird die **else**-Klausel zur letzten **if**-Anweisung ausgeführt.

Bei einer Mehrfallunterscheidung ist die im Abschnitt 3.7.2.3 vorzustellende **switch**-Anweisung gegenüber einer verschachtelten **if-else** - Konstruktion zu bevorzugen, wenn die Fallzuordnung über die verschiedenen Werte *eines* Ausdrucks (z. B. vom Typ **int**) erfolgen kann.

Beim Schachteln von bedingten Anweisungen kann es zum genannten **dangling-else** - Problem[1] kommen, wobei ein Missverständnis zwischen Programmierer und Compiler hinsichtlich der Zuordnung einer **else**-Klausel besteht. Im folgenden Code-Fragment[2]

```
if (i > 0)
    if (j > i)
        k = j;
else
    k = 13;
```

lassen die Einrücktiefen vermuten, dass der Programmierer die **else**-Klausel auf die *erste* **if**-Anweisung bezogen zu haben glaubt:

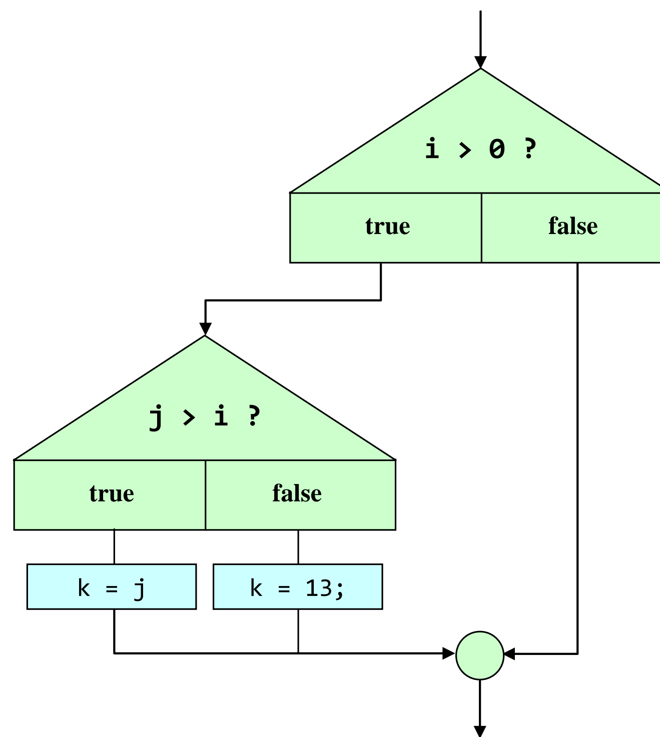


Der Compiler ordnet eine **else**-Klausel jedoch dem in Aufwärtsrichtung nächstgelegenen **if** zu, das nicht durch Blockklammern abgeschottet ist und noch keine **else**-Klausel besitzt. Im Beispiel bezieht er die **else**-Klausel also auf die *zweite* **if**-Anweisung, sodass de facto folgender Programmablauf resultiert:

---

1 Deutsche Übersetzung von *dangling*: *baumelnd*.

2 Fügt man das Quellcodesegment mit den „fehlerhaften" Einrücktiefen in ein Editorfenster unserer Entwicklungsumgebung IntelliJ ein, wird der „Layout-Fehler" übrigens automatisch behoben. IntelliJ verhindert also, dass der Logikfehler durch einen „Layout-Fehler" getarnt wird.



Bei `i` ≤ 0 geht der Programmierer vom neuen `k`-Wert 13 aus, der beim tatsächlichen Programmablauf jedoch *nicht* unbedingt zu erwarten ist.

Mit Hilfe von Blockklammern kann man die gewünschte Zuordnung erzwingen:

```
if (i > 0)
   {if (j > i)
      k = j;}
else
   k = 13;
```

Eine alternative Lösung besteht darin, auch dem zweiten **if** eine **else**-Klausel zu spendieren und dabei die leere Anweisung zu verwenden:

```
if (i > 0)
   if (j > i)
      k = j;
   else
      ;
else
   k = 13;
```

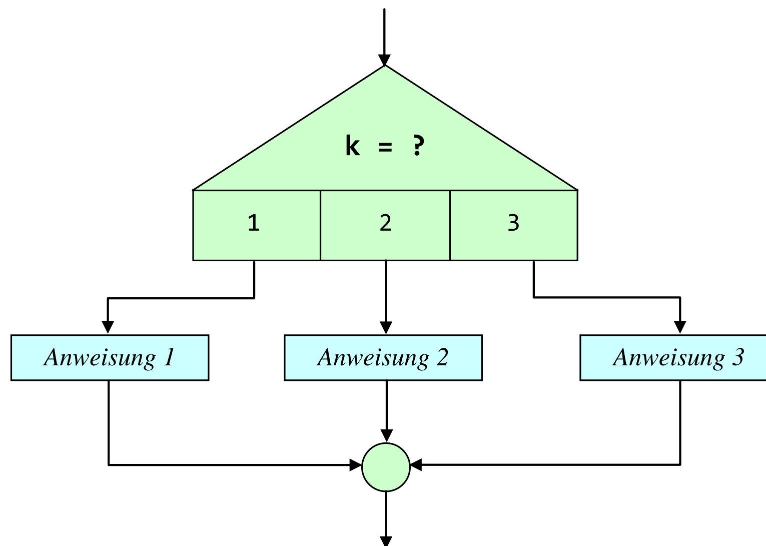
Gelegentlich kommt als Alternative zu einer **if-else** - Anweisung, die zur Berechnung eines Wertes bedingungsabhängig zwei unterschiedliche Ausdrücke benutzt, der Konditionaloperator (vgl. Abschnitt 3.5.9) in Frage, z. B.:

if-else - Anweisung	Konditionaloperator

`double arg = 3, d;`
`if (arg >= 0)`
`    d = arg * arg;`
`else`
`    d = 0;`
	`double arg = 3, d;`
	`d = arg >= 0 ? arg * arg : 0;`

3.7.2.3 *switch-Anweisung*

Wenn eine Fallunterscheidung mit mehr als zwei Alternativen wie im folgenden Flussdiagramm in Abhängigkeit vom Wert *eines* Ausdrucks vorgenommen werden soll,



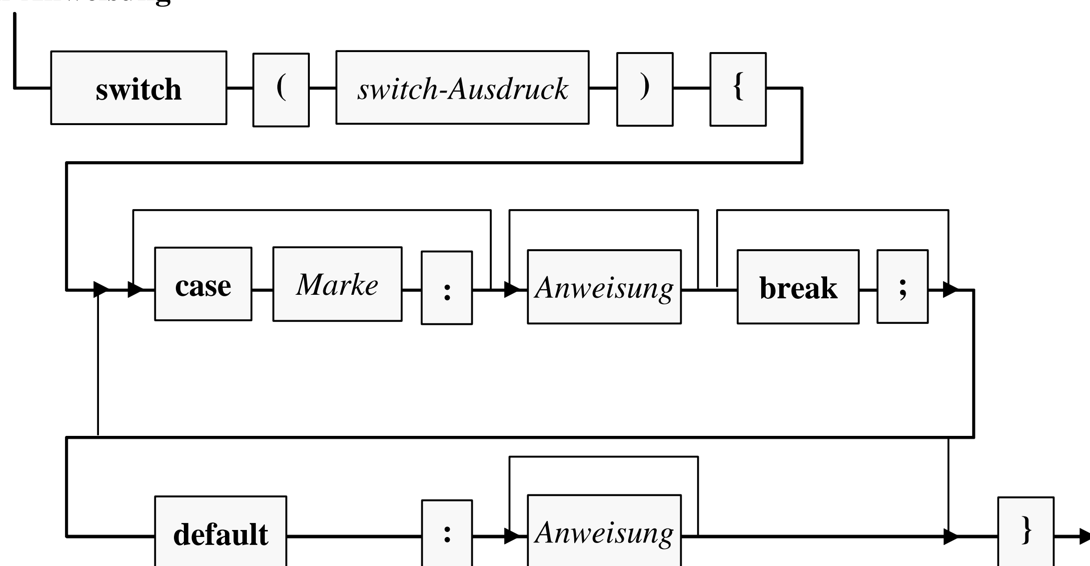
dann ist eine **switch**-Anweisung weitaus handlicher als eine verschachtelte **if-else** - Konstruktion.

In Bezug auf den Datentyp des steuernden Ausdrucks ist Java recht flexibel und erlaubt:

- Integrale primitive Datentypen mit maximal 4 Bytes:
  byte, **short**, **char** oder **int** (nicht **long**!)
- Verpackungsklassen (siehe unten) für integrale primitive Datentypen mit maximal 4 Bytes:
  Byte, **Short**, **Character** oder **Integer** (nicht **Long**!)
- Aufzählungstypen (siehe unten)
- Ab Java 7 sind auch Zeichenfolgen (Objekte der Klasse **String**) erlaubt.

Wegen ihrer großen Variabilität wird die **switch**-Anweisung mit einem Syntaxdiagramm beschrieben. Wer die Syntaxbeschreibung im Quellcode-Layout bevorzugt, kann ersatzweise einen Blick auf die gleich folgenden Beispiele werfen.

switch-Anweisung



Weil später noch ein praxisnahes (und damit auch etwas kompliziertes) Beispiel folgt, ist hier ein ebenso einfaches wie sinnfreies Exemplar zur Erläuterung der Syntax angemessen:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Prog {
  public static void main(String[] args) {
    int zahl = 2;
    final int marke1 = 1;
    switch (zahl) {
      case marke1:
        System.out.println("Fall 1 (mit break-Stopper)");
        break;
      case marke1 + 1:
        System.out.println("Fall 2 (mit Durchfall)");
      case 3:
      case 4:
        System.out.println("Fälle 3 und 4");
        break;
      default:
        System.out.println("Restkategorie");
    }
  }
}
```

</td><td>

```
Fall 2 (mit Durchfall)
Fälle 3 und 4
```

</td></tr>
</table>

Als **case**-Marken sind *konstante* Ausdrücke erlaubt, deren Wert schon der Compiler ermitteln kann (Literale, finalisierte Variablen oder daraus gebildete Ausdrücke). Anderenfalls könnte der Compiler z. B. nicht verhindern, dass mehrere Marken denselben Wert haben. Außerdem muss der Datentyp einer Marke natürlich kompatibel zum Typ des **switch**-Ausdrucks sein.

Stimmt beim Ablauf des Programms der Wert des **switch**-Ausdrucks mit einer **case**-Marke überein, dann wird die zugehörige Anweisung ausgeführt, ansonsten (falls vorhanden) die **default**-Anweisung.

Nach der Ausführung einer „angesprungenen“ Anweisung wird die **switch**-Konstruktion nur dann verlassen, wenn der Fall mit einer **break**-Anweisung abgeschlossen wird, oder wenn kein weiterer Fall mehr folgt. Ansonsten werden auch noch die Anweisungen der nächsten Fälle (ggf. inkl. **default**) ausgeführt, bis der „Durchfall“ nach unten entweder durch eine **break**-Anweisung gestoppt wird, oder die **switch**-Anweisung endet. Mit dem etwas gewöhnungsbedürftigen **Durchfall**-Prinzip kann man für geeignet angeordnete Fälle mit wenig Schreibaufwand kumulative Effekte kodieren, aber auch ärgerliche Programmierfehler durch vergessene **break**-Anweisungen produzieren.[1]

Soll für mehrere Werte des **switch**-Ausdrucks dieselbe Anweisung ausgeführt werden, setzt man die zugehörigen **case**-Marken (inklusive Schlüsselwort **case**) hintereinander und lässt die Anweisung auf die letzte Marke folgen. Leider gibt es keine Möglichkeit, eine *Serie* von Fällen durch Angabe der Randwerte (z. B. von *a* bis *k*) festzulegen.

Im folgenden Beispielprogramm wird die Persönlichkeit des Benutzers mit Hilfe seiner Farb- und Zahlpräferenzen analysiert. Während bei einer Vorliebe für Rot oder Schwarz die Diagnose sofort feststeht, wird bei den restlichen Farben auch noch die Lieblingszahl berücksichtigt:

---

[1] Im Verlauf des Kurses werden wir noch zwei weitere Optionen zum Verlassen einer **switch**-Anweisung kennenlernen:

- Per **return**-Anweisung wird nicht nur die **switch**-Anweisung verlassen, sondern die Methode.
- Per **throw**-Anweisung wird entweder ein die **switch**-Anweisung enthaltender **try**-Block verlassen oder die Methode.

```
class PerST {
    public static void main(String[] args) {
        String farbe = args[0].toLowerCase();
        int zahl = Integer.parseInt(args[1]);
        switch (farbe) {
            case "rot":
                System.out.println("Sie sind durchsetzungsfreudig und impulsiv.");
                break;
            case "schwarz":
                System.out.println("Nehmen Sie nicht alles so tragisch.");
                break;
            default:
                System.out.println("Ihre Emotionalität ist unauffällig.");
                if (zahl%2 == 0)
                    System.out.println("Sie haben einen geradlinigen Charakter.");
                else
                    System.out.println("Sie machen wohl gerne krumme Touren.");
        }
    }
}
```

Das Programm `PerST` demonstriert nicht nur die **switch**-Anweisung (hier mit einem steuernden Ausdruck vom Typ **String**), sondern auch den Zugriff auf **Programmargumente** über den **String[]** - Parameter der **main()** - Methode. Benutzer des Programms sollen beim Start ihre bevorzugte Farbe sowie ihre Lieblingszahl über Programmargumente (Kommandozeilenparameter) angeben. Wer z. B. die Farbe Blau und die Zahl 17 bevorzugt, sollte das Programm folgendermaßen starten:

```
>java PerST Blau 17
```

Im Programm wird jeweils nur *eine* Anweisung benötigt, um ein Programmargument in eine **String**- bzw. **int**-Variable zu befördern. Die zugehörigen Erklärungen werden Sie mit Leichtigkeit verstehen, sobald Methodenparameter sowie Arrays und Zeichenfolgen behandelt worden sind. An dieser Stelle greifen wir späteren Erläuterungen mal wieder etwas vor (hoffentlich mit motivierendem Effekt):

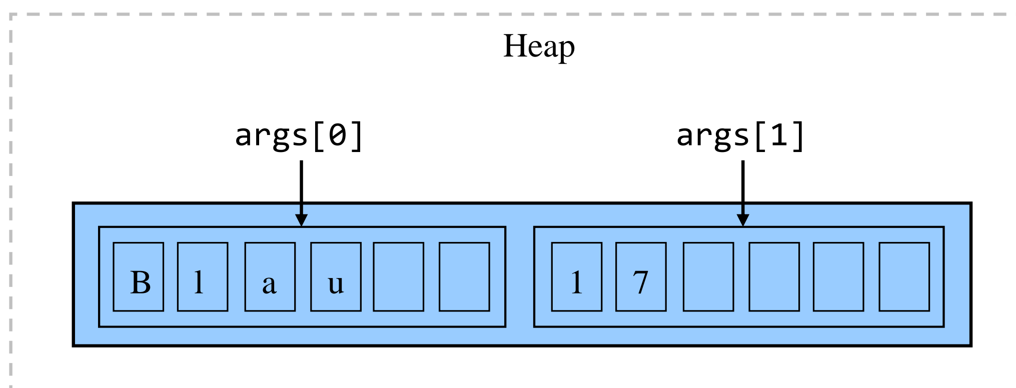
- Bei einem **Array** handelt es sich um ein Objekt, das eine Serie von Elementen desselben Typs aufnimmt, auf die man per Index, d.h. durch die mit eckigen Klammern begrenzte Elementnummer, zugreifen kann.
- In unserem Beispiel kommt ein Array mit Elementen vom Datentyp **String** zum Einsatz, wobei es sich um Zeichenfolgen handelt. Literale mit diesem Datentyp sind uns schon öfter begegnet (z. B. `"Hallo"`).
- In der Parameterliste einer Methode kann die gewünschte Arbeitsweise näher spezifiziert werden. Die **main()** - Methode einer Startklasse besitzt einen (ersten und einzigen) Parameter vom Datentyp **String[]** (Array mit **String**-Elementen). Der Datentyp dieses Parameters ist fest vorgegeben, sein Name ist jedoch frei wählbar (im Beispiel: `args`). In der Methode **main()** kann man auf `args` genauso zugreifen wie auf eine lokale Variable.
- Beim Programmstart werden der Methode **main()** von der Java Virtual Machine (JVM) als Elemente des **String[]** - Arrays `args` die Programmargumente übergeben, die der Anwender beim Start hinter den Namen der Startklasse, jeweils durch Leerzeichen getrennt, in die Kommandozeile geschrieben hat (siehe obiges Beispiel).

- Das erste Programmargument landet im ersten Element des Zeichenfolgen-Arrays `args` und wird mit `args[0]` angesprochen, weil Array-Elemente mit 0 beginnend nummeriert werden. Als Objekt der Klasse **String** wird `args[0]` im Beispielprogramm aufgefordert, die Methode **toLowerCase()** auszuführen:
  ```
  String farbe = args[0].toLowerCase();
  ```
  Diese Methode erstellt ein neues **String**-Objekt, das im Unterschied zum angesprochenen Original auf Kleinschreibung normiert ist, was die spätere Verwendung im Rahmen der **switch**-Anweisung erleichtert. Die Adresse dieses Objekts landet als **toLowerCase()** - Rückgabewert in der lokalen **String**-Referenzvariablen `farbe`.
- Das zweite Element des Zeichenfolgen-Arrays `args` (mit der Nummer 1) enthält das zweite Programmargument. Zumindest bei kooperativen Benutzern des Beispielprogramms kann diese Zeichenfolge mit der statischen Methode **parseInt()** der Klasse **Integer** in eine Zahl vom Datentyp **int** gewandelt und anschließend der lokalen Variablen `zahl` zugewiesen werden:
  ```
  int zahl = Integer.parseInt(args[1]);
  ```

Nach einem Programmstart mit dem Aufruf

```
>java PerST Blau 17
```

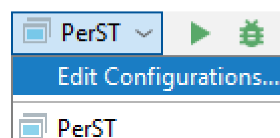
landet der **String**-Array `args` als Objekt im Heap-Bereich des programmeigenen Speichers:



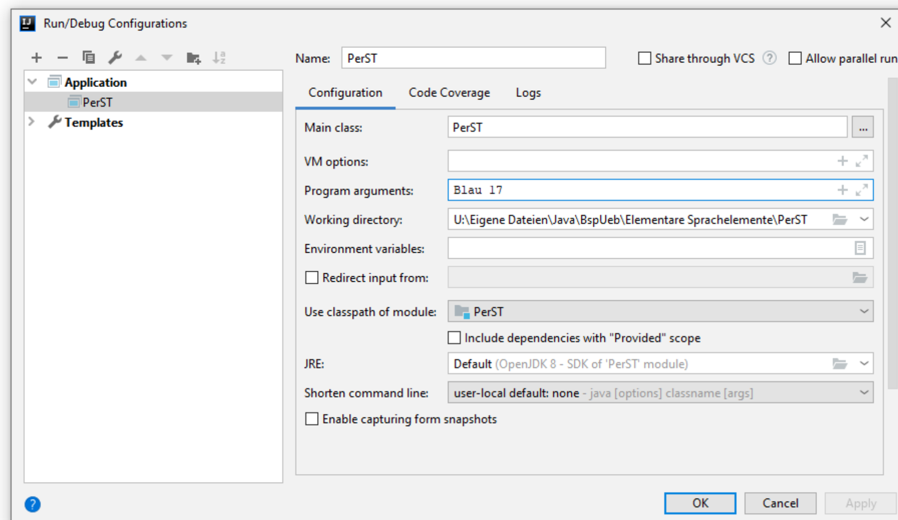
3.7.2.4 IntelliJ-Ausführungskonfiguration

Damit ein Java-Programm innerhalb unserer Entwicklungsumgebung ausgeführt werden kann, wird eine Ausführungskonfiguration (engl.: *Run Configuration*) benötigt. Eine solche wird vom Assistenten für ein neues IntelliJ-Projekt automatisch angelegt, und wir hatten bisher kaum einen Anlass zur Nachbesserung (vgl. Abschnitt 3.1.2). Für das im letzten Abschnitt vorgestellte Programm `PerST` müssen allerdings per Ausführungskonfiguration die vom Benutzer beim Programmstart übergebenen Argumente simuliert werden, sodass die automatisch erstellte Ausführungskonfiguration zu erweitern ist.

Wenn wir das Drop-Down - Menü zur Ausführungskonfiguration öffnen und das Item **Edit Configurations**

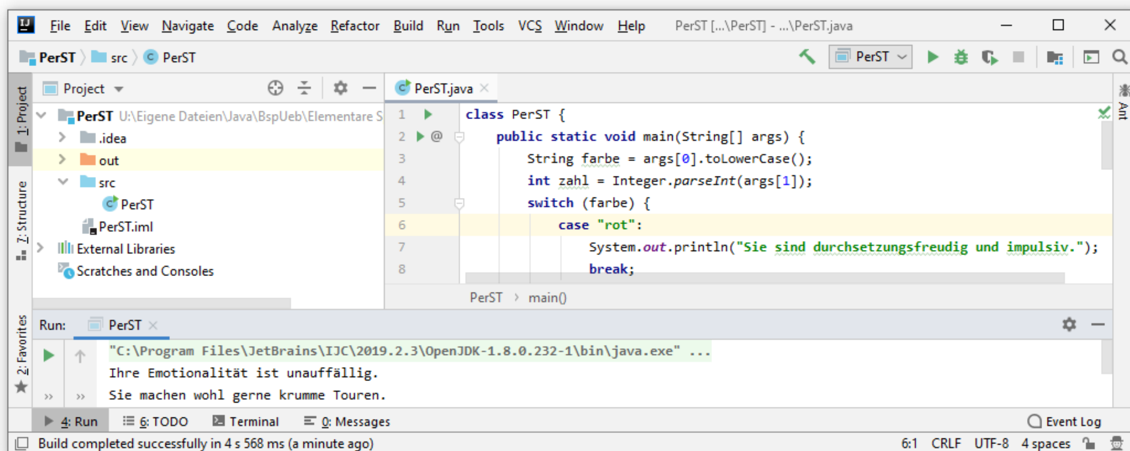


wählen, können wir im folgenden Dialog die gewünschten Programmargumente eintragen



und optional der Ausführungskonfiguration einen neuen **Namen** geben.

Anschließend lässt sich das Programm innerhalb der Entwicklungsumgebung mit Kommandozeilenargumenten ausführen, z. B.:



3.7.2.5 Vertiefung: switch-Ausdruck

Als Alternative zur **switch**-Anweisung (siehe Abschnitt 3.7.2.3) wurde der **switch**-Ausdruck in Java 12 eingeführt und in Java 13 erweitert.[1] Das Sprachmerkmal hat immer noch *Vorschaustatus*, d.h.:

- In späteren Java-Versionen können Details des **switch**-Ausdrucks geändert werden. Zwischen Java 12 und Java 13 ist z. B. das Schlüsselwort **break** durch **yield** ersetzt worden. Prinzipiell könnten die **switch**-Ausdrücke wieder komplett aus dem Java-Sprachumfang entfernt werden.
- Weil die **switch**-Ausdrücke noch den Vorschaustatus besitzen, sind sie per Voreinstellung blockiert und müssen beim Übersetzen sowie beim Ausführen eines Programms über Kommandozeilenoptionen freigegeben werden, z. B.:

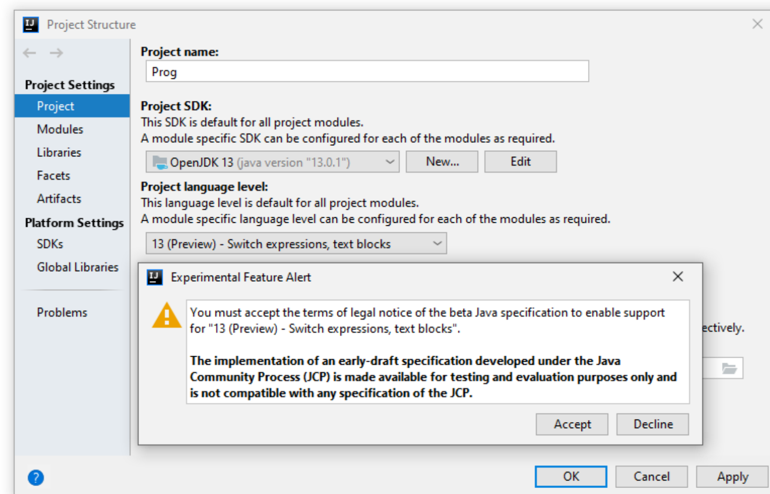
```
>javac.exe --release 13 --enable-preview Prog.java
>java.exe --enable-preview Prog
```

[1] Wir befinden uns gerade im Abschnitt über *Anweisungen*, und die **switch**-Ausdrücke hätten eigentlich im Abschnitt 3.5 behandelt werden müssen. Trotz dieses Arguments ist eine Behandlung der neuen **switch**-Ausdrücke *nach* den traditionellen (und noch lange stark verbreiteten) **switch**-Anweisungen didaktisch aber sehr viel sinnvoller.

In IntelliJ 2019.2 müssen wir uns nicht um Kommandozeilenoptionen kümmern, doch ist eine **Experimental Feature** - Anfrage zu akzeptieren, nachdem für ein Projekt das Sprachniveau auf

13 (Preview) - Switch expressions, text blocks

gesetzt worden ist, z. B.:



Auf den ersten Blick sind beim **switch**-Ausdruck keine gravierenden syntaktischen Änderungen im Vergleich zur **switch**-Anweisung festzustellen:

Quellcode	Ausgabe

<pre>class Prog {
  public static void main(String[] args) {
    int zahl = 1;
    String swr;
    swr = switch(zahl) {
      case 1:
        yield "Fall 1 (ohne Stopper,\nohne Durchfall)";
      case 2, 3:
        yield "Fälle 2 und 3";
      default:
        yield "Restkategorie";
    };
    System.out.println(swr);
  }
}</pre>
	<pre>Fall 1 (ohne Stopper,
	ohne Durchfall)</pre>

Im Detail bestehen aber doch einige Unterschiede bzw. Verbesserungen:

- Beim der neuen **switch**-Lösung haben wir es mit einem *Ausdruck* zu tun. Diese Lösung passt besser zur funktionalen Programmierung (siehe Kapitel 11), wo Zustandsänderungen durch Wertzuweisungen vermieden werden. Stattdessen werden die Ergebnisse von Ausdrücken oder Methoden als Eingabeparameter im nächsten Verarbeitungsschritt verwendet.
- Der Typ des **switch**-Ausdrucks ergibt sich aus der Verwendung des Ausdrucks im Rahmen einer Wertzuweisung oder einer **return**-Anweisung (siehe unten).
- Jeder **switch-case** muss per **yield**-Schlüsselwort einen Wert mit kompatiblem Typ liefern.[1]
- Es muss *kein Durchfall* per **break** verhindert werden.

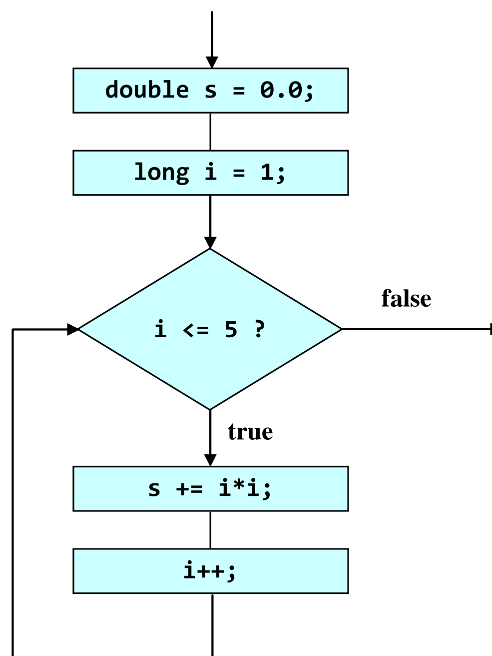
[1] Ein Ausstieg per **return**-Anweisung (siehe Abschnitt 4.3.1.2) ist also verboten. Ein Ausstieg per Ausnahmefehler (siehe Kapitel 11) ist aber möglich.

- Bei der gemeinsamen Behandlung mehrere Fälle wurde die Syntax vereinfacht. Hinter das nur einmal anzugebende angegebene Schlüsselwort **case** setzt man eine Komma-separierte Liste mit Sprungmarken (siehe Beispiel).
- Damit unter Verwendung eines **switch**-Ausdrucks eine vollständige Anweisung entsteht, muss ein Semikolon am Ende der Anweisung (also hinter der schließenden Klammer des **switch**-Blocks) stehen.

3.7.3 Wiederholungsanweisung

Eine Wiederholungsanweisung (oder schlicht: *Schleife*) kommt dann zum Einsatz, wenn eine (Verbund-)Anweisung *mehrfach* (eventuell mit systematischer Variation von Details) ausgeführt werden soll, wobei sich in der Regel schon der Gedanke daran verbietet, die Anweisung entsprechend oft in den Quelltext zu schreiben.

Im folgenden Flussdiagramm ist ein iterativer Algorithmus zu sehen, der die Summe der quadrierten natürlichen Zahlen von 1 bis 5 berechnet:[1]



Zur Realisation von iterativen Algorithmen enthält Java verschiedene Wiederholungsanweisungen (jeweils bestehend aus einer Schleifensteuerung und einer wiederholt auszuführenden Anweisung), die später in eigenen Abschnitten behandelt und hier mit vereinfachter Beschreibung im Überblick präsentiert werden:

- **Zählergesteuerte Schleife (for)**
  Bei der Ablaufsteuerung kommt eine **Zähl- oder Laufvariable** zum Einsatz, die *vor dem ersten* Schleifendurchgang initialisiert und *nach jedem* Durchlauf aktualisiert (z. B. inkrementiert) wird. Die zur Schleife gehörige (Verbund-)Anweisung wird ausgeführt, solange die Zählvariable einen festgelegten Grenzwert nicht überschritten hat.

---

[1] Das Verzweigungssymbol sieht aus darstellungstechnischen Gründen etwas anders aus als im Abschnitt 3.7.2, was aber keine Verwirrung stiften sollte. Obwohl im Beispiel eine Steigerung der Laufgrenze für die Variable `i` kaum in Frage kommt, soll an dieser Stelle das Thema *Ganzzahlüberlauf* (vgl. Abschnitt 3.6.1) in Erinnerung gerufen werden. Weil die Variable `i` vom Typ **long** ist, kann der Algorithmus bis zur Laufgrenze 3037000499 verwendet werden. Für größere `i`-Werte tritt beim Ausdruck `i*i` ein Überlauf auf, und das Ergebnis ist unbrauchbar. Eine einfache Möglichkeit zur Steigerung der maximalen sinnvollen Laufgrenze besteht darin, für eine Berechnung der Summanden per Gleitkommaarithmetik zu sorgen:

```
(double) i * i
```

- **Iterieren über die Elemente einer Kollektion (for)**
  Seit der Java-Version 5 (alias 1.5) ist es mit einer Variante der **for**-Schleife möglich, eine Anweisung für jedes Element eines Arrays oder einer anderen **Kollektion** (siehe Kapitel 10) ausführen zu lassen.
- **Bedingungsabhängige Schleife** (**while**, **do**)
  Bei jedem Schleifendurchgang wird eine **Bedingung** überprüft, und das Ergebnis entscheidet über das weitere Vorgehen:
  - **true**: Die zur Schleife gehörige Anweisung wird ein weiteres Mal ausgeführt.
  - **false**: Die Schleife wird beendet.

  Bei der *kopf*gesteuerten **while**-Schleife wird die Bedingung *vor Beginn* eines Durchgangs geprüft, bei der *fuß*gesteuerten **do**-Schleife hingegen *am Ende*. Weil man z. B. *nach* dem 3. Schleifendurchgang in keiner anderen Lage ist wie *vor* dem 4. Schleifendurchgang, geht es bei der Entscheidung zwischen Kopf- und Fußsteuerung lediglich darum, ob auf jeden Fall ein *erster* Schleifendurchgang stattfinden soll (**do**-Schleife) oder nicht (**while**-Schleife).

Die gesamte Konstruktion aus Schleifensteuerung und (Verbund-)anweisung stellt in syntaktischer Hinsicht *eine zusammengesetzte* Anweisung dar.

3.7.3.1 Zählergesteuerte Schleife (for)

Die Anweisung einer **for**-Schleife wird ausgeführt, solange eine Bedingung erfüllt ist, die normalerweise auf eine ganzzahlige Zählvariable Bezug nimmt.

Auf das Schlüsselwort **for** folgt die von runden Klammern umgebene Schleifensteuerung, wo die Vorbereitung der Laufvariablen (nötigenfalls samt Deklaration), die Fortsetzungsbedingung und die Aktualisierungsvorschrift untergebracht werden. Danach folgt die wiederholt auszuführende (Block-)Anweisung:

for (*Vorbereitung***;** *Bedingung***;** *Aktualisierung***)**   *Anweisung*

Zu den drei Bestandteilen der Schleifensteuerung sind einige Erläuterungen erforderlich, wobei hier etliche weniger typische bzw. sinnvolle Möglichkeiten weggelassen werden:

- ***Vorbereitung***
  In der Regel wird man sich auf *eine* Laufvariable beschränken und dabei einen Ganzzahltyp wählen. Somit kommen im Vorbereitungsteil der **for**-Schleifensteuerung in Frage:
  - eine Wertzuweisung für eine bereits deklarierte Variable, z. B.:
    ```
    i = 1
    ```
  - eine Variablendeklaration mit Initialisierung, z. B.
    ```
    long i = 1
    ```
    Diese Variante reduziert den Sichtbarkeitsbereich der Laufvariablen, und ein minimaler Sichtbarkeitsbereich ist generell empfehlenswert (siehe Abschnitt 3.3.9).

  Im folgenden Programm, das die Summe der quadrierten natürlichen Zahlen von 1 bis 5 berechnet, kommt die zweite Variante zum Einsatz:

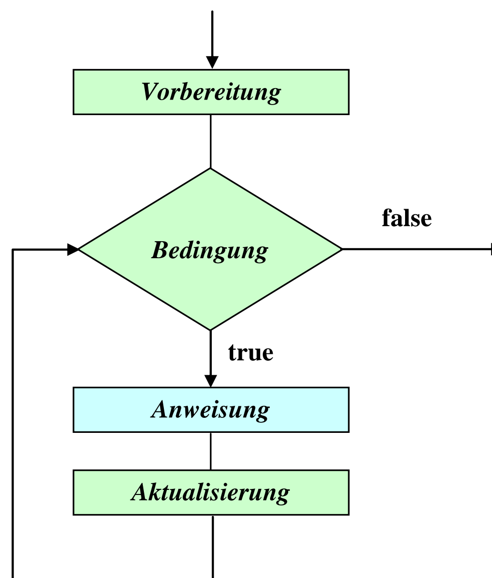
Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        double summe = 0.0;`
`        for (long i = 1; i <= 5; i++)`
`            summe += i*i;`
`        System.out.println("Quadratsumme = " + summe);`
`    }`
`}`
	`Quadratsumme = 55.0`

Der Vorbereitungsteil wird *vor dem ersten Durchlauf* ausgeführt. Eine hier deklarierte Variable ist *lokal* bzgl. der **for**-Schleife, ist also nur in deren Anweisung(sblock) sichtbar. Eine möglichst eingeschränkte Sichtbarkeit mindert das Risiko von Programmierfehlern (siehe Abschnitt 3.3.9).

- ***Bedingung***
  Üblicherweise wird eine Ober- oder Untergrenze für die Laufvariable gesetzt, doch erlaubt Java beliebige logische Ausdrücke. Die Bedingung wird *vor jedem Schleifendurchgang* geprüft. Resultiert der Wert **true**, wird der Anweisungsteil ausgeführt, anderenfalls wird die **for**-Schleife verlassen. Folglich kann es auch passieren, dass überhaupt kein Schleifendurchgang zustande kommt.
- ***Aktualisierung***
  Am Ende jedes Schleifendurchgangs (*nach* Ausführung der Anweisung) wird die Aktualisierung ausgeführt. Dabei wird meist die Laufvariable in- oder dekrementiert.

Im folgenden Flussdiagramm ist das Ablaufverhalten der **for**-Schleife dargestellt, wobei die Bestandteile der Schleifensteuerung an der grünen Farbe zu erkennen sind:



Zu den (zumindest stilistisch) bedenklichen Konstruktionen, die der Compiler klaglos akzeptiert, gehören **for**-Schleifenköpfe *ohne* Vorbereitung oder *ohne* Aktualisierung, wobei die trennenden Strichpunkte trotzdem zu setzen sind.

3.7.3.2 Iterieren über die Elemente von Arrays oder Kollektionen

Obwohl wir uns bisher mit Arrays (Objekten, die eine feste Anzahl von Elementen desselben Datentyps enthalten, siehe Abschnitt 5.1) nur anhand eines Beispiels (siehe Abschnitt 3.7.2.3) und mit Kollektionen (Containern für Elemente, siehe Kapitel 10) noch überhaupt nicht beschäftigt haben, soll die mit Java 5 (alias 1.5) eingeführte **for**-Schleifen - Variante für Arrays und Kollektionen doch

hier im Kontext mit den übrigen Wiederholungsanweisungen behandelt werden. Offiziell ist von der *erweiterten for-Schleife* (engl.: *enhanced for loop*) die Rede (siehe z. B. Gosling et al. 2019, Abschnitt 14.14.2). Bloch (2018, S. 264) bevorzugt die Bezeichnung *for-each - Schleife*. Konzentrieren Sie sich bitte auf das gleich präsentierte, leicht nachvollziehbare Beispiel, und lassen Sie sich durch die Begriffe *Array*, *Kollektion* und *Interface*, die zu später behandelten Themen gehören, nicht beunruhigen.

Das Programm `PerST` im Abschnitt 3.7.2.3 demonstriert, wie man über den **String[]** - Parameter der Methode **main()** auf die Zeichenfolgen zugreifen kann, welche der Benutzer beim Programmstart als Argumente angegeben hat. Im folgenden Programm wird durch eine erweiterte **for**-Schleife jedes Element im **String**-Array `args` mit den Programmargumenten ausgegeben:

<table>
<tr><th>Quellcode</th><th>Ausgabe nach einem Start mit
<code>>java Prog eins zwei drei</code></th></tr>
<tr><td>

```
class Prog {
    public static void main(String[] args) {
        for (String s : args)
            System.out.println(s);
    }
}
```

</td><td>

```
eins
zwei
drei
```

</td></tr>
</table>

Die Syntax der **for**-Variante für Kollektionen:

for (*Elementtyp Iterationsvariable* **:** *Serie***)**   *Anweisung*

Als Serie erlaubt der Compiler:

- einen Array (siehe Beispiel)
- ein Objekt einer Klasse, welche das Interface **Iterable** oder das generische Interface **Iterable<T>** implementiert (siehe Kapitel 9 und Abschnitt 10.4)

Im Schleifenkopf wird eine Iterationsvariable vom Datentyp der Serienelemente deklariert. Die Anweisung wird nacheinander für jedes Element der Serie ausgeführt, wobei die Iterationsvariable das gerade in Bearbeitung befindliche Serienelement anspricht.

Bloch (2018, S. 264ff) empfiehlt, wegen der folgenden Vorteile die for-each - Schleife nach Möglichkeit gegenüber der traditionellen **for**-Schleife zu bevorzugen:

- Oft werden die Flexibilität (und der Aufwand) bei der Initialisierung, Überprüfung und Aktualisierung der Indexvariablen nicht benötigt. In der for-each - Syntax beschränkt man sich auf die Elementzugriffsvariable und erhält einen besser lesbaren Quellcode.
- Durch den Verzicht auf eine Indexvariable entfallen Fehlermöglichkeiten.
- Weil in der for-each - Schleife für Array und Kollektionen dieselbe Syntax verwendet wird, macht der Wechsel der Container-Architektur wenig Aufwand.

Eine Einschränkung der **for**-Schleife für Arrays und Kollektionen besteht darin, dass man in ihrer Anweisung die Serienelemente nicht verändern kann, sodass z. B. der folgende Versuch zum „Löschen" der Elemente im **String**-Array `args` misslingt:

```
for (String s : args) {
    s = "erased";
}
```

Es liegt *kein* Syntaxfehler vor, doch die Anweisung hat keinen Effekt, weil die Iterationsvariable jeweils eine *Kopie* des aktuellen Serienelements erhält, sodass sich ein schreibender Zugriff nur innerhalb der **for**-Anweisung auswirkt.

Über eine traditionelle **for**-Schleife (vgl. Abschnitt 3.7.3.1) ist der Plan aus dem letzten Beispiel durchaus zu realisieren, wie das folgende Programm zeigt:

<table>
<tr><th>Quellcode</th><th>Ausgabe nach einem Start mit
<code>java Prog eins zwei drei</code></th></tr>
<tr><td>

```
class Prog {
    public static void main(String[] args) {
        for (String s : args)
            s = "erased";
        for (String s : args)
            System.out.println(s);

        for (int i = 0; i < args.length; i++)
            args[i] = "erased";
        for (String s : args)
            System.out.println(s);
    }
}
```

</td><td>

```
eins
zwei
drei
erased
erased
erased
```

</td></tr>
</table>

3.7.3.3 Bedingungsabhängige Schleifen

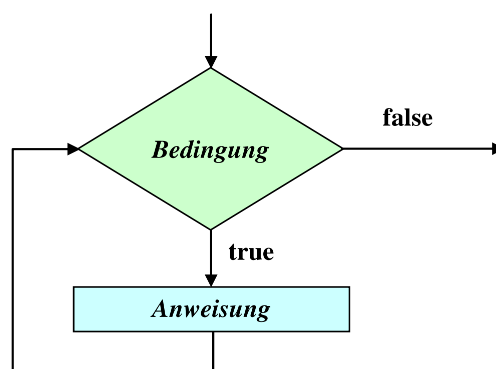
Wie die Erläuterungen zur **for**-Schleife gezeigt haben, ist die Überschrift zu diesem Abschnitt nicht sehr trennscharf, weil bei der **for**-Schleife ebenfalls eine beliebige Terminierungsbedingung angegeben werden darf. In vielen Fällen ist es eine Frage des persönlichen Geschmacks, welche Wiederholungsanweisung man zur Lösung eines konkreten Iterationsproblems einsetzt. Unter der aktuellen Abschnittsüberschrift werden traditionsgemäß die **while**- und die **do**-Schleife diskutiert.

3.7.3.3.1 while-Schleife

Die **while**-Anweisung kann als vereinfachte **for**-Anweisung beschrieben kann: Wer im Kopf einer **for**-Schleife auf Vorbereitung und Aktualisierung verzichten möchte, ersetzt besser das Schlüsselwort **for** durch **while** und erhält dann folgende Syntax:

while (*Bedingung*)   *Anweisung*

Wie bei der **for**-Anweisung wird die Bedingung *vor Beginn* eines Schleifendurchgangs geprüft. Resultiert der Wert **true**, so wird die Anweisung (ein weiteres Mal) ausgeführt, anderenfalls wird die **while**-Schleife verlassen, eventuell ohne eine einzige Ausführung der eingebetteten Anweisung:



Das im Abschnitt 3.7.3.1 vorgestellte Beispielprogramm zur Quadratsummenberechnung mit Hilfe einer **for**-Schleife kann leicht auf die Verwendung einer **while**-Schleife umgestellt werden:

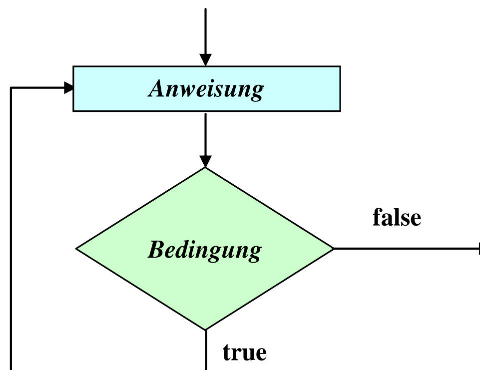
Quellcode	Ausgabe

<pre>class Prog {
    public static void main(String[] args) {
        long i = 1;
        double summe = 0.0;
        while (i <= 5) {
            summe += i*i;
            i++;
        }
        System.out.println("Quadratsumme = " + summe);
    }
}</pre>
	Quadratsumme = 55.0

Das Beispiel demonstriert einen Nachteil der **while**-Schleife gegenüber der im Abschnitt 3.7.3.1 beschriebenen **for**-Schleife: Die Laufvariable `i` muss außerhalb der **while**-Schleife deklariert werden, was zu einem unnötig großen Gültigkeitsbereich für diese lokale Variable führt (vgl. Abschnitt 3.3.9). Außerdem sind bei der **while**-Lösung der Schreibaufwand höher und die Lesbarkeit schlechter. Insgesamt ist die **for**-Schleife (in der traditionellen oder in der erweiterten Variante, siehe Abschnitte 3.7.3.1 bzw. 3.7.3.2) meist gegenüber der **while**-Schleife zu bevorzugen (Bloch 2018, S. 263).

3.7.3.3.2 do-Schleife

Bei der **do**-Schleife wird die Fortsetzungsbedingung *am Ende* der Schleifendurchläufe geprüft, sodass wenigstens *ein* Durchlauf stattfindet:



Das Schlüsselwort **while** tritt auch in der Syntax der **do**-Schleife auf:

do   *Anweisung* **while (***Bedingung***);**

do-Schleifen werden seltener benötigt als **while**-Schleifen, sind aber z. B. dann von Vorteil, wenn man vom Benutzer eine Eingabe mit bestimmten Eigenschaften einfordern möchte. Im folgenden Codesegment kommt die statische Methode `gchar()` aus der Klasse `Simput` zum Einsatz (siehe Abschnitt 3.4), die ein vom Benutzer eingetipptes und mit **Enter** quittiertes Zeichen als **char**-Wert abliefert:

```
char antwort;
do {
    System.out.println("Soll das Programm beendet werden (j/n)? ");
    antwort = Simput.gchar();
} while (antwort != 'j' && antwort != 'n' );
```

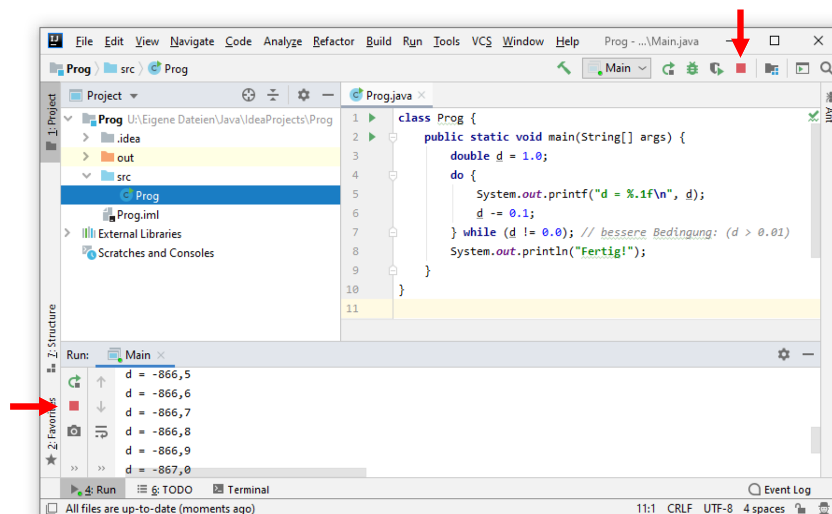
Bei einer **do**-Schleife mit Anweisungs*block* sollte man die **while**-Klausel unmittelbar hinter die schließende Blockklammer setzen (in dieselbe Zeile), um sie optisch von einer selbständigen **while**-Anweisung abzuheben (siehe Beispiel).

3.7.3.4 Endlosschleifen

Bei einer Wiederholungsanweisung (**for**, **while** oder **do**) kann es in Abhängigkeit von der Fortsetzungsbedingung passieren, dass der Anweisungsteil so lange wiederholt wird, bis das Programm von außen abgebrochen wird. Im folgenden Beispiel resultiert eine Endlosschleife aus einer ungeschickten Identitätsprüfung bei **double**-Werten (vgl. Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**):

```
class Prog {
    public static void main(String[] args) {
        double d = 1.0;
        do {
            System.out.printf("d = %.1f\n", d);
            d -= 0.1;
        } while (d != 0.0); // bessere Bedingung: (d > 0.01)
        System.out.println("Fertig!");
    }
}
```

Endlosschleifen sind als gravierende Programmierfehler unbedingt zu vermeiden. Befindet sich ein Programm in diesem Zustand muss es mit Hilfe des Betriebssystems abgebrochen werden, bei unseren Konsolenanwendungen unter Windows z. B. über die Tastenkombination **Strg+C**. Wurde der Dauerläufer innerhalb von IntelliJ gestartet, klickt man stattdessen auf das roten Quadrat in der Symbolleiste bzw. im **Run**-Fenster:



3.7.3.5 Schleifen(durchgänge) vorzeitig beenden

Mit der **break**-Anweisung, die uns schon im Abschnitt 3.7.2.3 als Bestandteil der **switch**-Anweisung begegnet ist, kann eine Schleife vorzeitig verlassen werden, wobei die Methode hinter der Schleife fortgesetzt wird.

Mit der **continue**-Anweisung erreicht man, dass der aktuelle Schleifendurchgang beendet und sofort mit dem nächsten begonnen wird (bei **for** und **while** nach Prüfung der Fortsetzungsbedingung). In der Regel kommen **break** und **continue** im Rahmen einer **if**-Anweisung zum Einsatz, z. B. im folgenden Programm zur (relativ simplen) Primzahlendiagnose:

```
class Primitiv {
  public static void main(String[] args) {
    boolean tg;
    long i, mtk, zahl;
    System.out.println("Einfacher Primzahlendetektor\n");
    while (true) {
      System.out.print("Zu untersuchende ganze Zahl von 2 bis 2^63-1 oder 0 zum Beenden: ");
      zahl = Simput.glong();
      if (Simput.checkError() || zahl == 1 || zahl < 0) {
        System.out.println("Keine Zahl oder illegaler Wert!\n");
        continue;
      }
      if (zahl == 0)
        break;
      tg = false;
      mtk = (long) (Math.sqrt(zahl) + 0.5);  //maximaler Teiler-Kandidat
      for (i = 2; i <= mtk; i++)
        if (zahl % i == 0) {
          tg = true;
          break;
        }
      if (tg)
        System.out.println(zahl + " ist keine Primzahl (Teiler: " + i + ").\n");
      else
        System.out.println(zahl + " ist eine Primzahl.\n");
    }
  System.out.println("\nVielen Dank für den Einsatz dieser Software!");
  }
}
```

Die zu untersuchende Zahl erfragt das Programm mit der statischen Methode `glong()` der im Abschnitt 3.4 vorgestellten Klasse `Simput`, die eine Rückgabe vom Typ **long** liefert. Ob die Benutzereingabe in eine **long**-Zahl gewandelt werden konnte, erfährt das Programm durch einen Aufruf der statischen `Simput`-Methode `checkError()`. Ist ein Fehler aufgetreten, liefert `checkError()` den Wert **true** zurück.

Bei einer irregulären Eingabe erscheint eine Fehlermeldung auf der Konsole, und der aktuelle Durchgang der **while**-Schleife wird per **continue** verlassen.

Auch nach der Eingabe 1 wird der aktuelle Schleifendurchgang per **continue** verlassen, weil keine Prüfung erforderlich ist. Der Benutzer wird darüber informiert, dass die 1 per Definition keine Primzahl ist.

Durch Eingabe der Zahl 0 kann das Programm beendet werden, wobei die absichtlich konstruierte **while** - „Endlosschleife" per **break** verlassen wird.

Man hätte die **continue**-Anweisung und die **break**-Anweisungen zwar vermeiden können (siehe Übungsaufgabe auf Seite 194), doch werden beim vorgeschlagenen Verfahren lästige Sonderfälle (unzulässige Werte, 0 als Terminierungssignal) auf übersichtliche Weise abgehakt, bevor der Kernalgorithmus startet.

Zum Verfahren der Primzahlendiagnose sollte noch erläutert werden, warum die Suche nach einem Teiler des Primzahlkandidaten bei seiner Wurzel enden kann (genauer: bei der größten ganzen Zahl ≤ Wurzel):

Sei d (≥ 2) ein echter Teiler der positiven, ganzen Zahl z, d.h. es gibt eine Zahl k (≥ 2) mit

$$z = k \cdot d$$

Dann ist auch k ein echter Teiler von z, und es gilt:

$$d \leq \sqrt{z} \quad \text{oder} \quad k \leq \sqrt{z}$$

Anderenfalls wäre das Produkt $k \cdot d$ größer als z. Wir haben also folgendes Ergebnis: Wenn eine Zahl z einen echten Teiler hat, dann besitzt sie auch einen echten Teiler kleiner oder gleich $\sqrt{z}$. Wenn man *keinen* echten Teiler kleiner oder gleich $\sqrt{z}$ gefunden hat, kann man die Suche einstellen, und z ist eine Primzahl.

Ist z. B. die Primzahl 23 mit der Wurzel 4,796... zu untersuchen, dann kann die Suche mit dem Teilerkandidaten 4 enden (größte ganze Zahl ≤ Wurzel). Es sind also nur die Teilerkandidaten 2, 3 und 4 zu untersuchen, sodass viel sinnloser Aufwand gespart wird. Nach dem Teilerkandidaten 2 auch noch ein Vielfaches dieser Zahl (z. B. 4) zu untersuchen, ist natürlich nicht sehr intelligent. Daher trägt das Programm den Namen *Primitiv*.

Zur Berechnung der Quadratwurzel verwendet das Beispielprogramm die statische Methode **sqrt()** aus der Klasse **Math**, über die man sich bei Bedarf in der API-Dokumentation informieren kann. Hinter der übertrieben aufwändigen wirkenden Anweisung

```
mtk = (long) (Math.sqrt(zahl) + 0.5);
```

stecken folgende Überlegungen:

- Der Laufindex `i` (Datentyp **long**) in der anschließenden **for**-Schleife wird wiederholt mit `mtk` verglichen. Damit dabei keine implizite Typumwandlung erforderlich ist, hat auch `mtk` den Typ **long** erhalten, so dass für das **sqrt()** - Ergebnis eine explizite Typanpassung erforderlich ist. Dabei kann es *nicht* zum Ganzzahlüberlauf kommen, weil das **sqrt()** - Argument eine **long**-Zahl ist.
- Ist eine ganze Zahl das Quadrat einer Primzahl, dann ist sie selbst keine Primzahl, sondern eine sogenannte *Sekundzahl*. Bei der Berechnung der Quadratwurzel aus einer Zahl im **long** - Wertebereich per Gleitkommaarithmetik kommt es in der Regel zu einer Abweichung vom mathematisch korrekten Ergebnis (vgl. Abschnitt 3.3.7.1). Bei einer Sekundzahl könnte das **sqrt()** - Ergebnis knapp unter dem korrekten ganzzahligen Wert liegen, was bei der Wandlung in **long** (durch Abschneiden der Nachkommastellen) zu einem Fehler führen würde. Infolgedessen würde die Sekundzahl falsch als Primzahl erkannt. Um dies zu verhindern, wird vor der Wandlung in **long** der Wert 0,5 addiert und so eine Rundung erzwungen.
  Die Maßnahme ist eigentlich überflüssig, denn laut JDK-Dokumentation liefert **Math.sqrt()** einen korrekt gerundeten Wert:[1]
  Returns the correctly rounded positive square root of a double value.
  Math.sqrt() berechnet tatsächlich z. B. für 994009 (= Quadrat der Primzahl 997) den korrekten Wert ohne jedes Anzeichen von Gleitkommaungenauigkeit.

3.8 Entspannungs- und Motivationseinschub: GUI-Standarddialoge

Nach etlichen recht anstrengenden Themen, soll dieser Abschnitt zur Entspannung und zur Regeneration Ihrer Motivation beitragen. Sie lernen GUI-Standarddialoge (*Graphical User Interface*) zur Abfrage von Werten und zur Präsentation von Meldungen kennen, welche die Klasse **JOptionPane** aus dem Paket **javax.swing** über statische Methoden zur Verfügung stellt. Den Standarddialog zur Meldungsausgabe haben wir in seiner einfachsten Form übrigens schon im Abschnitt 3.3.11.5 verwendet.

Die Klasse **JOptionPane** aus dem Paket **javax.swing** an dieser Stelle trotz der erklärten Absicht zum Wechsel von der traditionellen GUI-Lösung *Swing* auf die moderne Alternative *JavaFX* (alias

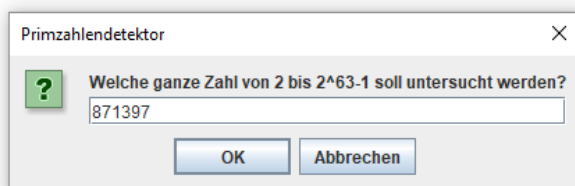
[1] https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html#sqrt-double-

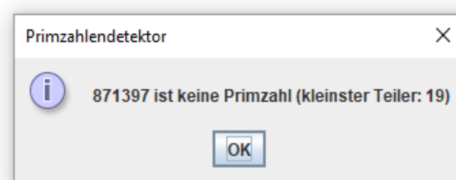
OpenJFX) zu verwenden, wirkt inkonsequent. Allerdings wurde schon im Abschnitt 3.3.11.5 demonstriert, dass es nur mit der Swing-Klasse **JOptionPane** gelingt, ohne Kontakt mit anspruchsvollen Java-Themen eine elementare GUI-Anwendung zu erstellen. Außerdem sind Kenntnisse über die Klasse **JOptionPane** von Nutzen, weil sie in vielen vorhandenen Programmen anzutreffen ist.

Wir erstellen zum Primzahldiagnoseprogramm aus Abschnitt 3.7.3.5 mit erstaunlich geringem Aufwand die folgende Variante

```
import javax.swing.JOptionPane;
class PrimitivJop {
   public static void main(String[] args) {
      String s;
      boolean tg;
      long i, mtk, zahl;
      while (true) {
         s = JOptionPane.showInputDialog(null,
               "Welche ganze Zahl von 2 bis 2^63-1 soll untersucht werden?",
               "Primzahlendetektor", JOptionPane.QUESTION_MESSAGE);
         zahl = Long.parseLong(s);
         if (zahl <= 1)
            continue;
         mtk = (long) (Math.sqrt(zahl) + 0.5);   //maximaler Teilerkandidat
         tg = false;
         for (i = 2; i <= mtk; i++)
            if (zahl % i == 0) {
               tg = true;
               break;
            }
         if (tg)
            s = String.valueOf(zahl) +
               " ist keine Primzahl (kleinster Teiler: " + String.valueOf(i)+")";
         else
            s = String.valueOf(zahl) + " ist eine Primzahl";
         JOptionPane.showMessageDialog(null,
             s, "Primzahlendetektor", JOptionPane.INFORMATION_MESSAGE);
      }
   }
}
```

mit grafischer Bedienoberfläche:





Die linke Dialogbox zur Erfassung des Primzahlkandidaten geht auf einen Aufruf der statischen **JOptionPane**-Methode **showInputDialog**() zurück:

> **public static String showInputDialog(Component** *parentComponent*,
> **Object** *message*, **String** *title*, **int** *messageType*)

Auf die Disziplin des Benutzers vertrauend lassen wir die als Rückgabewert gelieferte Zeichenfolge ohne Prüfung von der statischen **Long**-Methode **parseLong**() in einen **long**-Wert wandeln.

Die rechte Dialogbox mit dem Ergebnis der Primzahlendiagnose produzieren wir mit Hilfe der statischen **JOptionPane**-Methode **showMessageDialog**():

> **public static void showMessageDialog(Component** *parentComponent*,
> **Object** *message*, **String** *title*, **int** *messageType*)

Die auszugebende Zeichenfolge wird folgendermaßen erstellt:

- Von der statischen Methode **valueOf()** der Klasse **String** erhalten wir die Zeichenfolgen-Repräsentationen des darzustellenden **long**-Werts.
- Die Möglichkeit, mehrere Zeichenfolgen mit dem Plusoperator zu verketten, kennen wir schon seit Abschnitt 3.2.1, z. B.:

```
s = String.valueOf(zahl) + " ist eine Primzahl";
```

Weil der Klassenname **JOptionPane** im Quellcode mehrfach auftaucht, wird er zu Beginn importiert, damit anschließend kein Paketnamenspräfix erforderlich ist (vgl. Abschnitt 3.1.7).

Die statischen **JOptionPane**-Methoden **showInputDialog()** und **showMessageDialog()** kennen etliche Parameter (Argumente zur näheren Bestimmung der Ausführung), die in der folgenden Tabelle beschrieben werden:

Name	Erläuterung
parentComponent	Standarddialoge sind oft einem anderen (elterlichen) Fenster zu- oder untergeordnet. Die Angabe eines Fensterobjekts (an Stelle der Alternative **null**) hat zur Folge, dass der Standarddialog in der Nähe dieses Fensters erscheint.
message	Dieser Text erscheint in der Dialogbox.
title	Dieser Text erscheint in der Titelzeile der Dialogbox.
messageType	Dieser Parameter legt den Typ der Nachricht fest, der auch über das Icon am linken Rand der Dialogbox entscheidet. Als Werte sind die folgenden statischen und finalisierten Felder der Klasse **JOptionPane** erlaubt, die jeweils für einen **int**-Wert stehen:

JOptionPane-Konstante	**int**
JOptionPane.PLAIN_MESSAGE	-1
JOptionPane.ERROR_MESSAGE	0
JOptionPane.INFORMATION_MESSAGE	1
JOptionPane.WARNING_MESSAGE	2
JOptionPane.QUESTION_MESSAGE	3

In den folgenden Fällen liefert die Methode **showInputDialog()** *keine* als ganze Zahl im **long**-Wertebereich interpretierbare Rückgabe:

- Der Benutzer hat eine ungültige Zeichenfolge eingetragen, z. B. ...
  - sieben (überhaupt keine Zahl)
  - 3,14 (keine ganze Zahl)
  - 9223372036854775808 (ganze Zahl außerhalb des **long**-Wertebereichs).
- Der Benutzer hat den Input-Dialog abgebrochen (auf die Schaltfläche **Abbrechen** geklickt, auf das Schließkreuz am rechten Rand der Titelzeile geklickt oder die **Esc**-Taste gedrückt).

Unser Programm endet dann mit einer unbehandelten Ausnahme, z. B.:

```
Exception in thread "main" java.lang.NumberFormatException: null
  at java.base/java.lang.Long.parseLong(Long.java:552)
  at java.base/java.lang.Long.parseLong(Long.java:631)
  at PrimitivJop.main(PrimitivJop.java:11)
```

Im Kapitel 11 werden Sie erfahren, wie man solche Ausnahmen abfangen und behandeln kann.

Wird der Primzahlendetektor konsolenfrei (mit dem JVM-Werkzeug **javaw.exe**) gestartet, bemerkt der Benutzer nichts von der fehlenden Ausnahmebehandlung:

```
>javaw PrimitivJop
```

Wie man unter Windows eine Verknüpfungsdatei zum Programmstart per Doppelklick anlegt, wurde im Abschnitt 1.2.3 beschrieben.

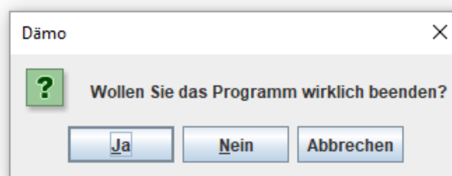
Von den zahlreichen weiteren Möglichkeiten der Klasse **JOptionPane** (siehe API-Dokumentation) soll noch die statische Methode **showConfirmDialog()** erwähnt werden:

> **public static int showConfirmDialog(Component** *parentComponent***,**
> **Object** *message***, String** *title***, int** *optionType***)**

Sie eignet sich für Ja/Nein - Fragen an den Benutzer, präsentiert ein konfigurierbares Ensemble von Schaltflächen (**OK**, **Ja**, **Nein**, **Abbrechen**) und teilt per **int**-Rückgabewert mit, über welche Schalfläche der Benutzer den Dialog beendet hat. Das folgende Beispielprogramm wird auf Benutzerwunsch über die statische Methode **exit()** der Klasse **System** beendet, wobei das Betriebssystem per **exit()** - Parameter den Returncode 0 erfährt:

```
import javax.swing.JOptionPane;
class Prog {
  public static void main(String[] args) {
    while (true)
      if (JOptionPane.showConfirmDialog(null,
          "Wollen Sie das Programm wirklich beenden?",
          "Dämo", JOptionPane.YES_NO_CANCEL_OPTION) == JOptionPane.YES_OPTION)
        System.exit(0);
  }
}
```

Über den Parameter *optionType* (Typ: **int**) steuert man die Schaltflächenausstattung, z. B.:



Über **int**-Werte oder äquivalente statische und finalisierte Felder der Klasse **JOptionPane** sind vier Ausstattungsvarianten wählbar:

optionType-Wert		Resultierende Schalter
JOptionPane-Konstante	**int**	
JOptionPane.DEFAULT_OPTION	-1	**OK**
JOptionPane.YES_NO_OPTION	0	**Ja**, **Nein**
JOptionPane.YES_NO_CANCEL_OPTION	1	**Ja**, **Nein**, **Abbrechen**
JOptionPane.OK_CANCEL_OPTION	2	**OK**, **Abbrechen**

Durch ihren Rückgabewert informiert die Methode **showConfirmDialog()** darüber, welchen Schalter der Benutzer betätigt hat. Bei der Schalterausstattung wie im obigen Beispiel (**JOptionPane.YES_NO_CANCEL_OPTION**) können die folgenden Rückgabewerte vom Typ **int** auftreten, die auch über statische und finalisierte Felder der Klasse **JOptionPane** ansprechbar sind:

Vom Benutzer gewählter Schalter	**showConfirmDialog()** - Rückgabewerte	
	JOptionPane-Konstante	**int**
Schließkreuz in der Titelzeile oder **Esc**-Taste	**JOptionPane.CLOSED_OPTION**	-1
Ja	**JOptionPane.YES_OPTION**	0
Nein	**JOptionPane.NO_OPTION**	1
Abbrechen	**JOptionPane.CANCEL_OPTION**	2

3.9 Übungsaufgaben zum Kapitel 3

Abschnitt 3.1 (Einstieg)

1) Welche **main()** - Varianten sind zum Starten eines Programms geeignet?

```
public static void main(String[] irrelevant) { … }
public void main(String[] args) { … }
public static void main() { … }
static public void main(String[] args) { … }
public static void main(String[] Args) { … }
static public void Main(String[] args) { … }
static public int main(String[] args) { … }
```

2) Welche von den folgenden Bezeichnern sind unzulässig?

```
4you
main
else
Alpha
lösung
```

3) Das folgende Programm gibt den Wert der Klassenvariablen **PI** aus der API-Klasse **Math** im Paket **java.lang** aus:

```
class Prog {
   public static void main(String[] args) {
      System.out.println("PI = " + Math.PI);
   }
}
```

Warum ist es hier *nicht* erforderlich, den Paketnamen anzugeben bzw. zu importieren?

Abschnitt 3.2 (Ausgabe bei Konsolenanwendungen)

1) Schreiben Sie ein Programm, das die Klassenvariable **PI** aus der API-Klasse **Math** wiederholt mit verschiedener Genauigkeit linksbündig ausgibt:

```
3,1        3,14       3,142
3,1416     3,14159    3,141593
```

2) Wie ist das fehlerhafte „Rechenergebnis“ des folgenden Programms zu erklären?

Quellcode	Ausgabe

`class Prog {`
`   public static void main(String[] args) {`
`      System.out.println("3.3 + 2 = " + 3.3 + 2);`
`   }`
	`3.3 + 2 = 3.32`

}	

Das zur exakten Beantwortung der Frage benötigte Hintergrundwissen (über die Auswertungsreihenfolge von Operatoren) wurde noch *nicht* vermittelt, sodass Sie nicht allzu viel Zeit investieren sollten. Vielleicht hilft der Tipp, dass ein geschickt positioniertes Paar runder Klammern zur gewünschten Ausgabe führt:

```
3.3 + 2 = 5.3
```

Abschnitt 3.3 (Variablen und Datentypen)

1) Entlarven Sie wieder einmal falsche Behauptungen:

1. Die lokalen Variablen einer Methode haben stets einen primitiven Datentyp.
2. Lokale Variablen befinden sich auf dem Stack.
3. Referenzvariablen werden auf dem Heap abgelegt.
4. Bei der objektorientierten Programmierung sollten möglichst keine primitiven Variablen verwendet werden.

2) Im folgenden Programm wird der **char**-Variablen `z` eine *Zahl* zugewiesen, die sie offenbar unbeschädigt an eine **int**-Variable weitergeben kann, wobei der `z`-Inhalt von **println()** aber als Buchstabe ausgegeben wird. Wie erklären sich diese Merkwürdigkeiten?

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
   public static void main(String args[]) {
      char z = 99;
      int i = z;
      System.out.println("z = " + z + "\ni = " + i);
   }
}</pre></td><td><pre>z = c
i = 99</pre></td></tr>
</table>

Wie kann man das Zeichen ‚c' über eine Unicode-Escape-Sequenz ansprechen?

3) Wieso klagt der OpenJDK 8 - Compiler über ein unbekanntes Symbol, obwohl die Variable `i` deklariert worden ist?

<table>
<tr><th>Quellcode</th><th>Fehlermeldung des OpenJDK 8 - Compilers</th></tr>
<tr><td><pre>class Prog {
   public static void main(String[] args) {{
      int i = 2;
   }
      System.out.println(i);
   }
}</pre></td><td><pre>Prog.java:5: error: cannot find symbol
                System.out.println(i);
                                   ^
  symbol: variable i
  location: class Prog
1 error</pre></td></tr>
</table>

4) Schreiben Sie bitte ein Java-Programm, das folgende Ausgabe macht:

```
Dies ist ein Java-Zeichenkettenliteral:
   "Hallo"
```

5) Beseitigen Sie bitte alle Fehler im folgenden Programm:

```
class Prog {
   static void main(String[] args) {
      float PI = 3,141593;
      double radius = 2,0;
      System.out.printLine("Der Flächeninhalt beträgt: + PI*radius*radius);
```

```
    }
}
```

Abschnitt 3.4 (Eingabe bei Konsolen)

1) Führen Sie für das IntelliJ-Projekt `Prog` (siehe Abschnitt 3.1.2), das wir zum Üben von diversen elementaren Sprachelementen verwenden, die im Abschnitt 3.4.2 beschriebene IntelliJ-Konfiguration aus, und lassen Sie das im Abschnitt 3.4.1 beschriebene Fakultätsprogramm mit `Simput.gint()` - Aufruf laufen.

Testen Sie auch die `Simput`-Methoden `gdouble()` und `gchar()`.

Abschnitt 3.5 (Operatoren und Ausdrücke)

1) Welche Werte und Datentypen besitzen die folgenden Ausdrücke?

```
6/4*2.0
(int)6/4.0*3
(int)(6/4.0*3)
3*5+8/3%4*5
```

2) Welcher Datentyp resultiert, wenn man eine **byte**- und eine **short**-Variable addiert?

3) Welche Werte haben die **int**-Variablen `erg1` und `erg2` am Ende des folgenden Programms?

```
class Prog {
    public static void main(String[] args) {
        int i = 2, j = 3, erg1, erg2;
        erg1 = (i++ == j ? 7 : 8) % 3;
        erg2 = (++i == j ? 7 : 8) % 2;
        System.out.println("erg1 = " + erg1 + "\nerg2 = " + erg2);
    }
}
```

4) Welche Wahrheitswerte erhalten im folgenden Programm die booleschen Variablen `la1` bis `la3`?

```
class Prog {
    public static void main(String[] args) {
        boolean la1, la2, la3;
        int i = 3;
        char c = 'n';

        la1 = 2 > 3 && 2 == 2 ^ 1 == 1;
        System.out.println(la1);

        la2 = (2 > 3 && 2 == 2) ^ (1 == 1);
        System.out.println(la2);

        la3 = !(i > 0 || c == 'j');
        System.out.println(la3);
    }
}
```

5) Erstellen Sie ein Java-Programm, das den Exponentialfunktionswert e^x zu einer vom Benutzer eingegebenen Zahl *x* bestimmt und ausgibt, z. B.:

Eingabe: `Argument: 1`
Ausgabe: `exp(1,000000) = 2,7182818285`

Hinweise:

- Suchen Sie mit Hilfe der Dokumentation zur Klasse **Math** im API-Paket **java.lang** eine passende Methode.
- Zum Einlesen des Arguments können Sie die Methode `gdouble()` aus unserer Eingabeklasse `Simput` verwenden, die eine vom Benutzer (mit Komma als Dezimaltrennzeichen) eingetippte und mit **Enter** quittierte Zahl als **double**-Wert abliefert (siehe Abschnitt 3.4).
- Über Möglichkeiten zur formatierten Ausgabe informiert der Abschnitt 3.2.2.

6) Erstellen Sie ein Programm, das einen DM-Betrag entgegen nimmt und diesen in Euro konvertiert. In der Ausgabe sollen ganzzahlige, korrekt gerundete Werte für Euro und Cent erscheinen, z. B.:

Eingabe: `DM-Betrag: 321`
Ausgabe: `164 Euro und 12 Cent`

Hinweise:

- Umrechnungsfaktor: 1 Euro = 1,95583 DM
- Zum Einlesen des DM-Betrags können Sie die Methode `gdouble()` aus unserer Eingabeklasse `Simput` verwenden (siehe Abschnitt 3.4).

7) Erstellen Sie ein Programm, das eine ganze Zahl entgegen nimmt und den Benutzer darüber informiert, ob die Zahl gerade ist oder nicht, z. B.:

Eingabe: `Ganze Zahl: 13`
Ausgabe: `ungerade`

Außer einem Methodenaufruf für die Eingabeaufforderung, z. B.:

```
System.out.print("Ganze Zahl: ");
```

soll das Programm **nur eine einzige** Anweisung enthalten.

Hinweis: Verwenden Sie die Methode `gint()` aus der Klasse `Simput`, um die Eingabe entgegenzunehmen (siehe Abschnitt 3.4).

Abschnitt 3.6 (Über- und Unterlauf bei numerischen Variablen)

1) Welche der folgenden Aussagen sind richtig bzw. falsch?

1. Kommt es bei einer Ganzzahlvariablen zum Überlauf, stoppt das Programm mit einem Laufzeitfehler.
2. Bei Objekten der Klasse **BigDecimal** kann weder ein Über- noch ein Unterlauf auftreten.
3. Bei einer versuchten Gleitkommadivision durch null stoppt das Programm mit einem Laufzeitfehler.
4. Man sollte bei numerischen Aufgaben grundsätzlich Objekte aus den Klassen **BigDecimal** und **BigInteger** verwenden.

Abschnitt 3.7 (Anweisungen (zur Ablaufsteuerung))

1) Warum liefert dieses Programm widersprüchliche Auskünfte über die boolesche Variable `b`?

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Prog {
    public static void main(String[] args) {
        boolean b = true;
        if (b = false)
            System.out.println("b ist false");
        else
            System.out.println("b ist true");
        System.out.println("\nKontr.ausgabe: b ist "+b);
    }
}
```

</td><td>

```
b ist true

Kontr.ausgabe: b ist false
```

</td></tr>
</table>

2) Das folgende Programm soll Buchstaben nummerieren:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Prog {
    public static void main(String[] args) {
        char bst = 'a';
        byte nr = 0;

        switch (bst) {
            case 'a': nr = 1;
            case 'b': nr = 2;
            case 'c': nr = 3;
        }
        System.out.println("Zu "+bst+
                " gehört die Nummer "+nr);
    }
}
```

</td><td>

```
Zu a gehört die Nummer 3
```

</td></tr>
</table>

Warum liefert es zum Buchstaben *a* die Nummer 3, obwohl für diesen Fall die Anweisung

```
nr = 1
```

vorhanden ist?

3) Erstellen Sie eine Variante des Primzahlen-Diagnoseprogramms aus Abschnitt 3.7.3.5, die ohne **break** und **continue** auskommt.

4) Wie oft wird die folgende **while**-Schleife ausgeführt?

```
class Prog {
    public static void main(String[] args) {
        int i = 0;
        while (i < 100);
        {
            i++;
            System.out.println(i);
        }
    }
}
```

5) Verbessern Sie das als Übungsaufgabe zum Abschnitt 3.5 in Auftrag gegebene Programm zur DM-Euro - Konvertierung so, dass es nicht für jeden Betrag neu gestartet werden muss. Vereinbaren Sie mit dem Benutzer ein geeignetes Verfahren für den Fall, dass er das Programm doch irgendwann einmal beenden möchte.

6) In dieser Aufgabe sollen Sie verschiedene Varianten von Euklids Algorithmus zur Bestimmung des größten gemeinsamen Teilers (GGT) zweier natürlicher Zahlen u und v implementieren und die Laufzeitunterschiede messen. Verwenden Sie als ersten Kandidaten den im Einführungsbeispiel zum Kürzen von Brüchen (Methode `kuerze()` der Klasse `Bruch`) benutzten Algorithmus (siehe Abschnitt 1.1.2). Sein Problem besteht darin, dass bei stark unterschiedlichen Zahlen u und v sehr viele Subtraktions-Operationen erforderlich sind. In der meist benutzten Variante des Euklidischen Verfahrens wird dieses Problem vermieden, indem an Stelle der Subtraktion die Modulo-Operation zum Einsatz kommt, basierend auf dem folgendem Satz der mathematischen Zahlentheorie:

> Für zwei natürliche Zahlen u und v (mit $u > v$) ist der GGT gleich dem GGT von u und $u \% v$ (u modulo v).

Begründung (analog zu Abschnitt 1.1.3): Für natürliche Zahlen u und v mit $u > v$ gilt:

$$x \text{ ist gemeinsamer Teiler von } u \text{ und } v$$
$$\Leftrightarrow$$
$$x \text{ ist gemeinsamer Teiler von } v \text{ und } u \% v$$

Der GGT-Algorithmus per Modulo-Operation läuft für zwei natürliche Zahlen u und v ($u \geq v > 0$) folgendermaßen ab:

> Es wird geprüft, ob u durch v teilbar ist.
> Trifft dies zu, ist v der GGT.
> Anderenfalls ersetzt man:
> > u durch v
> > v durch $u \% v$
>
> Das Verfahren startet neu mit den kleineren Zahlen.

Die Voraussetzung $u \geq v$ ist nicht wesentlich, weil beim Start mit $u < v$ der erste Algorithmusschritt die beiden Zahlen vertauscht.

Um den Zeitaufwand für beide Varianten zu messen, eignet sich die statische Methode **currentTimeMillis()** aus der Klasse **System** im Paket **java.lang** (siehe API-Dokumentation). Sie liefert als **long**-Wert die aktuelle Zeit in Millisekunden (seit dem 1. Januar 1970).

Für die Beispielwerte $u = 999000999$ und $v = 36$ liefern beide Euklid-Varianten sehr verschiedene Laufzeiten (CPU: Intel Core i3 mit 3,2 GHz):

```
GGT-Bestimmung mit Euklid (Differenz)

Erste Zahl:    999000999

Zweite Zahl:   36

GGT:           9

Benötigte Zeit: 35 Millisek.
```

```
GGT-Bestimmung mit Euklid (Modulo)

Erste Zahl:    999000999

Zweite Zahl:   36

GGT:           9

Benötigte Zeit: 0 Millisek.
```

7) Wegen der beschränkten Genauigkeit bei der Speicherung von binären Gleitkommazahlen (siehe Abschnitt 3.3.6) kann ein Rechner die **double**-Werte 1,0 und $1{,}0 + 2^{-i}$ ab einem bestimmten Exponenten i nicht mehr voneinander unterscheiden. Bestimmen Sie mit einem Testprogramm den größten ganzzahligen Exponenten i, für den man noch erhält:

$$1{,}0 + 2^{-i} > 1{,}0$$

In dem (zur freiwilligen Lektüre empfohlenen) Vertiefungsabschnitt 3.3.7.1 findet sich eine Erklärung für das Ergebnis.

4 Klassen und Objekte

Objektorientierte Software-Entwicklung besteht nach unserem bisherigen Kenntnistand im Wesentlichen aus der Definition von **Klassen**, die aufgrund einer vorangegangenen objektorientierten Analyse …

- als Baupläne für Objekte
- und/oder als Akteure

konzipiert werden. Wenn ein spezieller Akteur im Programm nur *einfach* benötigt wird, kann eine handelnde Klasse diese Rolle übernehmen.[1] Sind hingegen mehrere Individuen einer Gattung erforderlich (z. B. mehrere Brüche in einem Bruchrechnungsprogramm oder mehrere Fahrzeuge in der Speditionsverwaltung), dann ist eine Klasse mit Bauplancharakter gefragt.

Für eine Klasse und/oder ihre Objekte werden **Eigenschaften** (**Felder**) und **Handlungskompetenzen** (**Methoden**) deklariert bzw. definiert. Diese werden als **Member** der Klasse bezeichnet (dt.: *Mitglieder*).

In den Methoden eines Programms werden Aufgaben erledigt bzw. Algorithmen realisiert. Ein agierendes (eine Methode ausführendes) Objekt bzw. eine agierende Klasse muss nicht alles selbst erledigen, sondern kann vordefinierte (z. B. der Standardbibliothek entstammende) oder im Programm definierte Klassen einspannen, z. B.:

- Eine Klasse aus der Standardbibliothek wird beauftragt:
  ```
  double res = Math.exp(arg);
  ```
- Ein Objekt, das beim Laden einer Klasse aus der Standardbibliothek automatisch entsteht und über eine statische (klassenbezogene) Referenzvariable ansprechbar ist, wird beauftragt:
  ```
  System.out.println(arg);
  ```
- Ein explizit im Programm erstelltes Objekt aus einer im Programm definierten Klasse wird beauftragt:
  ```
  Bruch b1 = new Bruch();
  b1.frage();
  ```

Mit dem „Beauftragen“ eines Objekts oder einer Klasse bzw. mit dem „Zustellen einer Botschaft“ ist nichts anderes gemeint als ein Methodenaufruf.

Unsere vorläufige, auch im aktuellen Kapitel 4 zugrundeliegende Vorstellung von einem Computer-Programm lässt sich so beschreiben:

- Ein Programm besteht aus Klassen, die als Baupläne für Objekte und/oder als Akteure dienen.
- Die Akteure (Objekt und Klassen) haben jeweils einen Zustand (abgelegt in Feldern).
- Sie können Botschaften empfangen und senden (Methoden ausführen und aufrufen).

In der Hoffnung, dass die bisher präsentierten Eindrücke von der objektorientierten Programmierung (OOP) neugierig gemacht und nicht abgeschreckt haben, kommen wir nun zur systematischen

---

[1] Eine nur einfach zu besetzende Rolle von einer Klasse übernehmen zu lassen, ist keinesfalls in jeder Situation eine ideale Design-Entscheidung und wird im Manuskript hauptsächlich der Einfachheit halber bevorzugt. In einer späteren Phase auf dem Weg zum professionellen Entwickler sollte man sich unbedingt mit dem sogenannten *Singleton-Pattern* beschäftigen (siehe z. B. Bloch 2018, S. 17ff). Dabei geht es um Klassen, von denen innerhalb einer Anwendung garantiert nur ein Objekt entsteht. Hier fungiert also ein Objekt statt einer Klasse als Solist, was etliche Vorteile bietet, z. B.:

- Die Adresse des Solo-Objekts kann an Methoden als Parameter übergeben werden.
- Die Vererbungstechnik der OOP wird besser unterstützt (inkl. Polymorphie, siehe Abschnitt 7.7).
- Eine Singleton-Klasse kann Interfaces implementieren (siehe Kapitel 9).

Behandlung dieser Software-Technologie. Für die im Kapitel 1 speziell für größere Projekte empfohlene objektorientierte Analyse und Modellierung, z. B. mit Hilfe der Unified Modeling Language (UML), ist dabei leider keine Zeit vorhanden (siehe z. B. Balzert 2011; Booch et al. 2007).

4.1 Überblick, historische Wurzeln, Beispiel

4.1.1 Einige Kernideen und Vorzüge der OOP

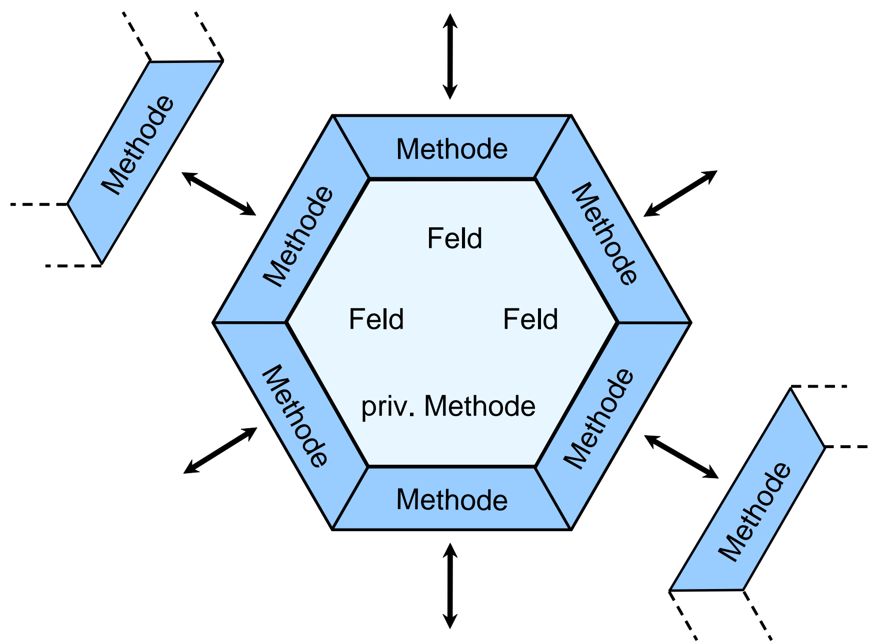
Lahres & Rayman (2009, Kapitel 2) nennen in ihrem *Praxisbuch Objektorientierung* unter Berufung auf **Alan Kay**, der den Begriff *Objektorientierte Programmierung* geprägt und die objektorientierte Programmiersprache *Smalltalk* entwickelt hat, als unverzichtbare OOP-Grundelemente:

- **Datenkapselung**
  Eine Klasse erlaubt in der Regel fremden Klassen keinen direkten Zugriff auf ihre Zustandsdaten. So wird das Risiko für das Auftreten inkonsistenter Zustände reduziert. Außerdem kann der Klassendesigner Implementierungsdetails ohne Nebenwirkungen auf andere Klassen ändern. Mit der Datenkapselung haben wir uns schon im Abschnitt 1.1 beschäftigt.
- **Vererbung**
  Aus einer vorhandenen Klasse lassen sich zur Lösung neuer Aufgaben spezialisierte Klassen ableiten, die alle Member der Basisklasse erben. Hier findet eine Wiederverwendung von Software ohne lästiges und fehleranfälliges Kopieren von Quellcode statt. Beim Design der abgeleiteten Klasse kann man sich darauf beschränken, neue Member zu definieren oder bei manchen Erbstücken (z. B. Methoden) Modifikationen zur Anpassung an die neue Aufgabe vorzunehmen.
- **Polymorphie**
  Über Referenzvariablen vom Typ einer Basisklasse lassen sich auch Objekte von abgeleiteten Klassen verwalten, wobei selbstverständlich nur solche Methoden aufgerufen werden dürfen, die schon in der Basisklasse definiert sind. Ist eine solche Methode in abgeleiteten Klassen unterschiedlich implementiert, führt jedes per Basisklassenreferenz angesprochene Objekt sein angepasstes Verhalten aus. Derselbe Methodenaufruf hat also unterschiedliche (polymorphe) Verhaltensweisen zur Folge. Welche Methode ausgeführt wird, entscheidet sich erst zur Laufzeit (späte Bindung). Dank Polymorphie ist eine lose Kopplung von Klassen möglich, und die Wiederverwendbarkeit von vorhandenem Code wird verbessert.

Java bietet sehr gute Voraussetzungen zur Nutzung dieser Konstruktionsprinzipien beim Entwurf von stabilen, wartungsfreundlichen, anpassungsfähigen und auf Wiederverwendung angelegten Software-Systemen, kann aber keinen Entwickler zur Realisation der Prinzipien zwingen.

4.1.1.1 Datenkapselung und Modularisierung

In der objektorientierten Programmierung (OOP) wird die vorher übliche Trennung von Daten und Operationen überwunden. Ein objektorientiertes Programm besteht aus **Klassen**, die durch **Felder (also Daten) und Methoden (also Operationen)** definiert sind. Eine Klasse wird in der Regel ihre Felder gegenüber anderen Klassen verbergen (**Datenkapselung**, engl.: **information hiding**) und so vor ungeschickten Zugriffen schützen. Die meisten Methoden einer Klasse sind hingegen von außen ansprechbar und bilden ihre **Schnittstelle** bzw. ihr API. Dies kommt in der folgenden Abbildung zum Ausdruck, die Sie schon aus Abschnitt 1.1.1 kennen:



Es kann aber auch *private Methoden* für den ausschließlich internen Gebrauch geben.

Öffentliche Felder einer Klasse gehören zu ihrer Schnittstelle und sollten finalisiert (siehe Abschnitt 4.2.5), also vor Veränderungen geschützt sein. Wir haben mit den statischen, öffentlichen und finalisierten Feldern **System.out** und **Math.PI** entsprechende Beispiele kennengelernt.

Klassen mit Datenkapselung realisieren besser als frühere Software-Technologien (siehe Abschnitt 4.1.2) das Prinzip der **Modularisierung**, das schon Julius Cäsar (100 v. Chr. - 44 v. Chr.) bei seiner beruflichen Tätigkeit als römischer Kaiser und Feldherr erfolgreich einsetzte (*Divide et impera!*).[1] Die Modularisierung ist ein probates, ja unverzichtbares Mittel der Software-Entwickler zur Bewältigung von umfangreichen Projekten.

Zugunsten einer häufigen und erfolgreichen Wiederverwendung sind Klassen mit hoher Komplexität (vielfältigen Aufgaben) und auch Methoden mit hoher Komplexität zu vermeiden. Als eine Leitlinie für den Entwurf von Klassen findet das von **Robert C. Martin**[2] erstmals formulierte **Prinzip einer einzigen Verantwortung** (engl.: *Single **R**esponsibility **P**rinciple*, SRP) (Martin 2002) bei den Vordenkern der objektorientierten Programmierung breite Zustimmung (siehe z. B. Lahres & Rayman 2009, Abschnitt 3.1). Multifunktionale Klassen tendieren zu stärkeren Abhängigkeiten von anderen Klassen, wobei die Wahrscheinlichkeit einer erfolgreichen Wiederverwendung sinkt. Ein negatives Beispiel wäre eine Klasse aus einem Personalverwaltungsprogramm, die sich sowohl um Gehaltsberechnungen als auch um die Interaktion mit dem Benutzer über eine grafische Bedienoberfläche kümmert.[3]

Aus der Datenkapselung und anderen Prinzipien der Modularisierung (z. B. Klassendesign nach dem Prinzip einer einzigen Verantwortung) ergeben sich gravierende Vorteile für die Software-Entwicklung:

---

1 Deutsche Übersetzung: Teile und herrsche!

2 Der als *Uncle Bob* bekannte Software-Berater und Autor erläutert auf der folgenden Webseite seine Vorstellungen von objektorientiertem Design: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

3 In einem sehr kleinen Programm ist es angemessen, wenn eine einzige Klasse für die „Geschäftslogik“ und die Benutzerinteraktion zuständig ist.

- **Vermeidung von Fehlern**
  Direkte Schreibzugriffe auf die Felder einer Klasse bleiben den klasseneigenen Methoden vorbehalten, die vom Designer der Klasse sorgfältig entworfen wurden. Damit sollten Programmierfehler nur sehr selten auftreten. In unserer Beispielklasse `Bruch` haben wir dafür gesorgt, dass unter keinen Umständen der Nenner eines Bruches auf den Wert 0 gesetzt wird. Anwender unserer Klasse können einen Nenner einzig über die Methode `setzeNenner()` verändern, die aber den Wert 0 nicht akzeptiert. Bei einer anderen Klasse mag es wichtig sein, dass für eine Gruppe von Feldern bei jeder Änderung gewisse Konsistenzbedingungen eingehalten werden.
- **Günstige Voraussetzungen für das Testen und die Fehlerbereinigung**
  Treten in einem Programm trotz Datenkapselung pathologische Variablenausprägungen auf, ist die Ursache relativ leicht aufzuklären, weil nur wenige Methoden verantwortlich sein können. Zur Sicherstellung wichtiger Bedingungen kann es sinnvoll sein, auch Feldzugriffe durch *klasseneigene* Methoden über die zuständigen Zugriffsmethoden vorzunehmen. Bei der Software-Entwicklung im professionellen Umfeld spielt das systematische Testen eines Programms (**Unit Testing**) eine entscheidende Rolle. Ein objektorientiertes Softwaresystem mit Datenkapselung und guter Modularisierung bietet günstige Voraussetzungen für eine möglichst umfassende Testung.
- **Innovationsoffenheit durch gekapselte Details der Klassenimplementation**
  Verborgene Details einer Klassenimplementation kann der Designer ändern, ohne die Kooperation mit anderen Klassen zu gefährden.
- **Produktivität durch wiederholt und bequem verwendbare Klassen**
  Selbständig agierende Klassen, die ein Problem ohne überflüssige Abhängigkeiten von anderen Programmbestandteilen lösen, sind potenziell in vielen Projekten zu gebrauchen (Wiederverwendbarkeit). Wer als Programmierer eine Klasse verwendet, braucht sich um deren inneren Aufbau nicht zu kümmern, sodass neben dem Fehlerrisiko auch der Einarbeitungsaufwand sinkt. Wir werden z. B. in GUI-Programmen einen recht kompletten Rich-Text-Editor über eine Klasse aus der Standardbibliothek integrieren, ohne wissen zu müssen, wie Text und Textauszeichnungen intern verwaltet werden.
- **Erfolgreiche Teamarbeit durch abgeschottete Verantwortungsbereiche**
  In großen Projekten können mehrere Programmierer nach der gemeinsamen Definition von Schnittstellen relativ unabhängig an verschiedenen Klassen arbeiten.

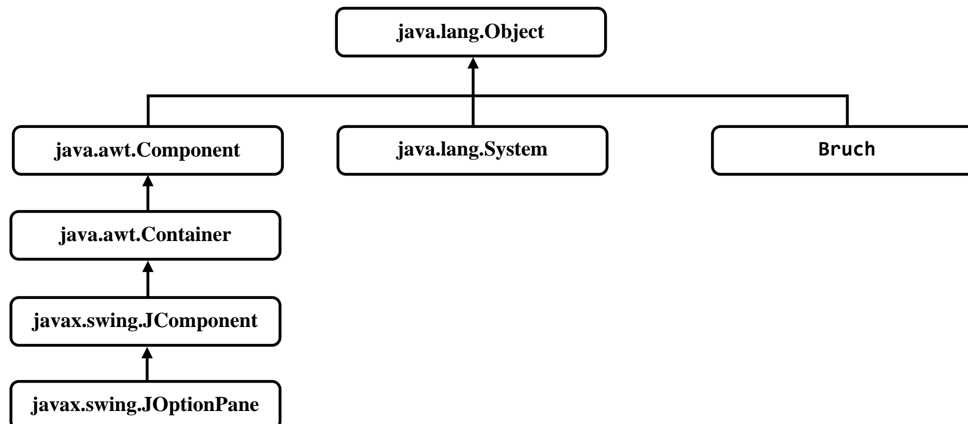
Durch die objektorientierte Programmierung werden auf vielfältige Weise **Kosten reduziert**:

- Vermeidung bzw. schnelles Aufklären von Programmierfehlern
- gute Chancen für die Wiederverwendung von Software
- gute Voraussetzungen für die Kooperation in Teams

4.1.1.2 Vererbung

Zu den Vorzügen der „super-modularen“ Klassenkonzeption gesellt sich in der OOP ein Vererbungsverfahren, das gute Voraussetzungen für die Erweiterung von Software-Systemen bei rationeller **Wiederverwendung** der bisherigen Code-Basis schafft: Bei der Definition einer *abgeleiteten Klasse* werden alle Eigenschaften (Felder) und Handlungskompetenzen (Methoden) der *Basisklasse* übernommen. Es ist also leicht möglich, ein Software-System um neue Klassen mit speziellen Leistungen zu erweitern. Durch systematische Anwendung des Vererbungsprinzips entstehen mächtige Klassenhierarchien, die in zahlreichen Projekten einsetzbar sind. Neben der direkten Nutzung vorhandener Klassen (über statische Methoden oder erzeugte Objekte) bietet die OOP mit der Vererbungstechnik eine weitere Möglichkeit zur Wiederverwendung von Software.

In Java wird das Vererbungsprinzip sogar auf die Spitze getrieben: Alle Klassen stammen von der Urahnklasse **Object** ab, die an der Spitze des hierarchisch organisierten Java-Klassensystems steht. Hier ist ein winziger Ausschnitt aus der Hierarchie zu sehen mit einigen Klassen, die uns im Manuskript schon begegnet sind (**JOptionPane**, **System**, `Bruch`):



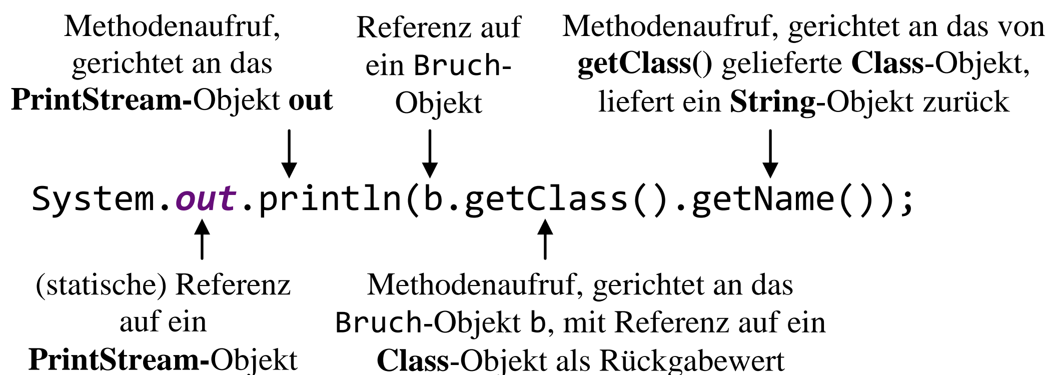
Zu jeder Klasse ist auch ihre Paketzugehörigkeit angegeben (unsere Klasse `Bruch` gehört zum unbenannten Standardpaket).

Wird bei einer Klassendefinition *keine* Basisklasse explizit angegeben (wie bei unserer Beispielklasse `Bruch` aus Abschnitt 1.1), erbt die neue Klasse implizit von der Urahnklasse **Object**. Weil sich im Handlungsrepertoire der Urahnklasse u.a. auch die Methode **getClass()** befindet, kann man Instanzen beliebiger Klassen nach ihrem Datentyp befragen. Im folgenden Programm wird ein `Bruch`-Objekt nach seiner Klassenzugehörigkeit befragt:

Quellcode	Ausgabe

`class Bruchrechnung {`
`   public static void main(String[] args) {`
`      Bruch b = new Bruch();`
`      System.out.println(b.getClass().getName());`
`   }`
`}`
	`Bruch`

Die Methode **getClass()** liefert als Rückgabewert ein Objekt der Klasse **Class** im Paket **java.lang**, das über die Methode **getName()** aufgefordert wird, eine Zeichenfolge mit dem Namen der Klasse zu liefern. Diese Zeichenfolge (ein Objekt der Klasse **String**) bildet schließlich den Parameter des **println()** - Aufrufs und landet auf der Konsole. In unserem Kursstadium ist es angemessen, die komplexe Anweisung unter Beteiligung von fünf Klassen (**System**, **PrintStream**, `Bruch`, **Class**, **String**), drei Methoden (**println()**, **getClass()**, **getName()**), zwei expliziten Referenzvariablen (**out**, `b`) und einer impliziten Referenz (**getClass()** - Rückgabewert) genau zu erläutern:



Durch die technischen Details darf nicht der Blick auf das wesentliche Thema des aktuellen Abschnitts verstellt werden: Eine abgeleitete Klasse erbt die Eigenschaften und Handlungskompetenzen ihrer Basisklasse. Wenn diese Basisklasse ihrerseits abgeleitet ist, kommen indirekt erworbene Erbstücke hinzu. Die als Beispiel betrachtete Klasse `Bruch` stammt direkt von der Klasse **Object** ab, und ihre Objekte beherrschen dank Vererbung u.a. die Methode **getClass()**, obwohl in der `Bruch`-Klassendefinition nichts davon zu sehen ist.

4.1.1.3 Polymorphie

Obwohl in unseren bisherigen Beispielen die Polymorphie noch nicht verwendet wurde, soll doch versucht werden, die Kernidee hinter diesem Begriff schon jetzt zu vermitteln. In diesem Abschnitt sind einige Vorgriffe auf das Kapitel 7 erforderlich. Wer sich jetzt noch nicht stark für den Begriff der Polymorphie interessiert, kann den Abschnitt ohne Risiko für den weiteren Kursverlauf überspringen.

Beim Klassendesign ist generell das **Open-Closed - Prinzip** beachtenswert:[1]

- Eine Klasse soll **offen** sein für Erweiterungen, die zur Lösung von neuen oder geänderten Aufgaben benötigt werden.
- Dabei darf es nicht erforderlich werden, vorhandenen Code zu verändern. Er soll **abgeschlossen** bleiben, möglichst für immer. In ungünstigen Fällen zieht eine Änderung am Quellcode weitere nach sich, sodass eine Kaskade von Anpassungen (eventuell unter Beteiligung von anderen Klassen) resultiert. Dadurch verursacht die Anpassung einer Klasse an neue Aufgaben hohe Kosten und oft ein fehlerhaftes Ergebnis.

Einen exzellenten Beitrag zur Erstellung von änderungsoffenem und doch abgeschlossenem Code leistet schon die Vererbungstechnik der OOP. Zur Modellierung einer neuen, spezialisierten Rolle kann man oft auf eine Basisklasse zurückgreifen und muss nur die zusätzlichen Eigenschaften und/oder Verhaltenskompetenzen ergänzen.

In Java können über eine Referenzvariable Objekte vom deklarierten Typ *und von jedem abgeleiteten Typ* angesprochen werden. In einer abgeleiteten Klasse können nicht nur zusätzliche Methoden erstellt, sondern auch geerbte überschrieben werden, um das Verhalten an spezielle Einsatzbereiche anzupassen. Ergeht ein Methodenaufruf an Objekte aus verschiedenen abgeleiteten Klassen, welche jeweils die Methode überschrieben haben, unter Verwendung von Basisklassenreferenzen, dann zeigen die Objekte ihr artgerechtes Verhalten. Obwohl alle Objekte mit einer Referenz vom selben Basisklassentyp angesprochen werden und denselben Methodenaufruf erhalten, agieren sie unterschiedlich. Welche Methode tatsächlich ausgeführt wird, entscheidet sich erst zur Laufzeit (späte Bindung). Genau in dieser Situation spricht man von *Polymorphie*, und diese Software-Technik leistet einen wichtigen Beitrag zur Realisation des Open-Closed - Prinzips.

Wird z. B. in einer Klasse zur Verwaltung von geometrischen Objekten eine Referenzvariable vom relativ allgemeinen Typ `Figur` deklariert und beim Aufruf der Methode `meldeInhalt()` verwendet, dann führt das angesprochene Objekt, das bei einem konkreten Programmeinsatz z. B. aus der abgeleiteten Klasse `Kreis` oder `Rechteck` stammt, seine spezifischen Berechnungen durch. Die Klasse zur Verwaltung von geometrischen Objekten kann ohne Quellcodeänderungen mit beliebigen, eventuell sehr viel später definierten `Figur`-Ableitungen kooperieren.

Weil in der allgemeinen Klasse `Figur` keine Inhaltsberechnungsmethode realisiert werden kann, wird hier die Methode `meldeInhalt()` zwar deklariert, aber nicht implementiert, sodass eine sogenannte *abstrakte* Methode entsteht. Enthält eine Klasse mindestens eine abstrakte Methode, ist sie

---

[1] Das Open-Closed - Prinzip wird von Robert C. Martin (*Uncle Bob*) in einem Text erläutert, der über folgende Web-Adresse zu beziehen ist: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

ihrerseits abstrakt und kann nicht zum Erzeugen von Objekten genutzt werden. Eine abstrakte Klasse ist aber gleichwohl als Datentyp erlaubt und spielt eine wichtige Rolle bei der Realisation von Polymorphie.[1]

Dank Polymorphie ist eine lose Kopplung von Klassen möglich, und die Wiederverwendbarkeit von vorhandenem Code wird verbessert. Um die Offenheit für neue Aufgaben zu ermöglichen, verwendet man beim Klassendesign für Felder und Methodenparameter mit Referenztyp einen möglichst allgemeinen Datentyp, der die benötigten Verhaltenskompetenzen vorschreibt, aber keine darüber hinausgehende Einschränkung enthält.

Dank Vererbung und Polymorphie kann objektorientierte Software **anpassungs- und erweiterungsfähig** bei weitgehend fixiertem Bestands-Code, also unter Beachtung des Open-Closed - Prinzips, gestaltet werden.

4.1.1.4 Realitätsnahe Modellierung

Klassen sind nicht nur ideale Bausteine für die rationelle Konstruktion von Software-Systemen, sondern sie erlauben auch eine gute Modellierung des Anwendungsbereichs. In der zentralen Projektphase der objektorientierten Analyse und Modellierung sprechen Software-Entwickler und Auftraggeber dieselbe Sprache, sodass Kommunikationsprobleme weitgehend vermieden werden.

Neben den Klassen zur Modellierung von Akteuren oder Ereignissen des realen Anwendungsbereichs sind bei einer typischen Anwendung aber auch zahlreiche Klassen beteiligt, die Akteure oder Ereignisse aus der Welt des Computers repräsentieren (z. B. Bildschirmfenster, Ereignisse).

4.1.2 Strukturierte Programmierung und OOP

In vielen älteren Programmiersprachen (z. B. C, Fortran, Pascal) sind zur Strukturierung von Programmen zwei Techniken verfügbar, die in weiterentwickelter Form auch bei der OOP genutzt werden:

- **Unterprogramme**
  Man zerlegt ein Gesamtproblem in mehrere Teilprobleme, die jeweils in einem eigenen *Unterprogramm* gelöst werden. Wird die von einem Unterprogramm erbrachte Leistung wiederholt (an verschiedenen Stellen eines Programms) benötigt, muss jeweils nur ein Aufruf mit dem Namen des Unterprogramms und passenden Parametern eingefügt werden. Durch diese Strukturierung ergeben sich kompakte und übersichtliche Programme, die leicht erstellt, analysiert, korrigiert und erweitert werden können. Praktisch alle älteren Programmiersprachen unterstützen solche Unterprogramme (Subroutinen, Funktionen, Prozeduren), und meist stehen umfangreiche Bibliotheken mit fertigen Unterprogrammen für diverse Standardaufgaben zur Verfügung. Beim Einsatz einer Unterprogrammsammlung klassischer Art muss der Programmierer passende Daten bereitstellen, auf die dann vorgefertigte Routinen losgelassen werden. Der Programmierer hat also seine Daten *und* das Arsenal der verfügbaren Unterprogramme (aus fremden Quellen oder selbst erstellt) zu verwalten und zu koordinieren.

---

[1] Neben den abstrakten Klassen, die mindestens *eine* abstrakte Methode (Definitionskopf ohne Implementation) enthalten, spielen bei der Polymorphie auch die sogenannten *Schnittstellen* eine wichtige Rolle als Datentypen für ein veränderungsoffenes Design. Eine Schnittstelle kann näherungsweise als Klasse mit *ausschließlich* abstrakten Methoden charakterisiert werden. Abstrakte Klassen und Schnittstellen (engl.: *Interfaces*) werden später ausführlich behandelt.

- **Problemadäquate Datentypen**
  Zusammengehörige Daten unter *einem* Variablennamen ansprechen zu können, vereinfacht das Programmieren erheblich. Mit dem Datentyp **struct** der Programmiersprache C oder dem analogen Datentyp **record** der Programmiersprache Pascal lassen sich problemadäquate Datentypen mit mehreren Bestandteilen konstruieren, die jeweils einen beliebigen, bereits bekannten Typ haben dürfen. So eignet sich etwa für ein Programm zur Adressenverwaltung ein neu definierter Datentyp mit Variablen für Name, Vorname, Telefonnummer etc. Alle Adressinformationen zu einer Person lassen sich dann in *einer* Variablen vom selbst definierten Typ speichern. Dies vereinfacht z. B. das Lesen, Kopieren oder Schreiben solcher Daten.

Die problemadäquaten Datentypen der älteren Programmiersprachen werden in der OOP durch Klassen ersetzt, wobei diese Datentypen nicht nur durch eine Anzahl von *Eigenschaften* (Feldern) beliebigen Typs charakterisiert sind, sondern auch *Handlungskompetenzen* (Methoden) besitzen, welche die Aufgaben der Funktionen bzw. Prozeduren der älteren Programmiersprachen übernehmen.

Im Vergleich zur strukturierten Programmierung bietet die OOP u.a. folgende Vorteile:

- Optimierte Modularisierung mit Zugriffsschutz
  Die Daten sind sicher in Objekten gekapselt, während sie bei traditionellen Programmiersprachen entweder als globale Variablen allen Missgriffen ausgeliefert sind oder zwischen Unterprogrammen „wandern“ (Goll et al. 2000, S. 21), was bei Fehlern zu einer aufwändigen Suche entlang der Verarbeitungskette führen kann.
- Gute Voraussetzungen für die Teamarbeit
  Durch die optimierte Modularisierung wird die (vor allem in großen Projekten wichtige) Kooperation in Entwicklerteams erleichtert.
- Rationelle (Weiter-)Entwicklung von Software nach dem Open-Closed - Prinzip durch Vererbung und Polymorphie
- Bessere Abbildung des Anwendungsbereichs
  Das erleichtert die Kommunikation zwischen dem Auftraggeber bzw. Anwender einerseits und dem Software-Architekten bzw. -Entwickler andererseits.
- Mehr Komfort für Bibliotheksbenutzer
  Jede rationelle Softwareproduktion greift in hohem Maß auf Bibliotheken mit bereits vorhandenen Lösungen zurück. Dabei sind die Klassenbibliotheken der OOP einfacher zu verwenden als klassische Funktionsbibliotheken.
- Erleichterte Wiederverwendung
  Die komfortable Nutzung von Lösungsbibliotheken sowie die rationelle Weiterentwicklung von Software durch Vererbung und Polymorphie führen zu einer erleichterten Wiederverwendung von vorhandener Software.

Dass objektorientierte Programmiersprachen im Vergleich zu ihren strukturierten Vorgängern etwas mehr Speicherplatz und CPU-Leistung verbrauchen, spielt schon lange keine Rolle mehr.

4.1.3 Auf-`Bruch` zu echter Klasse

In den Beispielprogrammen von Kapitel 3 wurde mit der Klassendefinition lediglich eine in Java unausweichliche formale Anforderung an Programme erfüllt. Die im Abschnitt 1.1 vorgestellte Klasse `Bruch` realisiert hingegen wichtige Prinzipien der objektorientierten Programmierung. Diese Klasse wird nun wieder aufgegriffen und in verschiedenen Varianten bzw. Ausbaustufen als Beispiel verwendet. Auf der Klasse `Bruch` basierende Programme sollen Schüler beim Erlernen der Bruchrechnung unterstützen. Eine objektorientierte Analyse der Problemstellung hat ergeben, dass in der elementaren Ausbaustufe des Programms lediglich eine Klasse zur Repräsentation von Brü-

chen benötigt wird. Später sind weitere Klassen zu ergänzen (z. B. Aufgabe, Übungsaufgabe, Testaufgabe, Schüler, Lernepisode, Testepisode, Fehler).

Wir nehmen nun bei der `Bruch`-Klassendefinition im Vergleich zur Variante im Abschnitt 1.1 einige Verbesserungen vor:

- Als zusätzliches Feld (zusätzliche Eigenschaft) erhält jeder Bruch ein `etikett` vom Datentyp der Klasse **String**. Damit wird eine beschreibende Zeichenfolge verwaltet, die z. B. beim Aufruf der Methode `zeige()` zusätzlich zu anderen Eigenschaften auf dem Bildschirm erscheint. Objekte der erweiterten Klasse `Bruch` besitzen also auch ein Feld mit *Referenztyp* (neben den Feldern `zaehler` und `nenner` vom primitiven Typ **int**).
- Weil die Klasse `Bruch` ihre Eigenschaften systematisch kapselt, also fremden Klassen keine direkten Zugriffe erlaubt, muss sie auch für das `etikett` zum Lesen bzw. Setzen des Wertes jeweils eine Methode bereitstellen.
- In der Methode `kuerze()` wird die performante Modulo-Variante von Euklids Algorithmus zur Bestimmung des größten gemeinsamen Teilers von zwei ganzen Zahlen verwendet (vgl. Übungsaufgabe auf Seite 195).

Im folgenden Quellcode der erweiterten Klasse `Bruch` sind die unveränderten Methoden gekürzt wiedergegeben:

```
public class Bruch {
   private int zaehler;          // wird automatisch mit 0 initialisiert
   private int nenner = 1;       // wird manuell mit 1 initialisiert
   private String etikett = "";  // die Referenztyp-Init. auf null wird ersetzt

   public void setzeZaehler(int z) {zaehler = z;}

   public boolean setzeNenner(int n) { ... }

   public void setzeEtikett(String eti) {
      if (eti.length() <= 40)
         etikett = eti;
      else
         etikett = eti.substring(0, 40);
   }

   public int gibZaehler() {return zaehler;}

   public int gibNenner() {return nenner;}

   public String gibEtikett() {return etikett;}

   public void kuerze() {
      // größten gemeinsamen Teiler mit dem Euklidischen Algorithmus bestimmen
      // (performante Variante mit Modulo-Operator)
      if (zaehler != 0) {
         int rest;
         int ggt = Math.abs(zaehler);
         int divisor = Math.abs(nenner);
         do {
            rest = ggt % divisor;
            ggt = divisor;
            divisor = rest;
         } while (rest > 0);
         zaehler /= ggt;
         nenner  /= ggt;
      } else
         nenner = 1;
   }
```

```
      public void addiere(Bruch b) { ... }

      public void frage() { ... }

      public void zeige() {
         String luecke = "";
         int el = etikett.length();
         for (int i=1; i<=el; i++)
            luecke = luecke + " ";
         System.out.println(" " + luecke  + "   " + zaehler + "\n" +
                            " " + etikett + " -----\n" +
                            " " + luecke  + "   " + nenner + "\n");
      }
   }
```

Für die bei diversen Demonstrationen in den folgenden Abschnitten verwendeten Startklassen (mit jeweils spezieller Implementierung) werden wir generell den Namen `Bruchrechnung` verwenden, z. B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b = new Bruch();
      b.setzeZaehler(4);
      b.setzeNenner(16);
      b.kuerze();
      b.setzeEtikett("Der gekürzte Bruch:");
      b.zeige();
   }
}
```

</td><td>

```
                       1
 Der gekürzte Bruch: -----
                       4
```

</td></tr>
</table>

Die Instanzvariablen `zaehler` und `nenner` der Klasse `Bruch` haben bei der Renovierung den Datentyp **int** beibehalten und sind daher nach wie vor mit einem potentiellen Überlaufproblem (vgl. Abschnitt 3.6.1) belastet, das im folgenden Programm demonstriert wird:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b1 = new Bruch(), b2 = new Bruch();
      b1.setzeZaehler(2147483647); b1.setzeNenner(1);
      b2.setzeZaehler(2); b2.setzeNenner(1);
      b1.addiere(b2);
      b1.zeige();
   }
}
```

</td><td>

```
    -2147483647

  -----
    1
```

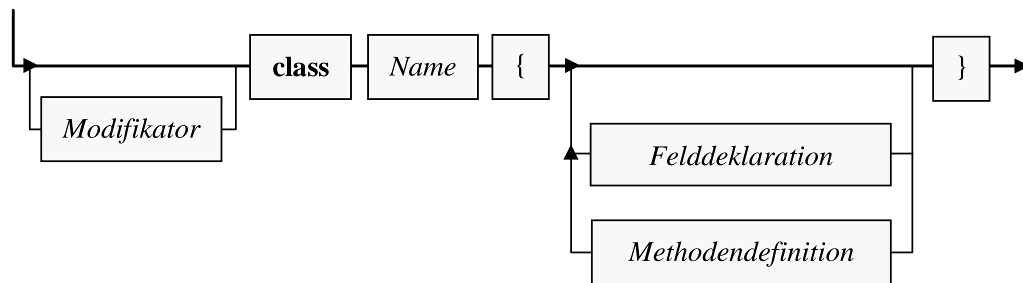
</td></tr>
</table>

Der Einfachheit halber verzichten wir auf die im Abschnitt 3.6.1 beschriebenen Techniken zur Vermeidung des Problems.

Im Unterschied zur Präsentation im Abschnitt 1.1 wird die `Bruch`-Klassendefinition anschließend gründlich erläutert. Dabei machen die im Abschnitt 4.2 zu behandelnden Instanzvariablen (Felder) relativ wenig Mühe, weil wir viele Details schon von den lokalen Variablen her kennen (siehe Abschnitt 3.3). Bei den Methoden gibt es mehr Neues zu lernen, sodass wir uns im Abschnitt 4.3 auf elementare Themen beschränken und später noch wichtige Ergänzungen vornehmen.

Wir arbeiten weiterhin mit dem aus Abschnitt 3.1.3.1 bekannten Syntaxdiagramm zur Klassendefinition, das aus didaktischen Gründen einige Vereinfachungen enthält:

Klassendefinition



Zwei Bemerkungen zum Kopf einer Klassendefinition:

- Im Beispiel ist die Klasse `Bruch` als **public** definiert, damit sie uneingeschränkt von anderen Klassen aus beliebigen Paketen (in berechtigten Modulen) genutzt werden kann.[1] Weil bei der Startklasse `Bruchrechnung` eine solche Nutzung durch andere Klassen nicht in Frage kommt, wird hier auf den (zum Starten durch die JVM *nicht* erforderlichen) Zugriffsmodifikator **public** verzichtet. Im Zusammenhang mit den Modulen und Paketen werden die Zugriffsmodifikatoren für Klassen systematisch behandelt.
- Klassennamen beginnen einer allgemein akzeptierten Java-Konvention folgend mit einem Großbuchstaben. Besteht ein Name aus mehreren Wörtern (z. B. **BigDecimal**), schreibt man der besseren Lesbarkeit wegen die Anfangsbuchstaben aller Wörter groß (*Pascal Casing*).[2] Tritt ein Akronym (z. B. HTTP) als Namensbestandteil auf, sollte nur der erste Buchstabe groß geschrieben werden (z. B. **HttpRequest**), weil anderenfalls die Namensbestandteile optisch schlecht separiert werden können. An die letztgenannte Konvention halten sich nicht alle Entwickler.

Hinsichtlich der **Dateiverwaltung** ist zu beachten:

- Die `Bruch`-Klassendefinition *muss* in einer Datei namens **Bruch.java** gespeichert werden, weil die Klasse als **public** definiert ist.
- Auch für den Quellcode der Startklasse `Bruchrechnung`, die nicht als **public** definiert ist, sollte analog eine Datei namens **Bruchrechnung.java** verwendet werden. Der Java-Compiler erlaubt zwar Quellcodedateien mit *mehreren* Top-Level - Klassen, von denen maximal eine die Sichtbarkeit **public** besitzen darf. Allerdings kann dabei zu Programmfehlern kommen, wenn eine Klassendefinition in mehreren Quellcodedateien steckt (siehe Bloch 2018, S. 115f). Der Übersichtlichkeit und Sicherheit halber sollte jede Java-Quellcodedatei nur *eine* Top-Level - Klasse enthalten.
- Dateien mit Java-Quellcode benötigen auf jeden Fall die Namenserweiterung **.java**.

4.2 Instanzvariablen (Felder)

Die Instanzvariablen (Felder) einer Klasse besitzen viele Gemeinsamkeiten mit den lokalen Variablen, die wir im Kapitel 3 über elementare Sprachelemente ausführlich behandelt haben, doch gibt es

[1] Dazu muss die Klasse später allerdings noch in ein explizites Paket aufgenommen werden. Noch gehört die Klasse `Bruch` zum Standardpaket, und dessen Klassen sind in anderen Paketen generell (auch bei Zugriffsstufe **public**) **nicht** verfügbar. Das mit Java 9 (im September 2017) eingeführte Modulsystem macht es zudem möglich, den Zugriff auf die Pakete in den Klassen eines Moduls auf *berechtigte* andere Module einzuschränken. Aktuell (im November 2019) setzen die meisten Java-Programme allerdings nur eine JVM-Version ≤ 8 voraus, sodass der Modifikator **public** den Zugriff für *alle* Klassen erlaubt.

[2] Bei einer Startklasse ist ein komplizierter Name zu vermeiden, wenn dieser vom Benutzer beim Programmstart eingetippt werden muss (mit korrekt eingehaltener Groß-/Kleinschreibung!).

auch wichtige Unterschiede, die im Mittelpunkt des aktuellen Abschnitts stehen. Unsere Klasse `Bruch` besitzt nach der Erweiterung um ein beschreibendes Etikett die folgenden Instanzvariablen:

- `zaehler` (Datentyp **int**)
- `nenner` (Datentyp **int**)
- `etikett` (Datentyp **String**)

Zu den beiden Feldern `zaehler` und `nenner` mit dem primitiven Datentyp **int** ist das Feld `etikett` mit dem Referenzdatentyp **String** dazugekommen. Jedes nach dem `Bruch`-Bauplan geschaffene Objekt erhält seine eigene Ausstattung mit diesen Variablen.

4.2.1 Sichtbarkeitsbereich, Existenz und Ablage im Hauptspeicher

Von den lokalen Variablen einer Methode unterscheiden sich die Instanzvariablen (Felder) einer Klasse vor allem bei der *Zuordnung* (vgl. Abschnitt 3.3.4):

- lokale Variablen gehören zu einer *Methode*
- Instanzvariablen gehören zu einem *Objekt*

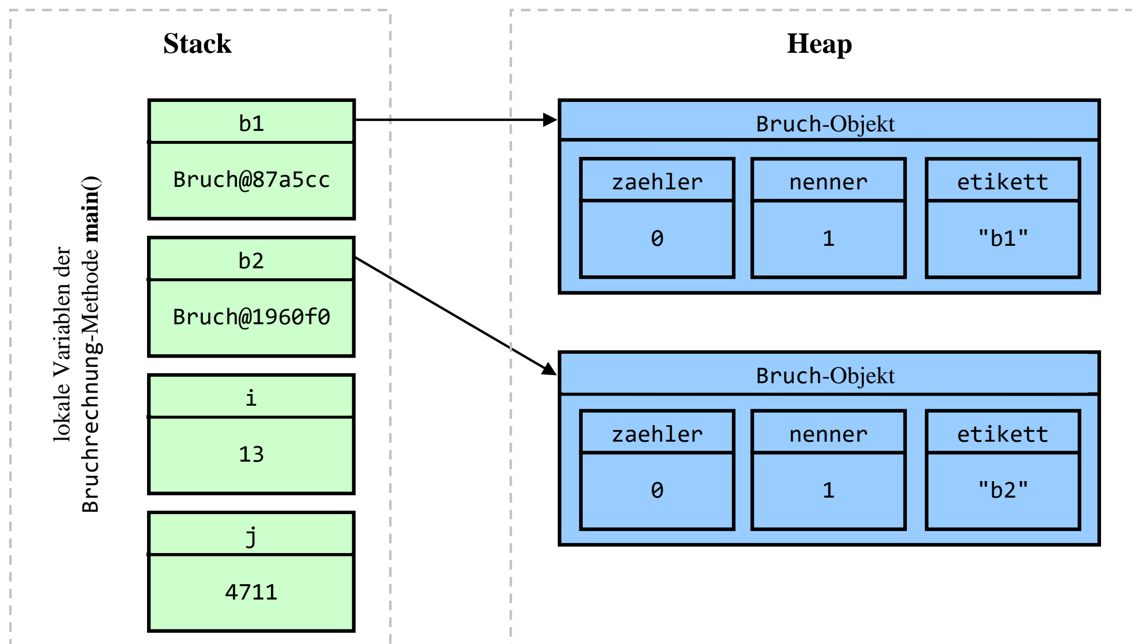
Daraus ergeben sich gravierende Unterschiede in Bezug auf den Sichtbarkeitsbereich (synonym: Gültigkeitsbereich), die Lebensdauer und die Ablage im Hauptspeicher:

	lokale Variable	**Instanzvariable**
Sichtbarkeit, Gültigkeit	Eine lokale Variable ist nur in ihrer eigenen Methode sichtbar (gültig). Nach der Deklarationsanweisung bleibt sie ansprechbar bis zur schließenden Klammer des Blocks, in dem sie deklariert worden ist. Ein eingeschachtelter Block gehört zum Sichtbarkeitsbereich des umgebenden Blocks.	Die Instanzvariablen eines existenten Objekts sind in einer Methode sichtbar, wenn ... • der Zugriff erlaubt (siehe 4.2.2) • und eine Referenz zum Objekt vorhanden ist. Instanzvariablen werden in klasseneigenen Instanzmethoden durch gleichnamige lokale Variablen überdeckt, können in dieser Situation jedoch über das vorgeschaltete Schlüsselwort **this** weiter angesprochen werden (siehe Abschnitt 4.2.4).
Lebensdauer	Sie existiert nur während der Ausführung der zugehörigen Methode.	Für jedes neue Objekt wird ein Satz mit allen Instanzvariablen seiner Klasse erzeugt. Die Instanzvariablen existieren bis zum Ableben des Objekts. Ein Objekt wird zur Entsorgung freigegeben, sobald keine Referenz auf das Objekt mehr im Programm vorhanden ist.
Ablage im Speicher	Sie wird auf dem sogenannten **Stack** (deutsch: *Stapel*) abgelegt. Dieses Segment des programmeigenen Speichers dient zur Durchführung von Methodenaufrufen.	Die Objekte landen mit ihren Instanzvariablen in einem Bereich des programmeigenen Speichers, der als **Heap** (deutsch: *Haufen*) bezeichnet wird.

Während die folgende **main()** - Methode

```
class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b1 = new Bruch(), b2 = new Bruch();
      int i = 13, j = 4711;
      b1.setzeEtikett("b1");
      b2.setzeEtikett("b2");
        . . .
   }
}
```

ausgeführt wird, befinden sich auf dem Stack die lokalen Variablen `b1`, `b2`, `i` und `j`. Die beiden `Bruch`-Referenzvariablen (`b1`, `b2`) zeigen jeweils auf ein `Bruch`-Objekt auf dem Heap, das einen kompletten Satz der `Bruch`-Instanzvariablen besitzt:[1]



Hier wird aus didaktischen Gründen ein wenig gemogelt: Die beiden Etiketten sind selbst Objekte und liegen „neben" den `Bruch`-Objekten auf dem Heap. In jedem `Bruch`-Objekt befindet sich eine Referenz-Instanzvariable namens `etikett`, die auf das zugehörige **String**-Objekt zeigt.

4.2.2 Deklaration mit Modifikatoren für den Zugriffsschutz und für andere Zwecke

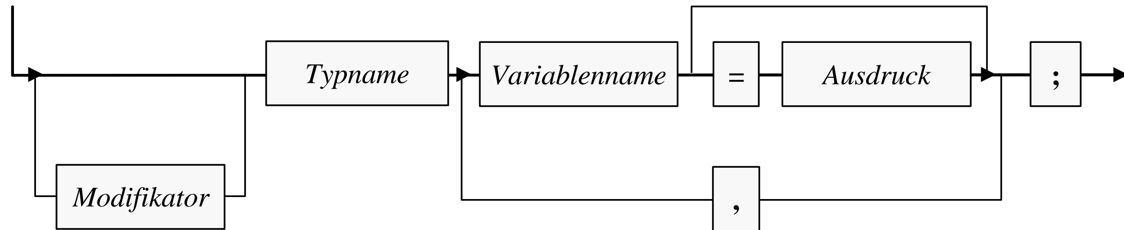
Während lokale Variablen im Rumpf einer Methode deklariert werden, erscheinen die Deklarationen der Instanzvariablen in der Klassendefinition *außerhalb* jeder Methodendefinition. Man sollte die Instanzvariablen der Übersichtlichkeit halber am Anfang der Klassendefinition deklarieren, wenngleich der Compiler auch ein späteres Erscheinen akzeptiert.[2]

Für die Deklaration einer lokalen Variablen haben wir **final** als einzigen Modifikator kennengelernt und in einem speziellen Syntaxdiagramm beschrieben (vgl. Abschnitt 3.3.10). Dieser Modifikator ist auch bei Instanzvariablen erlaubt (siehe Abschnitt 4.2.5). Außerdem kommen hier weitere Modifikatoren in Frage, die z. B. zur Spezifikation der **Schutzstufe** dienen. Insgesamt ist es sinnvoll, in

[1] Die Abbildung zeigt zu den beiden `Bruch`-Referenzvariablen (`b1`, `b2`) jeweils den Rückgabewert der (von **Object** geerbten) Methode **toString()** als Inhalt. Hinter dem @-Zeichen steht genau genommen der **hashCode()** - Wert (vgl. Abschnitt 10.5) der Klasse **Object**, der allerdings wesentlich auf der Speicheradresse basiert.

[2] Anders als bei lokalen Variablen von Methoden hat der Deklarationsort bei Instanzvariablen keinen Einfluss auf den Sichtbarkeitsbereich.

das Syntaxdiagramm zur Deklaration von Instanzvariablen den allgemeinen Begriff des Modifikators aufzunehmen:[1]



Die bei lokalen Variablen durch Typinferenz ermöglichte Ersetzung des Typnamens durch das Schlüsselwort **var** (siehe Abschnitt 3.3.8) ist bei Instanzvariablen verboten.

Im `Bruch`-Beispiel wird im Sinne einer perfekten Datenkapselung für *alle* Instanzvariablen mit dem Modifikator **private** angeordnet, dass nur klasseneigenen Methoden der direkte Zugriff erlaubt sein soll:

```
private int zaehler;
private int nenner = 1;
private String etikett = "";
```

Um fremden Klassen trotzdem einen (allerdings kontrollierten) Zugang zu den `Bruch`-Instanzvariablen zu ermöglichen, enthält die Klassendefinition jeweils ein Methodenpaar für den lesenden bzw. schreibenden Zugriff (z. B. `gibNenner()`, `setzeNenner()`).

Gibt man bei der Deklaration einer Instanzvariablen *keine* Schutzstufe an, dann haben alle anderen Klassen *im selben Paket* (siehe Kapitel 6) das direkte Zugriffsrecht, was in der Regel unerwünscht ist.

In der Klasse `Bruch` scheint die Datenkapselung auf den ersten Blick nur beim Nenner relevant zu sein, doch auch bei den restlichen Instanzvariablen bringt sie (potentiell) Vorteile:

- Zugunsten einer übersichtlichen Bildschirmausgabe soll das Etikett auf 40 Zeichen beschränkt bleiben. Mit Hilfe der Zugriffsmethode `setzteEtikett()` kann dies auf einfache Weise gewährleistet werden.
- Abgeleitete (erbende) Klassen (siehe Kapitel 7) können in die Zugriffsmethoden für `zaehler` und `nenner` neben der Null-Überwachung für den Nenner noch weitere Intelligenz einbauen und z. B. mit speziellen Aktionen reagieren, wenn der Wert auf eine Primzahl gesetzt wird.

Trotz ihrer überzeugenden Vorteile soll die Datenkapselung nicht zum Dogma erhoben werden.[2] Sie verliert an Bedeutung, wenn ...

---

[1] Es ist sinnlos und verboten, einen Modifikator *mehrfach* auf eine Instanzvariable anzuwenden. Im Syntaxdiagramm zur Instanzvariablendeklaration wird der Einfachheit halber darauf verzichtet, die Mehrfachvergabe durch eine aufwändige Darstellungstechnik zu verhindern.

[2] Bei öffentlichen Klassen (Zugriffsmodifikator **public**) ist die durch Datenkapselung realisierte Sicherheit und Flexibilität von weitaus größerer Bedeutung als bei Klassen, die nur im eigenen Paket einer Anwendung sichtbar sind (Bloch 2018, S.78). Wenn z. B. die Datenablage in einer Paket-privaten Klasse geändert werden muss, sind nur wenige andere Klassen betroffen, die vom selben Entwickler oder von derselben Firma kontrolliert werden.
Durch den Verzicht auf Zugriffsmethoden wird der Quellcode etwas kürzer und prägnanter. Leider sind die Eigenschaften der Programmiersprache C#, die als „syntactic sugar“ elegante Zugriffe auf gekapselte Felder erlauben, in Java nicht verfügbar.

- bei einem Feld Lese- und Schreibzugriffe uneingeschränkt erlaubt sein sollen, wenn es also insbesondere nicht erforderlich ist, die möglichen Werte zu restringieren.
- es nicht von Interesse ist, auf bestimmte Wertzuweisungen zu reagieren, um z. B. bestimmte Objekteigenschaften (man sagt auch: *Invarianten*) sicherzustellen.

Um allen Klassen in berechtigten Modulen den Direktzugriff auf eine Instanzvariable zu erlauben, wird in der Deklaration der Modifikator **public** angegeben, z. B.:[1]

```
public int zaehler;
```

Bei *finalisierten* Instanzvariablen, die nach einer initialen Wertzuweisung nicht mehr geändert werden können (siehe Abschnitt 4.2.5), ist keine Datenkapselung als Schutz gegen irreguläre Wertzuweisungen erforderlich, sofern sie ...

- entweder einen primitiven Datentyp besitzen
- oder als Referenzvariablen auf ein nicht-veränderbares Objekt zeigen (z. B. vom Typ **String**).

Wenn eine finalisierte, aber nicht geschützte Instanzvariable auf ein *veränderliches* Member-Objekt zeigt, kann das referenzierte Member-Objekt beschädigt werden.

Insgesamt ist in der Regel für Instanzvariablen die Datenkapselung, also die **private**-Deklaration zu empfehlen. Das gilt insbesondere für öffentlich zugängliche Klassen (definiert mit dem Modifikator **public**).

Im Zusammenhang mit den Modulen und Paketen (siehe Kapitel 6) werden wir uns noch ausführlich mit dem Thema *Zugriffsschutz* beschäftigen. Die wichtigsten Regeln für die Sichtbarkeit von Instanzvariablen können Sie aber jetzt schon verstehen:

- Bevor sich die Frage nach der Sichtbarkeit von Instanzvariablen stellt, muss die Klasse selbst sichtbar sein:
  - Eine Klasse ist grundsätzlich in anderen Klassen des eigenen Pakets sichtbar.
  - Für die Sichtbarkeit in *allen* Klassen (aus berechtigten Modulen) muss durch den Klassen-Zugriffsmodifikator **public** gesorgt werden.
- Per Voreinstellung ist der Zugriff auf Instanzvariablen allen Klassen des eigenen Pakets erlaubt.
- Mit einem Member-Zugriffsmodifikator lassen sich alternative Schutzstufen wählen, z. B.:
  - **private**
    Alle fremden Klassen werden ausgeschlossen (auch die im selben Paket).
  - **public**
    Alle Klassen in berechtigten Modulen dürfen zugreifen.

In Bezug auf die **Benennung** gibt es keine Unterschiede zwischen den Instanzvariablen und den lokalen Variablen (vgl. Abschnitt 3.3). Insbesondere sollten die folgenden Namenskonventionen eingehalten werden:

- Variablennamen beginnen mit einem Kleinbuchstaben.
- Besteht ein Name aus mehreren Wörtern (z. B. `currentSpeed`), schreibt man ab dem *zweiten* Wort die Anfangsbuchstaben groß (*Camel Casing*)

[1] Module wurden erst mit Java 9 eingeführt (im September 2017), und aktuell (im November 2019) setzen die meisten Java-Programme nur eine JVM-Version ≤ 8 voraus, sodass der Modifikator **public** den Zugriff für *alle* Klassen erlaubt.

4.2.3 Automatische Initialisierung auf den Voreinstellungswert

Während bei lokalen Variablen der Programmierer für die Initialisierung verantwortlich ist, erhalten die Instanzvariablen eines neuen Objekts automatisch die folgenden Voreinstellungswerte, wenn der Programmierer nicht eingreift:

Datentyp	**Voreinstellungswert**
byte, **short**, **int**, **long**	0
float, **double**	0.0
char	0 (Unicode-Zeichennummer)
boolean	**false**
Referenztyp	**null**

Im `Bruch`-Beispiel wird nur die automatische `zaehler`-Initialisierung unverändert übernommen:

- Beim `nenner` eines Bruches wäre die Initialisierung auf 0 bedenklich, weshalb eine explizite Initialisierung auf den Wert 1 vorgenommen wird.
- Wie noch näher zu erläutern sein wird, ist **String** in Java *kein* primitiver Datentyp, sondern eine Klasse. Variablen von diesem Typ können einen Verweis auf ein Objekt aus dieser Klasse aufnehmen. Solange kein zugeordnetes Objekt existiert, hat eine **String**-Instanzvariable den Wert **null**, zeigt also auf nichts. Weil der `etikett`-Wert **null** z. B. beim Aufruf der `Bruch`-Methode `zeige()` einen Laufzeitfehler (**NullPointerException**) zur Folge hätte, wird ein **String**-Objekt mit einer leeren Zeichenfolge erstellt und zur `etikett`-Initialisierung verwendet. Das Erzeugen des **String**-Objekts erfolgt *implizit* (ohne **new**-Operator, siehe unten), indem der **String**-Variablen `etikett` ein Zeichenfolgen-Literal zugewiesen wird.

4.2.4 Zugriff in klasseneigenen und fremden Methoden

In den Instanzmethoden einer Klasse können die Instanzvariablen des *aktuellen* (die Methode ausführenden) Objekts direkt über ihren Namen angesprochen werden, was z. B. in der `Bruch`-Methode `zeige()` zu beobachten ist:

```
System.out.println(" " + luecke  + "   " + zaehler + "\n" +
                   " " + etikett + " -----\n" +
                   " " + luecke  + "   " + nenner + "\n");
```

Im Beispiel zeigt sich syntaktisch kein Unterschied zwischen dem Zugriff auf die Instanzvariablen (`zaehler`, `nenner`, `etikett`) und dem Zugriff auf die lokale Variable `luecke`.

Gelegentlich kann es sinnvoll oder auch erforderlich sein, einem Instanzvariablennamen über das Schlüsselwort **this** (vgl. Abschnitt 4.4.8.2) eine explizite Referenz auf das aktuell handelnde Objekt voranzustellen, wobei das Schlüsselwort und der Feldname durch den **Punktoperator** zu trennen sind:

- Das kann optional der Klarheit halber geschehen, z. B.:

```
System.out.println(" " + luecke  + "   " + this.zaehler + "\n" +
                   " " + this.etikett + " -----\n" +
                   " " + luecke  + "   " + this.nenner + "\n");
```

- Instanzvariablen werden durch gleichnamige lokale Variablen oder Methodenparameter (siehe Abschnitt 4.3) überlagert, können jedoch in dieser (besser zu vermeidenden) Situation über das vorgeschaltete Schlüsselwort **this** weiter angesprochen werden.

Beim Zugriff auf eine Instanzvariable eines *anderen* Objekts derselben Klasse muss dem Variablennamen eine Referenz auf das Objekt vorangestellt werden, wobei die Bezeichner (für das Objekt bzw. für die Instanzvariable) durch den Punktoperator zu trennen sind. In der folgenden Anweisung aus der `Bruch`-Methode `addiere()` greift das handelnde Objekt lesend auf die Instanzvariablen eines anderen `Bruch`-Objekts zu, das über die Referenzvariable `b` angesprochen wird:

```
zaehler = zaehler*b.nenner + b.zaehler*nenner;
```

In einer *statischen* Methode der eigenen Klasse muss zum Zugriff auf eine Instanzvariable eines konkreten Objekts natürlich eine Referenz auf dieses Objekt vorhanden sein und dem Instanzvariablennamen vorangestellt werden (getrennt durch den Punktoperator).

Direkte Zugriffe auf die Instanzvariablen eines Objekts in Methoden *fremder* Klassen sind zwar nicht grundsätzlich verboten, verstoßen aber gegen das Prinzip der Datenkapselung, das in der OOP von zentraler Bedeutung ist. Würden die `Bruch`-Instanzvariablen ohne den Modifikator **private** deklariert, dann könnten der Zähler und der Nenner eines Bruches in der **main**() - Methode der Klasse `Bruchrechnung`, die sich im selben (nämlich dem unbenannten) Paket befindet, direkt angesprochen werden, z. B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b = new Bruch();
      b.zaehler = 1;
      b.nenner = 0;
      b.kuerze();
   }
}
```

</td><td>

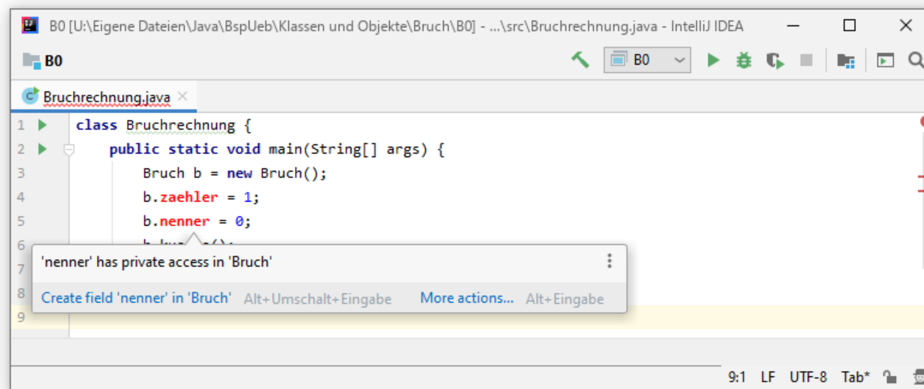
```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
  at Bruch.kuerze(Bruch.java:39)
  at Bruchrechnung.main(Bruchrechnung.java:6)
```

</td></tr>
</table>

In der von uns tatsächlich realisierten `Bruch`-Definition werden solche Zugriffe jedoch verhindert. Der OpenJDK 8 - Compiler meldet:

```
Bruchrechnung.java:4: error: zaehler has private access in Bruch
                b.zaehler = 1;
                 ^
Bruchrechnung.java:5: error: nenner has private access in Bruch
                b.nenner = 0;
                 ^
2 errors
```

Unsere Entwicklungsumgebung IntelliJ signalisiert die Problemstellen sehr deutlich im Quellcode-Editor:



4.2.5 Finalisierte Instanzvariablen

Neben der Schutzstufenwahl gibt es weitere Anlässe für den Einsatz von Modifikatoren in einer Felddeklaration. Mit dem Modifikator **final** können nicht nur lokale Variablen (siehe Abschnitt 3.3.10) sondern auch Instanzvariablen als *finalisiert* deklariert werden, sodass der Compiler nur eine einmalige Wertzuweisung erlaubt und eine Änderung dieses Wertes im weiteren Programmverlauf verhindert.

So wird verhindert, dass ein als fixiert geplanter Wert versehentlich (z. B. aufgrund eines Tippfehlers) doch geändert wird. Dank **final**-Deklaration kann der Compiler Regelverstöße verhindern, die ansonsten als gravierende Logikfehler großen Ärger bei den Kunden und großen Aufwand beim Software-Hersteller verursachen würden (Simons 2004, S. 60).

Während normale Felder automatisch mit der typspezifischen Null initialisiert werden (siehe Abschnitt 4.2.3), ist bei finalisierten Feldern eine *explizite* Initialisierung erforderlich. Diese darf bei der Deklaration (siehe Abschnitt 4.2.2), in einem Konstruktor (siehe Abschnitt 4.4.3) oder in einem Instanzinitialisierer (siehe Abschnitt 4.4.4) erfolgen.

In unserer Klasse `Bruch` könnten wir für eine fortlaufende Nummerierung der im Programmablauf erzeugten Objekte sorgen und in einer Instanzvariablen die individuelle Nummer aufbewahren. Bei einer finalisierten Instanzvariablen ist keine irrtümliche Wertänderung zu befürchten, sodass eventuell eine **public**-Deklaration wie im folgenden Beispiel in Frage kommt:

```
public final int nummer;
```

Für die obligatorische initiale Wertzuweisung kann z. B. in den Konstruktoren der Klasse gesorgt werden:

```
public Bruch() {nummer = ++anzahl;}
```

Diese Konstruktoren-Definition greift dem Kursverlauf in doppelter Weise vor:

- Wir haben die Konstruktoren einer Klasse noch nicht behandelt (siehe Abschnitt 4.4.3).
- In der Anweisung dieses Konstruktors wird das statische Feld `anzahl` der Klasse `Bruch` benutzt, das erst im Abschnitt 4.5.1 in die `Bruch`-Definition eingebaut wird, um die Anzahl der bisher erzeugten Objekte festzuhalten.

Trotzdem sollte das Beispiel illustriert haben, wann eine finalisierte Instanzvariable in Frage kommt, und wie sie zu verwenden ist.

Komplett unveränderliche Klassen (z. B. durch Finalisierung *aller* Felder) spielen in der Software-Entwicklung schon lange eine wichtige Rolle und gewinnen speziell im Zusammenhang mit dem relativ neuen Paradigma der funktionalen Programmierung an Bedeutung (siehe Kapitel 12). Code mit vielen veränderlichen Variablen ist fehleranfällig, relativ schwer zu verstehen und schlecht zu

parallelisieren, d.h. auf mehrere Prozessorkerne zu verteilen. Unveränderliche Klassen im Java-API sind z. B.:

- **String**
- Die Verpackungsklassen **Integer**, **Double** etc. für primitive Datentypen (siehe Abschnitt 5.3)

Bloch (2018, S. 82ff) nennt folgende Vorteile unveränderlicher Klassen:

- Wird bei der Kreation eines Objekts für einen gültigen Zustand gesorgt, ist die Gültigkeit während der gesamten Lebenszeit garantiert, was die Handhabung von Objekten sicher und einfach macht.
- Ein unveränderliches Objekt kann ohne Synchronisierungsaufwand von mehreren Threads genutzt werden (siehe Kapitel 15). Es wird also auf besonders einfache Weise Thread-Sicherheit erzielt.
- Unveränderliche Objekte können an mehreren Stellen einer Anwendung wiederverwendet werden, statt jeweils ein neues Objekt zu erzeugen (z. B. ein **String**-Objekt mit dem Inhalt „`N.N.`“). Diese Option zur Reduktion von Aufwand bei der Kreation und Entsorgung von Objekten benutzen z. B. die sogenannten *Fabrikmethoden* (siehe Abschnitt 4.4.5).

Die Klasse `Bruch` folgt dem modernen Trend hin zu unveränderlichen Objekten noch *nicht*, was vermutlich Programmiereinsteigern entgegenkommt. Sobald die sichere Verwendung der Klasse in einer Multithreading-Umgebung relevant wird, sollte eine unveränderliche Neukonzeption erwogen werden.

Dem funktionalen Programmierstil verpflichtete Methoden verändern nicht den Zustand von Objekten (z. B. die Koordinaten einer Position), sondern produzieren nach Bedarf neue Objekte (z. B. eine neue Position). Man kann sich vorstellen, dass die funktionale Programmierung nicht für alle Aufgabenstellungen angemessen ist. Sehr viele Objekte zu erstellen, verursacht einen hohen Zeitaufwand und bei großen Objekten auch einen hohen Speicherbedarf.

Bloch (2018, S. 86) kommt nach Abwägung von Vor- und Nachteilen der Unveränderlichkeit zur Empfehlung, Klassen nach Möglichkeit als komplett unveränderlich zu konzipieren.[1] Wenn die Unveränderlichkeit einer Klasse (z. B. aus Performanzgründen) nicht vollständig sein kann, dann sollte sie doch so weit wie möglich realisiert werden:

> Declare every field **private final** unless there's a good reason to do otherwise.

Traditionelle, veränderliche Klassen sind aber für viele Aufgaben weiterhin unverzichtbar. Wir werden als Alternative zur Klasse **String**, die für unveränderliche Zeichenfolgen optimiert ist, später die Klassen **StringBuilder** und **StringBuffer** kennenlernen, die für variable Zeichenfolgen konzipiert ist. In einem Testprogramm wird sich zeigen, dass die unveränderliche Klasse **String** für bestimmte Algorithmen nicht geeignet ist.

4.3 Instanzmethoden

Durch eine Bauplan-Klassendefinition werden Objekte mit einer Anzahl von Verhaltenskompetenzen entworfen, die per Methodenaufruf genutzt werden können. Objekte sind also Dienstleister, die eine Reihe von Nachrichten interpretieren und mit passendem Verhalten beantworten können. Ihre Instanzvariablen (Eigenschaften) sind bei konsequenter Datenkapselung für fremde Klassen unsichtbar (*information hiding*). Um fremden Klassen trotzdem (kontrollierte) Zugriffe auf eine In-

[1] Ein auf unserem aktuellen Ausbildungsstand schwer nachvollziehbarer Satz: Um die komplette Unveränderlichkeit von Objekten zu erzielen, muss man nicht nur Schreibzugriffe auf die Felder verhindern (z. B. durch das Finalisieren), sondern z. B. auch die Definition einer abgeleiteten Klasse unterbinden (wie z. B. bei der API-Klasse **String**).

stanzvariable zu ermöglichen, sind entsprechende Methoden zum Lesen bzw. Verändern erforderlich. Zu diesen speziellen Methoden (oft als *getter* und *setter* bezeichnet) gesellen sich diverse andere, die in der Regel komplexere Dienstleistungen erbringen.

Beim Aufruf einer Methode ist in der Regel über sogenannte **Parameter** die gewünschte Verhaltensweise festzulegen, und von vielen Methoden wird dem Aufrufer ein **Rückgabewert** geliefert, z. B. mit der angeforderten Information.

Ziel einer typischen Klassendefinition sind kompetente, einfach und sicher einsetzbare Objekte, die oft auch noch *reale* Objekte aus dem Aufgabenbereich der Software repräsentieren. Wenn ein anderer Programmierer z. B. ein Objekt aus unserer Beispielklasse `Bruch` verwendet, kann er es mit einen Aufruf der Methode `addiere()` veranlassen, einen per Parameter benannten zweiten Bruch zum eigenen Wert zu addieren, wobei das Ergebnis auch noch gleich gekürzt wird:

```
public void addiere(Bruch b) {
   zaehler = zaehler*b.nenner + b.zaehler*nenner;
   nenner = nenner*b.nenner;
   kuerze();
}
```

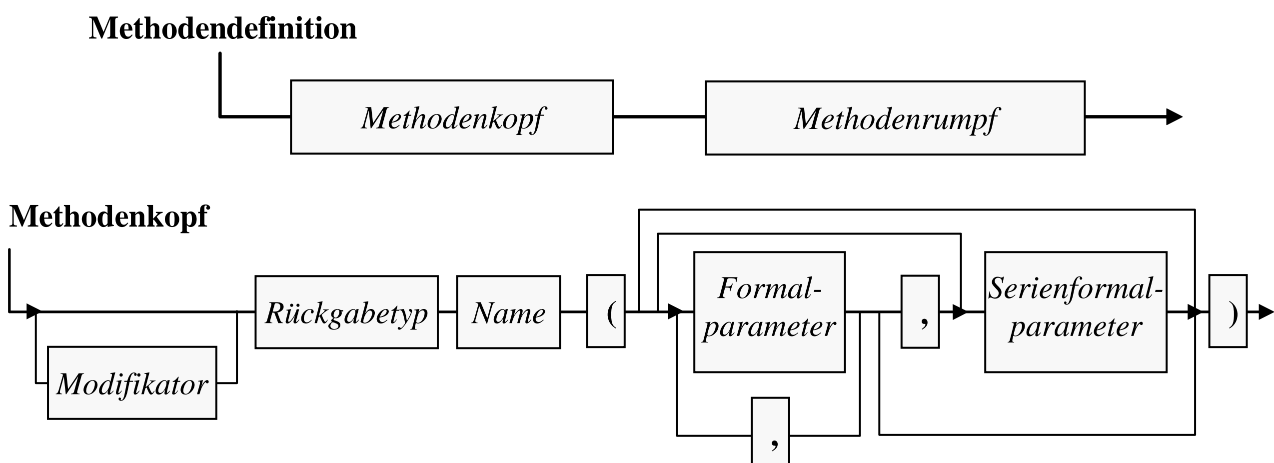
Weil diese Methode auch für beliebige fremde Klassen verfügbar sein soll, wird per Modifikator die Schutzstufe **public** gewählt.

Da es vom Verlauf der Auftragserledigung nichts zu berichten gibt, liefert `addiere()` keinen Rückgabewert. Folglich ist im Kopf der Methodendefinition der Rückgabetyp **void** angegeben.

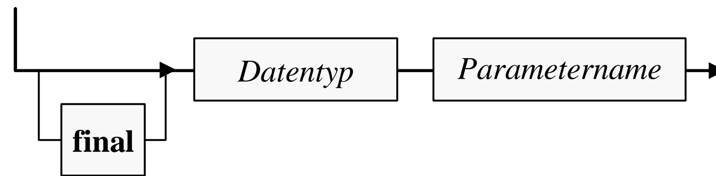
Während sich jedes Objekt mit seinem eigenen vollständigen Satz von Instanzvariablen auf dem Heap befindet, ist der Bytecode der Instanzmethoden nur *einmal* im Speicher vorhanden und wird von allen Objekten der Klasse verwendet. Er befindet sich in einem Bereich des programmeigenen Speichers, der als **Method Area** bezeichnet wird.

4.3.1 Methodendefinition

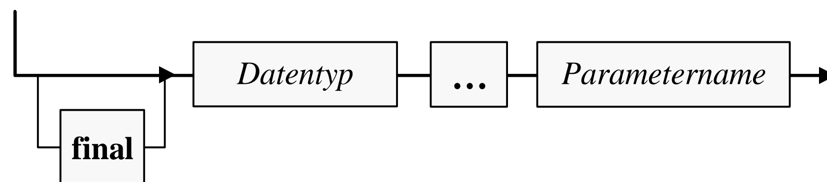
Die folgende Serie von Syntaxdiagrammen zur Methodendefinition unterscheidet sich von der im Abschnitt 3.1.3.2 präsentierten Variante durch eine genauere Erklärung der (im Abschnitt 4.3.1.4 zu behandelnden) Formalparameter:



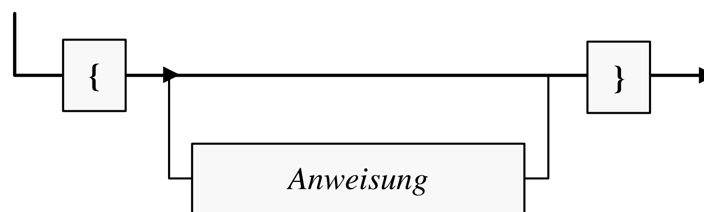
Formalparameter



Serienformalparamer



Methodenrumpf



In den nächsten Abschnitten werden die (mehr oder weniger) neuen Bestandteile dieser Syntaxdiagramme erläutert. Dabei werden Methodendefinition und -aufruf keinesfalls so sequentiell und getrennt dargestellt, wie es die Abschnittsüberschriften vermuten lassen. Schließlich ist die Bedeutung mancher Details der Methodendefinition am besten am Effekt auf den Methodenaufruf zu erkennen.

4.3.1.1 Modifikatoren

Bei einer Methodendefinition kann per Modifikator u.a. der voreingestellte Zugriffsschutz verändert werden. Wie für Instanz*variablen* gelten auch für Instanz*methoden* beim Zugriffsschutz folgende Regeln:

- Bevor sich die Frage nach der Sichtbarkeit von Instanzmethoden stellt, muss die Klasse selbst sichtbar sein:
  - Eine Klasse ist grundsätzlich in anderen Klassen des eigenen Pakets sichtbar.
  - Für die Sichtbarkeit in *allen* Klassen (aus berechtigten Modulen) muss durch den Klassen-Zugriffsmodifikator **public** gesorgt werden.[1]
- Per Voreinstellung ist der Zugriff auf Instanzmethoden allen Klassen des eigenen Pakets erlaubt.
- Mit einem Member-Zugriffsmodifikator lassen sich alternative Schutzstufen wählen, z. B.:
  - **private**
    Alle fremden Klassen werden ausgeschlossen (auch die im selben Paket).
  - **public**
    Alle Klassen in Paketen aus berechtigten Modulen dürfen zugreifen.

---

[1] Module wurden erst mit Java 9 eingeführt (im September 2017), und aktuell (im November 2019) setzen die meisten Java-Programme nur eine JVM-Version ≤ 8 voraus, sodass der Modifikator **public** den Zugriff für *alle* Klassen erlaubt.

In unserer Beispielklasse `Bruch` haben alle Methoden den Zugriffsmodifikator **public** erhalten. Damit diese Klasse mit ihren Methoden tatsächlich universell einsetzbar ist, muss sie allerdings noch in ein explizites Paket aufgenommen werden. Noch gehört die Klasse `Bruch` zum Standardpaket, und dessen Klassen sind in anderen Paketen generell *nicht* verfügbar. Im Kapitel 6 über Module und Pakete werden wir den Zugriffsschutz für Klassen und ihre Member ausführlich und endgültig behandeln.

Später (z. B. im Zusammenhang mit der Vererbung) werden uns noch Methoden-Modifikatoren begegnen, die anderen Zwecken als der Zugriffsregulation dienen (z. B. **final**, **abstract**).

4.3.1.2 Rückgabewert und return-Anweisung

Per Rückgabewert kann eine Methode auf elegante Weise Informationen an ihren Aufrufer übermitteln. Man ist auf einen einzigen Wert beschränkt, hat aber beim Typ die freie Wahl, sodass auch ein komplexes Informationsobjekt geliefert werden kann.

Wir haben schon im Abschnitt 3.5.2 gelernt, dass ein Methodenaufruf einen *Ausdruck* darstellt und als Argument von komplexeren Ausdrücken oder von Methodenaufrufen verwendet werden darf, sofern die Methode einen Wert von passendem Typ abliefert.

Bei der Definition einer Methode muss festgelegt werden, von welchem Datentyp ihr Rückgabewert ist. Erfolgt *keine* Rückgabe, ist der Ersatztyp **void** anzugeben.

Als Beispiel betrachten wir die `Bruch`-Methode `setzeNenner()`, die den Aufrufer durch einen Rückgabewert vom Datentyp **boolean** darüber informiert, ob sein Auftrag ausgeführt wurde (**true**) oder nicht (**false**):

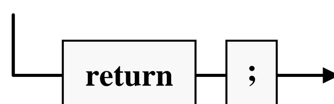
```
public boolean setzeNenner(int n) {
   if (n != 0) {
      nenner = n;
      return true;
   } else
      return false;
}
```

Ist der Rückgabetyp einer Methode von **void** verschieden, dann *muss* im Rumpf der Methode dafür gesorgt werden, dass jeder mögliche Ausführungspfad mit einer **return**-Anweisung endet, die einen Rückgabewert von kompatiblem Typ liefert:[1]

return-Anweisung für Methoden *mit* Rückgabewert

Bei Methoden *ohne* Rückgabewert (mit dem Rückgabetyp **void**) ist die **return**-Anweisung nicht unbedingt erforderlich, kann jedoch (in einer Variante *ohne* Ausdruck) dazu verwendet werden, um die Methode vorzeitig zu beenden, was meist im Rahmen einer bedingten Anweisung geschieht:

return-Anweisung für Methoden *ohne* Rückgabewert



[1] Wird eine Methode mit einem unbehandelten Ausnahmefehler abgebrochen (siehe Kapitel 11), ist allerdings keine **return**-Anweisung erforderlich.

Um ein Beispiel für die **return**-Anweisung ohne Rückgabewert in der `Bruch`-Klassendefinition unterzubringen, könnten wir in der Methode `kuerze()`

```
public void kuerze() {
   if (zaehler != 0) {
      int rest;
      int ggt = Math.abs(zaehler);
      int divisor = Math.abs(nenner);
      do {
         rest = ggt % divisor;
         ggt = divisor;
         divisor = rest;
      } while (rest > 0);
      zaehler /= ggt;
      nenner  /= ggt;
   } else
      nenner = 1;
}
```

auf die **if-else** - Fallunterscheidung verzichten und stattdessen in einer **if**-Anweisung beim Zählerwert 0 die Methode vorzeitig verlassen:

```
public void kuerze() {
   if (zaehler == 0) {
      nenner = 1;
      return;
   }
   int rest;
   int ggt = Math.abs(zaehler);
   int divisor = Math.abs(nenner);
   do {
      rest = ggt % divisor;
      ggt = divisor;
      divisor = rest;
   } while (rest > 0);
   zaehler /= ggt;
   nenner  /= ggt;
}
```

4.3.1.3 Namen

Bei der Benennung einer Methode sollten in Java die folgenden Konventionen eingehalten werden:

- Der Namen beginnt mit einem Kleinbuchstaben.
- Am Anfang sollte ein Verb stehen (z. B. `addiere()`, `kuerze()`).
- Folgen auf das Verb noch weitere Wörter, die meist keine Verben sind, (z. B. `setzeNenner()`, `compareTo()`), dann schreibt man ab dem zweiten Wort die Anfangsbuchstaben groß (*Camel Casing*).

Abgesehen von der handlungsorientierten Benennung durch Verwendung von Verben bestehen also keine Unterschiede zu den Namen von lokalen Variablen oder Feldern.

4.3.1.4 Formalparameter

Methodenparameter wurden Ihnen bisher vereinfachend als Informationen über die gewünschte Arbeitsweise einer Methode vorgestellt, die beim Aufruf übergeben werden. Tatsächlich ermöglichen Parameter aber den Informationsaustausch zwischen einem Aufrufer und einer angeforderten Methode in *beide* Richtungen.

Im Kopf der Methodendefinition werden über sogenannte **Formalparameter** Daten von bestimmtem Typ spezifiziert, die der Methode beim Aufruf zur Verfügung gestellt werden müssen.

In den Anweisungen des Methodenrumpfs werden die Formalparameter **wie lokale Variablen** verwendet, die mit den beim Aufruf übergebenen Aktualparameterwerten (siehe Abschnitt 4.3.2) initialisiert worden sind.

Methodeninterne Änderungen an den Inhalten dieser speziellen lokalen Variablen haben keinen Effekt auf die Außenwelt (siehe Abschnitt 4.3.1.4.1). Werden einer Methode *Referenzen* übergeben, kann sie jedoch im Rahmen ihrer Zugriffsrechte auf die zugehörigen Objekte einwirken (siehe Abschnitt 4.3.1.4.2) und so Informationen nach Außen transportieren.

Für jeden Formalparameter sind folgende Angaben zu machen:

- **Datentyp**
  Es sind beliebige Typen erlaubt (primitive Typen, Referenztypen). Man muss den Datentyp eines Formalparameters auch dann explizit angeben, wenn er mit dem Typ des linken Nachbarn übereinstimmt.
- **Name**
  Für Parameternamen gelten dieselben Regeln bzw. Konventionen wie für Variablennamen. Weil Formalparameter im Methodenrumpf wie lokale Variablen funktionieren, …
  - müssen sich die Parameternamen von den Namen der (anderen) lokalen Variablen unterscheiden,
  - werden namensgleiche Instanz- bzw. Klassenvariablen überlagert.
    Diese bleiben jedoch über ein geeignetes Präfix weiter ansprechbar. Durch einen Punktoperator separiert setzt man ...
    - das Schlüsselwort **this** vor eine Instanzvariable
    - den Klassennamen vor eine statische Variable

  Um Namenskonflikte (mit lokalen Variablen oder Feldern) zu vermeiden, hängen manche Programmierer an Parameternamen ein Suffix an, z. B. *par* oder einen Unterstrich.
- **Position**
  Die Position eines Formalparameters ist natürlich nicht gesondert anzugeben, sondern liegt durch die Methodendefinition fest. Sie wird hier als relevante Eigenschaft erwähnt, weil die beim späteren Aufruf der Methode übergebenen Aktualparameter gemäß ihrer Reihenfolge den Formalparametern zugeordnet werden. Java kennt keine Namensparameter, sondern nur Positionsparameter.

Ein Formalparameter kann wie jede andere lokale Variable mit dem Modifikator **final** auf den Initialisierungswert fixiert werden. Auf diese Weise lässt sich die (kaum jemals sinnvolle) Änderung des Initialisierungswertes verhindern. Welche Vorteile es hat, ungeplante Veränderungen von lokalen Variablen (und damit auch von Formalparametern) systematisch per **final**-Deklaration zu verhindern, wurde im Abschnitt 3.3.10 erläutert.

4.3.1.4.1 Parameter mit primitivem Datentyp

Über einen Parameter mit primitivem Datentyp werden Informationen in eine Methode kopiert, um diese mit Daten zu versorgen und/oder ihre Arbeitsweise zu steuern. Als Beispiel betrachten wir die folgende Variante der `Bruch`-Methode `addiere()`. Das beauftragte Objekt soll den via Parameterliste als Paar von Zähler und Nenner (`z`, `n`) übergebenen Bruch zu seinem eigenen Wert addieren und optional (Parameter `autokurz`) das Resultat gleich kürzen:

```
public boolean addiere(int z, int n, boolean autokurz) {
   if (n != 0) {
      zaehler = zaehler*n + z*nenner;
      nenner = nenner*n;
      if (autokurz)
         kuerze();
      return true;
   } else
      return false;
}
```

Methodeninterne Änderungen bei den über Formalparameternamen ansprechbaren lokalen Variablen bleiben ohne Effekt auf eine als Aktualparameter fungierende Variable der rufenden Methode. Im folgenden Beispiel übersteht die lokale Variable `imain` der Methode **main()** den Einsatz als Aktualparameter beim Aufruf der Instanzmethode `primParDemo()` ohne Folgen:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
    void primParDemo (int ipar) {
        System.out.println(++ipar);
    }

    public static void main(String[] args) {
        int imain = 4711;
        Prog p = new Prog();
        p.primParDemo(imain);
        System.out.println(imain);
    }
}</pre></td><td><pre>4712
4711</pre></td></tr>
</table>

Im Beispielprogramm ist die Klasse `Prog` startfähig; sie besitzt also eine öffentliche und statische Methode namens **main()** mit dem Rückgabetyp **void** und einem Parameter vom Typ **String[]**. In **main()** wird ein Objekt der Klasse `Prog` erzeugt und beauftragt, die Instanzmethode `primParDemo()` auszuführen. Mit dieser auch in den folgenden Abschnitten anzutreffenden, etwas umständlich wirkenden Konstruktion wird es vermieden, im aktuellen Abschnitt 4.3.1 über Details bei der Definition von *Instanzmethoden* zur Demonstration *statische* Methoden (außer **main()**) verwenden zu müssen. Bei den Parametern und beim Rückgabewert gibt es allerdings keine Unterschiede zwischen den Instanz- und den Klassenmethoden (siehe Abschnitt 4.5.3).

4.3.1.4.2 Parameter mit Referenztyp

Wir haben schon festgehalten, dass die Formalparameter einer Methode wie lokale Variablen funktionieren, die mit den Werten der Aktualparameter initialisiert worden sind. Methodeninterne Änderungen bei den Werten dieser lokalen Variablen wirken sich *nicht* auf die als Aktualparameter verwendeten Variablen der rufenden Methode aus. Auch bei einem Parameter mit *Referenztyp* (ab jetzt kurz als *Referenzparameter* bezeichnet) wird der Wert des Aktualparameters (eine Objektadresse) beim Methodenaufruf in eine lokale Variable kopiert. Dabei wird aber keinesfalls eine Kopie des referenzierten Objekts (auf dem Heap) erstellt. Vielmehr greift die aufgerufene Methode über ihre lokale Referenzvariable auf das Originalobjekt zu und kann dort durch den Zugriff auf Instanzvariablen oder durch Methodenaufrufe Veränderungen vornehmen, sofern sie dazu berechtigt ist.

Die ältere Version der `Bruch`-Methode `addiere()` verfügt über einen Referenzparameter mit dem Datentyp `Bruch`:

```
public void addiere(Bruch b) {
   zaehler = zaehler*b.nenner + b.zaehler*nenner;
   nenner = nenner*b.nenner;
   this.kuerze();
}
```

Durch einen Aufruf dieser Methode wird ein `Bruch`-Objekt beauftragt, den via Referenzparameter spezifizierten Bruch zu seinem eigenen Wert zu addieren (und das Resultat gleich zu kürzen). Zähler und Nenner des fremden `Bruch`-Objekts können per Referenzparameter und Punktoperator trotz Schutzstufe **private** direkt angesprochen werden, weil der Zugriff in einer Methode der Klasse `Bruch` stattfindet.

Dass in einer `Bruch`-Methodendefinition ein Referenzparameter vom Typ `Bruch` verwendet wird, ist übrigens weder „zirkulär" noch ungewöhnlich. Es ist vielmehr unvermeidlich, wenn `Bruch`-Objekte kooperieren sollen.

Beim Aufruf der Methode `addiere()` bleibt das per Referenzparameter ansprechbare Objekt unverändert. Sofern entsprechende Zugriffsrechte vorliegen, was bei einem Referenzparameter vom Typ der eigenen Klasse stets der Fall ist, kann eine Methode das Referenzparameterobjekt aber auch verändern. Wir erweitern unsere Klasse `Bruch` um eine Methode namens `dupliziere()`, die ein Objekt beauftragt, die Werte seiner Instanzvariablen auf ein anderes `Bruch`-Objekt zu übertragen, das per Referenzparameter bestimmt wird:

```
public void dupliziere(Bruch bc) {
   bc.zaehler = zaehler;
   bc.nenner = nenner;
   bc.etikett = etikett;
}
```

Hier liegt *kein* Verstoß gegen das Prinzip der Datenkapselung vor, weil der Zugriff auf die Instanzvariablen des Parameterobjekts durch eine klasseneigene Methode erfolgt, die vom Klassendesigner sorgfältig konzipiert sein sollte.

Im folgenden Programm wird das `Bruch`-Objekt `b1` beauftragt, die `dupliziere()` - Methode auszuführen, wobei als Parameter eine Referenz auf das Objekt `b2` übergeben wird:

Quellcode

```
class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b1 = new Bruch(), b2 = new Bruch();
      b1.setzeZaehler(1); b1.setzeNenner(2);
      b1.setzeEtikett("b1 = ");
      b2.setzeZaehler(5); b2.setzeNenner(6);
      b2.setzeEtikett("b2 = ");
      b1.zeige();
      b2.zeige();
      b1.dupliziere(b2);
      System.out.println("b2 nach dupliziere():\n");
      b2.zeige();
   }
}
```

Ausgabe

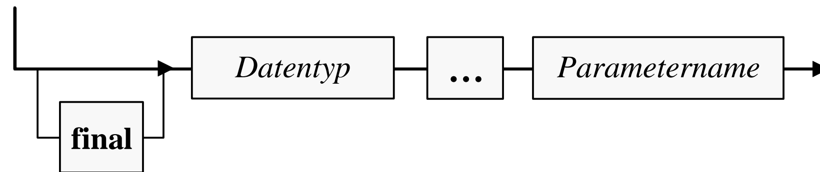
```
        1
 b1 =  -----
        2

        5
 b2 =  -----
        6

b2 nach dupliziere():

        1
 b1 =  -----
        2
```

Die Referenzparametertechnik eröffnet den (berechtigten) Methoden nicht nur unbegrenzte Wirkungsmöglichkeiten, sondern spart auch Zeit und Speicherplatz beim Methodenaufruf. Über einen Referenzparameter wird ein beliebig voluminöses Objekt in der aufgerufenen Methode verfügbar, ohne dass es (mit Zeit- und Speicheraufwand) kopiert werden müsste.

4.3.1.4.3 Serienparameter

Seit der Version 5.0 (alias 1.5) bietet Java auch Parameterlisten variabler Länge, wozu *am Ende* der Formalparameterliste eine *Serie* von Elementen desselben Typs über folgende Syntax deklariert werden kann:

Serienformalparamer



Man spricht von einem *Serienparameter* oder von einem *Varargs-Parameter*. Methoden mit einem Varargs-Parameter werden in der englischsprachigen Literatur gelegentlich als *variable arity* - Methoden bezeichnet und den *fixed arity* - Methoden gegenübergestellt, die keinen Varargs-Parameter besitzen.

Als Beispiel betrachten wir eine weitere Variante der `Bruch`-Methode `addiere()`, mit der ein Objekt beauftragt werden kann, *mehrere* fremde Brüche zum eigenen Wert zu addieren:

```
public void addiere(Bruch... bar) {
   for (Bruch b : bar)
      addiere(b);
}
```

Ob man zwischen den Typbezeichner und die drei Punkte (das sogenannte *Auslassungszeichen*, engl.: *ellipsis*) ein trennendes Leerzeichen setzt oder (wie im Beispiel) der Konvention folgend (vgl. Gosling et al. 2019, Abschnitt 8.4.1) darauf verzichtet, ist für den Compiler irrelevant.

Ein Serienparameter besitzt einen Array-Datentyp, zeigt also auf ein Objekt mit einer Serie von Instanzvariablen desselben Typs. Wir haben Arrays zwar noch nicht offiziell behandelt (siehe Abschnitt 5.1), aber doch schon gelegentlich verwendet, zuletzt im Zusammenhang mit der **for**-Schleifen - Variante für Arrays und andere Kollektionen (siehe Abschnitt 3.7.3.2). Im aktuellen Beispiel wird diese Schleifenkonstruktion dazu genutzt, um jedes Element im Array `bar` mit `Bruch`-Objekten durch einen Aufruf der ursprünglichen `addiere()` - Methode zum handelnden Bruch zu addieren.

Mit den `Bruch`-Objekten `b1` bis `b4` sind z. B. folgende Aufrufe erlaubt:

```
b1.addiere(b2);
b1.addiere(b2, b3);
b1.addiere(b2, b3, b4);
```

Es ist sogar erlaubt (im aktuellen Beispiel allerdings sinnlos), für einen Serienformalparameter beim Aufruf überhaupt keinen Aktualparameter anzugeben:

```
b1.addiere();
```

Weil per Serienparametersyntax letztlich ein Parameter mit Array-Datentyp deklariert wird, kann man beim Methodenaufruf an Stelle einer Serie von einzelnen Aktualparametern auch einen Array mit diesen Elementen übergeben. In der ersten Anweisung des folgenden Beispiels wird (dem Abschnitt 5.1.8 vorgreifend) ein Array-Objekt per Initialisierungsliste erzeugt. In der zweiten Anweisung wird dieses Objekt an die obige Serienparametervariante der `addiere()` - Methode übergeben:

```
Bruch[] ba = {b2, b3, b4};
b1.addiere(ba);
```

Eine weitere Methode mit Serienparameter kennen Sie übrigens schon aus dem Abschnitt 3.2.2 über die formatierte Ausgabe mit der **PrintStream**-Methode **printf()**, die folgenden Definitionskopf besitzt:

public PrintStream printf (String *format*, **Object...** *args*)

Dass die Methode **printf()** eine Referenz auf das handelnde **PrintStream**-Objekt als (meist ignorierten) Rückgabewert liefert, kann uns momentan gleichgültig sein.

Weil in diesem Abschnitt zwei Serienparameter-Beispiele mit einer Klasse als Elementtyp aufgetreten sind, soll vorsichtshalber betont werden, dass auch ein primitiver Elementtyp erlaubt ist, z. B.:

```
public void prInt(int... iar) {
    for (int i : iar)
        System.out.println(i);
}
```

4.3.1.5 Methodenrumpf

Über die Blockanweisung, die den Rumpf einer Methode bildet, haben Sie bereits erfahren:

- Hier werden die Formalparameter wie lokale Variablen verwendet. Ihre Besonderheit besteht darin, dass sie bei jedem Methodenaufruf über Aktualparameter vom Aufrufer initialisiert werden, sodass dieser den Ablauf der Methode beeinflussen kann.
- Die **return**-Anweisung dient zur Rückgabe eines Wertes an den Aufrufer und/oder zum Beenden der Methodenausführung. Bei einer Methode mit Rückgabe muss jeder Ausführungspfad mit einer **return**-Anweisung enden, die einen Wert von kompatiblem Typ liefert. Bei einer Methode mit dem Pseudorückgabetyp **void** kann die **return**-Anweisung (in der Variante *ohne* Ausdruck) optional dazu verwendet werden, um die Methode vorzeitig zu verlassen.

Ansonsten können beliebige Anweisungen unter Verwendung von elementaren und objektorientierten Sprachelementen eingesetzt werden, um den Zweck der Methode zu implementieren.

Im letzten Satz war bewusst von *dem Zweck* einer Methode die Rede und nicht von *den Zwecken*. Durch Mehrzweckmethoden verschlechtern sich Lesbarkeit und Wartungsfreundlichkeit, während das Fehlerrisiko steigt, weil z. B. die für eine Teilaufgabe benötigten Variablen auch im Code-Segment anderer Teilaufgaben gültig sind und durch Tippfehler unverhofft ins Spiel kommen oder in Mitleidenschaft gezogen werden können. Um diese Nachteile zu vermeiden, sollte für jede Aufgabe bzw. Aktivität eine eigene Methode definiert werden (Bloch 2018, S. 263).

Weil in einer Methode häufig andere Methoden aufgerufen werden, kommt es in der Regel zu mehrstufig verschachtelten Methodenaufrufen, wobei die Höhe des Stacks (Stapelspeichers) zur Verwaltung der Methodenaufrufe entsprechend wächst (siehe Abschnitt 4.3.3).

4.3.2 Methodenaufruf und Aktualparameter

Beim Aufruf einer Instanzmethode, z. B.:

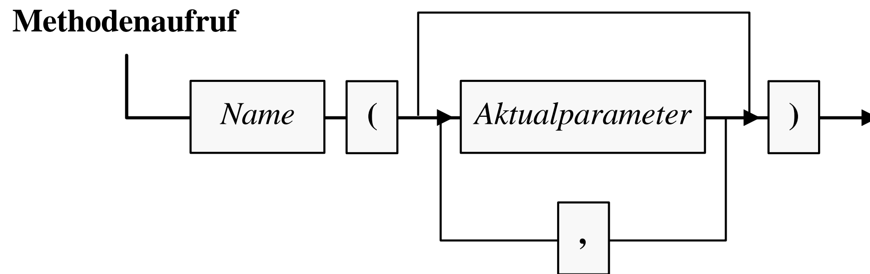
```
b1.zeige();
```

wird nach objektorientierter Denkweise eine *Botschaft* an ein Objekt geschickt:

„b1, zeige dich!“

Als Syntaxregel ist festzuhalten, dass zwischen dem Objektnamen (genauer: dem Namen der Referenzvariablen, die auf das Objekt zeigt) und dem Methodennamen der **Punktoperator** zu stehen hat. Eine analoge Syntaxregel haben Sie beim Zugriff auf Instanzvariablen kennengelernt.

Beim Aufruf einer Methode folgt ihrem Namen die in runde Klammern eingeschlossene Liste mit den **Aktualparametern**, wobei es sich um eine analog zur Formalparameterliste geordnete Sequenz von Ausdrücken mit kompatiblen Datentypen handeln muss.



Es muss grundsätzlich eine Parameterliste angegeben werden, ggf. eine leere wie im obigen Aufruf der Methode `zeige()`.

Als Beispiel mit Aktualparametern betrachten wir einen Aufruf der im Abschnitt 4.3.1.4.1 vorgestellten Variante der `Bruch`-Methode `addiere()`:

```
b1.addiere(1, 2, true);
```

Als Aktualparameter sind Ausdrücke zugelassen, deren Typ entweder direkt mit dem Formalparametertyp übereinstimmt oder mit diesem Typ kompatibel ist:

- Bei primitiven Datentypen findet automatisch eine erweiternde Typanpassung statt (vgl. Abschnitt 3.5.7.1).
- Hat der Formalparameter den Typ eine Klasse, werden auch Objekte aus einer abgeleiteten Klasse als Aktualparameter akzeptiert (siehe Kapitel 7).
- Hat der Formalparameter den Typ einer Schnittstelle, werden Objekte aus allen Klassen akzeptiert, welche die Schnittstelle implementieren. Dieser Satz steht der Vollständigkeit halber hier, obwohl er erst nach der Behandlung der Schnittstellen im Kapitel 9 verständlich ist.

Java kennt keine Namensparameter, sondern nur Positionsparameter. Um einen Parameter mit einem Wert zu versorgen, muss dieser Wert im Methodenaufruf an der korrekten Position stehen.

Außerdem müssen stets *alle* Parameter mit Ausnahme eines eventuell am Ende der Parameterliste stehenden Serienparameters (siehe Abschnitt 4.3.1.4.3) mit Werten versorgt werden. Oft existieren aber zu einer Methode mehrere *Überladungen* mit unterschiedlich langen Parameterlisten, sodass man durch Wahl einer Überladung doch die Option hat, auf manche Parameter zu verzichten (vgl. Abschnitt 4.3.4).

Liefert eine Methode einen Wert zurück, kann der aus ihrem Aufruf bestehende **Ausdruck** als Argument in komplexeren Ausdrücken verwenden werden, z. B.:

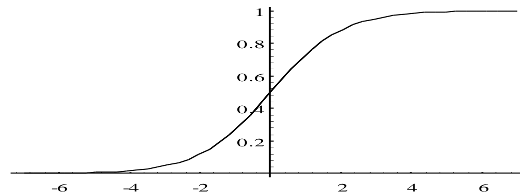
Quellcodesegment	Ausgabe

`double arg = 0.0, logist;`
`logist = Math.exp(arg)/(1+Math.exp(arg));`
`System.out.println(logist);`
	`0.5`

Im Beispiel wird die logistische Funktion

$$f(x) := \frac{e^x}{1 + e^x}$$

mit dem Graphen



unter Verwendung der statischen Methode **exp()** aus der Klasse **Math** im Paket **java.lang** an der Stelle 0,0 ausgewertet.

Außerdem ist ein Methodenaufruf als Aktualparameter erlaubt, wenn er eine Rückgabe mit kompatiblem Typ liefert, z. B.:

```
System.out.println(b.gibNenner());
```

Wie Sie schon aus Abschnitt 3.7.1 wissen, wird jeder Methodenaufruf durch ein angehängtes Semikolon zur vollständigen **Anweisung**, wobei ein Rückgabewert ggf. ignoriert wird.

Soll in einer Methodenimplementierung vom aktuell handelnden Objekt eine andere Instanzmethode ausgeführt werden, so muss beim Aufruf *keine* Objektbezeichnung angegeben werden. In den verschiedenen Varianten der `Bruch`-Methode `addiere()` soll das beauftragte Objekt den via Parameterliste übergebenen Bruch (bzw. die übergebenen Brüche) zu seinem eigenen Wert addieren und das Resultat (bei der Variante aus Abschnitt 4.3.1.4.1 parametergesteuert) gleich kürzen. Zum Kürzen kommt natürlich die entsprechende `Bruch`-Methode zum Einsatz. Weil sie vom gerade agierenden Objekt auszuführen ist, wird keine Objektbezeichnung benötigt, z. B.:

```
public void addiere(Bruch b) {
   zaehler = zaehler*b.nenner + b.zaehler*nenner;
   nenner = nenner*b.nenner;
   kuerze();
}
```

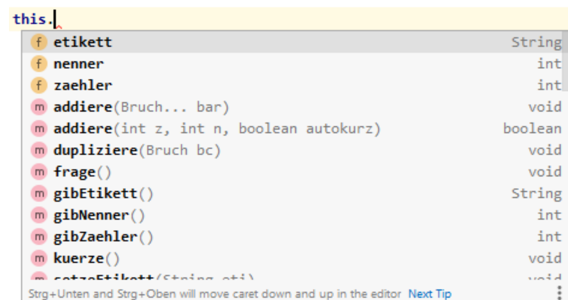
Wer auch solche Methodenaufrufe nach dem Schema

Empfänger.*Botschaft*

realisieren möchte, kann mit dem Schlüsselwort **this** das aktuelle Objekt ansprechen, z. B.:

```
this.kuerze();
```

Mit dem Schlüsselwort **this** samt angehängtem Punktoperator gibt man außerdem unserer Entwicklungsumgebung IntelliJ IDEA den Anlass, eine Liste mit allen für das agierende Objekt möglichen Methodenaufrufen und Feldnamen anzuzeigen, z. B.:



So kann man lästiges Nachschlagen und Tippfehler vermeiden.

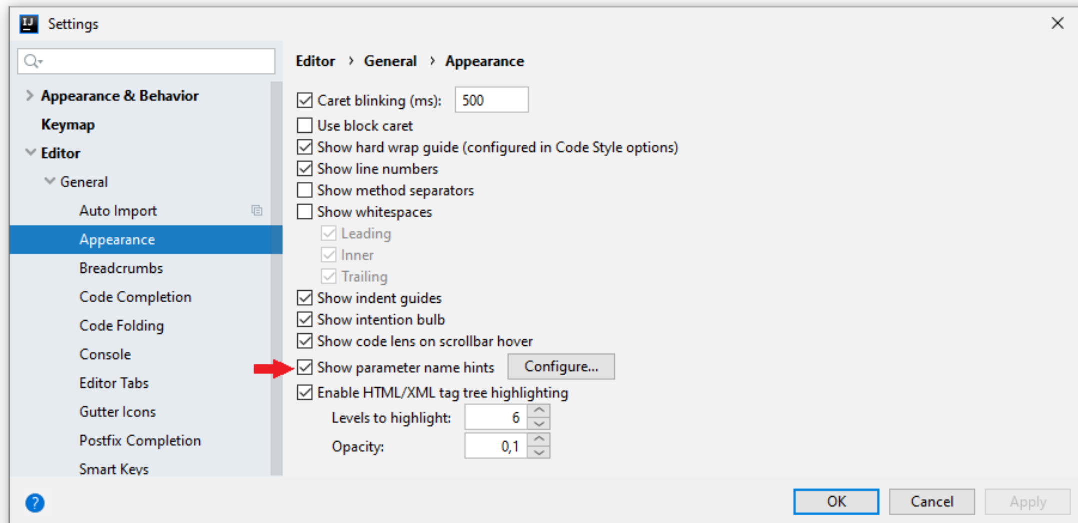
Unsere Entwicklungsumgebung zeigt bei manchen Methodenaufrufen für manche Parameter den Namen an, z. B.:

```
s = JOptionPane.showInputDialog( parentComponent: null,
        message: "Welche ganze Zahl von 2 bis 2^63-1 soll untersucht werden?",
        title: "Primzahlendetektor", JOptionPane.QUESTION_MESSAGE);
```

Dieses für Programmiereinsteiger potenziell irritierende Verhalten lässt sich nach

File > Settings > Editor > General > Appearance

ein- bzw. ausschalten sowie konfigurieren:



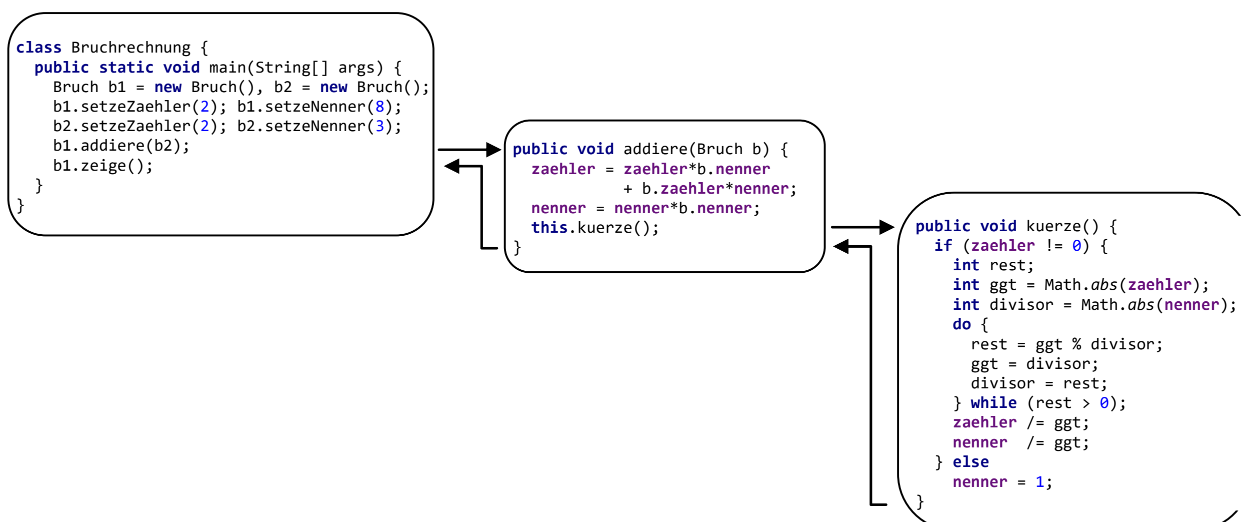
4.3.3 Debug-Einsichten zu (verschachtelten) Methodenaufrufen

Verschachtelte Methodenaufrufe stellen keine Besonderheit dar, sondern den selbstverständlichen Normalfall. Anhand der folgenden Bruchrechnungsstartklasse

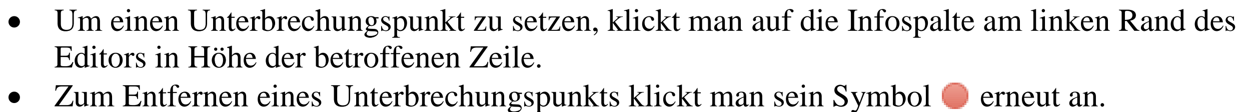
```
class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b1 = new Bruch(), b2 = new Bruch();
      b1.setzeZaehler(2); b1.setzeNenner(8);
      b2.setzeZaehler(2); b2.setzeNenner(3);
      b1.addiere(b2);
      b1.zeige();
   }
}
```

soll mit Hilfe unserer Entwicklungsumgebung IntelliJ IDEA untersucht werden, was bei folgender Aufrufverschachtelung geschieht:

- Die statische Methode **main()** der Klasse `Bruchrechnung` ruft die `Bruch`-Instanzmethode `addiere()`.
- Die `Bruch`-Instanzmethode `addiere()` ruft die `Bruch`-Instanzmethode `kuerze()`.

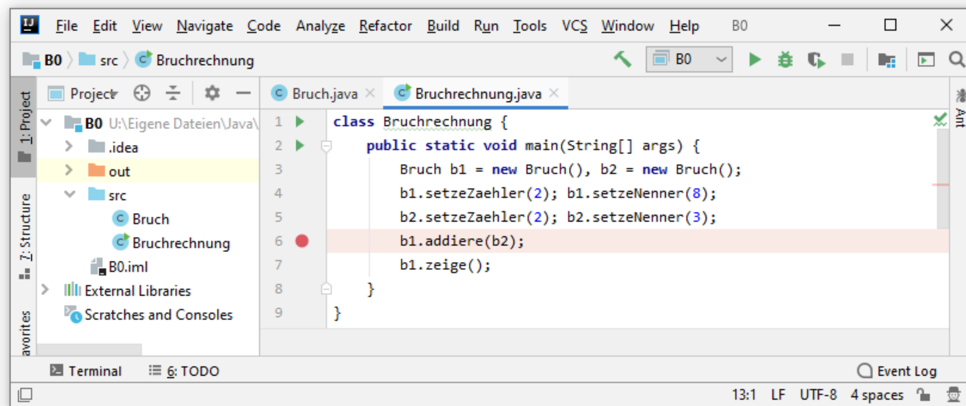


Wir verwenden die zur Fehlersuche konzipierte **Debug**-Technik von IntelliJ. Das aktuelle Bruchrechnungsprogramm soll bei der späteren Ausführung im Debug-Modus an mehreren Stellen durch einen sogenannten **Unterbrechungspunkt** (engl. *breakpoint*) angehalten werden, sodass wir jeweils die Lage im Hauptspeicher inspizieren können. Einen Unterbrechungspunkt einzurichten oder aufzuheben, macht wenig Mühe:



- Um einen Unterbrechungspunkt zu setzen, klickt man auf die Infospalte am linken Rand des Editors in Höhe der betroffenen Zeile.
- Zum Entfernen eines Unterbrechungspunkts klickt man sein Symbol erneut an.

Hier ist die **main()** - Methode des Beispielprogramms mit Unterbrechungspunkt zu sehen:



Setzen Sie weitere Unterbrechungspunkte …

- in der Methode **main()** vor dem `zeige()` - Aufruf,
- in der `Bruch`-Methode `addiere()` vor dem `kuerze()` - Aufruf,
- in der `Bruch`-Methode `kuerze()` vor der Anweisung

  ```
  ggt = divisor;
  ```

  im Block der **do-while** - Schleife.

Starten Sie die Klasse `Bruchrechnung` im Debug-Modus mit der Tastenkombination

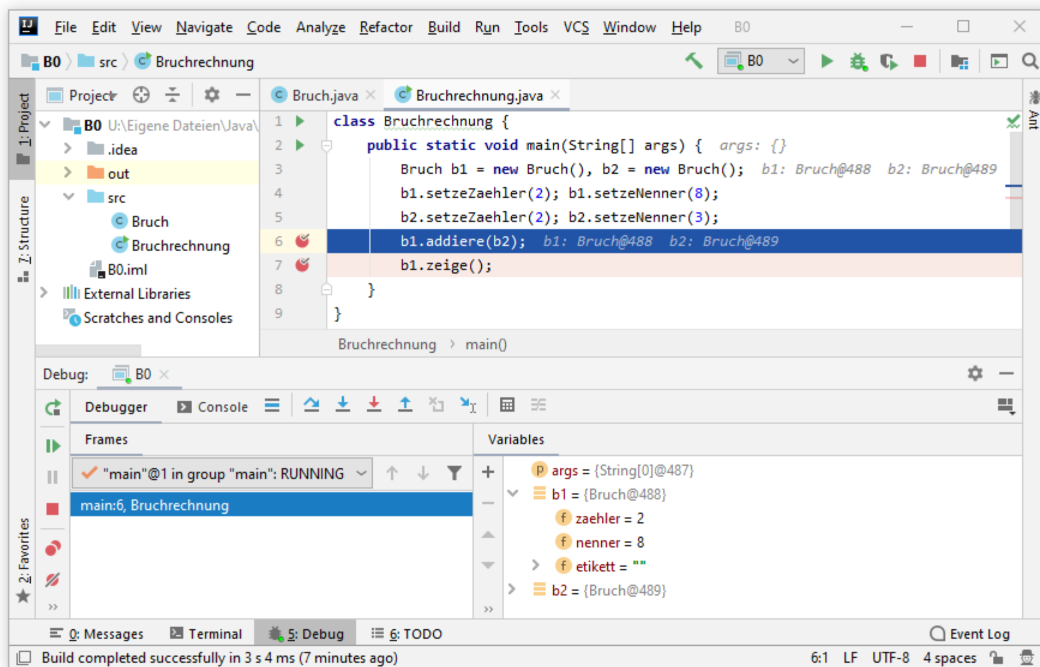
Umschalt + F9

oder über den Menübefehl

Run > Debug

oder über den Schalter in der Symbolleiste.

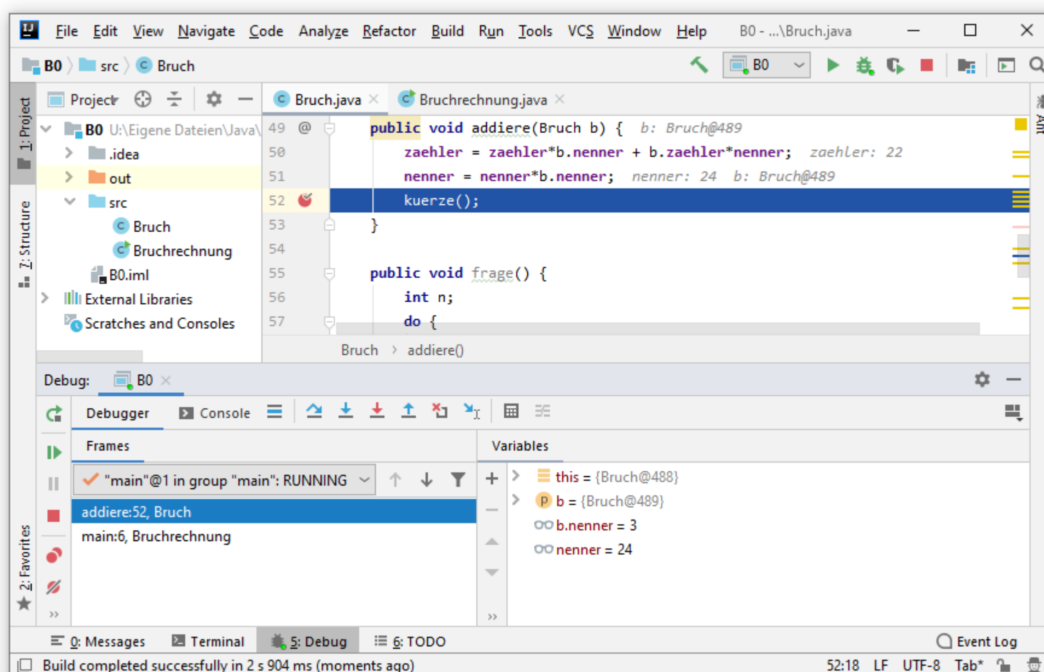
Das nun erscheinende **Debug**-Fenster zeigt auf der Registerkarte **Debugger** links die Stack-**Frames** des ausgeführten Programms. Bei Erreichen des ersten Unterbrechungspunkts (Anweisung `b1.addiere(b2);` in **main()**) ist nur der Stack Frame der Methode **main()** vorhanden:



Im **Variables**-Bereich auf der rechten Seite der **Debugger**-Registerkarte sind die lokalen Variablen der Methode **main()** zu sehen:

- Parameter `args`
- die lokalen Referenzvariablen `b1` und `b2`
  Man kann auch die Instanzvariablen der referenzierten `Bruch`-Objekte anzeigen lassen.

Lassen Sie das Programm mit dem Schalter ▶ aus der Symbolleiste des **Debug**-Fensters oder mit der Taste **F9** fortsetzen. Beim Erreichen des nächsten Unterbrechungspunkts (Anweisung `kuerze();` in der Methode `addiere()`) liegen die Stack-Frames der Methoden `addiere()` und **main()** übereinander:

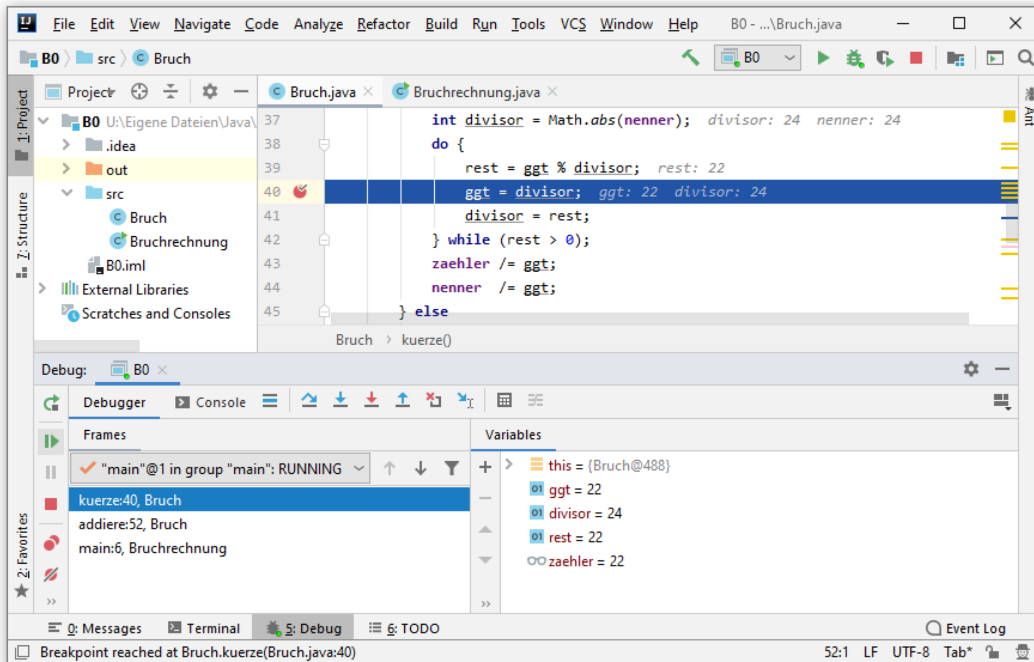


Der **Variables**-Bereich des **Debugger**-Fensters zeigt als lokale Variablen der Methode `addiere()`:

- **this** (Referenz auf das handelnde `Bruch`-Objekt)
- Parameter `b`

Man kann sich auch die Instanzvariablen der referenzierten `Bruch`-Objekte anzeigen lassen.

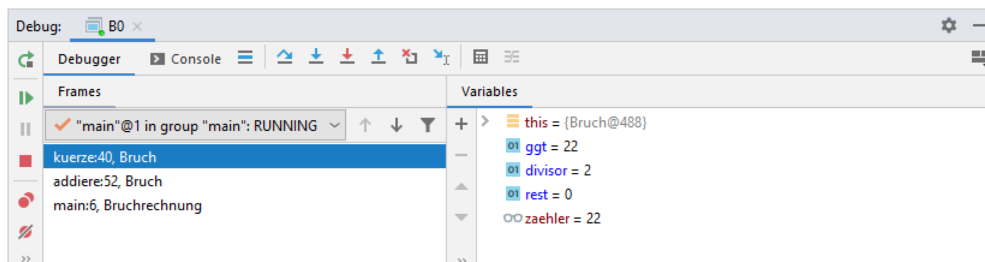
Beim Erreichen des nächsten Unterbrechungspunkts (Anweisung `ggt = divisor;` in der Methode `kuerze()`) liegen die Stack-Frames der Methoden `kuerze()`, `addiere()` und **main**() übereinander:



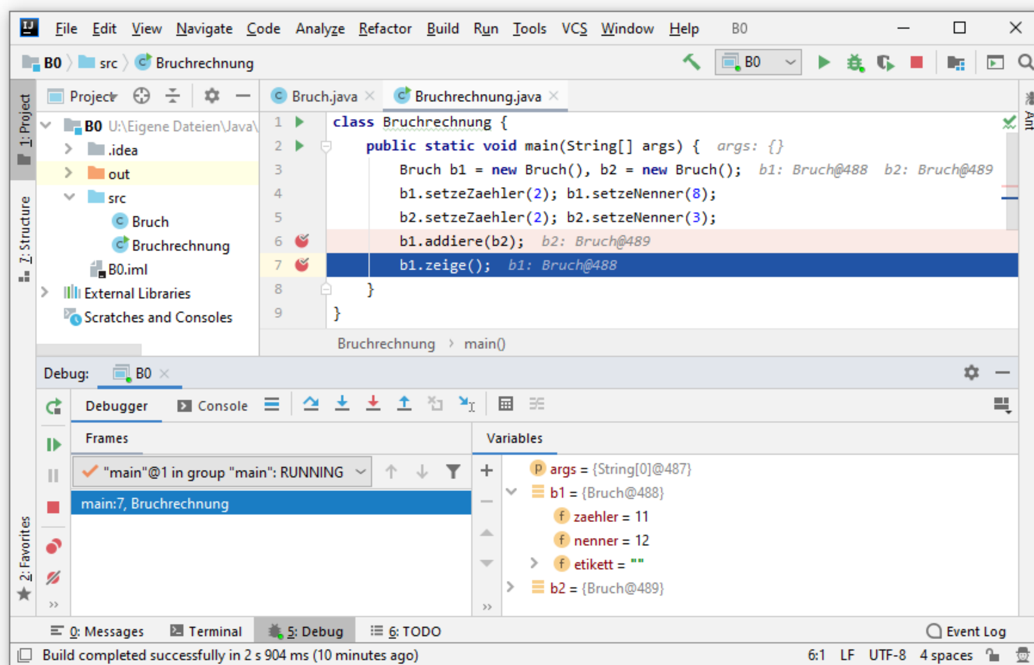
Der **Variables**-Bereich des **Debugger**-Fensters zeigt als lokale Variablen der Methode `kuerze()`:

- **this** (Referenz auf das handelnde `Bruch`-Objekt)
- die lokalen (im Block zur **if**-Anweisung deklarierten) Variablen `ggt`, `divisor` und `rest`.

Weil sich der dritte Unterbrechungspunkt in einer **do-while** - Schleife befindet, sind mehrere Fortsetzungsbefehle bis zum Verlassen der Methode `kuerze()` erforderlich, wobei die Werte der lokalen Variablen den Verarbeitungsfortschritt erkennen lassen, z. B.:

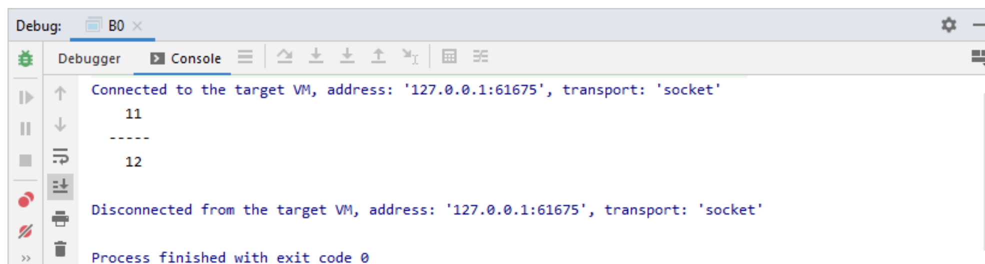


Bei Erreichen des letzten Unterbrechungspunkts (Anweisung `b1.zeige();` in **main**()) ist nur noch der Stack Frame der Methode **main**() vorhanden:



Die anderen Stack Frames sind verschwunden, und die dort ehemals vorhandenen lokalen Variablen existieren nicht mehr.

Beenden Sie das Programm durch einen letzten Fortsetzungsklick auf den Schalter . Anschließend zeigt die Registerkarte **Console** des **Debug**-Fensters die Ausgabe der Methode `zeige()`:

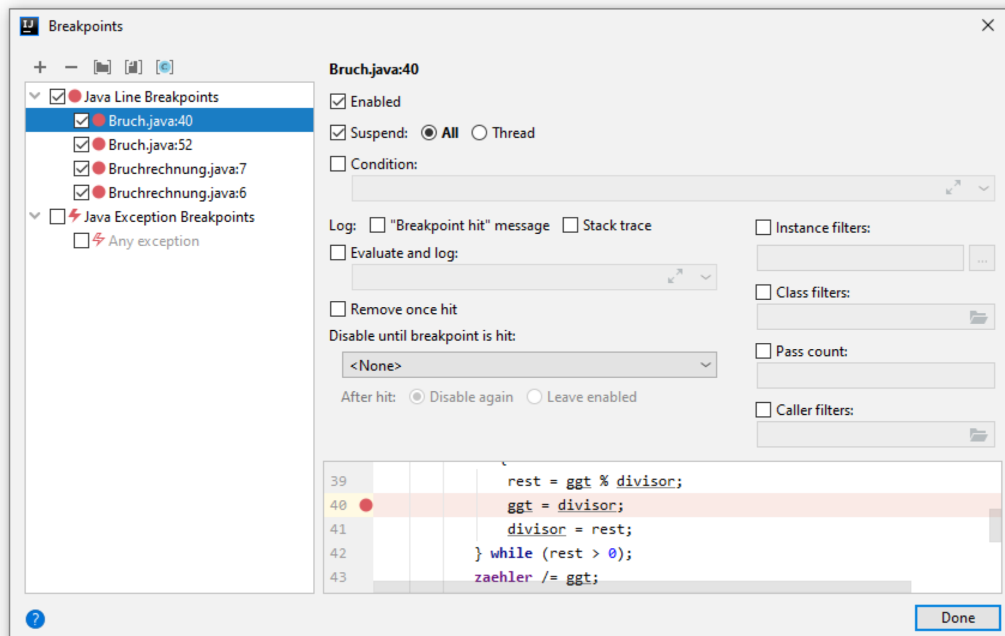


Über den Schalter des **Debug**-Fensters oder die Tastenkombination

Umschalt + Strg + F8

öffnet man das **Breakpoints**-Fenster zur Verwaltung der Unterbrechungspunkte:[1]

[1] Befindet sich beim Betätigen der Tastenkombination die Einfügemarke in einer Editorzeile mit Unterbrechungspunkt, dann erscheint zunächst ein Kontextmenü mit Eigenschaften des lokalen Unterbrechungspunkts, und die Tastenkombination **Umschalt + Strg + F8** ist erneut zu drücken, um das **Breakpoints**-Fenster zu öffnen.



Hier kann man z. B. einzelne oder auch alle Unterbrechungspunkte deaktivieren oder löschen.

Weil der verfügbare Speicher endlich ist, kann es bei einer Aufrufverschachtelung und der damit verbundenen Stapelung von Stack-Frames zu dem bereits genannten Laufzeitfehler vom Typ **StackOverflowError** kommen. Dies wird aber nur bei einem schlecht entworfenen bzw. fehlerhaften Algorithmus passieren.

4.3.4 Methoden überladen

Die beiden im Abschnitt 4.3.1.4 vorgestellten `addiere()` - Varianten können problemlos in der `Bruch`-Klassendefinition miteinander und mit der originalen `addiere()` - Variante koexistieren, weil die drei Methoden unterschiedliche Parameterlisten besitzen. Besitzt eine Klasse mehrere Methoden mit demselben Namen, liegt eine sogenannte *Überladung* vor.

Eine Überladung ist erlaubt, wenn sich die **Signaturen** der beteiligten Methoden unterscheiden. Zwei Methoden besitzen genau dann *dieselbe* Signatur, was *innerhalb einer Klasse* verboten ist, wenn die beiden folgenden Bedingungen erfüllt sind:[1]

- Die Namen der Methoden sind identisch.
- Die Formalparameterlisten sind gleich lang, und die Typen korrespondierender Parameter stimmen überein.

Für die Signatur einer Methode sind *irrelevant*:

[1] Bei den später zu behandelnden *generischen* Methoden muss die Liste der Kriterien für die Identität von Signaturen erweitert werden.

- Modifikatoren
  Insbesondere ändert der Modifikator **static** (also die Zuordnung der Methode zur Klasse, siehe Abschnitt 4.5.3) nichts an der Signatur.
- Rückgabetyp
  Die fehlende Signaturrelevanz des Rückgabetyps resultiert daraus, dass der Rückgabewert einer Methode in Anweisungen oft keine Rolle spielt (ignoriert wird). Folglich muss unabhängig vom Rückgabetyp entscheidbar sein, welche Methode aus einer Überladungsfamilie zu verwenden ist.
- Die *Namen* der Formalparameter

Ist bei einem Methodenaufruf die angeforderte Überladung nicht eindeutig zu bestimmen, meldet der Compiler einen Fehler. Um diese Konstellation in einer Variante unsere Klasse `Bruch` zu provozieren, sind einige Verrenkungen nötig:

- Die `Bruch`-Instanzvariablen `zaehler` und `nenner` erhalten den Datentyp **long**.
- Es werden zwei neue `addiere()` - Überladungen mit wenig sinnvollen Parameterlisten definiert:

```
public void addiere(long z, int n) {
   if (n == 0) return;
   zaehler = zaehler*n + z*nenner;
   nenner = nenner*n;
}
public void addiere(int z, long n) {
   if (n == 0) return;
   zaehler = zaehler*n + z*nenner;
   nenner = nenner*n;
}
```

Aufgrund dieser „Vorarbeiten“ enthält das folgende Programm

```
class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b = new Bruch();
      b.setzeZaehler(1);
      b.setzeNenner(2);
      b.addiere(3, 4);
      b.zeige();
   }
}
```

im Aufruf

```
b.addiere(3, 4);
```

eine Mehrdeutigkeit, weil keine `addiere()` - Überladung perfekt passt, und für zwei Überladungen gleich viele erweiternde Typanpassungen (vgl. Abschnitt 3.5.7) erforderlich sind. Der OpenJDK 8 - Compiler äußert sich so:

```
Bruchrechnung.java:6: error: reference to addiere is ambiguous
                b.addiere(3, 4);
                 ^
  both method addiere(long,int) in Bruch and method addiere(int,long) in Bruch match
1 error
```

Bei einem sinnvollen Entwurf von überladenen Methoden treten solche Mehrdeutigkeiten nur sehr selten auf.

Von einer Methode unterschiedlich parametrisierte Varianten in eine Klassendefinition aufzunehmen, lohnt sich z. B. in den folgenden Situationen:

- Für **verschiedene Datentypen** werden analog arbeitende Methoden benötigt. So besitzt z. B. die Klasse **Math** im Paket **java.lang** folgende Methoden, um den Betrag einer Zahl zu ermitteln:
  public static double abs(double *value*)
  public static float abs(float *value*)
  public static int abs(int *value*)
  public static long abs(long *value*)
  Seit der Java - Version 5 bieten *generische Methoden* (siehe Abschnitt 8.2) eine elegantere Lösung für die Unterstützung verschiedener Datentypen. Allerdings führt die generische Lösung bei primitiven Datentypen zu einem deutlich höheren Zeitaufwand für die Methodenausführung, sodass hier die Überladungstechnik weiterhin sinnvoll sein kann.
- Für eine Methode sollen **unterschiedliche umfangreiche Parameterlisten** angeboten werden, sodass zwischen einer bequem aufrufbaren Standardausführung (mit möglichst kurzer oder leerer Parameterliste) und einer individuell gestalteten Ausführungsvariante gewählt werden kann. So beherrscht z. B. die Klasse **String** zwei Instanzmethoden namens **substring()**, die eine Teilzeichenfolge als neues **String**-Objekt liefern. Während die erste Überladung nur einen Parameter für den Startindex der Teilzeichenfolge besitzt, verfügt die zweite Überladung über einen zusätzlichen Parameter für den Endindex:
  public String substring(int *beginIndex*)
  public String substring(int *beginIndex*, **int** *endIndex*)

4.4 Objekte

Im aktuellen Abschnitt geht es darum, wie Objekte erzeugt, genutzt und im obsoleten Zustand wieder aus dem Speicher entfernt werden.

4.4.1 Referenzvariablen deklarieren

Um irgendein Objekt aus der Klasse `Bruch` ansprechen zu können, benötigen wir eine **Referenzvariable** mit dem Datentyp `Bruch`. In der folgenden Anweisung wird eine solche Referenzvariable definiert und auch gleich initialisiert:

```
Bruch b = new Bruch();
```

Um die Wirkungsweise dieser Anweisung Schritt für Schritt zu untersuchen, beginnen wir mit einer einfacheren Variante *ohne* Initialisierung:

```
Bruch b;
```

Hier wird die Referenzvariable `b` mit dem Datentyp `Bruch` deklariert, der man folgende Werte zuweisen kann:

- die Adresse eines `Bruch`-Objekts
  In der Variablen wird kein komplettes `Bruch`-Objekt mit sämtlichen Instanzvariablen abgelegt, sondern ein **Verweis** (eine **Referenz**) auf einen Ort im Heap-Bereich des programmeigenen Speichers, an dem sich ein `Bruch`-Objekt befindet.[1]

[1] Sollte einmal eine Ableitung (Spezialisierung) der Klasse `Bruch` definiert werden, können deren Objekte ebenfalls über `Bruch`-Referenzvariablen verwaltet werden. Vom Vererbungsprinzip der objektorientierten Programmierung haben Sie schon einiges gehört, doch steht die gründliche Behandlung noch aus.

- **null**
  Dieses Referenzliteral steht für einen leeren Verweis. Eine Referenzvariable mit diesem Wert ist nicht undefiniert, sondern zeigt explizit auf nichts (vgl. Abschnitt 3.3.11.6).

Wir nehmen nunmehr offiziell und endgültig zur Kenntnis, dass Klassen als Datentypen verwendet werden können und haben damit bislang in Java-Programmen folgende Datentypen zur Verfügung:

- **Primitive Typen** (**boolean**, **char**, **byte**, **double**, ...)
- **Klassentypen**
  Es kommen Klassen aus dem Java-API, aus anderen Bibliotheken und selbst definierte Klassen in Frage. Ist eine Variable vom Typ einer Klasse, kann sie die Adresse eines Objekts aus dieser Klasse oder aus einer daraus abgeleiteten Klasse aufnehmen. Außerdem kann jede Referenzvariable den Wert **null** annehmen.

4.4.2 Objekte erzeugen

Damit z. B. der folgendermaßen deklarierten Referenzvariablen `b` vom Datentyp `Bruch`

```
Bruch b;
```

ein Verweis auf ein `Bruch`-Objekt zugewiesen werden kann, muss ein solches Objekt erst erzeugt werden, was per **new**-Operator geschieht, z. B. im folgenden Ausdruck:

```
new Bruch()
```

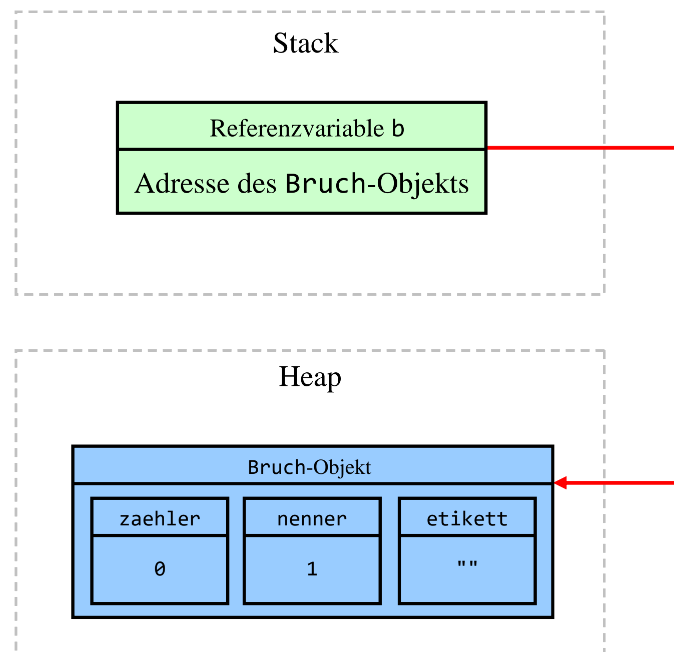
Als Operanden erwartet der **new**-Operator einen Klassennamen, dem eine Parameterliste zu folgen hat, weil der **new**-Operand als Name eines *Konstruktors* (siehe Abschnitt 4.4.3) fungiert. Als Wert des Ausdrucks resultiert eine Referenz (Speicheradresse), die (im Rahmen bestehender Rechte) einen Zugriff auf das neue Objekt (seine Instanzvariablen und -methoden) ermöglicht.

In der **main()** - Methode der folgenden Startklasse

```
class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b = new Bruch();
       . . .
   }
}
```

wird die vom **new**-Operator gelieferte Adresse in die lokale Referenzvariable `b` geschrieben. Es resultiert die folgende Situation im programmeigenen Arbeitsspeicher:[1]

[1] Hier wird aus didaktischen Gründen ein wenig gemogelt. Die Instanzvariable `etikett` ist vom Typ der Klasse **String**, zeigt also auf ein **String**-Objekt, das „neben" dem `Bruch`-Objekt auf dem Heap liegt. In der `Bruch`-Referenz-Instanzvariablen `etikett` befindet sich die Adresse des **String**-Objekts.



Während lokale Variablen (während der Methodenausführung) im **Stack**-Bereich des programmeigenen Arbeitsspeichers abgelegt werden, entstehen Objekte mit ihren Instanzvariablen auf dem **Heap**.

In einem Programm können *mehrere* Referenzvariablen auf *dasselbe* Objekt zeigen, z. B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b1 = new Bruch();
      b1.setzeZaehler(1);
      b1.setzeNenner(3);
      b1.setzeEtikett("b1 = ");
      Bruch b2 = b1;
      b2.zeige();
   }
}
```

</td><td>

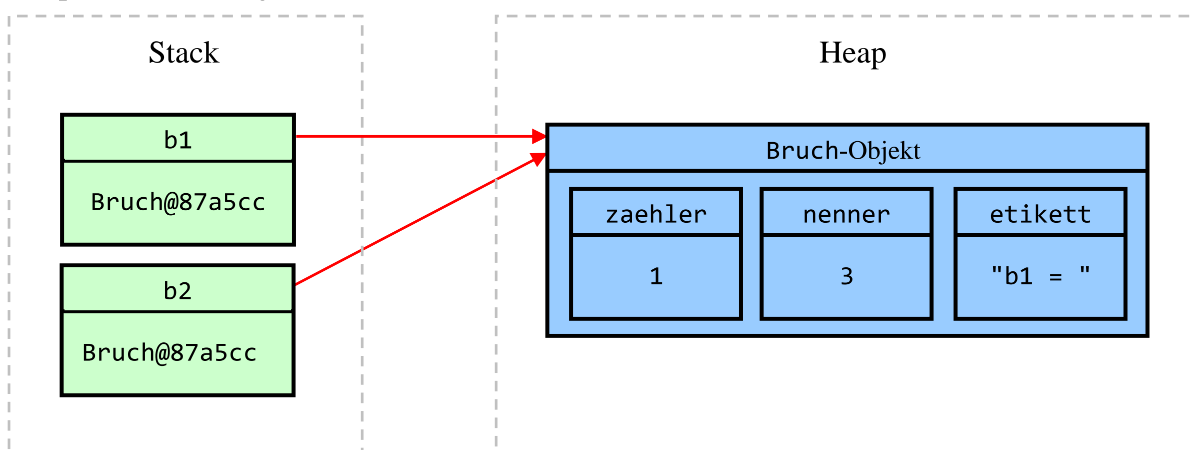
```
        1
 b1 =  -----
        3
```

</td></tr>
</table>

In der Anweisung

```
Bruch b2 = b1;
```

wird die neue Referenzvariable `b2` vom Typ `Bruch` angelegt und mit dem Inhalt von `b1` (also mit der Adresse des bereits vorhandenen `Bruch`-Objekts) initialisiert. Es resultiert die folgende Situation im Speicher des Programms:



Hier sollte nur die Möglichkeit der Mehrfachreferenzierung demonstriert werden. Bei einer ernsthaften Anwendung des Prinzips befinden sich die alternativen Referenzen an verschiedenen Stellen des Programms, z. B. in Instanzvariablen verschiedener Objekte. In einem Speditionsverwaltungsprogramm kennen z. B. alle Objekte zu einzelnen Fahrzeugen die Adresse des Planerobjekts, dem sie besondere Ereignisse wie Pannen melden.

Eventuell empfinden manche Leser den doppelten Auftritt des Klassennamens bei einer Referenzvariablendeklaration mit Initialisierung als störend redundant, z. B.:

```
Bruch b = new Bruch();
```

Hier sind aber zwei Sprachbestandteile (Variablendeklaration und Objektkreation) involviert, die beide den Klassennamen enthalten:

- In der Variablendeklaration wird der Datentyp angegeben.
- Wenn der **new**-Operator ein Objekt von bestimmtem Typ kreieren soll, kommt man um die Nennung des Klassennamens nicht herum. Es ist aber keinesfalls immer so, dass im **new**-Operanden als Klasse der deklarierte Datentyp Verwendung findet.

Bei der Referenzvariablendeklaration mit Initialisierung stehen beide Sprachbestandteile unmittelbar hintereinander, sodass bei oberflächlicher Betrachtung der Eindruck von Redundanz entsteht, wenn (wie in unseren einfachen Beispielen) der deklarierte Datentyp und die Klasse im **new**-Operanden identisch sind.

Wie Sie aus Abschnitt 3.3.8 wissen, kann seit Java 10 bei der Deklaration einer *lokalen Variablen* mit Initialisierung über das Schlüsselwort **var** dank Typinferenz etwas Schreibaufwand gespart und die doppelte Nennung des Klassennamens vermieden werden, z. B.:

```
var b = new Bruch();
```

Um Wartungsfreundlichkeit und Polymorphie zu ermöglichen, sind auch Basisklassen, abstrakte Klassen und Schnittstellen als Datentypen erlaubt und sinnvoll. Nutzt man solche Datentypen, dann stimmen bei der Referenzvariablendeklaration mit Initialisierung der deklarierte Datentyp und der Klassenname im **new**-Operanden *nicht* überein.

4.4.3 Konstruktoren

In diesem Abschnitt werden spezielle Methoden behandelt, die beim Erzeugen von neuen Objekten ausgeführt werden, um deren Instanzvariablen zu initialisieren und/oder andere Arbeiten zu verrichten (z. B. Öffnen einer Datei oder Netzwerkverbindung). Ziel der Konstruktor-Tätigkeit ist ein neues Objekt in einem validen Zustand, das für seinen Einsatz gut vorbereitet ist.[1] Wie Sie bereits wissen, wird zum Erzeugen von Objekten der **new**-Operator verwendet. Als Operand ist ein Konstruktor der gewünschten Klasse zu übergeben.

Hat der Programmierer zu einer Klasse *keinen* Konstruktor definiert, erhält diese Klasse automatisch einen **Standardkonstruktor**. Weil dieser Konstruktor keine Parameter besitzt, ergibt sich sein Aufruf aus dem Klassennamen durch Anhängen einer leeren Parameterliste, z. B.:

```
Bruch b = new Bruch();
```

Der Standardkonstruktor ruft den parameterfreien Konstruktor *der Basisklasse* auf und führt die Initialisierungen für Instanzvariablen aus, die bei der Deklaration oder in einem Instanzinitialisierer

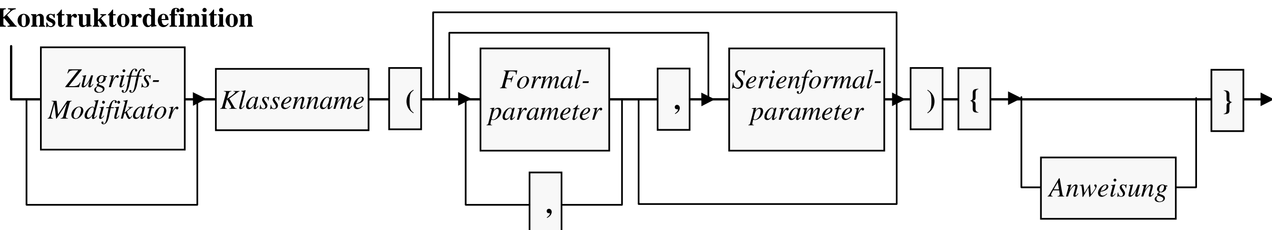
[1] Man ist geneigt, der Klasse eine aktive Rolle beim Erzeugen eines neuen Objekts zuzuschreiben. Allerdings lassen sich in einem Konstruktor die Instanz-Member des neuen Objekts genauso verwenden wie in einer Instanzmethode, was (wie die Abwesenheit des Modifikators **static**, vgl. Abschnitt 4.5.3) den Konstruktor in die Nähe einer Instanzmethode rückt. Laut Sprachbeschreibung zu Java 13 ist ein Konstruktor allerdings überhaupt kein Member, also weder Instanz- noch Klassenmethode (Gosling et al. 2019, Abschnitt 8.2). Für die Praxis der Programmierung ist es irrelevant, welchem Akteur man die Ausführung des Konstruktors zuschreibt.

(siehe Abschnitt 4.4.4) vorgenommen werden. Er hat dieselbe Schutzstufe wie die Klasse, sodass z. B. beim Standardkonstruktor der Klasse `Bruch` die Schutzstufe **public** resultiert.

In der Regel ist es beim Klassendesign sinnvoll, Konstruktoren *explizit* zu definieren, um das individuelle Initialisieren der Instanzvariablen von neuen Objekten zu ermöglichen. Dabei sind folgende Regeln zu beachten:

- Ein Konstruktor trägt denselben Namen wie die Klasse.
- In der Definition wird *kein* Rückgabetyp angegeben.
- Wie bei einer gewöhnlichen Methodendefinition ist eine Parameterliste anzugeben, ggf. eine leere. Parameter erlauben das individuelle Initialisieren der Instanzvariablen von neuen Objekten.
- Sobald man einen expliziten Konstruktor definiert hat, steht der Standardkonstruktor *nicht* mehr zur Verfügung. Ist weiterhin ein parameterfreier Konstruktor erwünscht, so muss dieser *zusätzlich* explizit definiert werden.
- Als Modifikatoren sind nur solche erlaubt, welche die Sichtbarkeit des Konstruktors (den Zugriffsschutz) regeln (z. B. **public**, **private**), sodass pro Konstruktor maximal *ein* Modifikator verwendet werden kann.
- Während der Standardkonstruktor die Schutzstufe der Klasse übernimmt, gelten für explizit definierte Konstruktoren beim Zugriffsschutz dieselben Regeln wie für andere Methoden. Per Voreinstellung sind sie also in allen Klassen *desselben Pakets* nutzbar. Mit der deklarierten Schutzstufe **private** kann man verhindern, dass ein Konstruktor von fremden Klassen benutzt wird.[1]
- Eine Klasse erbt die Konstruktoren ihrer Basisklasse *nicht*. Allerdings wird bei jeder Objektkreation ein Basisklassenkonstruktor aufgerufen. Wenn dies nicht explizit über das Schlüsselwort **super** als Bezeichnung für einen Basisklassenkonstruktor geschieht, wird der parameterfreie Basisklassenkonstruktor automatisch aufgerufen. Mit Fragen zur Objektkreation, die im Zusammenhang mit der Vererbung stehen, werden wir uns im Abschnitt 7.4 beschäftigen.
- In einer Klasse sind beliebig viele Konstruktoren möglich, die alle denselben Namen und jeweils eine individuelle Parameterliste haben müssen. Das Überladen (vgl. Abschnitt 4.3.4) ist also auch bei Konstruktoren erlaubt.

Das Syntaxdiagramm zur Definition eines Konstruktors wirkt wegen der Flexibilität beim Aufbau der Parameterliste etwas kompliziert:



Hinsichtlich der (Serien-)Formalparameter bestehen keine Unterscheide zwischen einem Konstruktor und einer Methode (siehe Abschnitt 4.3.1).

[1] Gelegentlich ist es sinnvoll, *alle* Konstruktoren durch den Modifikator **private** für die Nutzung durch fremde Klassen zu sperren. Dies hat allerdings zur Folge, dass keine abgeleitete Klasse definiert werden kann (siehe Abschnitt 7.4).

Bei den öffentlich zugänglichen Konstruktoren ist darauf zu achten, dass die Datenkapselung nicht ausgehebelt wird, indem Instanzvariablen auf beliebige Werte gesetzt und somit defekte Objekte erzeugt werden können.

Die folgende Variante unserer Klasse `Bruch` enthält einen expliziten Konstruktor mit Parametern zur Initialisierung aller Instanzvariablen und einen zusätzlichen, parameterfreien Konstruktor mit leerem Anweisungsteil. Beide sind aufgrund der Schutzstufe **public** allgemein verwendbar:

```
public class Bruch {
   private int zaehler;
   private int nenner = 1;
   private String etikett = "";

   public Bruch(int z, int n, String eti) {
      setzeZaehler(z);
      setzeNenner(n);
      setzeEtikett(eti);
   }

   public Bruch() {}

    . . .
}
```

Weil im parametrisierten Konstruktor die „beantragten“ Initialisierungswerte *nicht* direkt den Feldern zugewiesen, sondern durch die Zugriffsmethoden geschleust werden, bleibt die Datenkapselung erhalten. Wie jede andere Methode einer Klasse muss auch ein Konstruktor so entworfen sein, dass die Objekte der Klasse unter allen Umständen konsistent und funktionstüchtig sind. In der Klassendokumentation sollte darauf hingewiesen werden, dass dem Wunsch, den Nenner eines neuen `Bruch`-Objekts per Konstruktor auf den Wert 0 zu setzen, *nicht* entsprochen wird, und dass stattdessen der Wert 1 resultiert.[1]

Im folgenden Testprogramm werden beide Konstruktoren eingesetzt:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b1 = new Bruch(1, 2, "b1 = ");
      Bruch b2 = new Bruch();
      b1.zeige();
      b2.zeige();
   }
}</pre></td><td><pre>   1
b1 = -----
        2

   0

   1</pre></td></tr>
</table>

Konstruktoren können nicht direkt aufgerufen, sondern nur per **new**-Operator genutzt werden. Als Ausnahme von dieser Regel ist es allerdings möglich, im Anweisungsblock eines Konstruktors einen anderen Konstruktor derselben Klasse über das Schlüsselwort **this** aufrufen, z. B.:

```
public Bruch() {
   this(0, 1, "unbenannt");
}
```

[1] Eine sinnvolle Reaktion auf den Versuch, ein defektes Objekt zu erstellen, kann darin bestehen, im Konstruktor eine sogenannte *Ausnahme* zu werfen und dadurch den Aufrufer über das Scheitern seiner Absicht zu informieren. Das tut z. B. der folgende Konstruktor der API-Klasse **FileOutputStream** im Paket **java.io**:

public FileOutputStream (File *file***) throws FileNotFoundException**

Mit der Kommunikation über Ausnahmeobjekte werden wir uns im Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.** beschäftigen.

Auf diese Weise lässt es sich vermeiden, dass eine Konstruktorüberladung zu Wiederholungen im Quellcode führt.

Wird wie bei der folgenden minimalistischen Klasse mit immerhin zwei Konstruktoren

```
class Prog {
    int ivar = 4711;
    Prog() {}
    Prog(int ip) {ivar = ip;}
}
```

eine Instanzvariable im Rahmen der Deklaration initialisiert, landen die zugehörigen Bytecode-Anweisungen am Anhang jedes Konstruktors. Um diese Aussage zu verifizieren, verwenden wir das JDK-Werkzeug **javap**, das u.a. den Bytecode zu einer Klasse auflisten kann. Der Aufruf

```
>javap -c Prog
```

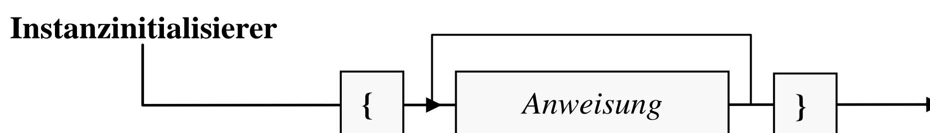
führt im Beispiel zum Ergebnis (siehe Anweisung **sipusch**):

```
Compiled from "Prog.java"
class Prog {
  int ivar;

  Prog();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: sipush        4711
       8: putfield      #2                  // Field ivar:I
      11: return

  Prog(int);
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: sipush        4711
       8: putfield      #2                  // Field ivar:I
      11: aload_0
      12: iload_1
      13: putfield      #2                  // Field ivar:I
      16: return
}
```

Analog verfährt der Compiler auch mit einem Instanzinitialisierer, der anschließend vorgestellt wird.

4.4.4 Instanzinitialisierer

Zur Initialisierung von Instanzvariablen kann in eine Klassendefinition eine Blockanweisung an jeder Position eingefügt werden, an der eine Felddeklaration oder eine Methodendefinition erlaubt ist. Es sind sogar beliebig viele Instanzinitialisierer erlaubt. Der Compiler fügt den Code aller Instanzinitialisierer am Anfang *jedes* Konstruktors ein. Dies geschieht in der Auftretensreihenfolge der Initialisierer, unmittelbar hinter dem Code aufgrund von initialisierenden Felddeklarationen.



Bei gewöhnlichen Klassen werden Instanzinitialisierer nur selten verwendet, weil sich Objektinitialisierungen sehr übersichtlich mit Konstruktoren erledigen lassen. Wird eine Initialisierung in mehreren Konstruktoren benötigt, kann man sie einem elementaren Konstruktor vornehmen, der von anderen Konstruktoren aufgerufen wird (siehe Abschnitt 4.4.3). In den später vorzustellenden anonymen Klassen (siehe Abschnitt 12.1.1.2) werden Instanzinitialisierer aber gelegentlich benötigt, weil dort mangels Klassenname keine Konstruktoren definiert werden können.

Um jetzt schon ein Beispiel für einen Instanzinitialisierer präsentieren zu können, statten wir eine normale Klasse damit aus. In der folgenden Klasse wird mit Hilfe der statischen Methode `gint()` aus unserer Bequemlichkeitsklasse `Simput` (vgl. Abschnitt 3.4) der Wert einer Instanzvariablen beim Benutzer erfragt:

```
class Prog {
    private int alter;

    {
        System.out.print("Ihr Alter: ");
        alter = Simput.gint();
    }

    public static void main(String[] args) {
        Prog p = new Prog();
        System.out.println(p.alter);
    }
}
```

Der `gint()` - Aufruf ist auch in einer einfachen Variablendeklaration mit Initialisierung möglich:

```
private int alter = Simput.gint();
```

In einem Instanzinitialisierer ist aber (wie in einem Konstruktor) eine Anweisungssequenz erlaubt, was im Beispiel zur Ausgabe einer Instruktion genutzt wird. Außerdem kann man hier Ausnahmen abfangen und werfen, um Laufzeitfehler zu beheben oder zu melden (siehe Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.**).

4.4.5 Objekte aus der Fabrik

Manche Klassen bieten statische Methoden zum Erzeugen von neuen Objekten des eigenen Typs an, z. B. die Klasse **Box** im Paket **javax.swing**:

```
Box box = Box.createHorizontalBox();
```

Man spricht hier von *Fabrikmethoden* (engl.: *factory methods*). Bloch (2018, S. 5ff) plädiert sogar nachdrücklich dafür, beim Design einer Klasse Fabrikmethoden gegenüber Konstruktoren zu bevorzugen und nennt u.a. die folgenden Vorteile von Fabrikmethoden:

- Während ein Konstruktor stets ein *neues* Objekt erzeugt, kann eine Fabrikmethode sich dafür entscheiden, Aufwand sparend eine Referenz auf ein bereits vorhandenes und funktionsgleiches Objekt zu liefern.
- Eine Fabrikmethode kann situationsgerecht ein Objekt aus einer abgeleiteten Klasse liefern.

Im Fall der Klasse **Box** ist ein öffentlicher Konstruktor verfügbar, sodass man das Ergebnis der vorigen Anweisung auch so realisieren kann:

```
Box box = new Box(BoxLayout.X_AXIS);
```

Ein Klassendesigner hat aber auch die Option, Fabrikmethoden anzubieten und auf öffentlich zugängliche Konstruktoren zu verzichten.

Im Anweisungsteil einer Fabrikmethode wird natürlich ein Objektkreationsausdruck mit **new**-Operator und Konstruktor benötigt, z. B.:

```
public static Box createHorizontalBox() {
    return new Box(BoxLayout.X_AXIS);
}
```

Zu einer Klasse ohne öffentlichen Konstruktor kann allerdings keine abgeleitete Klasse definiert werden (siehe Abschnitt 7.4).

4.4.6 Abräumen überflüssiger Objekte durch den Garbage Collector

Stellt die Laufzeitumgebung einen Speichermangel fest, tritt der **Garbage Collector** (Müllsammler) in Aktion und löscht Objekte vom Heap-Speicher, die nutzlos geworden sind, weil im Programm keine Referenz mehr auf diese Objekte zeigt.

In unseren bisherigen Bruchrechnungs-Beispielprogrammen entsteht jedes `Bruch`-Objekt in der **main()** - Methode der Startklasse. Beim Verlassen dieser Methode verschwindet die einzige Referenz auf das Objekt, und es ist reif für den Garbage Collector. Der muss sich aber keine Mühe geben, weil das Programm mit dem Ablauf der **main()** - Methode ohnehin endet.

In einem größeren Programm sind die Objektlebensläufe weniger simpel und einheitlich:

- Es ist möglich (und normal), dass ein Objekt die erzeugende Methode überlebt, weil eine Referenz nach Außen transportiert worden ist (z. B. per Rückgabewert, vgl. Abschnitt 4.4.8).
- Man kann ein Objekt während einer Methodenausführung aufgeben (dem Garbage Collector überlassen), indem man alle Referenzen auf das Objekt aktiv entfernt. Um eine Referenz aufzulösen, setzt man die betroffene Referenzvariable entweder auf den Wert **null** oder weist ihr eine andere Referenz zu, z. B.:
  ```
  b1 = null;
  ```
  Die explizite Beseitigung von Referenzen ist allerdings mit Aufwand verbunden und nur unter speziellen Umständen empfehlenswert (siehe Bloch 2018, S. 26ff).

Vermutlich sind Programmiereinsteiger vom Garbage Collector nicht sonderlich beeindruckt. Schließlich war im Manuskript noch nie davon die Rede, dass man sich um den belegten Speicher nach Gebrauch kümmern müsse. Der in einer Methode von lokalen Variablen belegte Speicher wird bei *jeder* Programmiersprache freigegeben, sobald die Ausführung der Methode beendet ist. Demgegenüber muss der von überflüssig gewordenen *Objekten* belegte Speicher bei älteren Programmiersprachen (z. B. C++) nach Gebrauch explizit wieder frei gegeben werden. In Anbetracht der Objektmengen, die ein typisches Programm (z. B. ein Grafikeditor) benötigt, ist einiger Aufwand erforderlich, um eine Verschwendung von Speicherplatz zu verhindern. Mit seinem vollautomatischen Garbage Collector vermeidet Java lästigen Aufwand und zwei kritische Fehlerquellen:

- Weil der Programmierer keine Verpflichtung (und Berechtigung) zum Entsorgen von Objekte hat, kann es nicht zu Programmabstürzen durch Zugriff auf voreilig zerstörte Objekte kommen.
- Es entstehen keine **Speicherlöcher** (engl.: *memory leaks*) durch versäumte Speicherfreigaben bei überflüssig gewordenen Objekten.

Der Garbage Collector wird im Normalfall nur dann tätig, wenn die virtuelle Maschine Speicher benötigt, sodass der genaue Zeitpunkt für die Entsorgung eines Objekts kaum vorhersehbar ist.

Bei großen Anwendungen können die im Hintergrund ablaufenden GC-Aktivitäten zu spürbaren temporären Leistungseinbußen führen, sodass Programmierer über Maßnahmen zur GC-Optimierung nachdenken müssen. Auf der folgenden Webseite finden sich diesbezügliche Erläuterungen und Tipps der Firma Oracle:

https://docs.oracle.com/javase/9/gctuning/introduction-garbage-collection-tuning.htm

Allerdings ist nicht für jeden „Hänger“ der GC verantwortlich. Als alternative Ursachen kommen z. B. in Frage: Langsame Zugriffe auf externe Ressourcen (Netzwerk, Datenbankserver), gegenseitige Behinderung von Threads, systemseitiger Speichermangel mit der Notwendigkeit zu Festplattenzugriffen.

4.4.7 finalize() und Cleaner

Im Zusammenhang mit obsolet gewordenen Objekten ist die (um Garbage Collector automatische erledigte) Freigabe von Speicher nicht die einzige Aufgabe. Viele Klassen bzw. Objekte verwenden knappe Ressourcen (z. B. Datei-, Netzwerk oder Datenbankverbindungen, Threads), die möglichst schnell zurückgegeben werden sollten, um eine Behinderung anderer Interessenten zu vermeiden. In diesen Fällen sollte eine Methode namens **close()** implementiert werden, die bei jedem obsolet gewordenen Objekt aufzurufen ist, um die belegten Ressourcen freizugeben (siehe z. B. Abschnitt 11.8).

Java bietet seit der Version 1 dem Klassendesigner die Möglichkeit, die beim Ableben eines Objekts erforderlichen Aufräumarbeiten in einer Methode namens **finalize()** unterzubringen, die ggf. vom Garbage Collector aufgerufen wird. Der potentielle Nutzen dieser Methode besteht darin, dass die Aufräumarbeiten nicht vom expliziten Aufruf der Methode **close()** abhängen, den ein nachlässiger Programmierer möglicherweise vergisst. Allerdings haben gravierende Probleme dazu geführt, dass die Methode **finalize()** seit Java 9 als abgewertet (engl.: *deprecated*) deklariert ist:[1]

- Es ist nicht vorhersehbar, ob und wann **finalize()** aufgerufen wird, sodass per **finalize()** nicht für die korrekte Funktion einer Klasse gesorgt, sondern lediglich die Wahrscheinlichkeit für einen Schaden durch die fehlerhafte Anwendung der Klasse reduziert werden kann.
- Durch die Existenz der Methode **finalize()** steigt der Zeitaufwand bei der Erstellung und bei der Entsorgung eines Objekts drastisch an.
- Ein in **finalize()** auftretender Ausnahmefehler bleibt unentdeckt.

Weil die Methode **finalize()** in der Standardbibliothek noch weit verbreitet ist, soll das Einsatzmuster mit der folgenden, vorübergehend in die Klasse `Bruch` aufgenommenen Methodendefinition demonstriert werden:

```
protected void finalize() throws Throwable {
   super.finalize();
   System.out.println(this + " finalisiert");
}
```

Hier tauchen einige Bestandteile auf, die bald ausführlich zur Sprache kommen und hier ohne großes Grübeln hingenommen werden sollten:

- `protected`
  In der Klasse **Object** ist für **finalize()** die Schutzstufe **protected** festgelegt (abgeleitete Klassen sind zugriffsberechtigt, siehe unten), und dieser Zugriffsschutz darf beim Überschreiben der Methode in einer (implizit von **Object**) abgeleiteten Klasse nicht verschärft werden. Die ohne Angabe eines Modifikators eingestellte Schutzstufe *Paket* enthält gegenüber **protected** eine Einschränkung und ist damit hier unzulässig.

---

[1] Zitat aus der API-Dokumentation zu Java 9 (https://docs.oracle.com/javase/9/docs/api/java/lang/Object.html):
The finalization mechanism is inherently problematic. Finalization can lead to performance issues, deadlocks, and hangs.
Unsere Entwicklungsumgebung IntelliJ stellt abgewertete Methoden mit durchgestrichenem Namen dar.

- `throws Throwable`
  Die **finalize()** - Methode der Klasse **Object** löst ggf. eine Ausnahme aus der Klasse **Throwable** aus (siehe Kapitel 11). Diese muss von der eigenen **finalize()** - Implementierung beim Aufruf der Basisklassenvariante entweder abgefangen oder weitergereicht werden, was durch den Zusatz **throws Throwable** im Methodenkopf anzumelden ist.
- `super.finalize();`
  Bereits die Urahnklasse **Object** aus dem Paket **java.lang**, von der alle Java-Klassen abstammen, verfügt über eine **finalize()** - Methode. Überschreibt man in einer abgeleiteten Klasse die **finalize()** - Methode der Basisklasse, dann sollte am Anfang der eigenen Implementation die überschriebene Variante aufgerufen werden, wobei das Schlüsselwort **super** die Basisklasse anspricht.
- `this`
  In der aus didaktischen Gründen eingefügten Kontrollausgabe wird mit dem Schlüsselwort **this** (vgl. Abschnitt 4.4.8.2) das aktuell handelnde Objekt angesprochen. Bei der automatischen Konvertierung der Referenz in eine Zeichenfolge wird die vom Laufzeitsystem verwaltete Objektbezeichnung zu Tage gefördert.

Um die baldige Freigabe von externen Ressourcen (z. B. Datenbank- oder Netzwerkverbindung) zu erreichen, sollte man sich *nicht* auf die Methode **finalize()** verlassen, weil sie nur dann vom Garbage Collector aufgerufen wird, wenn ein *Speichermangel* auftritt. Durch einen Aufruf der statischen Methode **gc()** aus der Klasse **System** kann man den sofortigen Einsatz des Müllsammlers *vorschlagen*, z. B. vor einer Aktion mit großem Speicherbedarf:

```
System.gc();
```

Allerdings ist nicht sicher, ob der Garbage Collector tatsächlich tätig wird. Außerdem ist nicht vorhersehbar, in welcher Reihenfolge die obsoleten Objekte entfernt werden.

Im folgenden Beispielprogramm werden zwei `Bruch`-Objekte erzeugt und nach einer Ausgabe ihrer Identifikation durch das Entfernen der Referenzen wieder aufgegeben:

```
class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b1 = new Bruch();
      Bruch b2 = new Bruch();
      System.out.println("b1: " + b1 + ", b2: " + b2 + "\n");
      b1 = b2 = null;
      System.gc();
   }
}
```

Ob anschließend der Garbage Collector aufgrund der expliziten Aufforderung `System.gc()` tatsächlich tätig wird, ist an den Kontrollausgaben der `finalize()` - Methode (siehe oben) zu erkennen. Bei Tests mit verschiedenen Versionen des Java-Compilers (**Project language level** im IntelliJ-Dialog **Project Structure > Project**) traten die folgenden Varianten auf:

1) Die Objekte werden finalisiert in der Reihenfolge `b1`, `b2` (z. B. zu beobachten bei Java 8):
```
b1: Bruch@21a722ef, b2: Bruch@63e68a2b

Bruch@21a722ef finalisiert
Bruch@63e68a2b finalisiert
```
2) Die Objekte werden finalisiert in der Reihenfolge `b2`, `b1` (z. B. zu beobachten bei Java 8):
```
b1: Bruch@60f32dde, b2: Bruch@7d487b8b

Bruch@7d487b8b finalisiert
Bruch@60f32dde finalisiert
```

3) Die Objekte werden *nicht* finalisiert (zu beobachten ab Java 9):
```
b1: Bruch@4fca772d, b2: Bruch@9807454
```

Seit Java 9 ist die Klasse **Cleaner** vorhanden, um als Ersatz für die abgewertete Methode **finalize()** die Wahrscheinlichkeit für einen Schaden durch einen unterlassenen **close()** - Aufruf zu reduzieren.[1] Hinsichtlich der Unbestimmtheit des Aufrufs besteht keine Verbesserung gegenüber **finalize()**. Allerdings verursacht **Cleaner** weniger Kosten, und beim Einsatz auftretende Ausnahmefehler können abgefangen werden.

Nach Bloch (2018, S. 29) sind weder die Methode **finalize()** noch die Klasse **Cleaner** zu empfehlen:

> Finalizers are unpredictable, often dangerous, and generally unnecessary. ...
> Cleaners are less dangerous than finalizers, but still unpredictable, slow, and generally unnecessary.

4.4.8 Objektreferenzen verwenden

Methodenparameter mit Referenztyp wurden schon im Abschnitt 4.3.1.4.2 behandelt. In diesem Abschnitt geht es um Methodenrückgabewerte mit Referenztyp und um das Schlüsselwort **this**, mit dem sich in einer Methode das aktuell handelnde Objekt ansprechen lässt.

4.4.8.1 Rückgabe mit Referenztyp

Soll ein methodenintern erzeugtes Objekt das Ende der Methodenausführung überleben, muss eine Referenz außerhalb der Methode geschaffen werden, was z. B. über einen Rückgabewert mit Referenztyp geschehen kann.

Als Beispiel erweitern wir die `Bruch`-Klasse um die Methode `klone()`, welche ein Objekt beauftragt, einen neuen Bruch anzulegen, mit den Werten der eigenen Instanzvariablen zu initialisieren und die Referenz an den Aufrufer zu übergeben:

```
public Bruch klone() {
   return new Bruch(zaehler, nenner, etikett);
}
```

Im folgenden Programm wird das durch `b2` referenzierte `Bruch`-Objekt in der von `b1` ausgeführten Methode `klone()` erzeugt. Es ist ansprechbar und dienstbereit, nachdem der erzeugende Methodenaufruf längst der Vergangenheit angehört:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b1 = new Bruch(1, 2, "b1 = ");
      b1.zeige();
      Bruch b2 = b1.klone();
      b2.zeige();
   }
}
```

</td><td>

```
        1
b1 =  -----
        2

        1
b1 =  -----
        2
```

</td></tr>
</table>

[1] https://docs.oracle.com/javase/9/docs/api/java/lang/ref/Cleaner.html

4.4.8.2 *this als Referenz auf das aktuelle Objekt*

Gelegentlich ist es sinnvoll oder erforderlich, dass ein handelndes Objekt sich selbst ansprechen und z. B. seine Adresse bei einem Methodenaufruf als Aktualparameter verwenden kann. Das ist mit dem Schlüsselwort **this** möglich, das innerhalb einer Instanzmethode wie eine Referenzvariable funktioniert. Im folgenden Beispiel ermöglicht die **this**-Referenz den Zugriff auf Instanzvariablen, die von namensgleichen Formalparametern überdeckt werden:

```
public boolean addiere(int zaehler, int nenner, boolean autokurz) {
   if (nenner != 0) {
      this.zaehler = this.zaehler * nenner + zaehler * this.nenner;
      this.nenner = this.nenner * nenner;
      if (autokurz)
         this.kuerze();
      return true;
   } else
      return false;
}
```

Außerdem wird beim `kuerze()` - Aufruf durch die (nicht erforderliche) **this**-Referenz verdeutlicht, dass die Methode vom aktuell handelnden Objekt ausgeführt wird. Später werden Sie noch weit relevantere **this**-Verwendungen kennenlernen.

4.5 Klassenvariablen und -methoden

Neben den *Instanz*variablen und -methoden unterstützt Java auch *klassenbezogene* Varianten. Syntaktisch werden diese Mitglieder in der Deklaration bzw. Definition durch den Modifikator **static** gekennzeichnet, und man spricht oft von *statischen* Feldern bzw. Methoden. Ansonsten gibt es bei der der Deklaration bzw. Definition kaum Unterschiede zwischen einem Instanz-Member und dem analogen statischen Member.

Bei den statischen Membern gilt (wie bei Instanz-Membern) für den Zugriffsschutz:

- Per Voreinstellung ist der Zugriff allen Klassen im selben Paket erlaubt.
- Mit einem Modifikator lassen sich alternative Schutzstufen wählen, z. B.:
  - **private**
    Alle fremden Klassen werden ausgeschlossen.
  - **public**
    Alle Klassen in berechtigten Modulen dürfen zugreifen.[1]

4.5.1 Klassenvariablen

In unserem Bruchrechnungsbeispiel soll ein statisches Feld dazu dienen, die Anzahl der bei einem Programmeinsatz bisher erzeugten `Bruch`-Objekte aufzunehmen:

```
public class Bruch {
   private int zaehler;
   private int nenner = 1;
   private String etikett = "";

   private static int anzahl;
```

[1] Module wurden mit Java 9 eingeführt (im September 2017), und die meisten aktuellen Java-Programme wurden für eine JVM-Version (≤ 8) erstellt, sodass dort der der Modifikator **public** den Zugriff für *alle* Klassen erlaubt.

```
    public Bruch(int z, int n, String eti) {
       setzeZaehler(z);
       setzeNenner(n);
       setzeEtikett(eti);
       anzahl++;
    }

    public Bruch() {anzahl++;}

     . . .
 }
```

Die Klassenvariable `anzahl` ist als **private** deklariert, also nur in Methoden der eigenen Klasse sichtbar. Sie wird in den beiden Konstruktoren inkrementiert.

Während jedes Objekt einer Klasse über einen eigenen Satz mit allen Instanzvariablen verfügt, die beim Erzeugen des Objekts auf dem Heap landen, existiert eine klassenbezogene Variable nur *einmal*. Sie wird beim Laden der Klasse in der sogenannten *Method Area* des programmeigenen Speichers angelegt.

Wie für Instanz- gilt auch für Klassenvariablen:

- Sie werden außerhalb jeder Methodendefinition deklariert.
- Sie werden (sofern nicht finalisiert, siehe unten) automatisch mit dem typspezifischen Nullwert initialisiert (vgl. Abschnitt 4.2.3), sodass im Beispiel die Variable `anzahl` mit dem **int**-Wert 0 startet.

Im Editor unserer Entwicklungsumgebung IntelliJ IDEA werden statische Variablen per Voreinstellung durch *kursive Schrift* gekennzeichnet (siehe obigen Quellcode).

In Instanz- oder Klassenmethoden der eigenen Klasse lassen sich Klassenvariablen ohne jedes Präfix ansprechen (siehe obige `Bruch`-Konstruktoren). Sofern Methoden *fremder* Klassen der direkte Zugriff auf eine Klassenvariable gewährt wird, müssen diese dem Variablennamen einen Vorspann aus Klassennamen und Punktoperator voranstellen, z. B.:

```
System.out.println("Hallo");
```

Wir verwenden seit Beginn des Kurses in fast jedem Programm die Klassenvariable **out** aus der Klasse **System** (im Paket **java.lang**). Diese zeigt auf ein Objekt der Klasse **PrintStream**, dem wir Ausgabeaufträge übergeben. Vor Schreibzugriffen ist diese öffentliche Klassenvariable durch das Finalisieren geschützt.

Mit dem Modifikator **final** können nicht nur lokale Variablen (siehe Abschnitt 3.3.10) und Instanzvariablen (siehe Abschnitt 4.2.5) sondern auch statische Variablen als finalisiert deklariert werden. Dadurch entfällt die automatische Initialisierung mit dem typspezifischen Nullwert. Die somit erforderliche explizite Initialisierung kann bei der Deklaration oder in einem statischen Initialisierer (siehe Abschnitt 4.5.4) erfolgen. Im weiteren Programmverlauf ist bei finalisierten Klassenvariablen keine Wertänderung mehr möglich.

Von einem *konstanten Feld* spricht man nach Bloch (2018, S. 290) bei einem statischen und unveränderlichen Feld (dekoriert mit **static** und **final**), wenn außerdem eine von den folgenden Bedingungen erfüllt ist:

- Das Feld besitzt einen primitiven Datentyp.
- Das Feld zeigt auf ein Objekt, das nicht geändert werden kann (z. B. vom Datentyp **String**).

Ein Beispiel ist das **double**-Feld **PI** in der API-Klasse **Math** (Paket **java.lang**), das eine Approximation der Kreiszahl π enthält:

```
public static final double PI = 3.141592653589793238464;
```

Im Namen eines konstanten Felds verwendet man per Konvention ausschließlich Großbuchstaben. Besteht ein Name aus mehreren Wörtern, werden diese der Lesbarkeit halber durch einen Unterstrich getrennt, z. B.:

```
public final static int DEFAULT_SIZE = 100;
```

Wenn sich die Java-Designer an die eben beschriebene Notationskonvention gehalten haben, sollte das statische Feld **out** in der Klasse **System** (mit klein geschriebenem Namen) nicht konstant sein:

```
System.out.println("Hallo");
```

Wie die (z. B. im Abschnitt 4.5.2 präsentierte) Deklaration von **System.out** zeigt, ist die Variable als **final** deklariert:

```
public static final PrintStream out;
```

Allerdings ist ein **PrintStream**-Objekt nicht unveränderlich, weil es z. B. einen variablen Ausgabepuffer besitzt, sodass **System.out** tatsächlich kein konstantes Feld ist.[1]

In der folgenden Tabelle sind wichtige Unterschiede zwischen Klassen- und Instanzvariablen zusammengestellt:

	Instanzvariablen	**Klassenvariablen**
Deklaration	Ohne Modifikator **static**	Mit Modifikator **static**
Zuordnung	Jedes Objekt besitzt einen eigenen Satz mit allen Instanzvariablen.	Klassenbezogene Variablen sind nur einmal vorhanden.
Existenz	Instanzvariablen werden beim Erzeugen des Objekts angelegt und initialisiert. Sie werden ungültig, wenn das Objekt nicht mehr referenziert ist.	Klassenvariablen werden beim Laden der Klasse angelegt und initialisiert.[2]

4.5.2 Wiederholung zur Kategorisierung von Variablen

Mittlerweile haben wir verschiedene Variablensorten kennengelernt, wobei die Sortenbezeichnung unterschiedlich motiviert war. Um einer möglichen Verwirrung vorzubeugen, bietet dieser Abschnitt eine Zusammenfassung bzw. Wiederholung. Die folgenden Begriffe sollten Ihnen keine Probleme mehr bereiten:

---

[1] Im Abschnitt 14.3.1.6 wird sich zeigen, dass der Inhalt von **System.out** trotz **final**-Deklaration mit der statischen **System**-Methode **setOut**() geändert werden kann. Diesen Verstoß im Verhalten des Java-Compilers gegen die **final**-Deklaration rechtfertigt die JLS (Gosling et al. 2019, Abschnitt 17.5.4) so:

> Normally, a field that is `final` and `static` may not be modified. However, `System.in`, `System.out`, and `System.err` are `static final` fields that, for legacy reasons, must be allowed to be changed by the methods `System.setIn`, `System.setOut`, and `System.setErr`.

[2] Das *Entladen* einer Klasse zur Speicheroptimierung ist einer Java-Implementierung prinzipiell erlaubt, aber mit Problemen verbunden und folglich an spezielle Voraussetzungen gebunden (siehe Gosling et al 2019, Abschnitt 12.7). Eine vom regulären Klassenlader der JVM geladene Klasse wird nicht vor dem Ende des Programms entladen (Ullenboom 2012a, Abschnitt 11.5).

- **Lokale Variablen ...**
  werden in Methoden deklariert,
  landen auf dem Stack,
  werden **nicht** automatisch initialisiert,
  sind gültig von der Deklaration bis zum Ende des Blocks, der die Deklaration enthält.
- **Instanzvariablen ...**
  werden außerhalb jeder Methode deklariert,
  landen (als Bestandteile von Objekten) auf dem Heap,
  werden (falls nicht finalisiert) automatisch mit dem typspezifischen Nullwert initialisiert,
  sind verwendbar, wo eine Referenz zum Objekt vorliegt und Zugriffsrechte bestehen.
- **Klassenvariablen ...**
  werden außerhalb jeder Methode mit dem Modifikator **static** deklariert,
  landen (als Bestandteile von Klassen) in der Method Area des programmeigenen Speichers,
  werden (falls nicht finalisiert) automatisch mit dem typspezifischen Nullwert initialisiert,
  sind verwendbar, wo Zugriffsrechte bestehen.
- **Referenzvariablen ...**
  zeichnen sich durch ihren speziellen *Inhalt* aus (Referenz auf ein Objekt). Es kann sich um lokale Variablen (z. B. `b1` in der **main()** - Methode von `Bruchrechnung`), um Instanzvariablen (z. B. `etikett` in der `Bruch`-Definition) oder um Klassenvariablen handeln (z. B. `anzahl` in der `Bruch`-Definition).

Man kann die Variablen kategorisieren nach ...

- **Datentyp (Inhalt)**
  Hinsichtlich des Variableninhalts sind Werte von primitivem Datentyp und Objektreferenzen zu unterscheiden.
- **Zuordnung**
  Eine Variable kann zu einem Objekt (Instanzvariable), zu einer Klasse (statische Variable) oder zu einer Methode (lokale Variable) gehören. Damit sind weitere Eigenschaften wie Ablageort, Initialisierung und Gültigkeitsbereich festgelegt (siehe oben).

Aus den Dimensionen *Datentyp* und *Zuordnung* ergibt sich eine (2 × 3)-Matrix zur Einteilung der Java-Variablen:

		Einteilung nach **Zuordnung**		
		Lokale Variable	Instanzvariable	Klassenvariable

Einteilung nach **Datentyp** (Inhalt)	Prim. Datentyp	`// aus der Bruch-`
		`// Methode frage()`
		`int n;`
			`// aus der Klasse Bruch`
			`private int zaehler;`
				`// aus der Klasse Bruch`
				`public static int anzahl;`

	Referenz	`// aus der Bruch-`
		`// Methode zeige()`
		`String luecke = "";`
			`// aus der Klasse Bruch`
			`private String etikett = "";`
				`// aus der Klasse System`
				`public static final`
				`PrintStream out;`

4.5.3 Klassenmethoden

Es ist vielfach sinnvoll oder gar erforderlich, einer *Klasse* Handlungskompetenzen (Methoden) zu verschaffen, die nicht von der Existenz konkreter Objekte abhängen. So muss z. B. beim Start eines Java-Programms die **main()** - Methode der Startklasse ausgeführt werden, bevor irgendein Objekt existiert. Sofern Klassenmethoden vorhanden sind, kann man auch eine Klasse als *Akteur* auf der objektorientierten Bühne betrachten.

Sind *ausschließlich* Klassenmethoden vorhanden, ist das Erzeugen von Objekten nicht sinnvoll. Man kann fremde Klassen durch den Zugriffsmodifikator **private** für die Konstruktoren daran hindern. Auch das Java-API enthält etliche Klassen, die ausschließlich klassenbezogene Methoden besitzen und damit *nicht* zum Erzeugen von Objekten konzipiert sind. Mit der Klasse **Math** aus

dem API-Paket **java.lang** haben wir ein wichtiges Beispiel bereits kennengelernt. Im **Math**-Quellcode wird das Instanzieren folgendermaßen verhindert:

```
/**
 * Don't let anyone instantiate this class.
 */
private Math() {}
```

Die folgende Anweisung zeigt, wie die statische **Math**-Methode **pow()** von einer fremden Klasse aufgerufen werden kann:

```
System.out.println(Math.pow(2, 3));
```

Vor den Namen der auszuführenden Methode setzt man (durch den Punktoperator getrennt) den Namen der angesprochenen Klasse, der eventuell durch den Paketnamen vervollständigt werden muss. Ob der Paketname angegeben werden muss, hängt von der Paketzugehörigkeit der Klasse und von den am Anfang des Quellcodes vorhandenen **import**-Deklarationen ab.

Da unsere `Bruch`-Klasse mittlerweile über eine (private) Klassenvariable für die Anzahl der erzeugten Objekte verfügt, bietet sich die Definition einer Klassenmethode an, mit der diese Anzahl auch von fremden Klassen ermittelt werden kann. Bei der Definition einer Klassenmethode wird (analog zum Vorgehen bei Klassenvariablen) der Modifikator **static** angegeben, z. B.:

```
public static int hanz() {
   return anzahl;
}
```

Ansonsten gelten die Aussagen von Abschnitt 4.3 über die Definition und den Aufruf von Instanzmethoden analog auch für Klassenmethoden.

Im folgenden Programm wird die `Bruch`-Klassenmethode `hanz()` in der `Bruchrechnung`-Klassenmethode **main()** aufgerufen, um die Anzahl der bisher erzeugten Brüche zu ermitteln:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Bruchrechnung {
   public static void main(String[] args) {
      System.out.println(Bruch.hanz() + " Brüche erzeugt");
      Bruch b1 = new Bruch(1, 2, "Bruch 1");
      Bruch b2 = new Bruch(5, 6, "Bruch 2");
      b1.zeige();
      b2.zeige();
      System.out.println(Bruch.hanz() + " Brüche erzeugt");
   }
}</pre></td><td><pre>0 Brüche erzeugt
           1
 Bruch 1 -----
           2

           5
 Bruch 2 -----
           6

2 Brüche erzeugt</pre></td></tr>
</table>

Wird eine Klassenmethode von anderen Methoden der *eigenen* Klasse (objekt- oder klassenbezogen) verwendet, muss der Klassenname *nicht* angegeben werden. Wir könnten z. B. in der `Bruch`-Instanzmethode `klone()` die `Bruch`-Klassenmethode `hanz()` aufrufen:

```
public Bruch klone() {
   Bruch b = new Bruch(zaehler, nenner, etikett);
   System.out.println(hanz() + " Brüche erzeugt");
   return b;
}
```

Gelegentlich wird missverständlich behauptet, in einer statischen Methode könnten keine Instanzmethoden aufgerufen werden, z. B. (Mössenböck 2005, S. 153):

> Objektmethoden können Klassenmethoden aufrufen aber nicht umgekehrt.

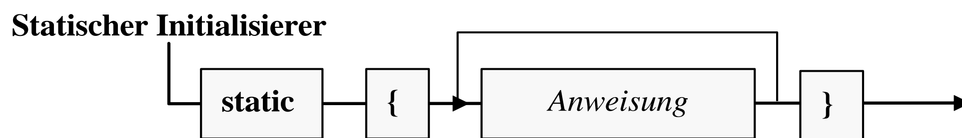
Sofern eine statische Methode eine Referenz zu einem Objekt besitzt, das sie eventuell selbst erzeugt hat, kann sie im Rahmen der eingeräumten Zugriffsrechte (bei Objekten der eigenen Klasse

also uneingeschränkt) Instanzmethoden dieses Objekts aufrufen. In einer Klassenmethode eine Instanzmethode ohne vorangestellte Objektreferenz aufzurufen, wäre reichlich sinnlos. Wer einen Auftrag an ein Objekt schicken möchte, muss den Empfänger natürlich benennen.

4.5.4 Statische Initialisierer

Analog zur Initialisierung von Instanzvariablen durch die Instanzkonstruktoren (siehe Abschnitt 4.4.3) oder die (in der Regel nur bei anonymen Klassen verwendeten) Instanzinitialisierer (siehe Abschnitt 4.4.4), die beim Erzeugen eines Objekts ausgeführt werden, bietet Java zur Vorbereitung von Klassenvariablen und eventuell auch zu weiteren Maßnahmen auf Klassenebene statische Initialisierer, die beim Laden der Klasse ausgeführt werden (siehe z. B. Gosling et al. 2019, Abschnitt 8.7).

Syntaktisch unterscheidet sich ein statischer Initialisierer von einem Instanzinitialisierer durch das vorangestellte Schlüsselwort **static**:



Zugriffsmodifikatoren sind verboten und überflüssig, weil ein statischer Initialisierer ohnehin nur vom Laufzeitsystem aufgerufen wird (beim Laden der Klasse).

Eine Klassendefinition kann *mehrere* statische Initialisierungsblöcke enthalten. Beim Laden der Klasse werden sie nach der Reihenfolge im Quelltext ausgeführt.

Bei einer etwas künstlichen (und in weiteren Ausbaustufen nicht mitgeschleppten) Erweiterung der Klasse `Bruch` soll der parameterfreie Konstruktor zufallsabhängige, aber pro Programmeinsatz identische Werte zur Initialisierung der Felder `zaehler` und `nenner` verwenden:

```
public Bruch() {
    zaehler = ZAEHLER_VOREINST;
    nenner = NENNER_VOREINST;
    anzahl++;
}
```

Dazu erhält die `Bruch`-Klasse private, statische und finalisierte Felder, die von einem statischen Initialisierer beim Laden der Klasse auf Zufallswerte gesetzt werden sollen:

```
private static final int ZAEHLER_VOREINST;
private static final int NENNER_VOREINST;
```

Im statischen Initialisierer wird ein Objekt der Klasse **Random** aus dem Paket **java.util** erzeugt und dann durch **nextInt()** - Methodenaufrufe mit der Produktion von **int**-Zufallswerten aus dem Bereich von 0 bis 4 beauftragt. Daraus entstehen Startwerte für die Felder `zaehler` und `nenner`:

```
public class Bruch {
   private int zaehler;
   private int nenner = 1;
   private String etikett = "";
   private static int anzahl;
   private static final int ZAEHLER_VOREINST;
   private static final int NENNER_VOREINST;
```

```
    static {
       java.util.Random zuf = new java.util.Random();
       ZAEHLER_VOREINST = zuf.nextInt(5) + 1;
       NENNER_VOREINST = zuf.nextInt(5) + ZAEHLER_VOREINST;
       System.out.println("Klasse Bruch geladen");
    }

      . . .
 }
```

Außerdem protokolliert der statische Initialisierer noch das Laden der Klasse, z. B.:

Quellcode	Ausgabe

<pre>class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b = new Bruch();
      b.zeige();
   }
}</pre>
	<pre>Klasse Bruch geladen
	    5
	  ----- 	
	    9</pre>

Ein wesentlicher, gegeben den momentanen Kursfortschritt noch nicht vorführbarer Nutzen eines statischen Initialisierers im Vergleich zur Initialisierung von statischen Feldern bei der Deklaration besteht darin, dass Ausnahmen abgefangen und geworfen werden können (siehe Kapitel 11).

4.6 Rekursive Methoden

Innerhalb einer Methode darf man selbstverständlich nach Belieben *andere* Methoden aufrufen. Es ist aber auch zulässig und manchmal sogar sinnvoll, dass eine Methode *sich selbst* aufruft. Solche *rekursiven* Aufrufe erlauben eine elegante Lösung für ein Problem, das sich sukzessive auf stets einfachere Probleme desselben Typs reduzieren lässt, bis man schließlich zu einem direkt lösbaren Problem gelangt.

Als Beispiel betrachten wir die Ermittlung des größten gemeinsamen Teilers (GGT) zu zwei ganzen Zahlen, die in der `Bruch`-Methode `kuerze()` benötigt wird.[1] Sie haben bereits zwei *iterative* (mit einer Schleife realisierte) Varianten des Euklidischen Lösungsverfahrens kennengelernt: Im Abschnitt 1.1 wurde ein sehr einfacher Algorithmus benutzt, den Sie später in einer Übungsaufgabe (siehe Seite 195) durch einen effizienteren Algorithmus (unter Verwendung des Modulo-Operators) ersetzt haben. Im aktuellen Abschnitt betrachten wir noch einmal die effizientere Variante, wobei zur Vereinfachung der Darstellung der GGT-Algorithmus vom restlichen Kürzungsverfahren getrennt und in eine eigene (private) Methode namens `ggTi()` ausgelagert wird:[2]

---

[1] Bislang sind wir beim GGT stets von zwei *natürlichen* Zahlen {0, 1, 2, ...} ausgegangen, jedoch macht die Verallgemeinerung auf *ganze* Zahlen {..., -2, -1, 0, 1, 2, ...} keine Schwierigkeiten. Unabhängig von den Vorzeichen der beiden ganzen Zahlen ist ihr größter gemeinsamer Teiler stets positiv. Ist t ein gemeinsamer Teiler, dann trifft dies auch auf $-t$ zu, und es gilt trivialerweise $t > -t$.

[2] Wenn die Methode `ggTi()` oder die gleich darzustellende Methode `ggTr()` beim Aufruf den Wert 0 als zweiten Aktualparameter erhält, dann kommt es zu einem Laufzeitfehler (`java.lang.ArithmeticException: / by zero`). Vorsichtsmaßnahmen gegen diesen Fehler sind nicht unbedingt erforderlich, weil die Methoden als **private** deklariert sind und ausschließlich in `kuerze()` aufgerufen werden. Dabei ist der zweite Aktualparameter stets der Nenner eines `Bruch`-Objekts, also niemals gleich 0. Wenn die Methoden `ggTi()` oder `ggTr()` permanent in der Klasse `Bruch` verbleiben würden und es somit später zu weiteren klasseninternen Verwendungen kommen könnte, dann wäre eine Absicherung gegen eine Division durch 0 durchaus sinnvoll.

```
private int ggTi(int a, int b) {
   int rest;
   do {
      rest = a % b;
      a = b;
      b = rest;
   } while (rest > 0);
   return Math.abs(a);
}

public void kuerze() {
   if (zaehler != 0) {
      int ggt = ggTi(zaehler, nenner);
      zaehler /= ggt;
      nenner  /= ggt;
   } else
      nenner = 1;
}
```

Die mit einer **do-while** - Schleife operierende Methode `ggTi()` kann durch die folgende rekursive Variante `ggTr()` ersetzt werden:

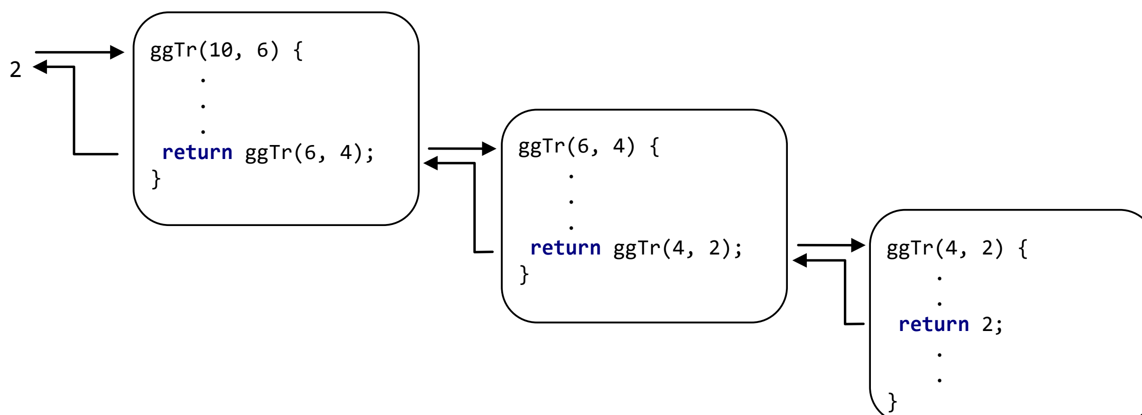
```
private int ggTr(int a, int b) {
   int rest = a % b;
   if (rest == 0)
      return Math.abs(b);
   else
      return ggTr(b, rest);
}
```

Statt eine Schleife zu benutzen, arbeitet die rekursive Methode nach folgender Logik:

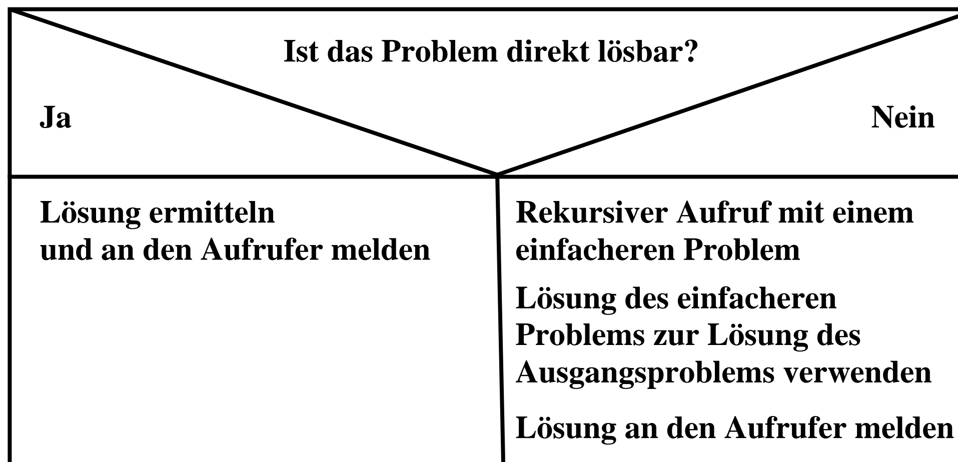
- Ist der Parameter `a` durch den Parameter `b` restfrei teilbar, dann ist `b` der GGT von `a` und `b`, und der Algorithmus endet mit der Rückgabe des Betrags von `b`:
  ```
  return Math.abs(b);
  ```
- Anderenfalls wird das Problem, den GGT von `a` und `b` zu bestimmen, auf das einfachere Problem zurückgeführt, den GGT von `b` und `(a % b)` zu bestimmen, und die Methode `ggTr()` ruft sich selbst mit neuen Aktualparametern auf. Dies geschieht elegant im Ausdruck der **return**-Anweisung:
  ```
  return ggTr(b, rest);
  ```

Im iterativen Algorithmus wird übrigens derselbe Trick zur Reduktion des Problems verwendet, und den zugrunde liegenden Satz der mathematischen Zahlentheorie kennen Sie schon aus der oben erwähnten Übungsaufgabe im Abschnitt 3.9.

Wird die Methode `ggTr()` z. B. mit den Argumenten 10 und 6 aufgerufen, kommt es zu folgender Aufrufverschachtelung:



Generell läuft eine rekursive Methode mit Lösungsübermittlung per Rückgabewert nach der im folgenden **Struktogramm** beschriebenen Logik ab:



Im Beispiel ist die Lösung des einfacheren Problems sogar identisch mit der Lösung des ursprünglichen Problems.

Wird bei einem fehlerhaften Algorithmus der linke Zweig nie oder zu spät erreicht, dann erschöpfen die geschachtelten Methodenaufrufe die Stack-Kapazität, und es kommt zu einem Ausnahmefehler, z. B.:

```
Exception in thread "main" java.lang.StackOverflowError
```

Zu einem rekursiven Algorithmus (per Selbstaufruf einer Methode) existiert stets auch ein iterativer Algorithmus (per Wiederholungsanweisung). Rekursive Algorithmen lassen sich zwar oft eleganter formulieren als die iterativen Alternativen, benötigen aber durch die hohe Zahl von Methodenaufrufen in der Regel mehr Rechenzeit.

4.7 Komposition

Bei Instanz- und Klassenvariablen sind beliebige Datentypen zugelassen, auch Referenztypen (siehe Abschnitt 4.2). Z. B. ist in der aktuellen `Bruch`-Definition eine Instanzvariable vom Referenztyp **String** vorhanden. Es ist also möglich, Objekte vorhandener Klassen als Bestandteile von neuen, komplexeren Klassen zu verwenden. Neben der später noch ausführlich zu behandelnden Vererbung ist diese *Komposition* (alias: *Aggregation*) eine effektive Technik zur Wiederverwendung von vorhandenen Typen beim Design von neuen Typen.[1] Außerdem ist sie im Sinne einer realitätsnahen Modellierung unverzichtbar, denn auch ein reales Objekt (z. B. eine Firma) enthält andere Objekte[2] (z. B. Mitarbeiter, Kunden), die ihrerseits wiederum Objekte enthalten (z. B. ein Gehaltskonto und einen Terminkalender bei den Mitarbeitern) usw.

Man kann den Standpunkt einnehmen, dass die Komposition eine selbstverständliche, wenig spektakuläre Angelegenheit sei, eigentlich nur ein neuer Begriff für eine längst vertraute Situation (Instanzvariablen mit Referenztyp). Es ist tatsächlich für den weiteren Lernerfolg im Kurs unkritisch, wenn Sie den Rest des aktuellen Abschnitts mit einem recht länglichen Beispiel zur Komposition überspringen.

---

[1] Durch die Verwendung von Schnittstellentypen (siehe Kapitel 9) wird die Effektivität der Komposition zur Realisation von Software-Recycling (und auch von Polymorphie) noch erheblich gesteigert, siehe z. B. https://en.wikipedia.org/wiki/Composition_over_inheritance.

[2] Die betroffenen Personen mögen den Fachterminus *Objekt* nicht persönlich nehmen.

Wir erweitern das Bruchrechnungsprogramm um eine Klasse namens `Aufgabe`, die Trainingssitzungen unterstützen soll und dazu mehrere `Bruch`-Objekte verwendet. In der `Aufgabe`-Klassendefinition tauchen vier Instanzvariablen vom Typ `Bruch` auf:

```
public class Aufgabe {
   private Bruch b1, b2, lsg, antwort;
   private char op = '+';

   public Aufgabe(char op_, int b1Z, int b1N, int b2Z, int b2N) {
      if (op_ == '*')
         op = op_;
      b1 = new Bruch(b1Z, b1N, "1. Argument:");
      b2 = new Bruch(b2Z, b2N, "2. Argument:");
      lsg = new Bruch(b1Z, b1N, "Das korrekte Ergebnis:");
      antwort = new Bruch();
      init();
   }

   private void init() {
      switch (op) {
         case '+': lsg.addiere(b2);
                   break;
         case '*': lsg.multipliziere(b2);
                   break;
      }
   }

   public boolean korrekt() {
       Bruch temp = antwort.klone();
       temp.kuerze();
       if (lsg.gibZaehler() == temp.gibZaehler() &&
           lsg.gibNenner() == temp.gibNenner())
           return true;
       else
           return false;
   }

   public void zeige(int was) {
      switch (was) {
         case 1: System.out.println("   " + b1.gibZaehler() +
                 "                " + b2.gibZaehler());
                 System.out.println(" ------     " + op + "     -----");
                 System.out.println("   " + b1.gibNenner() +
                 "                " + b2.gibNenner());
                 break;
         case 2: lsg.zeige(); break;
         case 3: antwort.zeige(); break;
      }
   }

   public void frage() {
      System.out.println("\nBerechne bitte:\n");
      zeige(1);
      do {
         System.out.print("\nWelchen Zähler hat Dein Ergebnis:     ");
         antwort.setzeZaehler(Simput.gint());
      } while (Simput.checkError());
      System.out.println("                                     ------");
      do {
         System.out.print("Welchen Nenner hat Dein Ergebnis:      ");
         antwort.setzeNenner(Simput.gint());
      } while (Simput.checkError());
   }
```

```
    public void pruefe() {
       frage();
       if (korrekt())
          System.out.println("\n Richtig!");
       else {
          System.out.println("\n Falsch!");
          zeige(2);
       }
    }

    public void neueWerte(char op_, int b1Z, int b1N, int b2Z, int b2N) {
       op = op_;
       b1.setzeZaehler(b1Z); b1.setzeNenner(b1N);
       b2.setzeZaehler(b2Z); b2.setzeNenner(b2N);
       lsg.setzeZaehler(b1Z); lsg.setzeNenner(b1N);
       init();
    }
 }
```

Die vier `Bruch`-Objekte in einer `Aufgabe` dienen folgenden Zwecken:

- `b1` und `b2` werden dem Anwender (in der `Aufgabe`-Methode `frage()`) im Rahmen einer Aufgabenstellung vorgelegt, z. B. zum Addieren.
- In `antwort` landet der Lösungsversuch des Anwenders.
- In `lsg` steht das korrekte Ergebnis.

In der Klasse `Bruch` wird die Instanzmethode `multipliziere()` nachgerüstet, die analog zur Methode `addiere()` arbeitet:

```
public void multipliziere(Bruch b) {
   zaehler = zaehler*b.zaehler;
   nenner = nenner*b.nenner;
   kuerze();
}
```

Im folgenden Programm wird die Klasse `Aufgabe` für ein Bruchrechnungstraining verwendet:

```
class Bruchrechnung {
   public static void main(String[] args) {
      Aufgabe auf = new Aufgabe('*', 3, 4, 2, 3);
      auf.pruefe();
      auf.neueWerte('+', 1, 2, 2, 5);
      auf.pruefe();
   }
}
```

Man kann immerhin schon ahnen, wie die praxistaugliche Endversion des Programms einmal arbeiten wird:

```
Berechne bitte:

  3             2
------    *   -----
  4             3

Welchen Zaehler hat Dein Ergebnis:     6
                                     ------
Welchen Nenner hat Dein Ergebnis:      12

 Richtig!
```

```
Berechne bitte:

   1               2
 ------     +    -----
   2               5

Welchen Zähler hat Dein Ergebnis:     3
                                    ------
Welchen Nenner hat Dein Ergebnis:      7

 Falsch!
                         9
 Das korrekte Ergebnis: -----
                         10
```

4.8 Mitgliedsklassen und lokale Klassen

Bisher haben wir mit *Top-Level - Klassen* gearbeitet, die eigenständig auf Paketebene (also *nicht* im Quellcode einer übergeordneten Klasse) definiert werden. Für Klassen, die nur in einem eingeschränkten Bereich (in einer anderen Klasse oder in einer Methode) benötigt werden, erlaubt Java auch die Definition innerhalb einer umgebenden Klasse und sogar innerhalb einer Methode. Das hat folgende Vorteile:

- Es lassen sich mehr Implementierungsdetails verstecken (erweitertes Information Hiding). So ist z. B. eine lokale (methodenintern definierte) Klasse nur innerhalb der umgebenden Methode sichtbar, während eine Top-Level - Klasse mindestens in ihrem Paket sichtbar ist.
- Mitgliedsklassen und lokale Klassen haben einen privilegierten Zugriff auf Variablen in der Umgebung (auch auf private Felder).
- Die zusammengehörigen Klassendefinitionen bleiben übersichtlich und wartungsfreundlich an einem Ort konzentriert.

Weil die resultierenden Konstruktionen etwas kompliziert sind, sollten sich Programmiereinsteiger zunächst auf eine oberflächliche Lektüre von Abschnitt 4.8 beschränken und sich erst dann für Details interessieren, wenn diese später relevant werden.

Bei der Begriffsverwendung orientiert sich das Manuskript am Java-Tutorial (Oracle 2019a).[1]

4.8.1 Mitgliedsklassen

Eine Mitgliedsklasse befindet sich im Quellcode einer umgebenden Klassendefinition, aber *nicht* in einer Methodendefinition, z. B.:

```java
class Top {
  . . .
 class MemberClass {
   . . .
 }
  . . .
}
```

Im Java-Tutorial (Oracle 2019a) werden sie auch als *eingeschachtelte Klassen* (engl. *nested classes*) bezeichnet. Manche Autoren verwenden diesen Ausdruck allerdings in einem allgemeineren Sinn und beziehen dabei auch die lokalen Klassen ein.

Einige Eigenschaften von Mitgliedsklassen sind:

[1] http://docs.oracle.com/javase/tutorial/java/javaOO/whentouse.html

- Während für Top-Level - Klassen (Klassen auf Paktebene) nur der Zugriffsmodifikator **public** erlaubt ist, können bei Mitgliedsklassen auch die Zugriffsmodifikatoren **private** und **protected** verwendet werden (vgl. Abschnitt 6.3.2). Für Mitgliedsklassen ist der Zugriffsschutz (die Sichtbarkeit) also genauso geregelt wie bei anderen Klassenmitgliedern (z. B. Feldern oder Methoden).
- Als Klassenmitglieder können eingeschachtelte Klassen den Modifikator **static** erhalten. Wenn der Modifikator fehlt, spricht man von einer *inneren Klasse* (siehe Abschnitt 4.8.1.1).
- Mitgliedsklassen dürfen geschachtelt werden.
- Der Compiler erzeugt auch für jede Mitgliedsklasse eine eigene **class**-Datei, wobei die Umgebung in die Benennung eingeht. Befindet sich z. B. in der Klasse `Top` die Klasse `Mitglied`, dann entsteht die Datei **Top$Mitglied.class**.

Eine Mitgliedsklasse sollte ausschließlich im Rahmen der umgebenden Klasse zum Einsatz kommen. Anderenfalls ist eine Top-Level - Klasse zu bevorzugen (Bloch 2018, S. 112).

4.8.1.1 Innere Klassen

Ein Objekt einer inneren Klasse ist grundsätzlich mit einem „Hüllen“ - Objekt der umgebenden Klasse assoziiert. Das folgende Programm enthält die umgebende Klasse `Mantel` und darin die innere Klasse `Manteltasche`. Eine Manteltasche ohne umgebenden Mantel ist nicht vorstellbar bzw. erforderlich:

```
class Mantel {
   private String name;
   private Manteltasche links, rechts;
   private int anzKnoepfe;
   Mantel(String n, int zahl) {
      name = n;
      anzKnoepfe = zahl;
      links = new Manteltasche(anzKnoepfe);
      rechts = new Manteltasche(anzKnoepfe);
   }

   Manteltasche gibLinkeTasche() {return links;}
   Manteltasche erstelleTasche(int anzK) {return new Manteltasche(anzK);}

   class Manteltasche {
      private int anzKnoepfe;
      Manteltasche(int anzahl) {
         anzKnoepfe = anzahl;
      }
      void report() {
         System.out.println("Tasche an Mantel \"" + name +
                        "\" mit "+ anzKnoepfe + " Knöpfen");
      }
   }
}

class InnerClassDemo {
   public static void main(String[] args) {
      Mantel mantel = new Mantel("Martin", 3);
      // Für die nächsten 3 Anweisungen ist mindestens die Sichtbarkeit Paket
      // bei Mantel.Manteltasche erforderlich.
      Mantel.Manteltasche tasche = mantel.gibLinkeTasche();
      tasche.report();
      Mantel.Manteltasche isoTasche = mantel.new Manteltasche(3);
   }
}
```

Es produziert die Ausgabe:

```
Tasche an Mantel "Martin" mit 3 Knöpfen
```

Weil die innere Klasse `Manteltasche` die voreingestellte Zugriffsstufe *Paket* benutzt, ist sie in der Klasse `InnerClassDemo` sichtbar. Mit dem Zugriffsmodifikator **private** für die innere Klasse könnte die Sichtbarkeit auf die umgebende Klasse `Mantel` eingeschränkt werden.

Einige Eigenschaften von inneren Klassen (also von Mitgliedsklassen ohne Modifikator **static**):

- Bei der Erstellung eines Objekts der inneren Klasse muss ein Objekt der äußeren Klasse als „Hülle" beteiligt sein, was folgendermaßen geschehen kann:
  - In einem Konstruktor der äußeren Klasse wird ein Objekt der inneren Klasse erstellt, z. B.:
    ```
    links = new Manteltasche(anzKnoepfe);
    ```
  - Ein Objekt der äußeren Klasse führt eine Instanzmethode aus und kreiert dort das innere Objekt, z. B.:
    ```
    Manteltasche erstelleTasche(int anzK) {return new Manteltasche(anzK);}
    ```
  - Bei der Kreation des inneren Objekts wird explizit das äußere Objekt als Umgebung benannt, indem das Schlüsselwort **new** durch einen Punkt getrennt auf die Referenz zum äußeren Objekt folgt, z. B.:
    ```
    Mantel.Manteltasche tasche = mantel.new Manteltasche(3);
    ```
- Ein Objekt der inneren Klasse hat Zugriff auf:
  - Statische Member der umgebenden Klasse (auch auf die privaten)
    Wird ein Bezeichner der umgebenden Klasse in der inneren Klasse überdeckt, dann lässt sich die Variante der umgebenden Klasse durch Voranstellen des Klassennamens ansprechen.
  - Felder und Methoden des umgebenden Objekts der äußeren Klasse (auch auf die privaten)
    Wird ein Bezeichner der umgebenden Klasse in der inneren Klasse überdeckt, dann lässt sich das umgebende Objekt über die **this**-Referenz mit vorangestelltem Klassennamen ansprechen, z. B.:
    ```
    Mantel.this.name
    ```
- Methoden und Konstruktoren der äußeren Klasse können auf alle Felder, Methoden und Konstruktoren der inneren Klasse zugreifen (auch auf die privaten).
- In einer inneren Klasse sind keine statischen Methoden erlaubt. Statische Variablen müssen finalisiert sein.

Ergänzend zum obigen spielerischen, zur Demonstration des Prinzips durchaus geeigneten Beispiel soll noch ein reales Beispiel aus dem (äußerst wichtigen) Java Collections Framework (siehe Kapitel 10) ergänzt werden. In der Klasse **HashMap<K,V>** (Paket **java.util**) wird die innere Klasse **KeyIterator** definiert, welche die Schnittstelle **Iterator<K>** implementiert (siehe Abschnitt 10.4). Ein **KeyIterator**-Objekt kann die Menge mit den Schlüsseln des zugehörigen **HashMap<K,V>** - Objekts durchlaufen und nach Aufforderung durch die Methode **next()** den nächsten Schlüssel liefern:

```
final class KeyIterator extends HashIterator implements Iterator<K> {
    public final K next() { return nextNode().key; }
}
```

Dabei erweist es sich als vorteilhaft, dass ein **KeyIterator**-Objekt in einem umgebenden **HashMap<K,V>** - Objekt existiert und (in der Methode **nextMode()**) auf dessen Felder zugreifen kann. Offenbar passt das Konzept der inneren Klasse hier perfekt.

Dass ein Objekt einer inneren Klasse das umgebende Objekt der äußeren Klasse kennt, hat auch die folgenden Konsequenzen (Bloch 2018, S. 113):

- Bei der Objektkreation ist ein Zusatzaufwand (hinsichtlich Platz und Zeit) erforderlich.
- Die im inneren Objekt vorhandene Referenz kann das Abräumen des äußeren Objekts verhindern und ein Speicherloch (engl.: *memory leak*) verursachen.

Wenn für ein Objekt einer Mitgliedsklasse die spezielle Beziehung zu einem Hüllenobjekt der umgebenden Klasse nicht erforderlich ist, dann sollte eine *statische* Mitgliedsklasse definiert werden.

4.8.1.2 Statische Mitgliedsklassen

Wird eine Mitgliedsklasse als **static** deklariert, handelt es sich *nicht* um eine innere Klasse. Sie verhält sich dann wie eine *Top-Level* - Klasse, hat aber Zugriff auf alle Mitglieder der umgebenden Klasse (auch bei **private**-Deklaration) und muss einen Doppelnamen führen. Im folgenden Beispiel werden die Klassen `Mantel` und `Motte` definiert. Während eine Manteltasche ohne umgebenden Mantel nicht vorstellbar ist (siehe Beispiel für eine innere Klasse im Abschnitt 4.8.1.1), kann eine Motte durchaus unabhängig von einem Mantel existieren:

Quellcode

```java
class Mantel {
  private static int KCAL = 3000;
  private int randomID = (int) (Math.random()*Integer.MAX_VALUE);

  static class Motte {
    Mantel mantel;
    void frissMantel() {
      mantel = new Mantel();
      System.out.println("Mantel " + mantel.randomID +
        "\nmit einem Nährwert\nvon " + Mantel.KCAL +
        " kcal verspeist.");
    }
  }
}


class StaticMemberClassDemo {
  public static void main(String[] args) {
   Mantel.Motte motte = new Mantel.Motte();
   motte.frissMantel();
  }
}
```

Ausgabe

```
Mantel 1536246767
mit einem Nährwert
von 3000 kcal verspeist.
```

Im Unterschied zu einem Objekt einer inneren Klasse befindet sich ein Objekt einer statischen Mitgliedsklasse *nicht* „in“ einem Objekt der äußeren Klasse.

Die statische Mitgliedsklasse kann auf statische Mitglieder der äußeren Klasse zugreifen (auch auf die privaten). Wenn Bezeichner für statische Member der äußeren Klasse verdeckt worden sind, muss der Klassenname vorangestellt werden.

Jede Mitgliedsklasse (ob statisch oder nicht) kann Instanzvariablen vom Typ der äußeren Klasse verwenden und hat dabei volle Zugriffsrechte (auch auf private Member).

Eine statische Mitgliedsklasse ist angemessen, wenn

- die Mitgliedklasse ausschließlich von der umgebenden Klasse benutzt werden soll,
- aber die Objekt der Mitgliedsklasse ihre Methoden ohne Rückgriff auf ein Objekt der umgebenden Klasse ausführen.

Ergänzend zum obigen spielerischen, zur Demonstration des Prinzips durchaus geeigneten Beispiel soll noch ein reales Beispiel aus dem (äußerst wichtigen) Java Collections Framework (siehe Kapi-

tel 10) ergänzt werden. In der Klasse **HashMap<K,V>** (Paket **java.util**) wird die Mitgliedsklasse **Node<K,V>** definiert, die ein einzelnes Element in der **HashMap<K,V>** - Kollektion modelliert. Bei der Kommunikation mit einem **Node<K,V>** - Objekt (z. B. über die Methoden **getValue()** oder **setValue()**) ist keine Referenz auf das **HashMap<K,V>** - Objekt erforderlich, sodass eine *statische* Mitgliedsklasse zum Einsatz kommt:

```
static class Node<K,V> implements Map.Entry<K,V> {
 . . .
   public final V getValue() { return value; }

   public final V setValue(V newValue) {
       V oldValue = value;
       value = newValue;
       return oldValue;
   }
 . . .
}
```

4.8.2 Lokale Klassen

In einer Methode dürfen lokale Klassen definiert werden, z. B.:

Quellcode	Ausgabe

<pre>class Aussen {
   private int instVar = 1;
 	
   void erledigeMitLokalerKlasse(int par) {
      int lokVar = 2;
      class LokaleKlasse {
         int informiere() {
            return instVar + lokVar + par;
         }
      }
      LokaleKlasse w = new LokaleKlasse();
      System.out.println(w.informiere());
   }
}
 	
class LocalClassDemo {
   public static void main(String[] args) {
      Aussen p = new Aussen();
      p.erledigeMitLokalerKlasse(3);
   }
}</pre>
	6

Einige Eigenschaften von lokalen Klassen:

- Eine lokale Klasse kann auf die *finalisierten* lokalen Variablen der umgebenden Methode zugreifen. Seit Java 8 wird nur noch die *effektive Finalität* vorausgesetzt. Diese besteht, wenn nach der Initialisierung keine Wertveränderung mehr stattfindet. Der Modifikator **final** ist nicht mehr erforderlich. Seit Java 8 kann eine lokale Klasse auch auf effektiv finale *Parameter* der umgebenden Methode zugreifen. Weil der Zugriff auf (effektiv) finale Variablen bzw. Parameter beschränkt ist, sind natürlich nur *lesende* Zugriffe möglich. Unsere Entwicklungsumgebung IntelliJ zeigt übrigens (seit der Version 2018.2) per Voreinstellung für lokale Variablen (inkl. Parameter) an, ob sie nach der Initialisierung geändert werden, sodass effektiv finale Variablen leicht an der *fehlenden* Unterstreichung zu erkennen sind. So wird eine lokale Variable angezeigt, die keine effektive Finalität besitzt:

```
int notEFin = 1;
notEFin = 13;
```

- Außerdem kann eine lokale Klasse auf die statischen Variablen und Methoden der Klasse zugreifen, zu der die umgebende Methode gehört. Wird eine lokale Klasse in einer Instanzmethode definiert, kann sie auch auf die Instanzvariablen und -methoden des handelnden Objekts zugreifen. Genau dann existiert ein umgebendes Objekt wie bei einer inneren Klasse (siehe Abschnitt 4.8.1.1)
- Felder der lokalen Klasse und lokale Variablen in ihren Methoden überdecken gleichnamige Variablen der umgebenden Klasse. Überdeckte statische Variablen der umgebenden Klasse können in der lokalen Klasse über den Klassennamen als Präfix angesprochen werden, z. B. bei einer umgebenden Klasse namens `Aussen` mit der statischen Variablen `statVar`:
  ```
  Aussen.statVar
  ```
  Überdeckte Instanzvariablen der umgebenden Klasse können in der lokalen Klasse über einen Präfix aus dem Klassennamen und dem Schlüsselwort **this** angesprochen werden, z. B. bei einer umgebenden Klasse namens `Aussen` mit der Instanzvariablen `instVar`:
  ```
  Aussen.this.instVar
  ```
- Wie für eine innere Klasse gilt auch für eine lokale Klasse, dass statische Methoden verboten sind, und statische Variablen finalisiert sein müssen.
- Der Gültigkeitsbereich von lokalen Klassen ist wie bei lokalen Variablen geregelt (siehe Abschnitt 3.3.9). Im Block, der eine lokale Klasse definiert und ein Objekt dieser Klasse erzeugt, kann auf dessen Felder und Methoden zugegriffen werden.
- Der Compiler erzeugt auch für jede lokale Klasse eine eigene **class**-Datei, in deren Namen die Bezeichner für die umgebende und die lokale Klasse eingehen, sodass im Beispiel die folgende Datei entsteht: **Aussen$1LokaleKlasse.class**.

Weitere Informationen über lokale Klassen bietet das Java-Tutorial (Oracle 2019a).[1]

Später werden wir noch die *anonymen Klassen* kennenlernen, die meist innerhalb einer Methode definiert werden und folglich viele Gemeinsamkeiten mit den lokalen Klassen besitzen (siehe Abschnitt 12.1.1.2). Bei der Definition wird kein Name vergeben, aber gleich ein Objekt instanziert, das zudem das einzige seiner Art bleibt. Meist genügen wenige Zeilen Quellcode, um ein Objekt mit einem eingeschränkten, meist aus einer einzigen Methode bestehenden Handlungsrepertoire zu erstellen und zum Einsatz zu bringen, z. B. zur Ereignisbehandlung in einem Programm mit grafischer JavaFX-Bedienoberfläche:

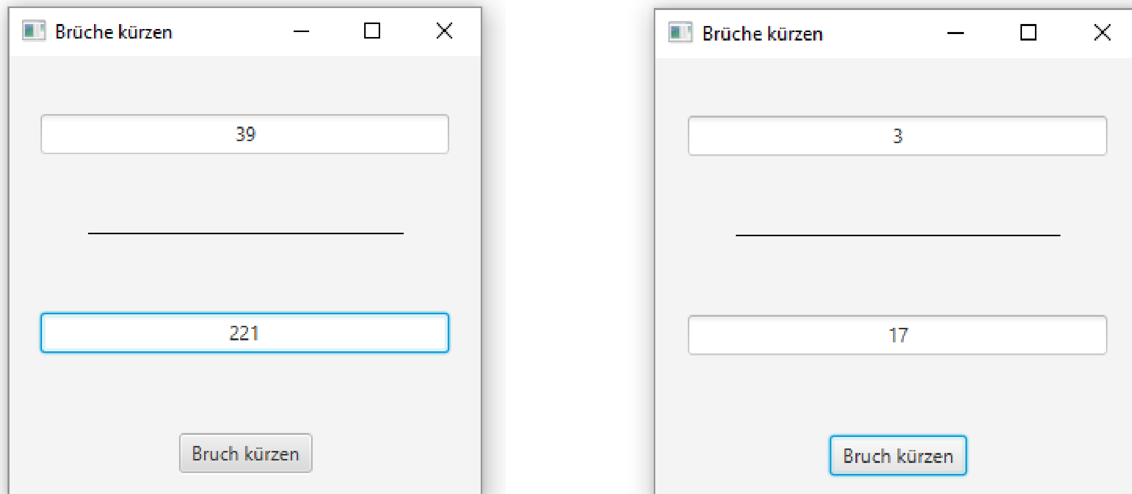
```
public void start(Stage primaryStage) {
 . . .
   button.setOnAction(new EventHandler< ActionEvent>() {
      @Override
      public void handle(ActionEvent event) {
         label.setText("Hallo JavaFX");
      }
   });
 . . .
}
```

[1] http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html

4.9 Bruchrechnungsprogramm mit JavaFX-GUI

Nachdem Sie nun wesentliche Teile der objektorientierten Programmierung mit Java kennengelernt haben, ist ein weiterer Ausblick auf die nicht mehr sehr ferne Entwicklung von Programmen mit grafischer Benutzerschnittstelle (engl. *Graphical User Interface*, *GUI*) als Belohnung und Motivationsquelle angemessen. Schließlich gilt es in diesem Manuskript auch die Erfahrung zu vermitteln, dass man beim Programmieren Erfolg und damit Spaß haben kann.

Wir erstellen unter Verwendung der Klasse `Bruch`, die schon im Abschnitt 1.1 vorgestellt wurde und die im Verlauf von Kapitel 4 wiederholt als Beispiel gedient hat, mit Hilfe des (per Voreinstellung aktivierten) IntelliJ - Plugins **JavaFX** und des GUI-Designers **Scene Builder** ein Bruchkürzungsprogramm mit grafischer Benutzerschnittstelle in JavaFX-Technik:



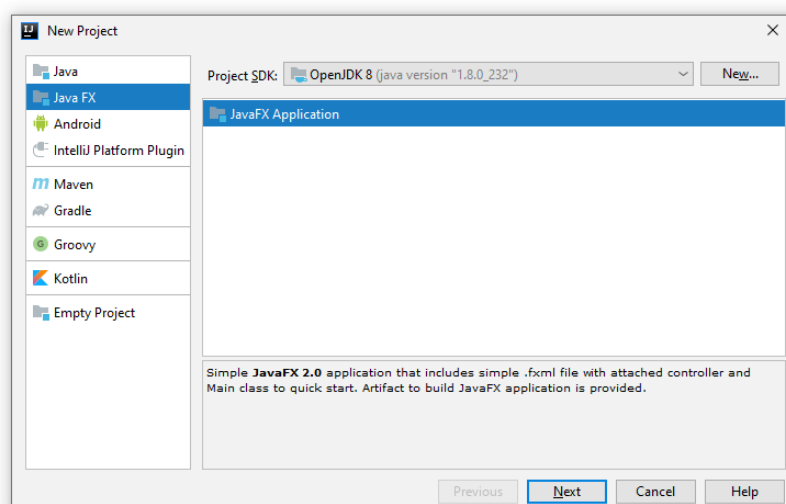
Aufgrund der individuellen Oberflächengestaltung kommt die im Abschnitt 3.8 vorgestellte Klasse **JOptionPane** mit statischen Methoden zur bequemen Realisation von Standarddialogen *nicht* in Frage.

4.9.1 JavaFX-Projekt mit dem OpenJDK 8 anlegen

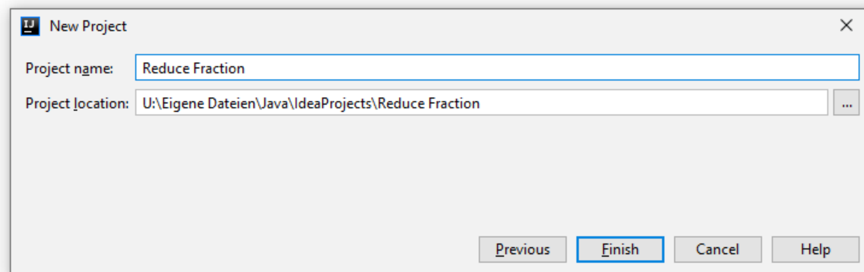
Wir starten mit dem Menübefehl

File > New > Project

und wählen im folgenden Fenster



eine neue **JavaFX Application** basierend auf dem **OpenJDK 8**. Im einzigen Assistentendialog vereinbaren wir den Projektnamen `Reduce Fraction` und einen Projektordner:



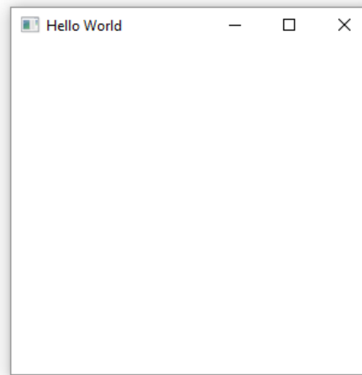
Durch die Verwendung der englischen Sprache werden Umlaute im Projektnamen vermieden. Diese haben die bisherige Arbeit mit unserer Entwicklungsumgebung IntelliJ nicht gestört, sabotieren aber die als Sahnehäubchen für die anstehende Entwicklungsarbeit geplante automatischen Erstellung einer ausführbaren Java-Archivdatei (siehe Abschnitt 6.1.3.5).

Der Assistent legt eine Hauptklasse mit dem Namen `Main` an, die von der API-Klasse **Application** abstammt. Das bei der Definition einer abgeleiteten Klasse zu verwendende Schlüsselwort **extends** wird im Kapitel 7 behandelt:

```
package sample;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));
        primaryStage.setTitle("Hello World");
        primaryStage.setScene(new Scene(root, 300, 275));
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Der Assistent hat für das neue Programm ein (wenn auch nicht vorbildlich benanntes) Paket angelegt (siehe erste Zeile im Quellcode). Wir haben die Paketierungstechnik bisher aus didaktischen Gründen ausgeblendet, werden uns bald (im Kapitel 6) damit befassen. Erste Erfahrungen mit dieser Technik kommen also jetzt durchaus gelegen.

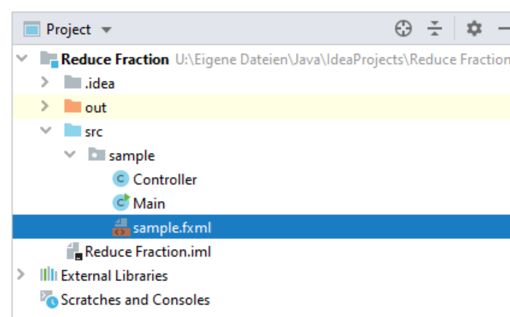
Das vom Assistenten erstellte Programm ist schon startfähig, zeigt aber bisher nur eine leere Szenerie:



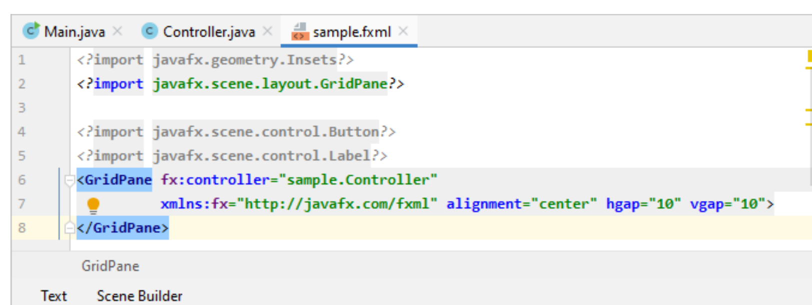
4.9.2 Bedienoberfläche bzw. FXML-Datei mit dem Scene Builder gestalten

Wir wollen beim JavaFX-Einsatz die Bedienoberfläche in einer FXML-Datei deklarieren. Diese Datei werden wir gelegentlich manuell editieren, was unsere Entwicklungsumgebung durch Code-Vervollständigung unterstützt. Meist werden wir den FXML-Code bzw. die Bedienoberfläche jedoch mit Hilfe des **Scene Builders** gestalten. Dieser grafische GUI-Designer wird von der Firma Oracle unter der Oracle BSD Lizenz angeboten und von der Firma Gluon vertrieben (siehe Abschnitt 2.5 zur Installation).

Der IntelliJ-Assistent für neue JavaFX-Projekte hat die Datei **sample.fxml** erstellt, im **src**-Ordner des Projekts abgelegt



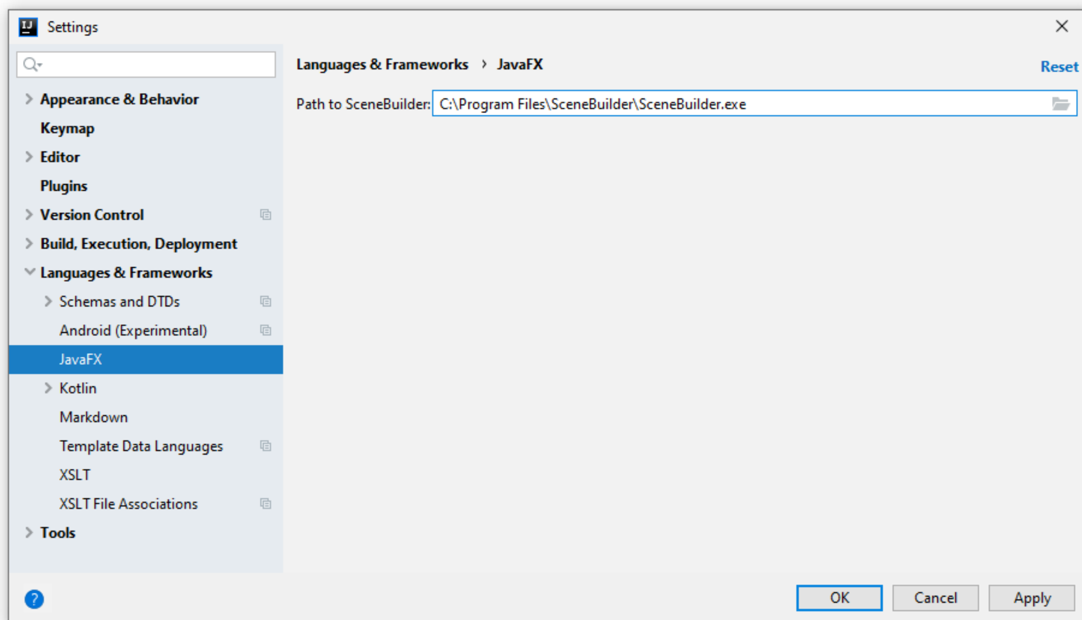
und im Editor geöffnet:



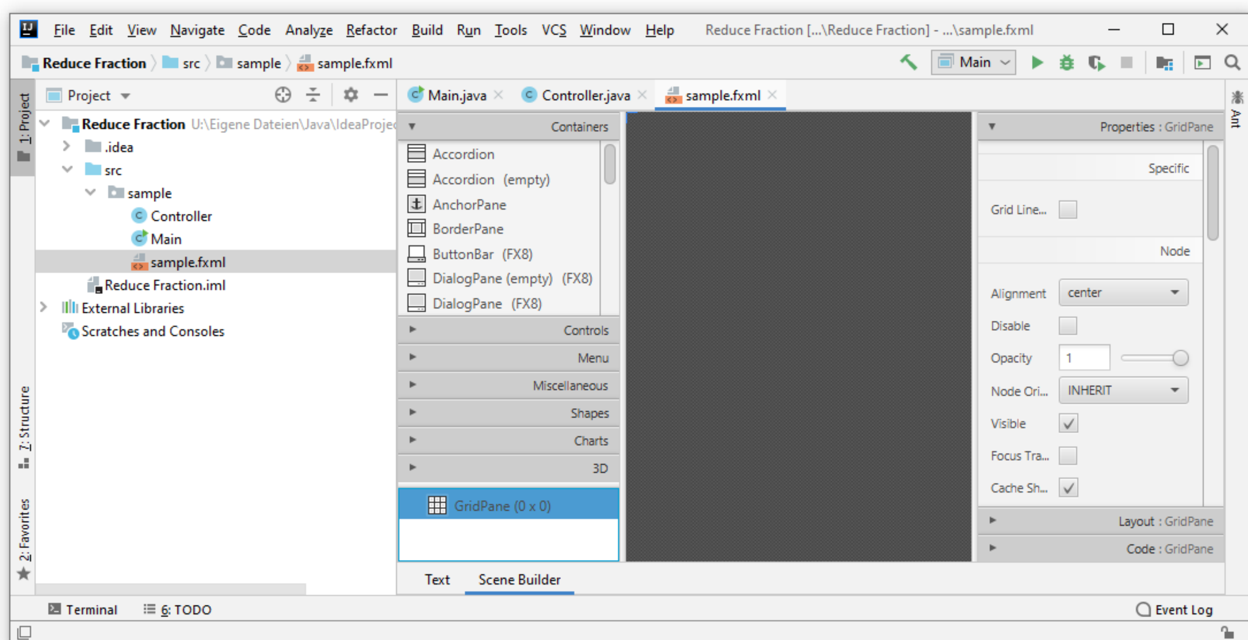
Falls noch nicht geschehen (vgl. Abschnitt 2.5), informieren wir IntelliJ nach

File > Settings > Languages & Frameworks > JavaFX

über die ausführbare Datei der zu verwendenden Scene Builder - Version, z. B.:

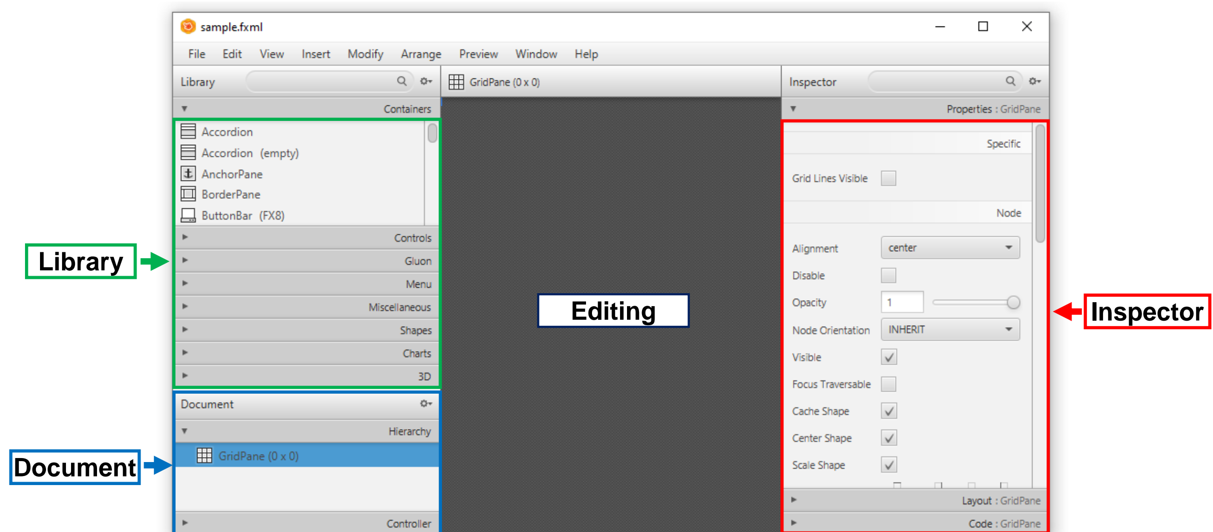


Die FXML-Datei lässt sich *innerhalb* der Entwicklungsumgebung mit dem Scene Builder öffnen durch einen Wechsel zur Editor-Registerkarte **Scene Builder**, z. B.:



Allerdings macht der integrierte Einsatz des Scene Builders gelegentlich Probleme, sodass der kaum aufwändigere externe Einsatz (als selbständiges Programm mit eigenem Fenster) zu bevorzugen war. Diese Betriebsart hat außerdem den Vorteil, dass man das Hauptmenü des Szene Builders nutzen kann.

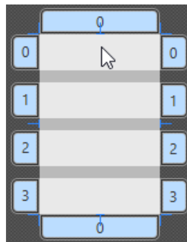
Man öffnet die FXML-Datei zum geöffneten JavaFX-Projekt im selbständig agierenden grafischen GUI-Designer über das Item **Open In SceneBuilder** aus dem Kontextmenü der FXML-Datei. Das nächste Bildschirmfoto zeigt den im eigenständigen Fenster aktive Scene Builder, wobei zur Erleichterung der folgenden Erläuterungen für vier Zonen der Bedienoberfläche Namen vergeben werden:



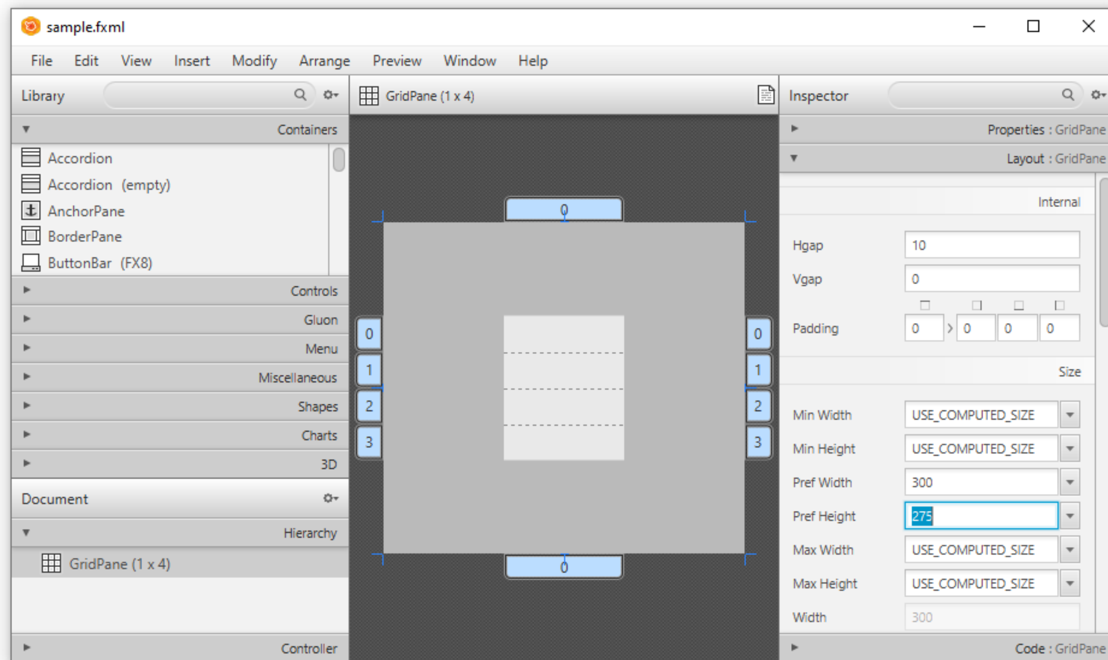
Über das Kontextmenü zum **GridPane**-Wurzelelement der Bedienoberfläche (siehe **Document**-Zone am linken Fensterrand) fügen wir nacheinander vier Zeilen und eine Spalte in das Layout ein:

GridPane > Add Row Above
Add Row Above
Add Row Above
Add Row Above
Add Column Before

Wir markieren den kompletten **GridPane**-Container per Mausklick an der gezeigten Stelle



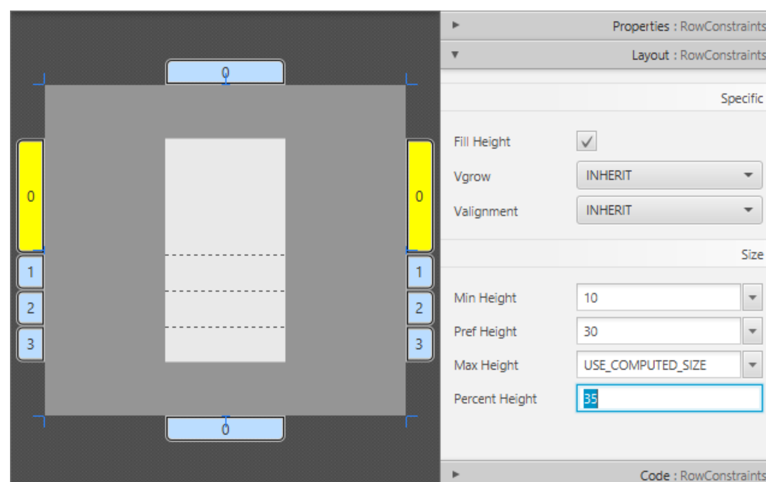
und setzen dann im **Layout**-Segment der **Inspector**-Zone die Eigenschaft **Vgap** (vertikaler Zeilenabstand) auf den Wert 0 und die Eigenschaften **Pref Width** bzw. **Pref Height** (bevorzugte Breite bzw. Höhe) auf 300 bzw. 275 (passend zu der Größe des Anwendungsfensters, siehe Quellcode der Klasse `Main.java` im Abschnitt 4.9.1). Der aktuelle Stand in der **Editing**-Zone sollte ungefähr so aussehen:



Auf die vier Zeilen des **GridPane**-Containers sollen verteilt werden:

Element	**Anteilige Höhe**
TextField-Steuerelement für den Zähler	35%
Horizontale Linie für den Bruchstrich	10%
TextField-Steuerelement für den Nenner	35%
Button-Steuerelement zum Kürzen	20%

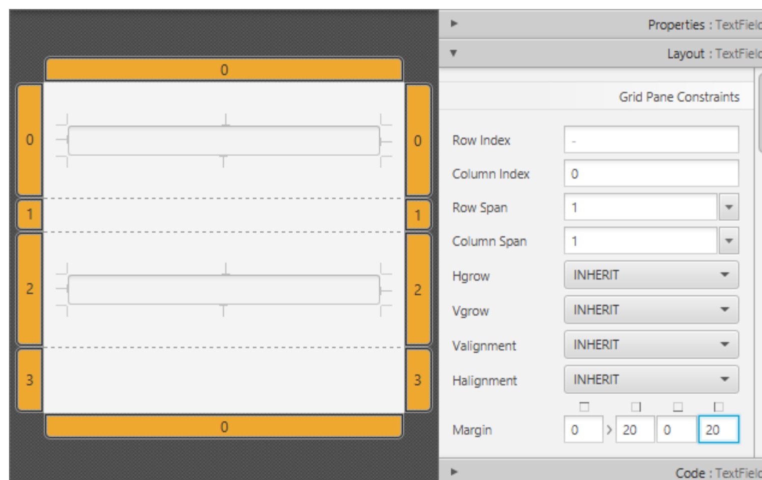
Nach der Markierung einer Zeile per Mausklick auf eine seitliche Lasche kann die Eigenschaft **Percent Height** auf den gewünschten Wert gesetzt werden, z. B.:



Analog verpassen wir der einzigen Spalte eine prozentuale Breite von 100%.

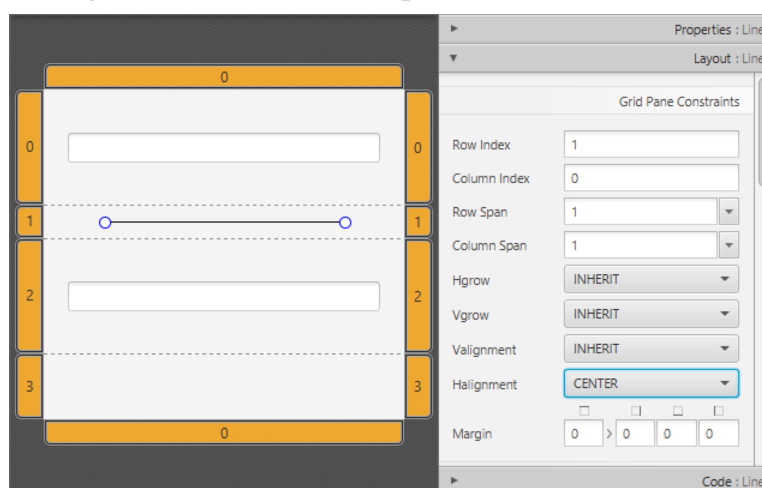
Im nächsten Schritt übertragen wir aus der **Library**-Abteilung **Controls** jeweils eine **TextField**-Komponenten per Drag & Drop in die erste bzw. dritte **GridPane**-Zeile:

Wir markieren beide Textfelder, was am einfachsten über die **Documents**-Zone gelingt, und verpassen ihnen über das **Layout**-Segment der **Inspector**-Zone einen linken sowie rechten Rand mit der Breite 20 (Eigenschaft **Margin**):



Außerdem sollte über das **Properties**-Segment der **Inspector**-Zone die Eigenschaft **Alignment** auf den Wert **CENTER** gesetzt werden, damit im aktiven Programm die vom Benutzer eingetragenen Zeichenfolgen zentriert werden.

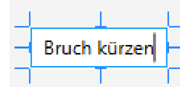
Nun fügen wir aus der **Library**-Abteilung **Shapes** eine **Line**-Komponente per Drag & Drop in die Zelle (1, 0) des **GridPane**-Containers ein und setzen über das **Layout**-Segment der **Inspector**-Zone die **Halignment**-Eigenschaft der **Line**-Komponente auf den Wert **CENTER**:



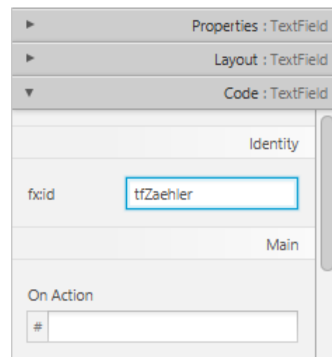
Schließlich befördern wir aus der **Library**-Abteilung **Controls** eine **Button**-Komponente per Drag & Drop in die unterste Zelle des **GridPane**-Containers und setzen die **Layout**-Eigenschaft **Halignment** auf den Wert **CENTER**.

Um die Beschriftung des Schalters zu ändern, haben wir folgende Möglichkeiten:

- Über das **Properties**-Segment der **Inspector**-Zone die Eigenschaft **Text** ändern.
- Nach einem Doppelklick auf den Schalter in der **Editing**-Zone die Beschriftung editieren:

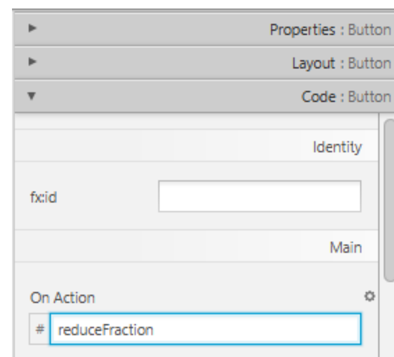


Wir werden bald eine Ereignisbehandlungsmethode erstellen, um auf das Betätigen des Schalters regieren zu können. Darin werden wir auf GUI-Komponenten Bezug nehmen. Um dies zu ermöglichen, erhalten die betroffenen **TextField**-Komponenten nun eine Kennung, die später als Feldname verwendet wird. Wir markieren das obere Textfeld und vergeben über das **Code**-Segment der **Inspector**-Zone `tfZaehler` als **fx:id**:



Analog vergeben wir die Kennung `tfNenner` an das untere Textfeld.

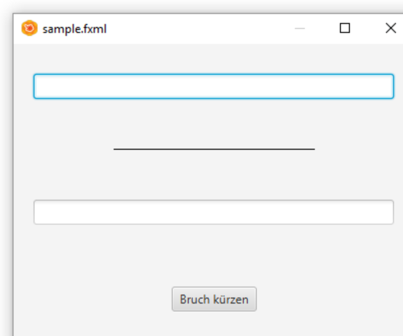
Im fertigen Programm soll durch einen Mausklick auf den Befehlsschalter eine Ereignisbehandlungsmethode namens `reduceFraction()` gestartet werden. Wir markieren das **Button**-Objekt und tragen im **Code**-Segment der **Inspector**-Zone den Namen der (noch nicht vorhandenen) Methode in das Feld **On Action** ein:



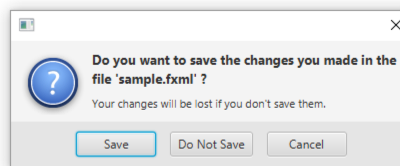
Wir fordern eine Vorschau des momentanen Entwicklungsstands unseres Programms über den Menübefehl

Preview > Show Preview in Window

an:



Auch bei selbständigen Scene Builder - Einsatz klappt die Kooperation mit IntelliJ. Beim Verlassen des Scene Builders sichert man die Änderungen,



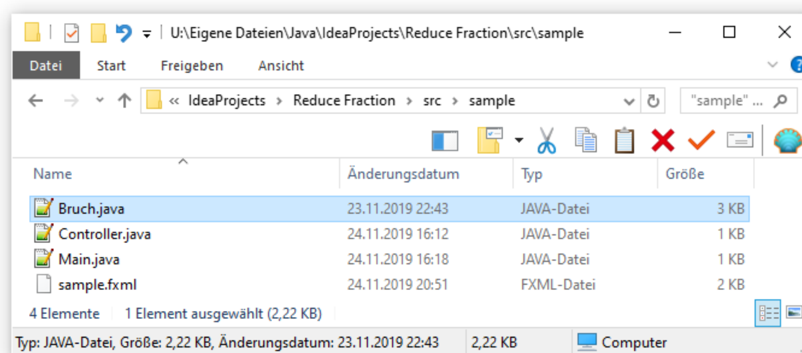
und IntelliJ übernimmt nach einem Mausklick auf das Editorfenster den aktuellen Stand.

4.9.3 Klasse `Bruch` einbinden

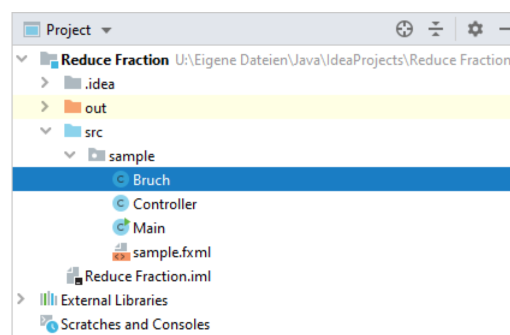
Aus den Benutzereingaben in die Textfelder des oben entworfenen Anwendungsfensters soll ein Objekt unserer Klasse `Bruch` entstehen, das bei einem Mausklick auf den Schalter mit dem Kürzen beauftragt wird. Leider stellen sich nun Designschwächen der Klasse `Bruch` heraus, die am einfachsten zu umgehen sind, indem wir den *Quellcode* der Klasse `Bruch` in das aktuelle Projekt aufnehmen und dann modifizieren, was generell keine gute Idee ist. Kopieren Sie mit dem Windows-Explorer (also mit den Mitteln des Betriebssystems) aus dem Ordner

…\BspUeb\Klassen und Objekte\Bruch\B3 (mit Konstruktoren)

die Datei **Bruch.java** in den Ordner mit den Quellcodedateien des Pakets `sample` (z. B. **...\Reduce Fraction\src\sample**):



IntelliJ nimmt nach einem Mausklick auf das **Project**-Fenster den neuen Stand zur Kenntnis:



Die Klasse `Bruch` wird nach dem Öffnen aus zwei Gründen als fehlerhaft markiert:

- Sie verwendet die nicht auffindbare Klasse `Simput`
- Sie gehört nicht zum Paket `sample`.

In **Bruch.java** wird die Klasse `Simput` genutzt, die über den Klassenpfad des aktuellen Projekts nicht auffindbar ist. Wir beseitigen den Fehler, indem wir die im aktuellen Projekt überflüssige `Bruch`-Methode `frage()` entfernen, wo die Klasse `Simput` zur Interaktion mit dem Benutzer im Rahmen einer Konsolenanwendung verwendet wird.

Man kann die Unbequemlichkeit bei der Wiederverwendung der Klasse `Bruch` als Indiz für einen Verstoß gegen das im Abschnitt 4.1.1.1 angesprochene *Prinzip einer einzigen Verantwortung* (***S**ingle **R**esponsibility **P**rinciple*, SRP) interpretieren. Die Klasse `Bruch` sollte sich auf die Kernkompetenzen von Brüchen (z. B. Initialisieren, Kürzen, Addieren) beschränken und die Benutzerinteraktion anderen Klassen überlassen. Nachdem die Klasse `Bruch` mehrfach als positives Beispiel zur Erläuterung von objektorientierten Techniken gedient hat, taugt sie nun Negativbeispiel und konkretisiert die Warnung aus Abschnitt 4.1.1.1, dass multifunktionale Klassen zu stärkeren Abhängigkeiten von anderen Klassen tendieren, wobei die Wahrscheinlichkeit einer erfolgreichen Wiederverwendung sinkt.

Weil sich das per Assistentenhilfe angelegte JavaFX-Projekt in einem Paket namens `sample` befindet, muss auch die Klasse `Bruch` in dieses Paket aufgenommen werden. Dazu ist am Anfang des Quellcodes die folgende Zeile einzufügen:

```
package sample;
```

Danch müsste die letzte Kritik von IntelliJ an der Klasse `Bruch` behoben sein.

4.9.4 Controller-Klasse vervollständigen

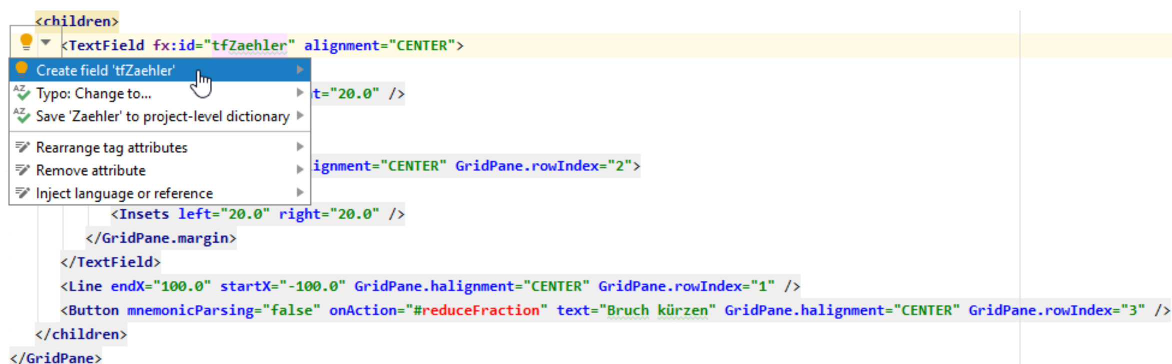
Der IntelliJ-Assistent für JavaFX-Projekte hat eine Klasse namens `Controller` angelegt (siehe **src**-Ordner im **Project**-Fenster), die für die Behandlung der von Benutzer (z. B. durch Mausklicks) ausgelösten Ereignisse zuständig ist. Aktuell sind allerdings noch keine entsprechenden Kompetenzen vorhanden:

```
package sample;

public class Controller {
}
```

Die Klasse `Controller` muss mit ihren vollqualifizierten Namen (inkl. Paketname) im Wurzelelement der FXML-Datei **sample.fxml** mit der GUI-Deklaration eingetragen werden. Diese Arbeit hat der Assistent für neue JavaFX-Projekte für uns erledigt:

```
<GridPane alignment="center" hgap="10" prefHeight="275.0" prefWidth="300.0"
xmlns:fx="http://javafx.com/fxml/1" xmlns="http://javafx.com/javafx/11.0.1"
fx:controller="sample.Controller">
. . .
</GridPane>
```

Wir sorgen im nächsten Schritt dafür, dass für die per FXML deklarierten Bedienelemente zugehörige Instanzvariablen in die Klasse `Controller` aufgenommen werden, damit die Bedienelemente in Ereignisbehandlungsmethoden angesprochen werden können . Dazu öffnen wir im IntelliJ-Editor das **Text**-Registerblatt zur Datei **sample.fxml**, setzen einen Mausklick auf den Variablennamen `tfZaehler` zum oberen Textfeld, öffnen das Drop-Down-Menü zur erscheinenden Glühbirne, um die Verbesserungsvorschläge der Entwicklungsumgebung zu erfahren. Das erste Angebot **Create field 'tfZaehler'** passt zu unserem aktuellen Ziel:



Nach Übernahme des ersten Vorschlags erscheint im Quellcode der Klasse `Controller` die passende Felddeklaration mit dem Datentyp **TextField**:

```
public class Controller {
    public TextField tfZaehler;
}
```

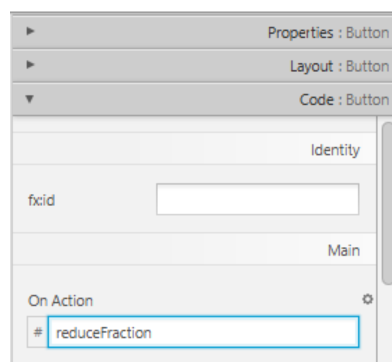
Der an einem roten Rahmen erkennbare Refaktorierungs-Modus gibt uns Gelegenheit, den vorgeschlagenen Datentyp zu ändern. Weil das in unserer gegenwärtigen Lage nicht erforderlich ist, beenden wir das Refaktorieren per **Enter**-Taste. Analog ergänzen wir die **TextField**-Variable zum zweiten Textfeld.

IntelliJ platziert neben die Felddeklarationen jeweils einen Link zum korrespondierenden Element der FXML-Datei:

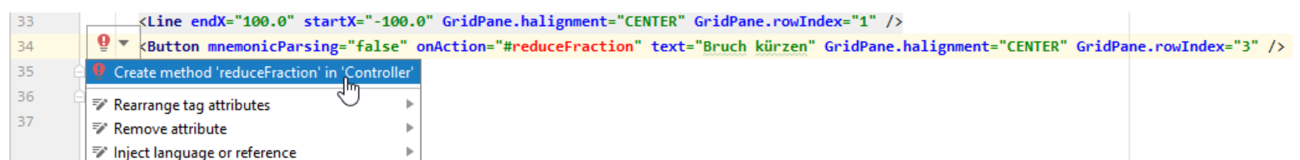
```
5   public class Controller {
6       public TextField tfZaehler;
7       public TextField tfNenner;
8   }
```

Mit der Schutzstufe **public** für die Felder verstößt IntelliJ gegen das Prinzip der Datenkapselung. Im weiteren Verlauf des Abschnitts werden wir uns um dieses Thema kümmern.

Aufgrund einer per Scene Builder vorgenommenen Konfiguration zum Befehlsschalter (vgl. Abschnitt 4.9.2)



befindet sich in der Datei **sample.fxml** die **onAction** - Ereignisbehandlungsmethode `reduceFraction()` als noch nicht realisierte Ankündigung. Wenn wir die Einfügemarke auf den rot dargestellten Methodennamen setzen und dann die Tastenkombination **Alt + Enter** drücken oder das Drop-Down - Menü zur roten Glühbirne öffnen, bietet IntelliJ an, die noch fehlende Methode in der Klasse `Controller` zu erstellen:



Wir akzeptieren das Angebot und beenden per **Enter** die von IntelliJ angeregten Refaktorierungen:

```
public void reduceFraction(ActionEvent actionEvent) {
}
```

Wir vervollständigen die Methode und verzichten dabei auf jede Absicherung gegen fehlerhafte Eingaben:[1]

[1] Die dazu sinnvollerweise zu verwendende Technik der Ausnahmebehandlung steht uns noch nicht zur Verfügung.

```
public void reduceFraction(ActionEvent actionEvent) {
   b.setzeZaehler(Integer.parseInt(tfZaehler.getText()));
   b.setzeNenner(Integer.parseInt(tfNenner.getText()));
   b.kuerze();
   tfZaehler.setText(Integer.toString(b.gibZaehler()));
   tfNenner.setText(Integer.toString(b.gibNenner()));
}
```

Das in der Methode benötigte `Bruch`-Objekt wird in der Klasse `Controller` über eine Felddeklaration mit Initialisierung bereitgestellt:

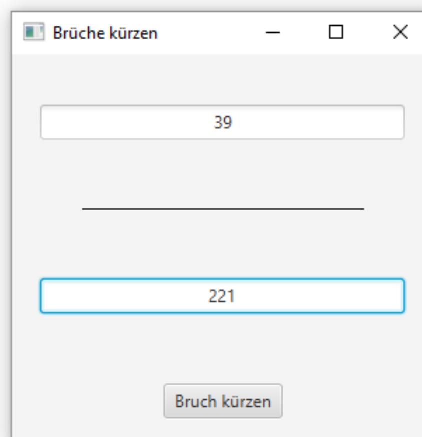
```
private Bruch b = new Bruch();
```

Zähler und Nenner dieses `Bruch`-Objekts werden auf altgewohnte Weise über die Zugriffsmethoden `setzeZaehler()` und `setzeNenner()` der Klasse `Bruch` mit Werten versorgt, die bei den **TextField**-Objekten mit der Methode **getText()** erfragt werden. Die gekürzten Werte des Bruchs wandern in umgekehrter Richtung zurück zu den **TextField**-Objekten.

In der Methode **start()** der Klasse `Main` ist nur eine triviale Änderung erforderlich, die für eine sinnvolle Titelzeilenbeschriftung sorgt:

```
primaryStage.setTitle("Brüche kürzen");
```

Das Programm ist nun funktionstüchtig:[1]



Dass IntelliJ in der Klasse `Controller` die Instanzvariablen zu den per FXML-deklarierten Bedienelementen mit der Schutzstufe **public** angelegt hat, missfällt Ihnen vermutlich. Wenn wir für die angemessene Datenkapselung sorgen,

```
private TextField tfZaehler;
private TextField tfNenner;
```

führt ein Klick auf den Befehlsschalter allerdings zu einem Ausnahmefehler statt zum gewünschten Verhalten. Wir müssen mit der Annotation[2] **@FXML** dafür sorgen, dass auch über *private* Instanzvariablen die GUI-Komponenten angesprochen werden können, die (per Scene Builder) in der FXML-Datei eine Kennung (**fx.id**) erhalten haben:

---

[1] Im **Run**-Fenster von IntelliJ erscheint beim Anwendungsstart eine Warnung, weil in der vom Assistenten erstellten FXML-Datei das JavaFX-API 11.0.1 avisiert, zum Starten aber das OpenJFX-SDK 8.0.232 verwendet wird:

```
WARNING: Loading FXML document with JavaFX API of version 11.0.1 by JavaFX runtime of version
8.0.232-ojdkbuild
```

Die Warnung unterbleibt, wenn in der FXML-Datei eine kompatible JavaFX-API - Version angegeben wird:

```
xmlns="http://javafx.com/javafx/8.0"
```

[2] Annotationen werden im Kapitel 7 zusammen mit den Schnittstellen behandelt.

```
@FXML
private TextField tfZaehler;
@FXML
private TextField tfNenner;
```

Damit die Annotation FXML bekannt wird, sorgt man mit IntelliJ-Hilfe für das Importieren dieser Klasse:

```
import javafx.fxml.FXML;
```

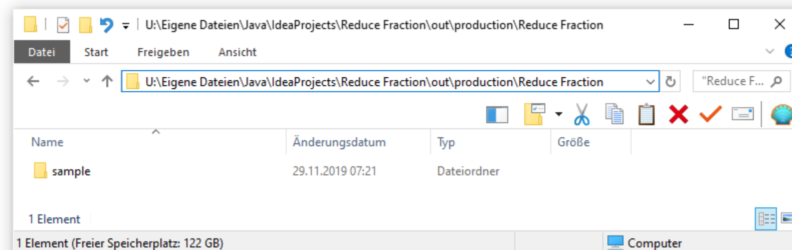
Das mit dem OpenJDK/OpenJFX 8 entwickelte IntelliJ-Projekt `Reduce Fraction` ist im folgenden Ordner zu finden

...\BspUeb\JavaFX\Reduce Fraction\Reduce Fraction mit Java 8

4.9.5 Programmstart

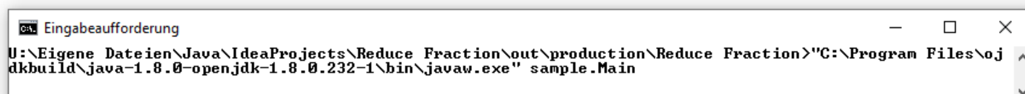
Ist die im Abschnitt 1.2.1 beschriebene OpenJDK 8 - Installation ausgeführt worden, kann das Programm außerhalb der Entwicklungsumgebung z. B. so gestartet werden:

- Konsolenfester öffnen und auf das Verzeichnis positionieren, das den Paketordner mit den **class**-Dateien enthält, z. B.:

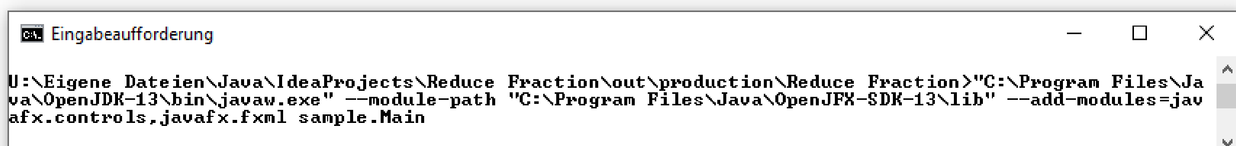


- Weil JavaFX (alias OpenJFX) in der OpenJDK 8 - Installation enthalten ist, kann das Programm folgendermaßen per **javaw.exe** gestartet werden, wobei der vollqualifizierte Name der Startklasse (inklusive Paketname) anzugeben ist:

  Wenn sich **javaw.exe** *nicht* im Windows-Pfad für ausführbare Dateien befindet, muss der Dateiname inklusive Pfadangabe geschrieben werden, z. B.:



Das mit dem OpenJDK/OpenJFX 8 entwickelte Programm kann auch mit der JVM im OpenJDK 13 ausgeführt werden, wenn zusätzlich das OpenJFX-SDK 13 installiert worden ist. Im Startkommando ist ein Modulpfad anzugeben (vgl. Abschnitt 6.2), z. B.:



Das Startverfahren lässt sich für Endbenutzer unter Windows z. B. durch die Erstellung einer Verknüpfung vereinfachen (siehe Abschnitt 1.2.3).

Im Abschnitt 6.1.3.5 wird beschrieben, wie man das Programm in eine **jar**-Datei verpackt, sodass es leicht verteilt und auf einem Rechner mit dem OpenJDK 8 und OpenJFX 8 per Doppelklick gestartet werden kann.

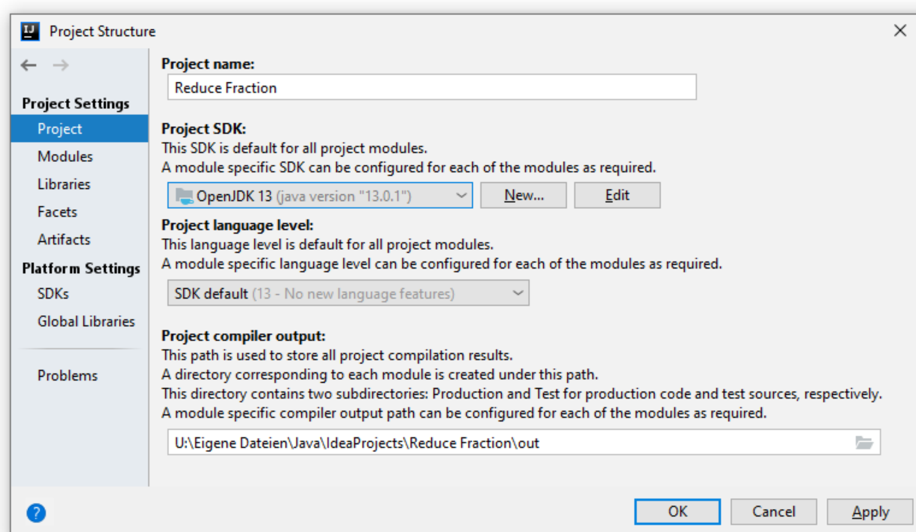
Ein professionelles Java-Programm wird ohnehin inklusive eigener JVM ausgeliefert, sodass dem Endbenutzer komplizierte Erläuterungen zum Starten unter verschiedenen Java-Versionen erspart bleiben.

4.9.6 OpenJDK/OpenJFX 13

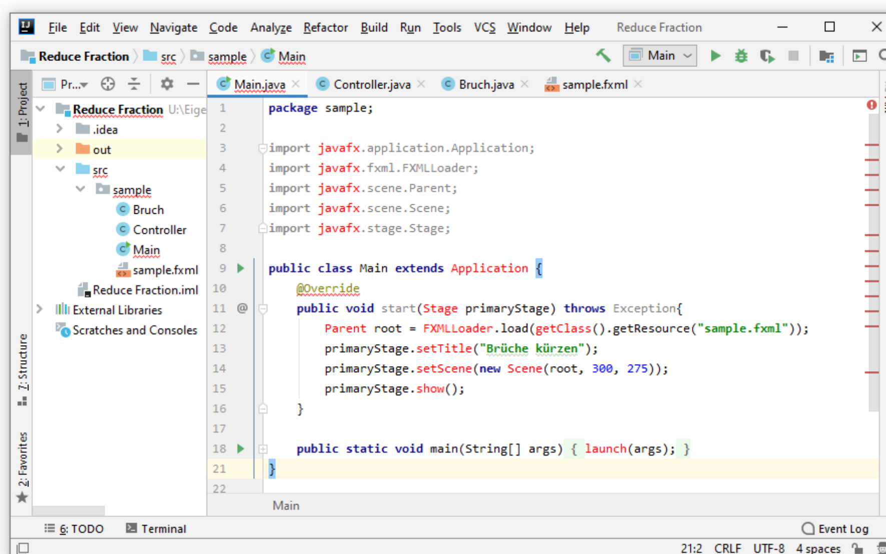
Um die JavaFX-Entwicklung mit Java 13 zu üben, öffnen wir über den Menübefehl

File > Project Structure

oder per Mausklick auf das Symbol den Dialog **Project Structure** und wechseln dort nötigenfalls zum Abschnitt **Project**, um dort das **OpenJDK 13** als **Project SDK** einzustellen:



Weil im OpenJDK 13 die JavaFX-Bibliothek (die zugehörigen Pakete) fehlen, zeigt der Quellcode-Editor reichlich rote Tinte:



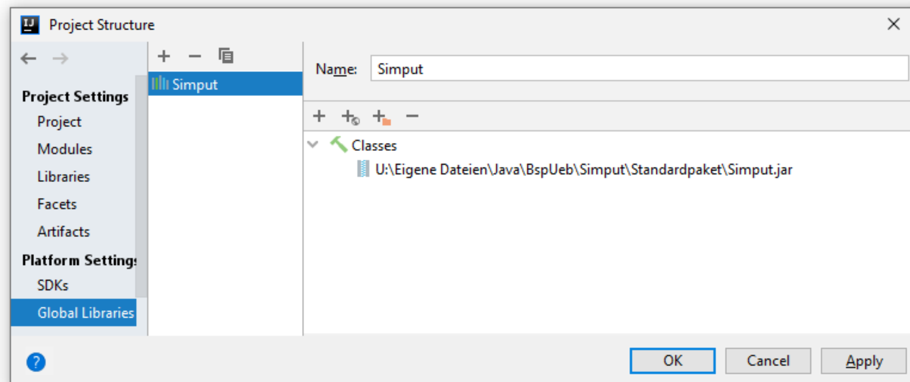
Unter der Voraussetzung, dass gemäß Abschnitt 2.5 das OpenJFX 13 in den Ordner

C:\Program Files\Java\OpenJFX-SDK-13

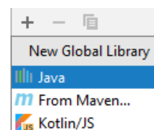
installiert worden ist, lässt sich das Problem lösen, ...

- indem eine globale Bibliothek mit dem OpenJFX 13 eingerichtet wird
- und in die Abhängigkeiten des Projekts aufgenommen wird.

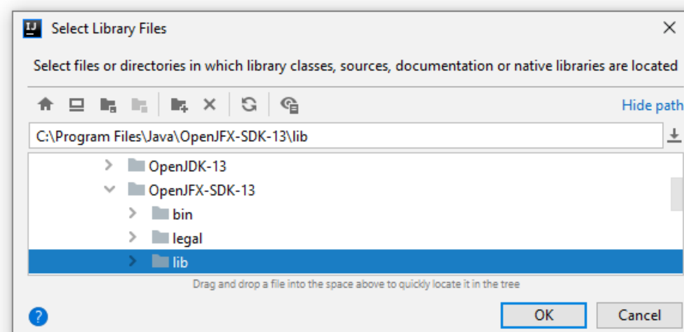
Im Abschnitt **Global Libraries** des Dialogs **Project Structure** klicken wir auf den Schalter **+** über der bereits im Abschnitt 3.4.2 eingerichteten Bibliothek mit der Klasse `Simput`,



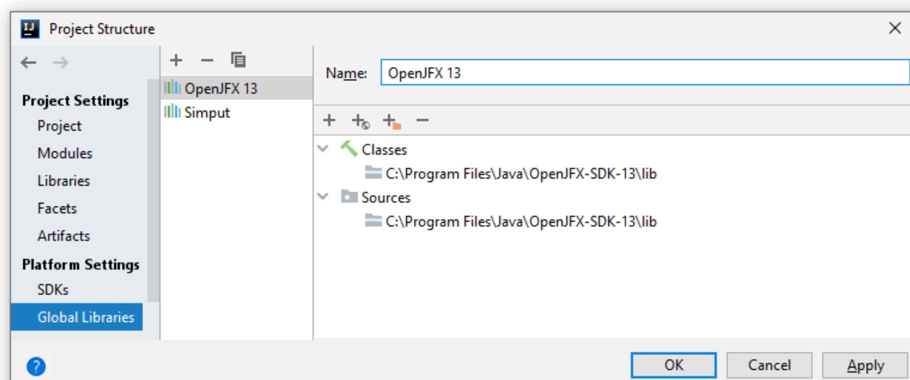
entscheiden uns für die Kategorie **Java**



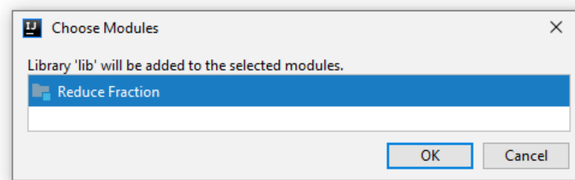
und wählen anschließend den Unterordner **lib**:



Damit die nun definierte, passend umbenannte globale Bibliothek **`OpenJFX 13`**



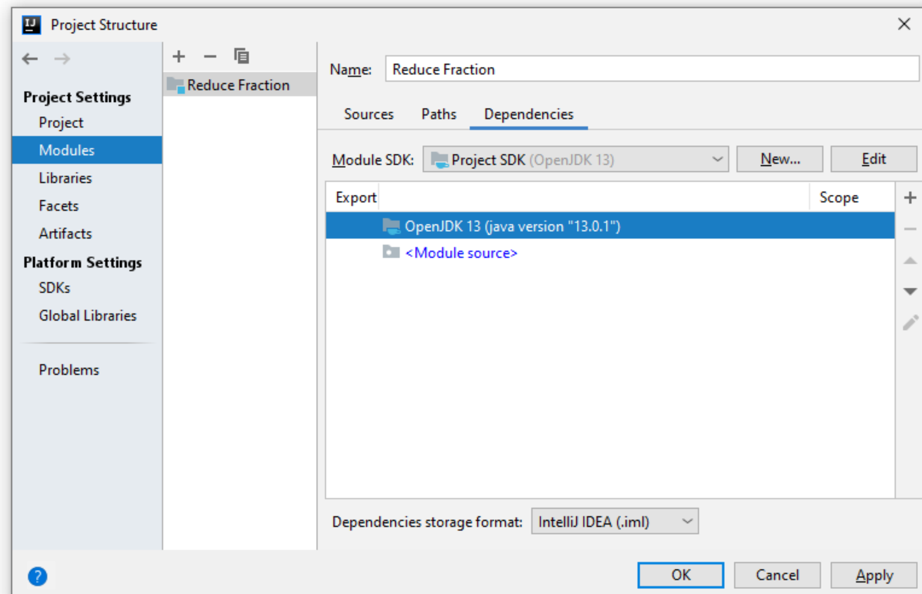
in einem konkreten Projekt bzw. Modul benutzt werden kann, muss sie in die Liste der Abhängigkeiten des Moduls aufgenommen werden. Für das aktuell geöffnete Projekt kann dies bequem per Mausklick geschehen, z. B.:



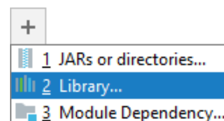
Bei einem anderen Projekt öffnet man nach

File > Project Structure > Modules

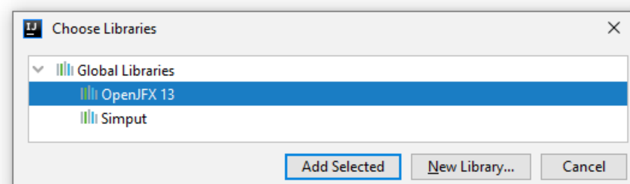
im folgenden Fenster für das meist einzige vorhandene IntelliJ-Modul die Registerkarte **Dependencies**:



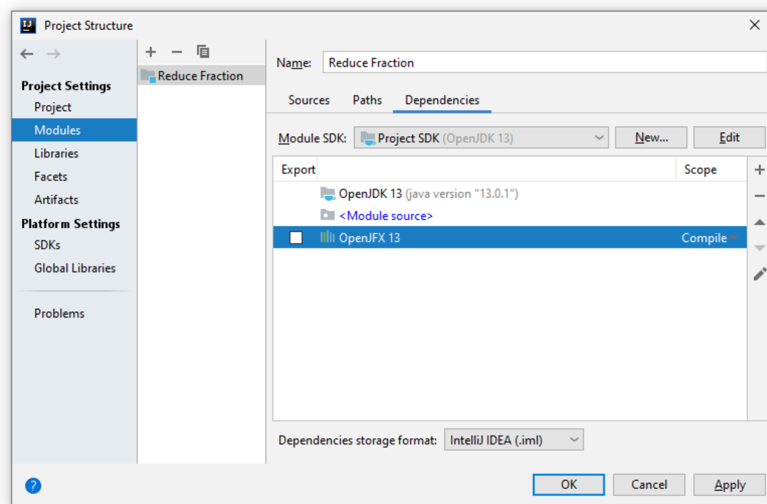
Nach einem Klick auf den Schalter + am rechten Fensterrand (!) entscheidet man sich für die Kategorie **Library**



und wählt dann per **Add Selected** die globale Bibliothek `OpenJFX 13`,

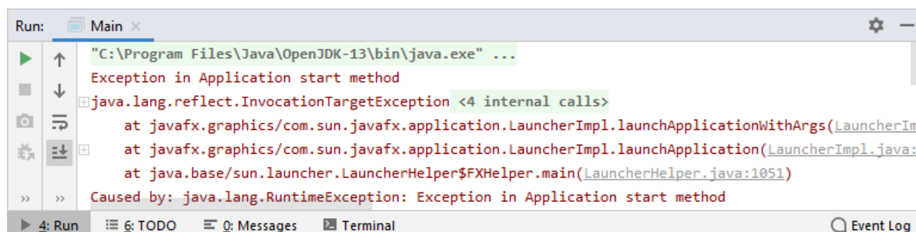


die anschließend in der Liste der Abhängigkeiten erscheint:

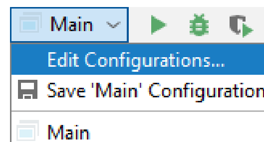


Nach dem Quittieren mit **OK** und einer Neuindizierung des Projekts ist die rote Tinte verschwunden.

Ein Startversuch scheitert aber trotzdem:



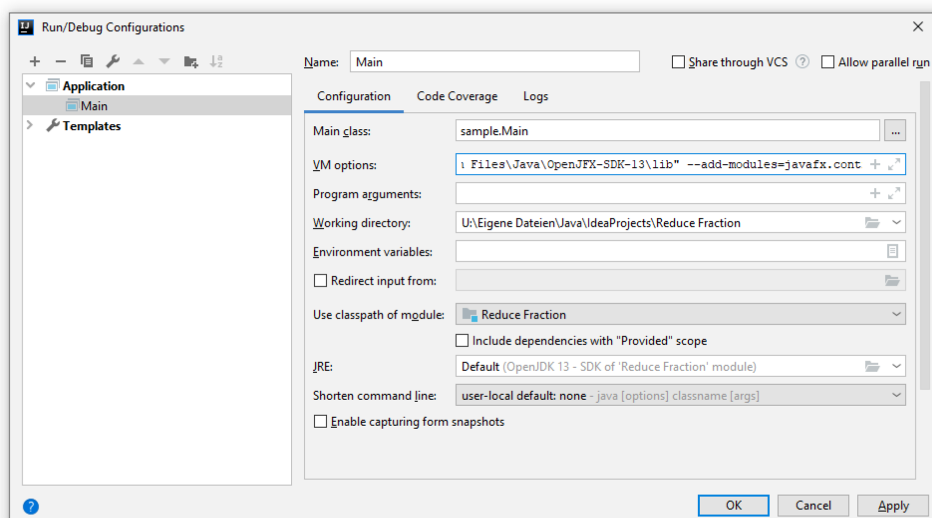
Zur Lösung des Problems öffnet man über



das Fenster **Run/Debug Configurations** und trägt dort die folgenden **VM options**

```
--module-path "C:\Program Files\Java\OpenJFX-SDK-13\lib"
--add-modules=javafx.controls,javafx.fxml
```

ein:



Wie das Programm unter Verwendung des OpenJDK/OpenJFX 13 außerhalb von IntelliJ ausgeführt werden kann, wurde schon im Abschnitt 4.9.5 erläutert.

Das mit dem OpenJDK/OpenJFX 13 entwickelte IntelliJ-Projekt `Reduce Fraction` ist im folgenden Ordner zu finden

...\BspUeb\JavaFX\Reduce Fraction\Reduce Fraction mit Java 13

4.10 Übungsaufgaben zum Kapitel 4

1) Welche der folgenden Aussagen über Variablen sind richtig bzw. falsch?

1. Die Instanzvariablen einer Klasse werden meist als **privat** deklariert.
2. Durch Datenkapselung (Schutzstufe **private**) werden die Objekte einer Klasse darin gehindert, Instanzvariablen anderer Objekte derselben Klasse zu verändern.
3. Bei einer Felddeklaration ohne Zugriffsmodifikator gilt in Java die Schutzstufe **private**.
4. Referenzvariablen werden automatisch mit den Wert **null** initialisiert.

2) Wie erhält man eine Instanzvariable mit uneingeschränktem Zugriff für die Methoden der eigenen Klasse, die von Methoden *fremder* Klassen zwar gelesen, aber nicht geändert werden kann?

3) Welche der folgenden Aussagen über Methoden sind richtig bzw. falsch?

1. Methoden müssen generell als **public** deklariert werden, denn sie gehören zur Schnittstelle einer Klasse.
2. Ändert man den Rückgabetyp einer Methode, dann ändert sich auch ihre Signatur.
3. Beim Methodenaufruf müssen die Datentypen der Aktualparameter exakt mit den Datentypen der Formalparameter übereinstimmen.
4. Lokale Variablen einer Methode überdecken gleichnamige Instanzvariablen.

4) Was halten Sie von der folgenden Variante der `Bruch`-Methode `setzeNenner()`?

```
public boolean setzeNenner(int n) {
   if (n != 0)
      nenner = n;
   else
      return false;
}
```

5) Könnten in *einer* `Bruch`-Klassendefinition die beiden folgenden `addiere()` - Methoden koexistieren, die sich durch die Reihenfolge der Parameter für Zähler und Nenner des zu addierenden Bruchs unterscheiden?

```
public void addiere(int zpar, int npar) {
   if (npar == 0) return;
   zaehler = zaehler*npar + zpar*nenner;
   nenner = nenner*npar;
   kuerze();
}

public void addiere(int npar, int zpar) {
   if (npar == 0) return;
   zaehler = zaehler*npar + zpar*nenner;
   nenner = nenner*npar;
   kuerze();
}
```

6) Erstellen Sie die Klassen `Time` und `Duration` zur Verwaltung von Zeitpunkten (der Einfachheit halber nur innerhalb eines Tages) und Zeitintervallen (von beliebiger Länge).

Neben der Beschäftigung mit syntaktischen Details der Klassendefinition ist es in Ihrer jetzigen Lernphase wichtig, den Entwurf von Klassen zu üben. Dazu bieten die Klassen `Time` und Duration eine geeignete, nicht allzu komplizierte Aufgabe. In der Praxis sollten Sie beim Umgang mit Zeitpunkten und Zeitintervallen das in Java 8 erheblich renovierte Date/Time - API der Standardbibliothek verwenden.[1]

Die beiden Klassen `Time` und `Duration` sollen über Instanzvariablen für Stunden, Minuten und Sekunden sowie über folgende Methoden verfügen:

- Konstruktoren mit unterschiedlichen Parameterausstattungen
- Methoden zum Abfragen bzw. Setzen von Stunden, Minuten und Sekunden
  Beim Versuch zur Vereinbarung eines irregulären Werts (z. B. Uhrzeit mit einer Stundenangabe größer als 23) sollte die betroffene Methode die Ausführung verweigern und den Rückgabewert **false** liefern.
- Eine Methode mit dem Namen `toString()` und dem Rückgabetyp **String**, die zu einem `Time`- bzw. `Duration`-Objekt eine gut lesbare Zeichenfolgenrepräsentation liefert[2]
  Tipp: In der Klasse **String** steht die statische Methode **format()** zur Verfügung, die analog zur **PrintStream**-Methode **printf()** (alias **format()**, siehe Abschnitt 3.2.2) arbeitet und eine formatierte **String**-Rückgabe liefert. Im folgenden Beispiel enthält die Formatierungszeichenfolge den Platzhalter **%02d** für eine ganze Zahl, die bei Werten kleiner als 10 mit einer führenden Null ausgestattet wird:
  ```
  return String.format("%02d:%02d:%02d Uhr", hours, minutes, seconds);
  ```

In der Klasse `Time` sollen außerdem Methoden mit folgenden Leistungen vorhanden sein:

- `getDistenceTo()`
  Berechnung der Zeitdistanz zu einem anderen, als Parameter übergebenen Zeitpunkt am selben oder am folgenden Tag
- `addDuration()`
  Addieren eines als Parameter übergebenen Zeitintervalls zu einem Zeitpunkt mit einer neuen Uhrzeit als Ergebnis

Erstellen Sie eine Testklasse zur Demonstration der `Time`-Methoden `getDistenceTo()` und `addDuration()`. Ein Programmlauf soll z. B. folgende Ausgaben produzieren:

a) Distanz zwischen zwei Zeitpunkten ermitteln:
```
Von 17:34:55 Uhr bis 12:24:12 Uhr vergehen    18:49:17 h:m:s.
```

b) Zeitdauer zu einem Zeitpunkt addieren:
```
20:23:00 h:m:s nach 17:34:55 Uhr sind es 13:57:55 Uhr.
```

---

1 Siehe z. B. https://docs.oracle.com/javase/tutorial/datetime/index.html

2 Dabei wird die **toString()** - Methode der Basisklasse **Object** überschrieben.

7) Lokalisieren Sie bitte in der folgenden Abbildung mit einer Kurzform der Klasse `Bruch`

```
public class Bruch {
    private int zaehler;
    private int nenner = 1;
    private String etikett = "";
    static private int anzahl;

    public Bruch(int z, int n, String eti) {
        setzeZaehler(z);
        setzeNenner(n);
        setzeEtikett(eti);
        anzahl++;
    }
    public Bruch() {anzahl++;}

    public void setzeZaehler(int z) {zaehler = z;}
    public boolean setzeNenner(int n) {
        if (n != 0)    {
            nenner = n;
            return true;
        } else
            return false;
    }
    public void setzeEtikett(String eti) {
        int rind = eti.length();
        if (rind > 40)
            rind = 40;
        etikett = eti.substring(0, rind);
    }
    public int gibZaehler() {return zaehler;}
    public int gibNenner() {return nenner;}
    public String gibEtikett() {return etikett;}

    public void kuerze() {
      . . .
    }
    public void addiere(Bruch b) {
        zaehler = zaehler*b.nenner + b.zaehler*nenner;
        nenner = nenner*b.nenner;
        kuerze();
    }
    public boolean frage() {
      . . .
    }

    public void zeige() {
      . . .
    }
    public void dupliziere(Bruch bc) {
        bc.zaehler = zaehler;
        bc.nenner = nenner;
        bc.etikett = etikett;
    }
    public Bruch klone() {
        return new Bruch(zaehler, nenner, etikett);
    }

    static public int hanz() {return anzahl;}
}

class Bruchrechnung {
    public static void main(String[] args) {
        Bruch b1 = new Bruch(890, 25, "");
        b1.zeige(); b1.kuerze(); b1.zeige();
    }
}
```

Markierungen in der Abbildung (Pfeil auf die Codezeile):

1 → `private int zaehler;`
2 → `static private int anzahl;`
3 → `public Bruch() {anzahl++;}`
4 → `int rind = eti.length();`
5 → `kuerze();`
6 → `public void dupliziere(Bruch bc) {`
7 → `public Bruch klone() {`
8 → `static public int hanz() {return anzahl;}`
9 → `Bruch b1 = new Bruch(890, 25, "");`

neun Begriffe der objektorientierten Programmierung, und tragen Sie die Positionen in die folgende Tabelle ein:

Begriff	Pos.
Definition einer Instanzmethode mit Referenzrückgabe	
Deklaration einer lokalen Variablen	
Definition einer Instanzmethode mit Referenzparameter	
Deklaration einer Instanzvariablen	
Methodenaufruf	

Begriff	Pos.
Konstruktordefinition	
Deklaration einer Klassenvariablen	
Objekterzeugung	
Definition einer Klassenmethode	

Zum Eintragen benötigen Sie nicht unbedingt eine gedruckte Variante des Manuskripts, sondern können auch das interaktive PDF-Formular in der folgenden Datei

...\BspUeb\Klassen und Objekte\Begriffe lokalisieren.pdf

benutzen.[1]

8) Erstellen Sie eine Klasse mit einer statischen Methode zur Berechnung der Fakultät über einen rekursiven Algorithmus. Erstellen Sie eine Testklasse, welche die rekursive Fakultätsmethode benutzt.

9) Die folgende Aufgabe eignet sich nur für Leser(innen) mit Grundkenntnissen in linearer Algebra: Erstellen Sie eine Klasse für Vektoren im $\mathbb{R}^2$, die mindestens über Methoden mit den folgenden Leistungen verfügt:

- **Länge** ermitteln

  Der Betrag eines Vektors $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ ist definiert durch:

  $$|x| := \sqrt{x_1^2 + x_2^2}$$

  Verwenden Sie die Klassenmethode **Math.sqrt()**, um die Quadratwurzel aus einer **double**-Zahl zu berechnen.

- Vektor auf Länge Eins **normieren**
  Dazu dividiert man beide Komponenten durch die Länge des Vektors, denn mit

  $\tilde{x} := (\tilde{x}_1, \tilde{x}_2)$ sowie $\tilde{x}_1 := \frac{x_1}{\sqrt{x_1^2 + x_2^2}}$ und $\tilde{x}_2 := \frac{x_2}{\sqrt{x_1^2 + x_2^2}}$ gilt:

  $$|\tilde{x}| = \sqrt{\tilde{x}_1^2 + \tilde{x}_2^2} = \sqrt{\left(\frac{x_1}{\sqrt{x_1^2 + x_2^2}}\right)^2 + \left(\frac{x_2}{\sqrt{x_1^2 + x_2^2}}\right)^2} = \sqrt{\frac{x_1^2}{x_1^2 + x_2^2} + \frac{x_2^2}{x_1^2 + x_2^2}} = 1$$

- Vektoren (komponentenweise) **addieren**

  Die Summe der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

  $$x + y := \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

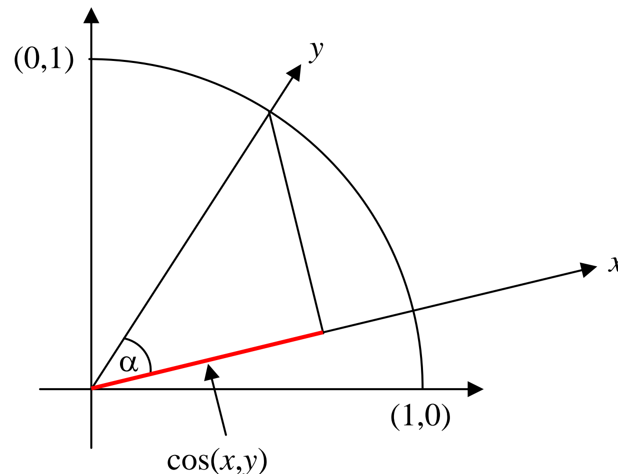
[1] Die Idee zu dieser Übungsaufgabe stammt aus Mössenböck (2003).

- **Skalarprodukt** zweier Vektoren ermitteln
  Das Skalarprodukt der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

$$x \cdot y := x_1 y_1 + x_2 y_2$$

- **Winkel** zwischen zwei Vektoren in Grad ermitteln
  Für den Kosinus des Winkels, den zwei Vektoren *x* und *y* im mathematischen Sinn (links herum) einschließen, gilt:[1]

$$\cos(x, y) = \frac{x \cdot y}{|x||y|}$$



  Um aus cos(*x*, *y*) den Winkel in Grad zu ermitteln, können Sie folgendermaßen vorgehen:
  - mit der Klassenmethode **Math.acos()** den zum Kosinus gehörigen Winkel im Bogenmaß ermitteln
  - mit der Klassenmethode **Math.toDegrees()** das Bogenmaß (*rad*) in Grad umrechnen (*deg*), wobei folgende Formel verwendet wird:

$$deg = \frac{rad}{2\pi} \cdot 360$$

- **Rotation** eines Vektors um einen bestimmten Winkelgrad
  Mit Hilfe der Rotationsmatrix

$$D := \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

  kann der Vektor *x* um den Winkel α (im Bogenmaß!) gedreht werden:

$$x' = D\,x = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \cos(\alpha)\,x_1 - \sin(\alpha)\,x_2 \\ \sin(\alpha)\,x_1 + \cos(\alpha)\,x_2 \end{pmatrix}$$

  Zur Berechnung der trigonometrischen Funktionen stehen die Klassenmethoden **Math.cos()** und **Math.sin()** bereit. Für die Umwandlung von Winkelgraden (*deg*) in das von **cos()** und **sin()** benötigte Bogenmaß (*rad*) steht die Methode **Math.toRadians()** bereit, die mit folgender Formel arbeitet:

$$rad = \frac{deg}{360} \cdot 2\pi$$

---

[1] Dies folgt aus dem Additionstheorem für den Kosinus.

Erstellen Sie ein Demonstrationsprogramm, das Ihre Vektor-Klasse verwendet und ungefähr den folgenden Programmablauf ermöglicht (Eingabe fett):

```
Vektor 1:                (1,00; 0,00)
Vektor 2:                (1,00; 1,00)

Laenge von Vektor 1:     1,00
Laenge von Vektor 2:     1,41

Winkel:                  45,00 Grad

Um wie viel Grad soll Vektor 2 gedreht werden: 45

Neuer Vektor 2           (0,00; 1,41)
Neuer Vektor 2 normiert  (0,00; 1,00)

Summe der Vektoren       (1,00; 1,00)
```

5 Elementare Klassen

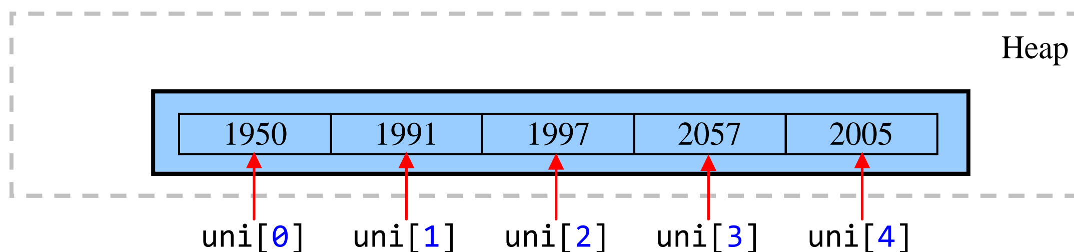
In diesem Kapitel wird gewissermaßen die objektorientierte Fortsetzung der elementaren Sprachelemente aus Kapitel 3 präsentiert. Es werden wichtige Bausteine für Programme behandelt, die in Java als *Klassen* realisiert sind (Arrays und Zeichenketten).Außerdem werden einige spezielle Datentypen vorgestellt (Verpackungsklassen und Aufzählungstypen).

Die Themen der folgenden Abschnitte sind:

- Arrays als Container für eine feste Anzahl von Elementen desselben Datentyps
- Klassen zur Verwaltung von Zeichenketten (**String**, **StringBuilder**, **StringBuffer**)
- Verpackungsklassen zur Integration primitiver Datentypen in das Java-Klassensystem
- Aufzählungstypen (Enumerationen)

5.1 Arrays

Ein Array ist ein Objekt, das eine feste Anzahl von Elementen desselben Datentyps als Instanzvariablen enthält.[1] In der folgenden Abbildung ist ein Array namens `uni` mit 5 Elementen vom Typ **int** zu sehen:

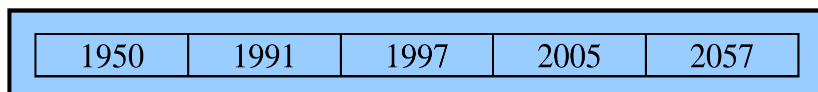


Beim Zugriff auf ein einzelnes Element gibt man nach dem Arraynamen den durch eckige Klammern begrenzten Index an, wobei die Nummerierung mit 0 beginnt und bei n Elementen folglich mit n - 1 endet.[2]

Man kann aber auch den kompletten Array ansprechen und z. B. als Aktualparameter an eine Methode übergeben. Aufgrund der Anweisung

```
Arrays.sort(uni);
```

werden die Elemente des Arrays `uni` durch die statische Methode **sort()** der Klasse **Arrays** (im Paket **java.util**) der Größe nach sortiert (vgl. Abschnitt 5.1.6), was zum folgenden Ergebnis führt:



Neben den Elementen enthält ein Array-Objekt noch Verwaltungsdaten (z. B. die finalisierte und öffentliche Instanzvariable **length** mit der Anzahl der Elemente).

Im Vergleich zur Verwendung einer entsprechenden Anzahl von Einzelvariablen ermöglichen Arrays eine gravierende Vereinfachung der Programmierung:

---

[1] Arrays werden in vielen Programmiersprachen auch *Felder* genannt. In Java bezeichnet man jedoch recht einheitlich die Instanz- oder Klassenvariablen als *Felder*, sodass der Name hier nicht mehr zur Verfügung steht.

[2] Technisch gesehen liegt ein Array-Zugriffs*ausdruck* mit dem Operator [] vor.

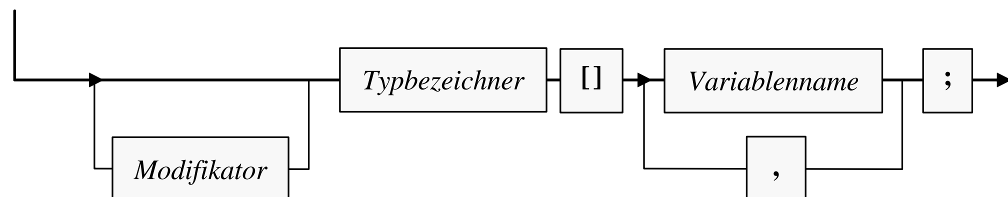
- Weil der Index auch durch einen *Ausdruck* (z. B. durch eine Variable) geliefert werden kann, sind Arrays im Zusammenhang mit den Wiederholungsanweisungen äußerst praktisch.
- Man kann die *gemeinsame* Verarbeitung *aller* Elemente (z. B. bei der Ausgabe in eine Datei) per Methodenaufruf mit Array-Aktualparameter veranlassen.
- Viele Algorithmen arbeiten mit Vektoren und Matrizen. Zur Modellierung dieser mathematischen Objekte sind Arrays unverzichtbar.

Wir beschäftigen uns erst jetzt mit den zur Grundausstattung praktisch jeder Programmiersprache gehörenden Arrays, weil diese Datentypen in Java als *Klassen* realisiert sind und folglich zunächst entsprechende Grundlagen zu erarbeiten waren.[1] Nach dem Einstieg mit *eindimensionalen* Arrays behandeln wir auch den mehrdimensionalen Fall.

5.1.1 Array-Variablen deklarieren

Im Vergleich zu der bisher bekannten Variablendeklaration (ohne Initialisierung) ist bei Array-Variablen hinter dem Typbezeichner zusätzlich ein Paar eckiger Klammern anzugeben:[2]

Deklaration einer Array-Variablen



Welche Modifikatoren zulässig bzw. erforderlich sind, hängt davon, ob die Array-Variable zu einer Methode, zu einer Klasse oder zu einer Instanz gehört. Die Array-Variable `uni` aus dem einleitend beschriebenen Beispiel gehört zu einer Methode (siehe unten) und ist folgendermaßen zu deklarieren:

```
int[] uni;
```

Bei der Deklaration entsteht nur eine Referenzvariable, jedoch noch kein Array-Objekt. Daher ist auch keine Array-Größe (Anzahl der Elemente) anzugeben.

Einer Array-Referenzvariablen kann als Wert die Adresse eines Arrays mit Elementen vom vereinbarten Typ oder das Referenzliteral **null** (Zeiger auf nichts) zugewiesen werden.

5.1.2 Array-Objekte erzeugen

Mit Hilfe des **new**-Operators erzeugt man ein Array-Objekt mit einem bestimmten Elementtyp und einer bestimmten Anzahl von Elementen. In der folgenden Anweisung entsteht ein Array mit 5 **int**-Elementen, und seine Adresse landet in der Variablen `uni`:

```
uni = new int[5];
```

Im **new**-Operanden *muss* hinter dem Datentyp zwischen eckigen Klammern die Anzahl der Elemente festgelegt werden, wobei ein beliebiger Ausdruck mit ganzzahligem Wert (≥ 0) erlaubt ist. Man

---

[1] Obwohl wir die wichtige Vererbungsbeziehung zwischen Klassen noch nicht offiziell behandelt haben, können Sie vermutlich schon den Hinweis verdauen, dass alle Array-Klassen direkt von der Urahnklasse **Object** im Paket **java.lang** abstammen.

[2] Alternativ dürfen bei der Deklaration die eckigen Klammern auch *hinter* dem Variablennamen stehen, z. B.

```
int uni[];
```

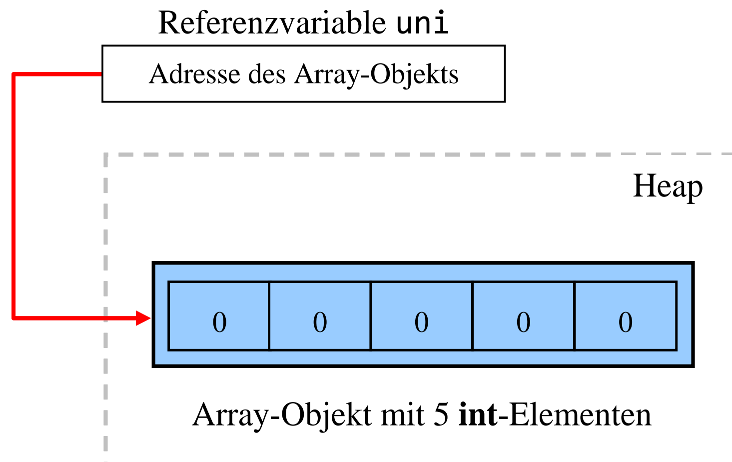
Hier wird eine Regel der Programmiersprache C unterstützt, wobei die Lesbarkeit des Quellcodes aber leidet.

kann also die Länge eines Arrays zur Laufzeit festlegen, z. B. in Abhängigkeit von einer Benutzereingabe.

Die Deklaration einer Array-Referenzvariablen *und* die Erstellung des Array-Objekts lassen sich natürlich auch in *einer* Anweisung erledigen, z. B.:

```
int[] uni = new int[5];
```

Mit der Verweisvariablen `uni` und dem referenzierten Array-Objekt auf dem Heap haben wir insgesamt die folgende Situation im Speicher:



Weil es sich bei den Array-Elementen um Instanzvariablen eines Objekts handelt, erfolgt eine automatische Null-Initialisierung nach den Regeln von Abschnitt 4.1.3. Die **int**-Elemente im Beispiel erhalten folglich den Startwert 0.

Die Anzahl der Elemente in einem Array wird begrenzt durch den größten positiven Wert des Datentyps **int** (= 2147483647).

Ein Array-Objekt wird vom Garbage Collector entsorgt, wenn im Programm keine Referenz mehr vorliegt (vgl. Abschnitt 4.4.6). Um eine Referenzvariable aktiv von einem Array-Objekt zu „entkoppeln“, kann man ihr z. B. das Referenzliteral **null** oder aber ein alternatives Referenzziel zuweisen.

Es ist auch möglich, dass mehrere Referenzvariablen auf dasselbe Array-Objekt zeigen, z. B.:

Quellcode	Ausgabe

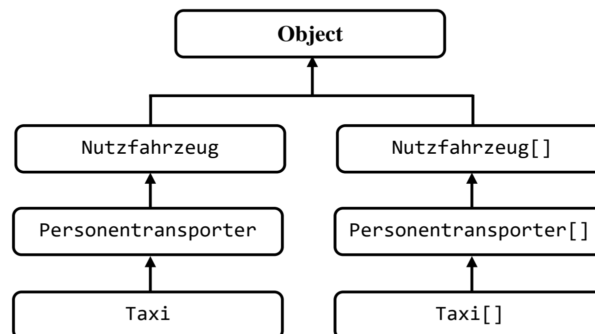
`class Prog {`
`    public static void main(String[] args) {`
`        int[] x = new int[3], y;`
`        x[0] = 1; x[1] = 2; x[2] = 3;`
`        y = x; // y zeigt nun auf dasselbe Array-Objekt wie x`
`        y[0] = 99;`
`        System.out.println(x[0]);`
`    }`
`}`
	99

5.1.3 Kovariante Einbindung von Arrays in die Klassenhierarchie

Um zu diskutieren, wie Arrays in die Java-Klassenhierarchie eingebunden sind, betrachten wir als Beispiel die folgenden Klassenhierarchie aus einem Speditionsverwaltungsprogramm:

- Die Klasse `Nutzfahrzeug` stammt direkt von **Object** ab:
  ```
  class Nutzfahrzeug {...}
  ```
- Die Klasse `Personentransporter` ist aus der Klasse `Nutzfahrzeug` abgeleitet:
  ```
  class Personentransporter extends Nutzfahrzeug {...}
  ```
- Die Klasse `Taxi` stammt von `Personentransporter` ab:
  ```
  class Taxi extends Personentransporter {...}
  ```

In der folgenden Abbildung ist zu sehen, wie die drei Klassen sowie die zugehörigen Array-Klassen `Nutzfahrzeug[]`, `Personentransporter[]` und `Taxi[]` in die Java-Klassenhierarchie eingehängt sind.



Im Kapitel 8 über Generizität werden wir diese Spezialisierungsbeziehungen zwischen Array-Klassen als **Kovarianz** bezeichnen[1] und als Design-Fehler kritisieren. Aufgrund der Kovarianz-Eigenschaft von Arrays übersetzt der Compiler nämlich z. B. die folgenden Anweisungen ohne jede Kritik:

```
Object[] arrObject = new String[5];
arrObject[0] = 13;
```

Weil **String** von **Object** abstammt, ist **Object[]** aufgrund der kovarianten Spezialisierungsbeziehungen von Array-Klassen eine Basisklasse von **String[]**, und eine Variable vom Typ einer Basisklasse kann in der objektorientierten Programmierung generell die Adresse eines Objekts aus einer abgeleiteten Klasse aufnehmen. Zur Laufzeit kommt es jedoch zu einem Ausnahmefehler vom Typ **ArrayStoreException**:

```
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer
 at Prog.main(Prog.java:5)
```

Der per

```
new String[5]
```

erzeugte Array kennt zur Laufzeit sehr wohl seinen tatsächlichen Elementtyp (**String**) und lehnt die Aufnahme eines **Integer**-Objekts ab (zu **Integer**-Objekten und Autoboxing siehe Abschnitt 5.3.2). Weil Programmierfehler nicht zur Laufzeit, sondern vom Compiler entdeckt werden sollten, ist die bei Arrays realisierte kovariante Zuweisungskompatibilität als Mangel einzuschätzen, von dem neben Java auch andere Programmiersprachen betroffen sind (z. B. C#).

---

[1] Die Abbildung illustriert das mutmaßliche Motiv für die Anwendung des statistischen Begriffs Kovarianz auf die Spezialisierungsbeziehungen von Array-Klassen: Für die „Rangreihe“ der drei Klassen und die „Rangreihe“ der zugehörigen Array-Klassen besteht tatsächlich eine perfekte Rangkorrelation.

5.1.4 Arrays verwenden

Der Zugriff auf ein Element eines Array-Objekts geschieht über eine zugehörige Referenzvariable, an deren Namen zwischen eckigen Klammern ein passender Index angehängt wird. Als Index ist ein beliebiger Ausdruck mit ganzzahligem Wert erlaubt, wobei natürlich die Feldgrenzen zu beachten sind. In der folgenden **for**-Schleife wird pro Durchgang ein zufällig gewähltes Element des **int**-Arrays inkrementiert,

```
for (i = 0; i < drl; i++)
   uni[zzg.nextInt(5)]++;
```

auf den die Referenzvariable `uni` aufgrund der in Abschnitt 5.1.2 beschriebenen Deklaration und Initialisierung zeigt:

```
int[] uni = new int[5];
```

Den Indexwert liefert die von einem Objekt der Klasse **Random** (im Paket **java.util**) ausgeführte Zufallszahlenmethode **nextInt()** mit dem Rückgabetyp **int**.

Wie in vielen anderen Programmiersprachen hat auch in Java das erste von *n* Array-Elementen die Nummer 0 und folglich das letzte die Nummer *n* - 1. Damit existiert z. B. nach der Anweisung

```
int[] uni = new int[5];
```

kein Element `uni[5]`. Ein Zugriffsversuch führt zum Laufzeitfehler vom Typ **ArrayIndexOutOf-BoundsException**, z. B.:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at Prog.main(Prog.java:4)
```

Wenn das verantwortliche Programm einen solchen Ausnahmefehler nicht behandelt (siehe Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.**), dann wird es vom Laufzeitsystem beendet. Man kann sich in Java generell darauf lassen, dass jede Überschreitung von Feldgrenzen verhindert wird, sodass es nicht zur Verletzung anderer Speicherbereiche und den entsprechenden Folgen (Absturz mit Speicherschutzverletzung, unerklärliches Programmverhalten) kommt.

Die (z. B. durch eine Benutzerentscheidung zur Laufzeit festgelegte) Länge eines Array-Objekts lässt sich über die finalisierte und öffentliche Instanzvariable **length** feststellen, z. B.:

<table>
<tr><th>Quellcode</th><th>Eingabe (fett) und Ausgabe</th></tr>
<tr><td><pre>class Prog {
  public static void main(String[] args) {
    System.out.print("Gew. Länge des Vektors: ");
    int[] wecktor = new int[Simput.gint()];
    System.out.println();

    for(int i = 0; i < wecktor.length; i++) {
      System.out.print("Wert von Element " + i + ": ");
      wecktor[i] = Simput.gint();
    }
    System.out.println();

    for(int i = 0; i < wecktor.length; i++)
      System.out.println(wecktor[i]);
  }
}</pre></td><td><pre>Gew. Länge des Vektors: 3

Wert von Element 0: 7
Wert von Element 1: 13
Wert von Element 2: 4711

7
13
4711</pre></td></tr>
</table>

5.1.5 Array-Kopien mit neuer Länge erstellen

Existiert ein Array-Objekt erst einmal, kann die Anzahl seiner Elemente nicht mehr geändert werden. Um einen Array zu „verlängern“, muss man also ...

- einen neuen, größeren Array erstellen,
- die vorhandenen Elemente dorthin kopieren
- und den alten Array dem Garbage Collector überlassen.

Unter Verwendung der statischen Methode **copyOf()** aus der Klasse **Arrays** (im Paket **java.util**) ist eine solche „Verlängerung“ in *einem* Aufruf zu erledigen. In der Dokumentation zur API-Klasse **Arrays** findet sich eine Familie von **copyOf()** - Überladungen für diverse Elementtypen, z. B. die folgende Variante für den Typ **int**:

public static int[] copyOf(int[] *original*, **int** *newLength*)

Hinzugekommene Elemente werden mit dem typspezifischen Nullwert initialisiert.

Einige später vorzustellende API-Kollektionsklassen zur Verwaltung von Elementlisten gehen im Bedarfsfall analog vor, um die Kapazität des intern zum Speichern der Elemente verwendeten Arrays zu erhöhen. Im Quellcode der API-Klasse **ArrayList** (im Paket **java.util**), die wir später als „größendynamischen“ Container mit Array-Innenleben kennenlernen werden, findet sich z. B. die folgende private Methode **grow()**:

```
private Object[] grow(int minCapacity) {
   return elementData = Arrays.copyOf(elementData, newCapacity(minCapacity));
}
```

Ist beim **copyOf()** - Aufruf die angegebene neue Länge *kleiner* als die alte, entsteht eine durch Streichung der Elemente mit den höchsten Indexnummern gekürzte Array-Kopie.

5.1.6 Nützliche Methoden in der Klasse Arrays

Neben der im letzten Abschnitt vorgestellten Methode **copyOf()** beherrscht die Klasse **Arrays** noch weitere nützliche statische Methoden, z. B.:

- **public static void sort(int[]** *iar*)
  Die in Überladungen für diverse Elementdatentypen vorhandene Methode **sort()** realisiert das Dual-Pivot-Quicksort-Verfahren von Vladimir Yaroslavskiy, Jon Bentley und Joshua Bloch.
- **public static int binarySearch(int[]** *iar*, **int** *ges*)
  Die in Überladungen für diverse Elementdatentypen vorhandene Methode **binarySearch()** durchsucht einen Array nach einem Objekt unter Verwendung des Halbierungsverfahrens. Sie liefert den Index des Treffers oder den Wert -1. Der Array muss zuvor sortiert werden; anderenfalls ist das Ergebnis undefiniert.
- **public static void fill(int[]** *iar,* **int** *val*)
  Die in Überladungen für diverse Elementdatentypen vorhandene Methode **fill()** setzt alle Elemente eines Arrays auf den gewünschten Wert.

5.1.7 Beispiel: Beurteilung des Java-Pseudozufallszahlengenerators

Oben wurde am Beispiel des 5-elementigen **int**-Arrays `uni` demonstriert, dass die Array-Technik im Vergleich zur Verwendung einzelner Variablen den Aufwand bei der Deklaration und beim Zugriff deutlich verringert. Insbesondere beim Einsatz in einer Schleifenkonstruktion erweist sich die Ansprache der einzelnen Elemente über einen Index als überaus praktisch. Die im bisherigen Verlauf von Abschnitt 5.1 zur Demonstration verwendeten Anweisungen lassen sich leicht zu einem Programm erweitern, das die Qualität des **Pseudozufallszahlengenerators** in Java überprüft. Dieser

Generator produziert Folgen von Zahlen mit einem bestimmten Verteilungsverhalten. Obwohl eine Serie perfekt vom Initialisierungswert des Pseudozufallszahlengenerators abhängt, kann sie in der Regel echte Zufallszahlen ersetzen. Manchmal ist es sogar von Vorteil, eine Serie über einen festen Initialisierungswert reproduzieren zu können. In der Regel verwendet man aber variable Initialisierungen, z. B. abgeleitet aus einer Zeitangabe. Der Einfachheit halber redet man oft von *Zufallszahlen* und lässt den Zusatz *Pseudo* weg.

Man kann übrigens mit moderner EDV-Technik unter Verwendung von physikalischen Prozessen durchaus *echte* Zufallszahlen produzieren, doch ist der Zeitaufwand im Vergleich zu Pseudozufallszahlen erheblich höher (siehe z. B. Lau 2009).

Nach der folgenden Anweisung zeigt die Referenzvariable `zzg` auf ein Objekt der Klasse **Random** aus dem API-Paket **java.util**, das als Pseudozufallszahlengenerator taugt:

```
java.util.Random zzg = new java.util.Random();
```

Durch Verwendung des parameterfreien **Random**-Konstruktors entscheidet man sich für die Anzahl der Millisekunden seit dem 1.1.1970, 00.00 Uhr, als Initialisierungswert für den Pseudozufall.[1]

Das angekündigte Programm zur Prüfung des Java-Pseudozufallszahlengenerators zieht 10.000 Zufallszahlen aus der Menge {0, 1, 2, 3, 4} und ermittelt die empirische Verteilung dieser Stichprobe:[2]

```
class UniRand {
   public static void main(String[] args) {
      final int drl = 10_000;
      int i;
      int[] uni = new int[5];
      java.util.Random zzg = new java.util.Random();
      for (i = 0; i < drl; i++)
         uni[zzg.nextInt(5)]++;

      System.out.println("Absolute Häufigkeiten:");
      for (int element : uni)
         System.out.print(element + " ");

      System.out.println("\n\nRelative Häufigkeiten:");
      for (int element : uni)
         System.out.print((double)element/drl + " ");
   }
}
```

Die **Random**-Methode **nextInt()** liefert beim Aufruf mit dem Aktualparameterwert 5 als Rückgabe eine **int**-Zufallszahl aus der Menge {0, 1, 2, 3, 4}, wobei die möglichen Werte mit der gleichen Wahrscheinlichkeit 0,2 auftreten sollten. Im Programm dient der Rückgabewert als Array-Index dazu, ein zufällig gewähltes `uni`-Element zu inkrementieren. Wie das folgende Ergebnisbeispiel zeigt, stellt sich die erwartete Gleichverteilung in sehr guter Näherung ein:

```
Absolute Haeufigkeiten:
1950 1991 1997 2057 2005

Relative Haeufigkeiten:
0.195 0.1991 0.1997 0.2057 0.2005
```

---

[1] Lieferant dieses Wertes ist die statische Methode **currentTimeMillis()** der Klasse **System** im API-Paket **java.lang** und obige Anweisung ist äquivalent mit:

```
java.util.Random zzg = new java.util.Random(System.currentTimeMillis());
```

[2] In der Sprache der Wahrscheinlichkeitstheorie erfolgt die Ziehung „mit Zurücklegen".

Ein χ^2-Signifikanztest mit der Gleichverteilung als Nullhypothese bestätigt durch eine Überschreitungswahrscheinlichkeit von 0,569 (weit oberhalb der kritischen Grenze 0,05), dass keine Zweifel an der Gleichverteilung bestehen:

uni

	Beobachtetes N	Erwartete Anzahl	Residuum
0	1950	2000,0	-50,0
1	1991	2000,0	-9,0
2	1997	2000,0	-3,0
3	2057	2000,0	57,0
4	2005	2000,0	5,0
Gesamt	10000		

Statistik für Test

	uni
Chi-Quadrat	2,932
df	4
Asymptotische Signifikanz	,569

Über die im Beispielprogramm verwendete Klasse **Random** aus dem Paket **java.util** können Sie sich z. B. mit Hilfe der API-Dokumentation informieren.

Statt ein **Random**-Objekt zu erzeugen und mit der Produktion von Pseudozufallszahlen zu beauftragen, kann man auch die statische Methode **random()** aus der Klasse **Math** benutzen, die gleichverteilte **double**-Werte aus dem Intervall [0, 1) liefert, z. B.:

```
uni[(int) (Math.random()*5)]++;
```

Werden sehr viele Pseudozufallszahlen benötigt, sollte statt der Klasse **Random** die seit Java 7 verfügbare und leistungsoptimierte Klasse **ThreadLocalRandom** aus dem Paket **java.util.concurrent** verwendet werden (Bloch 2018, S. 268). Im Beispielprogramm ist dazu die Anweisung

```
java.util.Random zzg = new java.util.Random();
```

zu ersetzen durch:

```
java.util.concurrent.ThreadLocalRandom zzg =
        java.util.concurrent.ThreadLocalRandom.current();
```

In einer Variante des Beispielprogramms benötigte die Klasse **ThreadLocalRandom** für 5 Millionen **nextInt()** - Aufrufe mit 70 Millisekunden tatsächlich etwas weniger Zeit als die Klasse **Random** (90 Millisekunden).

5.1.8 Initialisierungslisten

Bei einem Array mit wenigen Elementen ist die Möglichkeit von Interesse, beim Deklarieren der Referenzvariablen eine Initialisierungsliste mit den Werten für die Elementvariablen anzugeben und das Array-Objekt dabei implizit (ohne Verwendung des **new**-Operators) zu erzeugen, z. B.:

Quellcode

```
class Prog {
    public static void main(String[] args) {
        int[] wecktor = {1, 2, 3};
        System.out.println(wecktor[2]);
    }
}
```

Ausgabe

```
3
```

Die Deklarations- und Initialisierungsanweisung

```
int[] wecktor = {1, 2, 3};
```

ist äquivalent zu:

```
int[] wecktor = new int[3];
wecktor[0] = 1;
```

```
weckor[1] = 2;
weckor[2] = 3;
```

Initialisierungslisten sind nicht nur bei der Deklaration erlaubt, sondern auch bei der Objektkreation per **new**-Operator, z. B.:

```
int[] wecktor;
wecktor = new int[] {1, 2, 3};
```

Es ist auch eine *leere* Initialisierungsliste erlaubt, wobei ein gutes Anwendungsbeispiel nicht leicht zu finden ist, weil Länge eines Arrays bekanntlich nach der Erstellung nicht mehr geändert werden kann.

5.1.9 Objekte als Array-Elemente

Für die Elemente eines Arrays ist natürlich auch ein Referenztyp erlaubt. Im folgenden Beispiel wird ein Array mit Objekten aus unserer Beispielklasse `Bruch` erzeugt:

Quellcode

```
class Bruchrechnung {
   public static void main(String[] args) {
      Bruch b1 = new Bruch(1, 2, "b1 = ");
      Bruch b2 = new Bruch(5, 6, "b2 = ");
      Bruch[] bruvek = {b1, b2};
      bruvek[1].zeige();
   }
}
```

Ausgabe

```
        5
b2 =  -----
        6
```

Im nächsten Abschnitt lernen wir einen wichtigen Spezialfall von Arrays mit Referenztyp-Elementen kennen. Dort zeigen die Elementvariablen wiederum auf Arrays, sodass mehrdimensionale Arrays entstehen.

5.1.10 Mehrdimensionale Arrays

In der linearen Algebra und in vielen anderen Anwendungsbereichen werden auch *mehrdimensionale* Arrays benötigt. Ein zweidimensionaler Array wird in Java als *Array of Arrays* realisiert, z. B.:

Quellcode

```
class Prog {
  public static void main(String[] args) {
    int[][] matrix = new int[4][3];

    System.out.println("matrix.length  = " + matrix.length);
    System.out.println("matrix[0].length = "+matrix[0].length+"\n");

    for(int i = 0; i < matrix.length; i++) {
      for(int j = 0; j < matrix[i].length; j++) {
        matrix[i][j] = (i+1)*(j+1);
        System.out.print(" " + matrix[i][j]);
      }
      System.out.println();
    }
  }
}
```

Ausgabe

```
matrix.length    = 4
matrix[0].length = 3

 1 2 3
 2 4 6
 3 6 9
 4 8 12
```

Dieses Verfahren lässt sich verallgemeinern, um Arrays mit höherer Dimensionalität zu erzeugen, die aber nur selten benötigt werden.

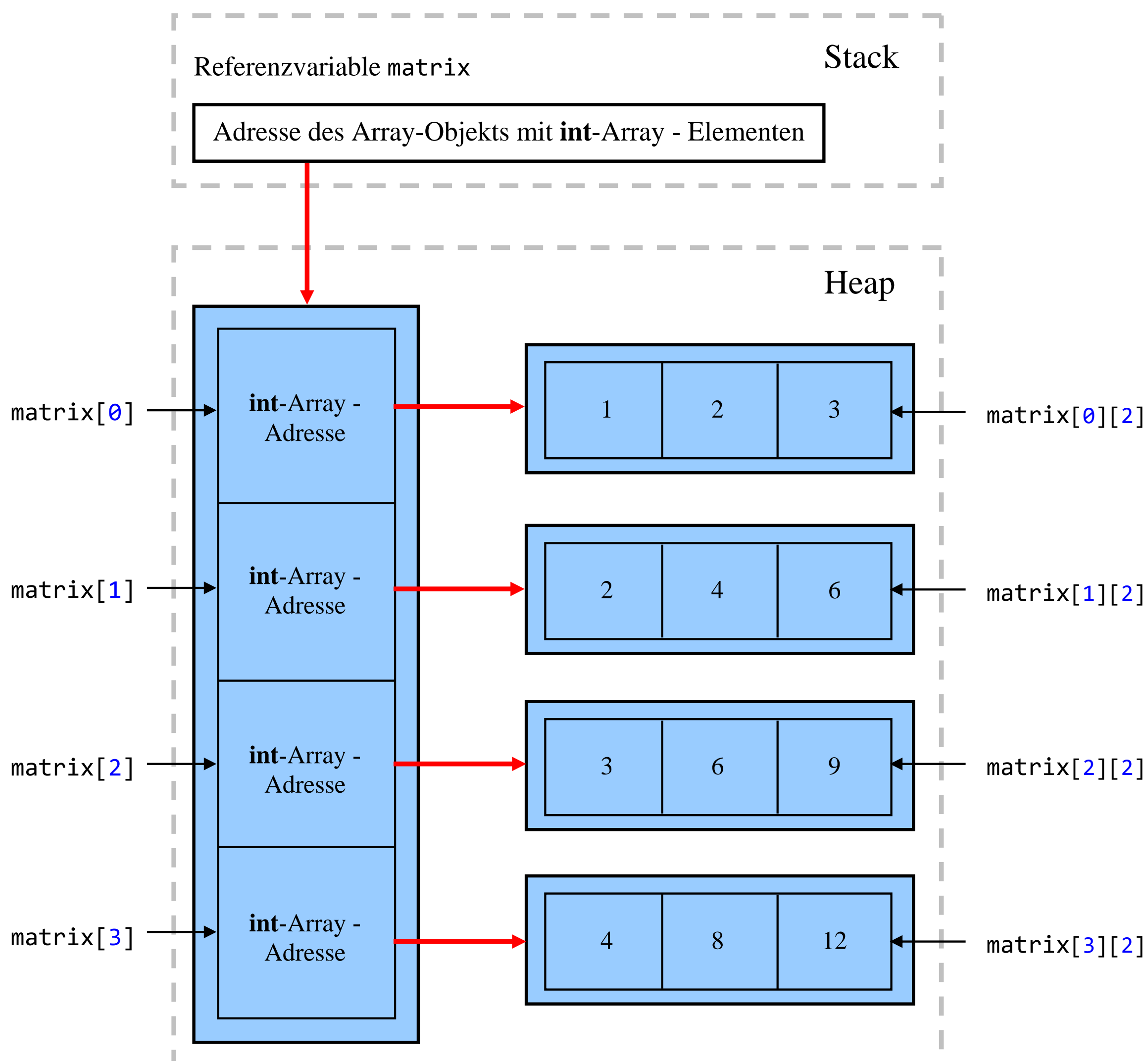
Die erforderliche Reihenfolge der Längenangaben bei der Kreation von geschachtelten Arrays ist etwas gewöhnungsbedürftig. Mit **T** als Namen für einen beliebigen Datentyp haben wir bisher die Logik kennengelernt, dass **T[]** einen Array mit Elementen vom Typ **T** bezeichnet. Daher sollte in der folgenden Anweisung

```
int[][] matrix = new int[4][3];
```

ein äußerer Array mit 3 Elementen vom Typ **int[4]** entstehen. Wie das Beispielprogramm zeigt, resultiert aber ein äußerer Array mit den 4 Elementen `matrix[0]` bis `matrix[3]`, bei denen es sich jeweils um eine Referenz auf einen Array vom Typ **int[3]** handelt. Die Größenangaben in der Deklaration werden den geschachtelten Arrays von außen nach innen zugeordnet. Bei einem zweidimensionalen Array (also bei einer Matrix) ist also zuerst die Anzahl der Zeilen und danach die Anzahl der Spalten anzugeben. Beim *Zugriff* auf Matrixelemente resultiert aus der in Java gewählten Reihenfolge der Längenangaben bzw. Indexwerte gerade die aus der Mathematik vertraute Spezifikationsreihenfolge (Zeilenindex, Spaltenindex), z. B.:

```
matrix[i][j] = (i+1)*(j+1);
```

In der folgenden Abbildung wird die Situation im Hauptspeicher beschrieben:



Im nächsten Beispielprogramm wird die Möglichkeit demonstriert, mehrdimensionale Arrays mit unterschiedlich langen Elementen anzulegen, sodass z. B. eine ausgesägte (engl. *jagged*) Matrix entsteht:

Quellcode	Ausgabe

<pre>class Prog {
  public static void main(String[] args) {
    int[][] matrix = new int[5][];
    for(int i = 0; i < matrix.length; i++) {
      matrix[i] = new int[i+1];
      System.out.printf("matrix[%d]", i);
      for(int j = 0; j < matrix[i].length; j++) {
        matrix[i][j] = i*j;
        System.out.printf("%3d", matrix[i][j]);
      }
      System.out.println();
    }
  }
}</pre>
	<pre>matrix[0] 0
	matrix[1] 0 1
	matrix[2] 0 2 4
	matrix[3] 0 3 6 9
	matrix[4] 0 4 8 12 16</pre>

Im Beispiel wird ein Array-Objekt namens `matrix` mit den fünf Elementen `matrix[0]` bis `matrix[4]` erzeugt, bei denen es sich jeweils um eine Referenz auf einen Array mit **int**-Elementen handelt:

```
int[][] matrix = new int[5][];
```

Die Array-Objekte für die Matrixzeilen entstehen später mit individueller Länge:

```
matrix[i] = new int[i+1];
```

Mit Hilfe dieser Technik kann man sich z. B. beim Speichern einer symmetrischen Matrix Platz sparend auf die untere Dreiecksmatrix beschränken.

Auch im mehrdimensionalen Fall können Initialisierungslisten eingesetzt werden, z. B.:

Quellcode	Ausgabe

<pre>class Prog {
  public static void main(String[] args) {
    int[][] matrix = {{1}, {1,2}, {1, 2, 3}};
      for(int i = 0; i < matrix.length; i++) {
        for(int ele : matrix[i])
          System.out.print(ele+" ");
        System.out.println();
      }
  }
}</pre>
	<pre>1
	1 2
	1 2 3</pre>

5.2 Klassen für Zeichenfolgen

Das Java-API bietet für den Umgang mit Zeichenfolgen, die grundsätzlich aus Unicode-Zeichen bestehen, mehrere, für unterschiedliche Einsatzzwecke optimierte Klassen an, die sich alle im Paket **java.lang** befinden:

- **String**
  Objekte der Klasse **String** können nach dem Erzeugen nicht mehr geändert werden.
- **StringBuilder**, **StringBuffer**
  Für *variable* Zeichenfolgen sollte unbedingt die Klasse **StringBuilder** oder die Klasse **StringBuffer** verwendet werden, weil deren Objekte nach dem Erzeugen noch verändert werden können.

5.2.1 Die Klasse String für konstante Zeichenfolgen

Nach Einschätzung von Oaks (2014, S. 198) ist **String** in Java die mit Abstand am häufigsten verwendete Java-Klasse, und es sind einige Anstrengungen unternommen worden, um für eine bequeme Verwendung sowie für eine gute Performanz zu sorgen. Man hat sich entschieden, die Klasse für den *lesenden* Zugriff auf Zeichenfolgen zu optimieren und die Objekte als unveränderlich zu konzipieren.

Bis Java 8 hat ein **String**-Objekt seine Daten intern in einem **char**-Array gespeichert und demzufolge pro Zeichen 2 Bytes verwendet (Unicode-Zeichensatz mit 2^{16} Zeichen, Kodierung UTF-16). Viele Anwendungen benötigen für **String**-Objekte aber lediglich den Latin-1 - Zeichensatz, der pro Zeichen mit *einem* Byte auskommt. Seit Java 9 speichert ein **String**-Objekt seine Daten in einem **byte**-Array und verwendet pro Zeichen aufgrund einer automatischen Bedarfsanalyse entweder ein Byte oder zwei Bytes. So wird Speicherplatz gespart, wobei weder vorhandene Java-Programme angepasst werden müssen, noch Lernaufwand seitens der Entwickler erforderlich ist. Die Klasse **String** liefert somit ein gutes Beispiel für die durch Datenkapselung ermöglichte Flexibilität bei der internen Datenablage.

5.2.1.1 Erzeugen von String-Objekten

In der folgenden Deklarations- und Initialisierungsanweisung

```
String s1 = "abcde";
```

wird:

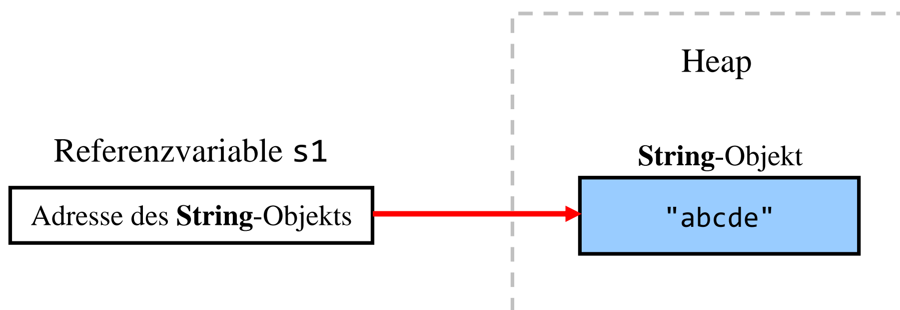
- eine **String**-Referenzvariable namens `s1` angelegt,
- ein neues **String**-Objekt auf dem Heap erzeugt, falls noch kein inhaltsgleiches vorhanden ist
- die Adresse des neu erstellten oder bereits angetroffenen Objekts in der Referenzvariablen abgelegt

Soviel objektorientierten Hintergrund sieht man der angenehm einfachen Anweisung auf den ersten Blick nicht an. In Java sind jedoch auch Zeichenketten*literale* als **String**-Objekte realisiert, sodass z. B.

```
"abcde"
```

einen Ausdruck darstellt, der als Wert einen Verweis auf ein **String**-Objekt auf dem Heap liefert.

Obige Anweisung erzeugt im Hauptspeicher die folgende Situation:



Die Klasse **String** besitzt auch Konstruktoren für die Objektkreation per **new**-Operator, wobei z. B. ein **StringBuilder**- oder ein **StringBuffer**-Objekt als Aktualparameter in Frage kommt. Auch ein **String**-Literal ist als Aktualparameter erlaubt, wenngleich sich diese Konstruktion im Abschnitt 5.2.1.3 als wenig sinnvoll herausstellen wird:

```
String s1 = new String("abcde");
```

5.2.1.2 String als WORM - Klasse

Nachdem ein **String**-Objekt auf dem Heap erzeugt wurde, ist es **unveränderlich** (engl.: *immutable*). In der Abschnittsüberschrift wird für diesen Sachverhalt eine Abkürzung aus der Elektronik ausgeliehen: WORM (**W**rite **O**nce **R**ead **M**any). Eventuell werden Sie die Unveränderlichkeit von **String**-Objekten in Zweifel ziehen und ein Gegenbeispiel der folgenden Art vorbringen:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        String testr = "abc";`
`        System.out.println("testr = " + testr);`
`        testr = testr + "def";`
`        System.out.println("testr = " + testr);`
`    }`
`}`
	`testr = abc`
	`testr = abcdef`

Die Anweisung

```
testr = testr + "def";
```

verändert aber *nicht* das per `testr` ansprechbare **String**-Objekt (mit dem Inhalt „abc“), sondern erzeugt ein neues **String**-Objekt (mit dem Inhalt „abcdef“) und schreibt dessen Adresse in die Referenzvariable `testr`.

5.2.1.3 Interner String-Pool und Identitätsvergleich

Geschieht wie im folgenden Beispiel

```
String s1 = "abcde";
```

die Initialisierung einer **String**-Referenzvariablen über ein Literal oder einen anderen *konstanten* Ausdruck, sodass schon der Compiler die resultierende Zeichenfolge kennt, dann kommt der sogenannte **interne String-Pool** ins Spiel:

- Ist hier bereits ein inhaltsgleiches **String**-Objekt vorhanden, wird dessen Adresse in die Referenzvariable geschrieben und auf eine Neukreation verzichtet.
- Anderenfalls wird im **String**-Pool ein neues Objekt angelegt und dessen Adresse in die Referenzvariable geschrieben.

So wird verhindert, dass für wiederholt im Quellcode auftretende Zeichenfolgenliterale jeweils Speicherplatz verschwendend ein neues Objekt entsteht. Diese Vorgehensweise ist sinnvoll, weil sich vorhandene **String**-Objekte garantiert nicht mehr ändern (siehe Abschnitt 5.2.1.2).

Außerdem ist für die im **String**-Pool registrierten Objekte garantiert, dass sie *unterschiedliche* Zeichenfolgen enthalten, was sich bald im Zusammenhang mit Identitätsvergleichen als nützlich (Rechenzeit sparend) herausstellen wird.

Kommt bei der Initialisierung eines **String**-Referenzvariablen ein Ausdruck mit Beteiligung von Variablen zum Einsatz, wird auf jeden Fall ein neues Objekt erzeugt und der interne **String**-Pool ist *nicht* beteiligt, z. B. bei der folgenden Variablen `s3`:

```
String de = "de";
String s3 = "abc" + de;
```

Ebenso wird auch bei Verwendung des **new**-Operators verfahren.

Ein **String**-Literal wie im folgenden Beispiel

```
String s1 = new String("abcde");
```

als Konstruktorparameter zu verwenden, ist nur selten sinnvoll, denn:

- Das Zeichenfolgenliteral führt zu einem neuen **String**-Objekt im internen **String**-Pool, falls dort noch kein inhaltsgleiches Objekt existiert.
- Per **new**-Operator entsteht auf jeden Fall ein neues **String**-Objekt auf dem Heap, das dieselbe Zeichenfolge enthält wie das Parameter-Objekt.

Weil die beiden **String**-Objekte unveränderlich sind, lohnt sich der Doppelaufwand nicht. Der **String**-Konstruktor mit Zeichenkettenliteral als Parameter kann in Ausnahmefällen aber doch sinnvoll sein, wenn unbedingt ein neues Objekt benötigt wird (z. B. als Monitorobjekt für die später zu behandelnde Thread-Synchronisation).

Für den Vergleich von **String**-Variablen *per Identitätsoperator* haben die obigen Ausführungen wichtige Konsequenzen, wie das folgende Programm zeigt:

Quellcode

```java
class Prog {
    public static void main(String[] args) {
        String s1 = "abcde";
        String s2 = "abc" + "de";
        String de = "de";
        String s3 = "abc" + de;
        String s4 = new String("abcde");
        System.out.print("(s1 == s2) = "+(s1==s2) + "\n"+
                "(s1 == s3) = " + (s1==s3) + "\n"+
                "(s1 == s4) = " + (s1==s4));
    }
}
```

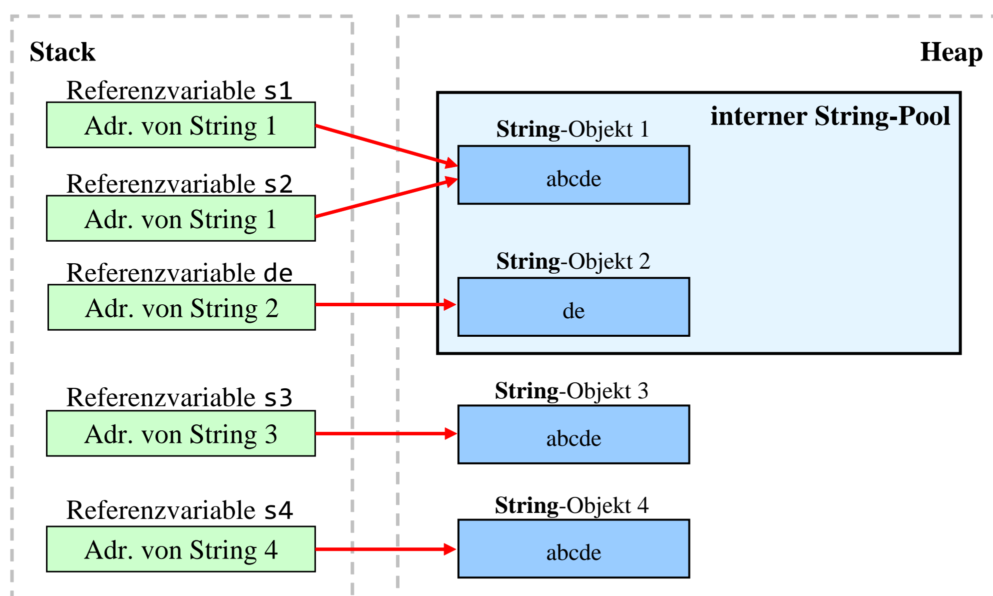
Ausgabe

```
(s1 == s2) = true
(s1 == s3) = false
(s1 == s4) = false
```

Das merkwürdige[1] Verhalten des Programms hat folgende Ursachen:

- Wendet man den Identitätsoperator auf zwei **String**-Referenzvariablen an, werden die in den Variablen gespeicherten *Adressen* verglichen, keinesfalls die Inhalte der referenzierten **String**-Objekte.
- Nur wenn die beiden am Vergleich beteiligten **String**-Referenzvariablen auf Objekte im internen **String**-Pool zeigen, ist garantiert, dass die Variablen genau dann für dieselbe Zeichenfolge stehen, wenn sie denselben Referenzwert haben.

Im Beispielprogramm werden vier **String**-Objekte mit folgenden Referenzen erzeugt:



[1] „Merkwürdig" bedeutet hier, dass sich eine Aufnahme in das Langzeitgedächtnis lohnt.

Später werden zwei für den Vergleich von **String**-Objekten relevante Methoden vorgestellt:

- Mit **equals()** zum Vergleich mit einem Kollegen aufgefordert, nimmt ein **String**-Objekt auf jeden Fall einen *Inhaltsvergleich* vor (siehe Abschnitt 5.2.1.4.2).
- Mit der Methode **intern()** wird die Aufnahme von **String**-Objekten in den internen **String**-Pool unterstützt, sodass anschließend Referenz- und Inhaltsvergleich äquivalent sind (siehe Abschnitt 5.2.1.5). Das Erscheinen eines Zeichenfolgenliterals im Quellcode ist also *nicht* der einzige Anlass für die Aufnahme eines **String**-Objekts in den **String**-Pool.

5.2.1.4 Methoden für String-Objekte

Von den ca. 70 öffentlichen Methoden der Klasse der **String** werden in diesem Abschnitt nur die wichtigsten angesprochen. Für spezielle Anwendungen lohnt sich also ein Blick in die Dokumentation zur Klasse **String**.

5.2.1.4.1 Verketten von Strings

Zum Verketten von Strings kann in Java der „+“ - Operator verwendet werden, wobei beliebige Datentypen bei Bedarf automatisch in Strings konvertiert werden. Im folgenden Beispiel wird mit Klammern dafür gesorgt, dass der Compiler die „+“ - Operatoren jeweils sinnvoll interpretiert (Verketten von **String**-Objekten bzw. Addieren von Zahlen):

Quellcode

```
class Prog {
    public static void main(String[] args) {
        System.out.println("4 + 3 = " + (4 + 3));
    }
}
```

Ausgabe

```
4 + 3 = 7
```

Es ist übrigens eine Besonderheit, dass **String**-Objekte mit dem „+“ - Operator verarbeitet werden können. Bei anderen Java-Klassen ist das aus C++ und C# bekannte *Überladen* von Operatoren *nicht* möglich.

5.2.1.4.2 Inhaltsvergleich

Für den Test auf identischen **Inhalt** kann man die **String**-Methode **equals()**

public boolean equals(String *vergl***)**

verwenden, um den im Abschnitt 5.2.1.3 erläuterten Tücken beim Vergleich von **String**-Referenzvariablen per Identitätsoperator aus dem Weg zu gehen. Im folgenden Programm werden zwei **String**-Objekte zunächst nach ihren Speicheradressen verglichen, dann nach dem Inhalt:

Quellcode

```
class Prog {
    public static void main(String[] args) {
        String s1 = "abc";
        String s2 = new String("abc");
        System.out.println(s1 == s2);
        System.out.println(s1.equals(s2));
    }
}
```

Ausgabe

```
false
true
```

5.2.1.4.3 Sortierungspriorität

Zum Vergleich von Zeichenfolgen hinsichtlich der Sortierungspriorität kann die **String**-Methode **compareTo()**

public int compareTo(String *vergl***)**

dienen, z. B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Prog {
    public static void main(String[] args) {
        String a = "Müller, Anja", b = "Müller, Kurt",
               c = "Müller, Anja";
        System.out.println("< : " + a.compareTo(b));
        System.out.println("= : " + a.compareTo(c));
        System.out.println("> : " + b.compareTo(a));
    }
}
```

</td><td>

```
< : -10
= : 0
> : 10
```

</td></tr>
</table>

Für die Methode **compareTo()** sind folgende **int**-Rückgabewerte garantiert:

<table>
<tr><th colspan="2"></th><th>compareTo() - Rückgabe</th></tr>
<tr><td rowspan="3">Die Sortierungspriorität des angesprochenen String-Objekts ist im Vergleich zum Parameterobjekt:</td><td>kleiner</td><td>negative Zahl</td></tr>
<tr><td>gleich</td><td>0</td></tr>
<tr><td>größer</td><td>positive Zahl</td></tr>
</table>

5.2.1.4.4 Länge einer Zeichenkette

Während bei Array-Objekten die Anzahl der Elemente in der finalisierten Instanz*variablen* **length** zu finden ist (vgl. Abschnitt 5.1), lässt sich die Länge einer Zeichenkette über die Instanz*methode* **length**() ermittelt:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Prog {
    public static void main(String[] args) {
        char[] cvek = {'a', 'b', 'c'};
        String str = "abc";
        System.out.println(cvek.length);
        System.out.println(str.length());
    }
}
```

</td><td>

```
3
3
```

</td></tr>
</table>

5.2.1.4.5 Zeichen(folgen) extrahieren, suchen oder ersetzen

Im folgenden Programm werden einige anschließend beschriebene **String**-Methoden demonstriert:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Prog {
    public static void main(String[] args) {
        String bsp = "Brgl";
        System.out.println(bsp.substring(1, 3));
        System.out.println(bsp.indexOf("g"));
        System.out.println(bsp.indexOf("x"));
        System.out.println(bsp.startsWith("r"));
        System.out.println(bsp.charAt(0));
    }
}
```

</td><td>

```
rg
2
-1
false
B
```

</td></tr>
</table>

a) Teilzeichenfolge extrahieren

Mit der Methode

public String substring(int *start*, **int** *ende***)**

lassen sich alle Zeichen zwischen den Positionen *start* (inklusive) und *ende* (exklusive) extrahieren.

b) Teilzeichenfolge suchen

Mit der Methode

public int indexOf(String *gesucht***)**

kann man einen **String** nach einer anderen Zeichenkette durchsuchen. Als Rückgabewert erhält man ...

- nach erfolgreicher Suche: die Startposition der ersten Trefferstelle
- nach vergeblicher Suche: -1

c) Zeichenfolge auf eine bestimmte Startsequenz überprüfen

Mit der Methode

public boolean startsWith(String *start*)

lässt sich feststellen, ob ein **String** mit einer bestimmten Zeichenfolge beginnt.

d) Das Zeichen an einer bestimmten Position ermitteln

Weil ein **String** *kein* Array ist, kann auf die einzelnen Zeichen *nicht* per Indexoperator ([]) zugegriffen werden. Mit der **String**-Methode

public char charAt(int *index***)**

steht aber ein Ersatz zur Verfügung, wobei die Nummerierung der Zeichen bei 0 beginnt. Ein Aufruf mit ungültiger Position führt zu einem Ausnahmefehler aus der Klasse

java.lang.StringIndexOutOfBoundsException

e) Aus einem String einen Char - Array erstellen

Wenn auf jeden Fall mit dem Indexoperator gearbeitet werden soll, kann aus einem **String** über die Methode

public char[] toCharArray()

ein **char**-Array mit identischem Inhalt erzeugt werden, z. B.:

Quellcode

```
class Prog {
    public static void main(String[] args) {
        String s = "abc";
        char[] c = s.toCharArray();
        for (int i = 0; i < c.length; i++)
            System.out.println(c[i]);
    }
}
```

Ausgabe

```
a
b
c
```

f) Zeichen oder Teilzeichenfolgen ersetzen

Von der Methode

public String replace(char *oldChar*, **char** *newChar*)

erhält man einen neuen **String**, der aus dem angesprochenen Original hervorgeht, indem ein altes Zeichen (an allen Trefferstellen) durch ein neues Zeichen ersetzt wird, z. B.:

```
String s2 = s1.replace('C','c');
```

Mit weiteren **replace()** - Überladungen kann man das erste Auftreten einer Teilzeichenfolge oder alle Teilzeichenfolgen, die einem regulären Ausdruck genügen, durch eine neue Teilzeichenfolge ersetzen lassen.

5.2.1.4.6 Groß-/Kleinschreibung normieren

Von den Methoden

public String toUpperCase()

bzw.

public String toLowerCase()

erhält man einen neuen **String**, der im Unterschied zum angesprochenen Original auf Groß- bzw. Kleinschreibung normiert ist, was vor Vergleichen oft sinnvoll ist, z. B.:

Quellcode	Ausgabe

<pre>class Prog {
    public static void main(String[] args) {
        String a = "Otto", b = "otto";
        System.out.println(a.toUpperCase().equals(b.toUpperCase()));
    }
}</pre>
	true

In der Anweisung mit dem **equals()** - Aufruf stoßen wir auf eine stattliche Anzahl von Punktoperatoren, sodass eine kurze Erklärung angemessen ist:

- Der Methodenaufruf `a.toUppercase()` erzeugt ein neues **String**-Objekt und liefert die zugehörige Referenz.
- Diese Referenz ermöglicht es, dem neuen Objekt Botschaften zu übermitteln, was unmittelbar zum Aufruf der Methode **equals()** genutzt wird.

5.2.1.5 Vertiefung: Aufwand beim Inhalts- bzw. Referenzvergleich

Wenn sehr viele Inhaltsvergleiche vorzunehmen sind, ist der im Abschnitt 5.2.1.3 beschriebene interne **String**-Pool eine erwägenswerte Option. Zeigen zwei Referenzvariablen auf Pool-Strings, folgt aus der Gleichheit der Adressen bereits die Inhaltsgleichheit. Folglich kann man statt des relativ aufwändigen Inhaltsvergleichs den erheblich flotteren Referenzvergleich durchführen.

Allerdings muss zunächst dafür gesorgt werden, dass die beteiligten Referenzvariablen auf Pool-Strings zeigen. Wie bereits im Abschnitt 5.2.1.3 berichtet wurde, gibt es neben der **String**-Initialisierung durch einen konstanten Ausdruck noch eine zweite Möglichkeit, ein **String**-Objekt im Pool abzulegen. Man ruft dazu die **String**-Instanzmethode **intern()** auf,

public String intern()

die zum angesprochenen String seine sogenannte *kanonische Repräsentation* liefert:

- Ist im internen Pool ein inhaltsgleicher **String** vorhanden (im Sinne der **equals()** - Methode), wird dessen Adresse als Rückgabe geliefert.
- Anderenfalls wird der angesprochene **String** in den Pool aufgenommen und seine Adresse als Rückgabe geliefert.

Für die Planung der **intern()** - Verwendung ist es relevant, wie die JVM den internen **String**-Pool realisiert und im Speicher ablegt (siehe Oaks 2014, S. 198ff; Vorontsov 2014). Zur Verwaltung der Pool-Strings wird eine Hash-Tabelle, also ein Kollektionsobjekt für Schlüssel-Wert - Paare verwendet (analog zur generischen Klasse **HashMap<K,V>** aus dem Java Collections Framework, siehe 10.6.2). Seit der Java-Version 7 befindet sich diese Hash-Tabelle auf dem allgemeinen Heap, während es sich bis Java 6 in der Method Area befand. Während die Klasse **HashMap<K,V>** aus dem Java Collections Framework ihre Größe dynamisch ändern kann, ist dies bei der Hash-Tabelle zur Verwaltung des internen **String**-Pools *nicht* möglich. In aktuellen Java-Versionen kann die Kapazität der Hash-Tabelle zum internen **String**-Pools allerdings beim JVM-Start über den Parameter **-XX:StringTableSize** festgelegt werden. Oaks (2014, S. 200) empfiehlt, ca. die doppelte Anzahl der anzunehmenden Pool-Strings anzugeben und dabei eine Primzahl zu verwenden.

Eine Überschreitung der Pool-Kapazität führt zu einer verschlechterten Leistung der Methode **intern()**. Bei Server-Anwendungen kann ein Risiko bestehen, wenn *Benutzer* die Kontrolle über die Aufnahme von Strings in den internen Pool haben, und dessen Kapazität überschreiten.

Das Internieren der von zu vergleichenden Variablen referenzierten **String**-Objekten ...

- verursacht zunächst Zeitaufwand
- beschleunigt aber anschließende **String**-Vergleiche.

Um einen Eindruck von der Rentabilität des Internierens zu gewinnen, werden im folgenden Programm `anz` Zufallszeichenfolgen der Länge `len` jeweils `wdh` mal mit einem zufällig gewählten Partner verglichen. Dies geschieht zunächst per **equals()** - Methode und dann nach dem zwischenzeitlichen Internieren per Adressenvergleich.

```
class StringIntern {
   public static void main(String[] args) {
      final int anz = 50_000, len = 20, wdh = 50;
      StringBuffer sb = new StringBuffer();
      java.util.Random ran = new java.util.Random();
      String[] sar = new String[anz];

      // Zufallszeichenfolgen mit Hilfe eines StringBuiler-Objekts erzeugen
      for (int i = 0; i < anz; i++) {
         for (int j = 0; j < len; j++)
            sb.append((char) (65 + ran.nextInt(26)));
         sar[i] = sb.toString();
         sb.delete(0, len);
      }

      long start = System.currentTimeMillis();
      int hits = 0;
      // Inhaltsvergleiche
      for (int n = 1; n <= wdh; n++)
         for (int i = 0; i < anz; i++)
            if (sar[i].equals(sar[ran.nextInt(anz)]))
               hits++;
      System.out.println((wdh * anz)+" Inhaltsvergleiche ("+hits+
            " hits) benötigen "+(System.currentTimeMillis()-start)+" Millisekunden");

      start = System.currentTimeMillis();
      hits = 0;

      // Internieren
      for (int j = 1; j < anz; j++)
         sar[j] = sar[j].intern();
      System.out.println("\nZeit für das Internieren: "+
            (System.currentTimeMillis()-start)+" Millisekunden");
```

```
        // Adressvergleiche
        for (int n = 1; n <= wdh; n++)
           for (int i = 0; i < anz; i++)
              if (sar[i] == sar[ran.nextInt(anz)])
                 hits++;
        System.out.println((wdh * anz)+" Adressvergleiche ("+hits+
              " hits) benötigen (inkl. Internieren) "+(System.currentTimeMillis()-start)+
              " Millisekunden");
     }
  }
```

Es hängt von den Aufgabenparametern `anz`, `len` und `wdh` ab, welche Vergleichstechnik überlegen ist:[1]

	Laufzeit in Millisekunden	
	equals() - Vergleiche	Intern. plus Adressvergl.
`anz = 50000, len = 20, wdh = 5`	56	93
`anz = 50000, len = 20, wdh = 50`	565	101
`anz = 50000, len = 20, wdh = 500`	3511	390

Erwartungsgemäß ist das Internieren umso rentabler, je mehr Vergleiche anschließend mit den Zeichenfolgen angestellt werden.

5.2.2 Die Klassen StringBuilder und StringBuffer für veränderliche Zeichenfolgen

Für häufig zu ändernde Zeichenfolgen sollte man statt der Klasse **String** unbedingt die Klasse **StringBuilder** oder die Klasse **StringBuffer** verwenden, weil hier beim Ändern einer Zeichenkette das zeitaufwändige Erstellen eines neuen Objektes entfällt.

Als Nachteile im Vergleich zur Klasse **String** sind zu nennen:

- Weil die Objekte nicht unveränderlich sind, scheiden Optimierungen wie der interne **String**-Pool aus.
- Es fehlt die syntaktische Unterstützung in der Programmiersprache, z. B. durch den überladenen „+“ - Operator.

Der einzige Unterschied zwischen den Klassen **StringBuilder** und **StringBuffer** besteht darin, dass die Klasse **StringBuffer** Thread-sicher ist, sodass ein Objekt dieser Klasse gefahrlos von mehreren Threads (Ausführungsfäden, siehe Kapitel 15) eines Programms genutzt werden kann. Diese Thread-Sicherheit ist aber mit Aufwand verbunden, sodass die Klasse **StringBuilder** zu bevorzugen ist, wenn eine variable Zeichenfolge nur von *einem* Thread genutzt wird. Bloch (2018, S. 84) hält die (ältere) Klasse **StringBuffer** generell für obsolet. Weil die beiden Klassen völlig analog aufgebaut sind, kann sich die anschließende Beschreibung auf die Klasse **StringBuilder** beschränken.

In der Klasse **StringBuilder** stehen u.a. die folgenden Konstruktoren zur Verfügung:

- **public StringBuilder**()
  Beispiel: `StringBuilder sb = new StringBuilder();`
- **public StringBuilder**(**String** *str*)
  Beispiel: `StringBuilder sb = new StringBuilder("abc");`

Im folgenden Programm wird eine Zeichenfolge 100.000-mal verlängert, zunächst mit Hilfe der Klasse **String**, dann mit Hilfe der Klasse **StringBuilder** und schließlich mit der Klasse **StringBuffer**:[1]

[1] Die Ergebnisse wurden auf einem PC mit der Intel-CPU Core i3 550 (3,2 GHz) unter Windows 10 (64 Bit) mit dem OpenJDK 8 ermittelt.

```
class SBBBench {
    final static int DRL = 100_000;
    static long vorher, diff;
    public static void main(String[] args) throws Exception {
        String s = "*";
        vorher = System.currentTimeMillis();
        for (int i = 0; i < DRL; i++)
            s = s + "*";
        diff = System.currentTimeMillis() - vorher;
        System.out.printf("Zeit für die %-30s %7d\n", "String-\"Verlängerung\":", diff);

        StringBuilder sbuild = new StringBuilder("*");
        runAppend(sbuild);

        StringBuffer sbuff = new StringBuffer("*");
        runAppend(sbuff);
    }

    static void runAppend (Appendable cs) throws Exception {
        vorher = System.currentTimeMillis();
        for (int i = 0; i < DRL; i++)
            cs.append("*");
        String s = cs.toString();
        diff = System.currentTimeMillis() - vorher;
        System.out.printf("Zeit für die %-30s %7d\n",
                          cs.getClass().getSimpleName()+" - Verlängerung: ", diff);
    }
}
```

Während bei Verwendung der Klasse **String** sehr viel Zeit verschwendet wird, unterscheiden sich die Laufzeiten[2] für die beiden anderen Klassen nur wenig, wobei aber der Preis für die Thread-Sicherheit der Klasse **StringBuffer** zu erkennen ist:

```
Zeit für die String-"Verlängerung":           8644
Zeit für die StringBuilder - Verlängerung:       9
Zeit für die StringBuffer - Verlängerung:       14
```

Um Redundanz zu vermeiden, wird im Programm die statische Methode `runAppend()` mit einem Parameter vom Typ der Schnittstelle **Appendable** verwendet. Mit Schnittstellen werden wir uns bald beschäftigen. Im Beispiel dient die Schnittstelle **Appendable** als gemeinsamer Datentyp für die Klassen **StringBuilder** und **StringBuffer**. Die in `runAppend()` verwendete Methode **append()** kommuniziert per Ausnahmeobjekt. Weil wir uns um Ausnahmen noch nicht kümmern wollen, melden wir in den Definitionsköpfen der Methoden `runAppend()` und **main()** an, dass von ihnen Ausnahmen zu erwarten sind.

Ein **StringBuilder**-Objekt beherrscht u.a. die folgenden **public**-Methoden:

---

[1] Ein IntelliJ-Projekt mit dem Programm ist im Ordner **...\BspUeb\Elementare Klassen\Zeichenfolgen\SBBBench** zu finden.

[2] Die Laufzeiten (in Millisekunden) wurden auf einem PC mit der Intel-CPU Core i3 550 (3,2 GHz) unter Windows 10 (64 Bit) mit dem OpenJDK 8 ermittelt.

Methode	Erläuterung
int length()	Diese Methode liefert die aktuelle Anzahl der Zeichen.
append()	Der **StringBuilder** wird um die Zeichenfolgen-Repräsentation des Argumentes verlängert, z. B.: `sb.append("*");` Es sind **append()** - Überladungen für zahlreiche Datentypen vorhanden.
insert()	Die Zeichenfolgen-Repräsentation des Arguments, das von nahezu beliebigem Typ sein kann, wird an einer bestimmten Stelle eingefügt, z. B.: `sb.insert(4, 3.14);`
delete()	Die Zeichen von einer Startposition (einschließlich) bis zu einer Endposition (ausschließlich) werden gelöscht, z. B.: `sb.delete(1, 3);`
replace()	Ein Bereich des **StringBuilder**-Objekts wird durch den Parameter-**String** ersetzt, z. B.: `sb.replace(1, 3, "xy");`
String toString()	Es wird ein **String**-Objekt mit dem Inhalt des **StringBuilder**-Objekts erzeugt. Dies ist z. B. erforderlich, um zwei **StringBuilder**-Objekte mit Hilfe der **String**-Methode **equals()** vergleichen zu können: `sb1.toString().equals(sb2.toString())`

5.2.3 Mehrzeilige Textblöcke

In Java 13 wurden, allerdings mit Vorschau-Status (vgl. Abschnitt 3.7.3.5), mehrzeilige Textblöcke eigeführt, die es erleichtern, Segmente aus anderen Sprachen (z. B. HTML, SQL, XML) in den Java-Quellcode aufzunehmen. Im folgenden Programm sind die traditionelle Syntax und die Textblock-Lösung am Beispiel einer integrierten HTML-Seite zu sehen:[1]

```
class Prog {
    public static void main(String[] args) {
        String htmlKl = "<html>\n" +
                        "    <body>\n" +
                        "        <p>Traditionelle Lösung</p>\n" +
                        "    </body>\n" +
                        "</html>\n";
        String htmlTB = """
                         <html>
                             <body>
                                 <p>Textblock-Lösung</p>
                             </body>
                         </html>
                        """;
    }
}
```

Ein Textblock wird durch *drei* doppelte Hochkommata eingeleitet und abgeschlossen. Die Leerzeichen vor dem Start der ersten Textzeile werden bei allen Zeilen des Textblocks automatisch entfernt, sodass eine für den Java-Quellcode passende Einrückung möglich ist. IntelliJ zeigt die Startposition für den Textblock durch eine vertikale Linie an:

[1] Das Beispiel stammt von https://openjdk.java.net/jeps/355

```
String htmlTB = """
                <html>
                    <body>
                        <p>Textblock-Lösung</p>
                    </body>
                </html>
                """;
```

5.3 Verpackungsklassen für primitive Datentypen

In Java existiert zu jedem primitiven Datentyp eine Wrapper-Klasse, in deren Objekte jeweils ein Wert des primitiven Typs verpackt werden kann (*to wrap* heißt *einpacken*):

Primitiver Datentyp	Wrapper-Klasse
byte	Byte
short	Short
int	Integer
long	Long
double	Double
float	Float
boolean	Boolean
char	Character

Diese Verpackung ist z. B. dann erforderlich, wenn eine Methode genutzt werden soll, die Referenzparameter verlangt.

Neben Ihrer Funktion, im Java-Typsystem die Kluft zwischen Wert- und Referenztypen zu überbrücken, stellen die Wrapper-Klassen nützliche Konvertierungsmethoden und Konstanten bereit (als statische Methoden bzw. Felder).

5.3.1 Wrapper-Objekte erstellen

In der Regel verfügen die Wrapper-Klassen über zwei Konstruktoren mit jeweils einem Parameter, der vom zugehörigen primitiven Typ bzw. vom Typ **String** ist, z. B. bei der Klasse **Integer**, deren Objekte einen **int**-Wert verpacken:

- **public Integer(int** *value*)
  Beispiel: `Integer iu = new Integer(4711);`
- **public Integer(String** *str*)
  Beispiel: `Integer iu = new Integer(args[0]);`

Seit Java 9 sind die Wrapper-Konstruktoren abgewertet (engl.: *deprecated*), worauf unsere Entwicklungsumgebung IntelliJ IDEA deutlich hinweist:

```
Integer iu = new Integer(4711);
```

Als Ersatz für die beiden Konstruktoren wird die statische Fabrikmethode **valueOf()** vorgeschlagen, die zwei analoge Überladungen besitzt, z. B. bei der Klasse **Integer**:

- **public static Integer valueOf(int** *value*)
- **public static Integer valueOf(String** *str*)

Die **valueOf()** - Methoden der Klasse **Integer** verwenden Zeit und Speicherplatz sparend einen Cache mit bereits erzeugten Wrapper-Objekten vom eigenen Typ, wobei allerdings nur die Werte von -128 bis 127 unterstützt werden:

- Liegt der Parameter im Bereich von -128 bis 127, und ist bereits ein Wrapper-Objekt mit diesem Wert vorhanden, dann wird dessen Adresse zurückgeliefert (analog zum internen **String**-Pool, vgl. Abschnitt 5.2.1.3).
- Anderenfalls wird ein neues Objekt erstellt und dessen Adresse geliefert.

Wenn nicht unbedingt ein *neues* Objekt benötigt wird, sollte an Stelle eines Wrapper-Konstruktors die Methode **valueOf()** verwendet werden.

Das eben beschriebene Verhalten der **valueOf()** - Methoden ist zulässig und sinnvoll, weil die Wrapper-Objekte **unveränderlich** sind (engl.: *immutable*). Nach dem Erzeugen eines Wrapper-Objekts kann sein Inhalt nicht mehr geändert werden. Daher besitzen die Wrapper-Klassen keinen parameterfreien Konstruktor. Es ist aber möglich, ein *Feld* mit Wrappertyp zu definieren, das aufgrund der automatischen Initialisierung auf **null** zeigt:

Quellcode	Ausgabe

`class Prog {`
`    static Integer iul;`
`    public static void main(String[] args) {`
`        System.out.println(iul);`
`    }`
`}`
	`null`

Wird diesem Feld ein **int**-Wert zugewiesen, entsteht ein neues **Integer**-Objekt.

Im nächsten Abschnitt geht es um das Erstellen von Wrapper-Objekten über einen Compiler-Automatismus.

5.3.2 Auto(un)boxing

Seit der Version 5 kann der Java-Compiler Werte eines primitiven Typs *automatisch* in Wrapper-Objekte verpacken, z. B.:

```
Integer iw = 4711;
```

Damit vereinfacht sich die Nutzung von Methoden, die **Object**-Parameter erwarten. Im folgenden Beispielprogramm wird ein Objekt der Klasse **ArrayList** aus dem Paket **java.util** als bequemer und flexibler Container verwendet:[1]

- Ein **ArrayList**-Container kann Objekte beliebigen Typs als Elemente aufnehmen.
- Die Größe des Containers wird automatisch an den Bedarf angepasst.

Um Werte primitiver Typen in einen **ArrayList**-Container einfügen zu können, müssen sie in Wrapper-Objekte verpackt werden, was aber dank Autoboxing keine Mühe macht:

```
class Autoboxing {
   public static void main(String[] args) {
      java.util.ArrayList al = new java.util.ArrayList();
      al.add("Otto");

      // AutoBoxing
      al.add(13);
      al.add(23.77);
      al.add('x');
```

[1] **ArrayList** ist eine *generische* Klasse (siehe Kapitel 8) und sollte unbedingt mit Elementen *eines* bestimmten Datentyps genutzt werden. Dieser ist beim Instanzieren anzugeben, wenn der Compiler die Typhomogenität überwachen soll. Wir verwenden ausnahmsweise den sogenannten *Rohtyp* der Klasse **ArrayList**, der sich aus didaktischen Gründen gut für den aktuellen Abschnitt eignet, ansonsten aber zu vermeiden ist.

```
         System.out.println("Der ArrayList-Container enthält:");
         for(Object o : al)
            System.out.printf(" %-7s  Typ: %-20s\n", o, o.getClass());
      }
   }
```

Wie die folgende Programmausgabe zeigt, sind tatsächlich diverse Wrapper-Klassen im Spiel:

```
Der ArrayList-Container enthält:
 Otto     Typ: java.lang.String
 13       Typ: java.lang.Integer
 23.77    Typ: java.lang.Double
 x        Typ: java.lang.Character
```

Dank Autoboxing klappt auch das Erzeugen eines Arrays mit Wrapper-Elementtyp per Initialisierungsliste unter Verwendung von Werten des zugehörigen primitiven Typs, z. B.:

```
Integer[] wia = {1, 2, 3};
```

In den folgenden Zeilen findet ein Auto(un)boxing statt:

```
Integer iw = 4711;
int i = iw;
```

Aus dem **Integer**-Objekt `iw` wird der eingepackte Wert entnommen und der **int**-Variablen `i` zugewiesen. Zeigt ein **Integer**-Objekt auf **null**, kommt es beim Autounboxing zur **NullPointerException**, wie das folgende Beispiel nach Bloch (2018, S. 274) demonstriert:

Quellcode	Ausgabe

`class Prog {`
`    static Integer iul;`
`    public static void main(String[] args) {`
`        System.out.println(iul == 13);`
`    }`
`}`
	`Exception in thread "main"`
	`java.lang.NullPointerException`
	` at Prog.main(Prog.java:4)`

Dank Autoboxing sind die primitiven Typen zuweisungskompatibel zur Klasse **Object**, wobei zum Auspacken aber eine explizite Typumwandlung erforderlich ist, z. B.:

```
Object o = 4711;
int i = (Integer) o;
```

Bisher haben wir die explizite Typumwandlung nur auf primitive Datentypen angewendet, sie spielt aber auch bei Referenztypen eine wichtige Rolle. Welche Konvertierungen erlaubt sind, ist der Java-Sprachspezifikation (Gosling et al. 2019, Abschnitt 5.1) zu entnehmen. Im konkreten Fall wird der deklarierte Typ (**Object**) durch eine Spezialisierung bzw. Ableitung (**Integer**) ersetzt. Der Compiler erlaubt die Konvertierung, wobei die Verantwortung beim Programmierer liegt.

5.3.3 Empfehlungen zur Verwendung von Wrapper-Objekten

Wrapper-Typen sind z. B. erforderlich ...

- als Typen für Elemente, Schlüssel und Werte in Kollektionen (siehe Kapitel 10)
- zur Konkretisierung von Typformalparametern in generischen Klassen (siehe Abschnitt 8.1).

Jedoch muss bei der Verwendung von Wrapper-Objekten stets der gravierende Unterschied im Vergleich zu Variablen mit einem primitiven Typ beachtet werden. Dieser Unterschied wird durch (Un)boxing etwas versteckt, aber nicht beseitigt. Wird der Unterschied missachtet, drohen ...

- Programmierfehler (siehe Abschnitt 5.3.3.1).
- Schlechtes Leistungsverhalten von Programmen
  Auf keinen Fall sollten die (unveränderlichen!) Wrapper-Objekte dazu verwendet werden, um sich häufig ändernde Werte aufzunehmen (siehe Abschnitt 5.3.3.2).

5.3.3.1 Identitätsoperator vermeiden

Die Anwendung des Identitätsoperators auf Wrapper-Objekt kann zu gravierenden Fehlern führen, wie das folgende Beispiel mit zwei **Integer**-Objekten demonstriert:

Quellcode:

```java
class PersonTest {
    public static void main(String[] args) {
        Integer iu = new Integer(4711);
        Integer iv = new Integer(4711);
        System.out.println(iu == iv);
        System.out.println(iu.equals(iv));
    }
}
```

Ausgabe:

```
false
true
```

Weil *Objekte* zu vergleichen sind, hängt das Ergebnis von den Speicheradressen ab. Sollen die verpackten **int**-Werte den Ausschlag geben, ist die **Integer**-Methode **equals()** zu verwenden.

Verwendet man die **Integer**-Methode **valueOf()** zur Objektkreation, dann liefert bei Initialisierungswerten von -128 bis 127 der (Speicheradressen-basierte) Identitätsoperator dasselbe Ergebnis wie die Methode **equals()**:

Quellcode:

```java
class Prog {
    public static void main(String[] args) {
        Integer ius = Integer.valueOf(47);
        Integer ivs = Integer.valueOf(47);
        System.out.println(ius == ivs);
        Integer iul = Integer.valueOf(4711);
        Integer ivl = Integer.valueOf(4711);
        System.out.println(iul == ivl);
    }
}
```

Ausgabe:

```
true
false
```

Den Grund kennen Sie aus Abschnitt 5.3.1: Liegt der **valueOf()** - Parameter im Bereich von -128 bis 127, und ist bereits ein **Integer**-Objekt mit diesem Wert vorhanden, dann wird dessen Adresse zurückgeliefert. Bei größeren Initialisierungswerten erstellt **valueOf()** auf jeden Fall ein neues Objekt. Also sollte unabhängig von der zur Objektkreation gewählten Technik bei Wrapper-Objekten der Identitätsoperator vermieden werden.

Als Alternativen zu **equals()** kommen die beiden folgenden **Integer**-Vergleichsmethoden in Frage:

- **public static int compare(int** *i*, **int** *j*)
  Die Rückgabe ist ...
  - < 0, wenn $i < j$
  - 0, wenn $i == j$
  - > 0, wenn $i > j$

  Werden **Integer**-Parameter angeboten, findet ein Autounboxing statt (vgl. Abschnitt 5.3.2).
- **public int compareTo(Integer** *i*)
  Die Rückgabe ist ...
  - < 0, wenn das angesprochene Objekt numerisch kleiner als das Parameterobjekt ist.
  - 0, wenn das angesprochene Objekt numerisch identisch ist mit dem Parameterobjekt.
  - > 0, wenn das angesprochene Objekt numerisch größer als das Parameterobjekt ist.

5.3.3.2 Wrapper-Objekte nicht für variable Werte verwenden

Weil Wrapper-Objekte unveränderlich sind, führt bei einer Wrapper-Referenzvariablen jede Wertzuweisung zu einer Objektkreation und damit zu einem erheblichen Zeitaufwand. Beim Vergleich des eingepackten Werts mit einer **int**-Zahl ist ein Unboxing erforderlich. Daher sollte auf keinen Fall ein Wrapper-Objekt einen Job übernehmen, den auch eine Variable mit dem zugehörigen primitiven Typ erfüllen kann.

Das folgende Programm benutzt ein **Integer**-Objekt in einer **for**-Schleife als Laufvariable, was zahlreiche (Un)boxing-Operationen und einen hohen Zeitaufwand (in Millisekunden) verursacht:

Quellcode:

```
class Prog {
    public static void main(String[] args) {
        double summe = 0.0;
        long start = System.currentTimeMillis();
        for (Integer i = 0; i < 10_000_000; i++)
            summe += i;
        System.out.println("Zeitaufwand:\t" +
                (System.currentTimeMillis()-start));
    }
}
```

Ausgabe:

```
127
```

Ersetzt man den Datentyp der Laufvariablen durch **int**, reduziert sich die Laufzeit auf 16 Millisekunden.[1]

5.3.4 Konvertierungsmethoden

Die Wrapper-Klassen stellen statische Methoden zum Konvertieren von Zeichenfolgen in einen Wert des zugehörigen (primitiven) Typs zur Verfügung, z. B. bei der Klasse **Double**:

- Die **Double**-Klassenmethode

  public static double parseDouble(String *str***)**
  throws NumberFormatException

  liefert einen **double**-Wert zurück, falls die Konvertierung der Zeichenfolge gelingt.
- Die bereits erwähnte Klassenmethode **valueOf()**

  public static Double valueOf(String *str***)**
  throws NumberFormatException

  liefert einen *verpackten* **double**-Wert zurück, falls die Konvertierung der Zeichenfolge gelingt. Wegen der aufwändigen Objektkreationen ist es nicht empfehlenswert, zahlreiche derartige Konvertierung vorzunehmen. Immerhin vermeidet **valueOf()** bei mehreren Aufrufen mit *derselben* Parameterzeichenfolge das Erstellen von Objekten mit identischem Inhalt (vgl. Abschnitt 5.3.1), z. B.:

  Quellcode:

  ```
  class Prog {
      public static void main(String[] args) {
          System.out.println(Double.valueOf(1.0).hashCode());
          System.out.println(Double.valueOf(1.0).hashCode());
      }
  }
  ```

  Ausgabe:

  ```
  1072693248
  1072693248
  ```

Wenn eine Konvertierung mit **parseDouble()** oder **valueOf()** scheitert, dann informieren die Methoden ihren Aufrufer durch das Werfen einer Ausnahme vom Typ **NumberFormatException**.

---

[1] Die Ergebnisse stammen von einem PC mit der Intel-CPU Core i3 550 (3,2 GHz) unter Windows 10 (64 Bit).

Über die potentiell zu erwartende Ausnahme wird in der Methodendefinition durch eine **throws**-Klausel am Ende des Methodenkopfs informiert. Bisher blieb im Manuskript bei der Beschreibung einer Methode, die potentiell Ausnahmeobjekte wirft, diese Kommunikationstechnik aus didaktischen Gründen unerwähnt. Die später noch ausführlich zu behandelnde Ausnahmetechnik ist gleich in einem Beispiel zu sehen.

Das folgende Beispielprogramm berechnet die Summe der numerisch interpretierbaren Kommandozeilenargumente:

```
class Summe {
   public static void main(String[] args) {
      double summe = 0.0;
      int fehler = 0;
      System.out.println("Ihre Eingaben:");
      for (String s : args) {
         System.out.println(" " + s);
         try {
            summe += Double.parseDouble(s);
         } catch(Exception e) {
            fehler++;
         }
      }
      System.out.println("\nSumme: " + summe + "\nFehler: "+fehler);
   }
}
```

Im Rahmen einer **try-catch** - Konstruktion, die später im Kapitel über Ausnahmebehandlung ausführlich besprochen wird, versucht das Programm für jedes Kommandozeilenargument eine numerische Interpretation mit der **Double**-Konvertierungsmethode **parseDouble()**.

Ein Aufruf mit

```
java Summe 3.5 4 5 6 sieben 8 9
```

liefert die Ausgabe:

```
Ihre Eingaben:
 3.5
 4
 5
 6
 sieben
 8
 9

Summe: 35.5
Fehler: 1
```

Um aus einem Werten eines primitiven Typs ein **String**-Objekt zu erstellen, kann man die statische Methode **valueOf()** der Klasse **String** verwenden, die in Überladungen für diverse Argumenttypen vorhanden ist, z. B.:

```
String s = String.valueOf(summe);
```

5.3.5 Konstanten für Grenz- bzw. Spezialwerte

In den numerischen Wrapper-Klassen sind öffentliche und finalisierte Klassenvariablen für diverse Grenz- bzw. Spezialwerte definiert, z. B. in der Klasse **Double**:

Konstante	Inhalt
MAX_VALUE	Größter (endlicher) Wert des Datentyps **double**
MIN_VALUE	Kleinster positiver Wert des Datentyps **double**
NaN	Not-a-Number - Ersatzwert für den Datentyp **double**
POSITIVE_INFINITY	Positiv-Unendlich - Ersatzwert für den Datentyp **double**
NEGATIVE_INFINITY	Negativ-Unendlich - Ersatzwert für den Datentyp **double**

Beispiel:

Quellcode

```
class Prog {
    public static void main(String[] args) {
        System.out.println("Max. double-Zahl:\n"+
                           Double.MAX_VALUE);
    }
}
```

Ausgabe

```
Max. double-Zahl:
1.7976931348623157E308
```

5.3.6 Character-Methoden zur Zeichen-Klassifikation

Die Wrapper-Klasse **Character** zum primitiven Typ **char** bietet einige öffentliche und statische Methoden zur Klassifikation von Unicode-Zeichen, die bei der Verarbeitung von Textdaten sehr nützlich sein können:

Methode	Erläuterung
boolean isDigit(char *ch*)	Die Methode liefert den Wert **true** zurück, wenn das übergebene Zeichen eine Ziffer ist, sonst **false**.
boolean isLetter(char *ch*)	Die Methode liefert den Wert **true** zurück, wenn das übergebene Zeichen ein Buchstabe ist, sonst **false**.
boolean isLetterOrDigit(char *ch*)	Die Methode liefert den Wert **true** zurück, wenn das übergebene Zeichen ein Buchstabe oder eine Ziffer ist, sonst **false**.
boolean isWhitespace(char *ch*)	Die Methode liefert den Wert **true** zurück, wenn ein Trennzeichen übergeben wurde, sonst **false**. Zu den Trennzeichen gehören in Java (siehe Gosling et al. 2019, Abschnitt 3.6): • Leerzeichen (\u0020) • Tabulatorzeichen (\u0009) • Wagenrücklauf (\u000D) • Zeilenvorschub (\u000A)
boolean isLowerCase(char *ch*)	Die Methode liefert den Wert **true** zurück, wenn das übergebene Zeichen ein Kleinbuchstabe ist, sonst **false**.
boolean isUpperCase(char *ch*)	Die Methode liefert den Wert **true** zurück, wenn das übergebene Zeichen ein Großbuchstabe ist, sonst **false**.

5.4 Aufzählungstypen

Angenommen, Sie wollen in eine Adressendatenbank auch den Charakter der erfassten Personen eintragen und sich dabei an den vier Temperamentstypen des griechischen Philosophen Hippokrates (ca. 460 - 370 v. Chr.) orientieren: melancholisch, cholerisch, phlegmatisch, sanguin. Um dieses Merkmal mit seinen vier möglichen Ausprägungen in einer Instanzvariablen zu speichern, haben Sie verschiedene Möglichkeiten, z. B.

- Eine **String**-Variable zur Aufnahme der Temperamentsbezeichnung
  Hier drohen Fehler durch inkonsistente Schreibweisen, z. B.:
  ```
  if (otto.temp == "Fleckmatisch") ...
  ```
- Eine **int**-Variable mit der Kodierungsvorschrift 0 = melancholisch, 1 = cholerisch, etc.
  Hier ist der Quellcode nur für Eingeweihte zu verstehen, z. B.:
  ```
  if (otto.temp == 3) ...
  ```

Durch Datenkapselung mit entsprechenden Zugriffsmethoden sowie sorgfältige Arbeitsweise des Klassendesigners könnte man immerhin für eine Instanzvariable vom Typ **String** oder **int** sicherstellen, dass ausschließlich die vier vorgesehenen Temperamentswerte auftreten. Im Quellcode vorhandene Versuche, einen irregulären Wert zu vergeben, würden allerdings erst zur Laufzeit entdeckt.

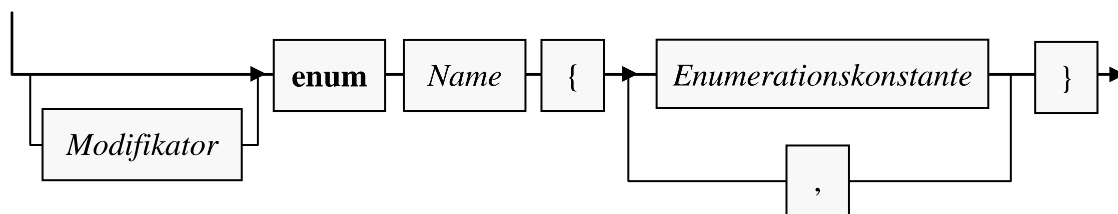
Java enthält mit den **Enumerationen** (**Aufzählungstypen**) eine Lösung, die folgende Vorteile bietet:

- Eine exakt definierte Menge gültiger Werte
  Damit ist die Einhaltung des Wertebereichs nicht mehr von der Sorgfalt des Klassendesigners abhängig, sondern wird vom Compiler sichergestellt. Außerdem werden im Quellcode vorhandene Versuche, einen irregulären Wert zu vergeben, schon zur Übersetzungszeit entdeckt und nicht erst zur Laufzeit.
- Gut lesbarer Quellcode
  Im obigen Beispiel kann z. B. der folgende logische Ausdruck verwendet werden:
  ```
  if (otto.temp == Temperament.PHLEGMATISCH) ...
  ```

5.4.1 Einfache Enumerationstypen

Bei der Definition eines einfachen Aufzählungstyps folgt nach dem optionalen Zugriffsmodifikator auf das Schlüsselwort **enum** und den Typbezeichner eine geschweift eingeklammerte Liste mit den Namen für eine feste Anzahl von Enumerationskonstanten:

Einfache Enumerationsdefinition



Weil Syntaxdiagramme zwar präzise, aber nicht unbedingt auf den ersten Blick verständlich sind, betrachten wir ergänzend ein Beispiel:

```
public enum Temperament {MELANCHOLISCH, CHOLERISCH, PHLEGMATISCH, SANGUIN}
```

Es hat sich eingebürgert, die Namen der Enumerationskonstanten komplett groß zu schreiben.

Ein Aufzählungstyp kann auf Paketebene sowie innerhalb einer Klasse definiert werden (vgl. Abschnitt 4.8.1), aber nicht innerhalb einer Methode. Für eine Top-Level - Enumeration sollte eine eigene Quellcodedatei verwendet werden. Bei einer Top-Level - Enumeration mit der Zugriffsstufe **public** ist dies obligatorisch. Wird ein Aufzählungstyp ausschließlich in einer bestimmten Klasse verwendet, kommt die Definition als statischer Mitgliedstyp in Frage (siehe Abschnitt 4.8.1.2).

Objekte der folgenden Klasse `Person` (der Einfachheit halber ohne Datenkapselung) erhalten eine Instanzvariable vom eben definierten Aufzählungstyp `Temperament`:

```
public class Person {
   public String vorname, name;
   public int alter;
   public Temperament temp;
   public Person(String vor, String nach, int alt, Temperament tp) {
      vorname = vor;
      name = nach;
      alter = alt;
      temp = tp;
   }
   public Person() {}
}
```

Weil Enumerationskonstanten mit dem Typnamen qualifiziert werden müssen, ist einige Tipparbeit erforderlich, die aber durch einen gut lesbaren Quellcode belohnt wird:[1]

```
class PersonTest {
   public static void main(String[] args) {
      Person otto = new Person("Otto", "Hummer", 35, Temperament.SANGUIN);
      if (otto.temp == Temperament.SANGUIN)
         System.out.println("Lustiger Typ");
}
```

Eine Variable mit Aufzählungstyp ist als steuernder Ausdruck einer **switch**-Anweisung erlaubt (vgl. Abschnitt 3.7.2.3), wobei die Enumerationskonstanten in den **case**-Marken aber ausnahmsweise *ohne* den Typnamen zu schreiben sind, z. B.:

```
switch (otto.temp) {
   case MELANCHOLISCH:  System.out.println("Nicht gut drauf"); break;
   case CHOLERISCH:     System.out.println("Mit Vorsicht zu genießen"); break;
   case PHLEGMATISCH:   System.out.println("Lahme Ente"); break;
   case SANGUIN:        System.out.println("Lustiger Typ");
}
```

Bisher konnte man den Eindruck gewinnen, als wäre eine Enumeration ein Ganzzahltyp mit einer kleinen Menge von benannten Werten. Tatsächlich ist eine Enumeration aber eine *Klasse* mit der Basisklasse **Enum** aus dem Paket **java.lang** und folgenden Besonderheiten:

- Die Enumerationskonstanten zeigen als statische und finalisierte Referenzvariablen auf Objekte der Enumerationsklasse, die beim Laden der Klasse automatisch erstellt werden. Nun ist klar, warum den Enumerationskonstanten (von Ausnahmen abgesehen) der Typname vorangestellt werden muss.
- Es ist nicht möglich, weitere Objekte der Enumerationsklasse (per **new**-Operator oder auf andere Weise) zu erzeugen.
- Die Objekte eines *einfachen* Enumerationstyps sind unveränderlich, sodass die (aus der JLS stammende) Bezeichnung *Enumerationskonstante* konsistent mit dem im Abschnitt 4.5.1 beschriebenen Begriff eines *konstanten Felds* verwendet wird. Die Objekte der im Abschnitt 5.4.2 beschriebenen *erweiterten* Enumerationstypen sind aber *nicht* unbedingt unveränderlich.
- Man kann eine Enumeration nicht beerben.
  Im Abschnitt 7.1 werden wir solche Klassen als *finalisiert* bezeichnen.

---

[1] Im Abschnitt 6.1.2.2 werden wir eine Möglichkeit kennenlernen, Wiederholungen des Aufzählungstypnamens im Quellcode zu vermeiden: Mit der Deklaration **import static** kann man alle statischen Variablen und Methoden eines Typs importieren, sodass sie anschließend wie klasseneigene angesprochen werden können, sofern entsprechende Zugriffsrechte bestehen. Wie gleich zu erfahren ist, handelt es sich bei den Enumerationskonstanten um statische und finalisierte Referenzvariablen.

In obigem Beispiel ist die `Person`-Eigenschaft `temp` eine Referenzvariable vom Typ `Temperament`. Sie zeigt …

- entweder auf eines der vier `Temperament`-Objekte
- oder auf **null**.

Die Enumerationsobjekte kennen ihre Position in der definierenden Liste und liefern diese als Rückgabewert der Instanzmethode **ordinal()**, z. B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class PersonTest {
   public static void main(String[] args) {
      Person otto = new Person("Otto", "Hummer",
                               35, Temperament.SANGUIN);
      System.out.println(otto.temp.ordinal());
   }
}</pre></td><td><pre>3</pre></td></tr>
</table>

Bei jeder Enumerationsklasse kann man mit der statischen Methode **values()** einen Array mit ihren Objekten anfordern, z. B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class PersonTest {
   public static void main(String[] args) {
      for (Temperament t : Temperament.values())
         System.out.println(t.name());
   }
}</pre></td><td><pre>MELANCHOLISCH
CHOLERISCH
PHLEGMATISCH
SANGUIN</pre></td></tr>
</table>

5.4.2 Erweiterte Enumerationstypen

Es ist möglich, eine Enumerationsklasse mit Instanzvariablen, Methoden und privaten Konstruktoren auszustatten. Objekte der folgenden Enumeration `TemperamentEx` geben über die Methoden `stable()` bzw. `extra()` Auskunft darüber, ob die zugehörige Persönlichkeit emotional stabil bzw. extravertiert ist:[1]

```
public enum TemperamentEx {
   MELANCHOLISCH(false, false),
   CHOLERISCH(false, true),
   PHLEGMATISCH(true, false),
   SANGUIN(true, true);

   private final boolean stable, extra;
   private TemperamentEx(boolean stab, boolean ex) {
      stable = stab;
      extra = ex;
   }

   public boolean stable() {return stable;}
   public boolean extra() {return extra;}
}
```

Diese Informationen befinden sich in Instanzvariablen, welche von einem Konstruktor initialisiert werden. Der Konstruktor ist nur innerhalb der Enumerationsklasse nutzbar. Dazu werden Aktualparameterlisten an die Enumerationskonstanten angehängt.

---

[1] Informationen zu den Persönlichkeitsdimensionen *emotionale Stabilität* und *Extraversion* sowie zum Zusammenhang mit den Typen des Hippokrates finden Sie z. B. in: Mischel, W. (1976). *Introduction to Personality*, S.22.

Im Beispiel sind die Instanzvariablen des erweiterten Enumerationstyps als **final** deklariert, sodass unveränderliche Enumerationsobjekte resultieren. Die Objekte eines erweiterten Enumerationstyps können aber auch als *veränderlich* konzipiert werden, z. B.:

```
private boolean stable, extra;
 . . .
public void changeExtra(boolean ex) {extra = ex;}
```

Über die sogenannte *konstanten-spezifische Methodenimplementation* erstellt man zu einer Enumeration eine Methode, die von den Objekten des Typs (ansprechbar über die Enumerationskonstanten) unterschiedlich ausgeführt wird. Ein Beispiel ist die von Bloch (2018, S. 162f) beschriebene Enumeration *Operation* mit Objekten für die vier Grundrechnungsarten und der jeweils speziell implementierten Methode **apply()**:

public double apply(double *x*, **double** *y*)

5.5 Übungsaufgaben zum Kapitel 5

Abschnitt 5.1 (Arrays)

1) Welche der folgenden Aussagen sind richtig bzw. falsch?

1. Die Länge eines Arrays muss zur Übersetzungszeit festgesetzt werden.
2. Die Länge eines Arrays muss beim Erzeugen (zur Laufzeit) festgesetzt werden.
3. Array-Elemente werden automatisch mit der typspezifischen Null initialisiert, weil es sich um Instanzvariablen handelt.
4. Von der erweiterten **for**-Schleife (siehe Abschnitt 3.7.3.2) werden auch Arrays unterstützt.
5. Die Länge eines Arrays lässt sich mit der Instanzmethode **length()** ermitteln.

2) Erstellen Sie ein Java-Programm, das 6 Lottozahlen (von 1 bis 49) zieht und sortiert ausgibt. Zum Sortieren können Sie z. B. das (sehr einfache) **Auswahlverfahren** (engl.: *Selection Sort*) verwenden:

- Für den Ausgangsvektor mit den Elementen 0, …, *n*-1 wird das Minimum gesucht und an den linken Rand befördert. Dann wird der Vektor mit den Elementen 1, …, *n*-1 analog behandelt, usw.
- Bei jeder Teilaufgabe muss man das kleinste Element eines Vektors an seinen linken Rand befördern, was auf folgende Weise geschehen kann:
    - Man geht davon aus, das Element am linken Rand sei das kleinste (genauer: *ein* Minimum).
    - Es wird sukzessive mit seinen rechten Nachbarn verglichen. Ist das Element an der Position *i* kleiner, so tauscht es mit dem „Linksaußen“ seinen Platz.
    - Nun steht am linken Rand ein Element, das die anderen Elemente mit Positionen kleiner oder gleich *i* nicht übertrifft. Es wird nun sukzessive mit den Elementen an den Positionen ab *i*+1 verglichen.
    - Nachdem auch das Element an der letzten Position mit dem Element am linken Rand verglichen worden ist, steht mit Sicherheit am linken Rand ein Element, zu dem sich kein kleineres findet.

Diese Aufgabe soll Erfahrung im Umgang mit Arrays und einen ersten Eindruck von Sortieralgorithmen vermitteln. Im Programmieralltag empfiehlt sich für derartige Probleme die statische Methode **sort()** der Klasse **Arrays** im Paket **java.util**.

3) Erstellen Sie ein Programm zur Primzahlensuche mit dem ***Sieb des Eratosthenes***.[1] Dieser Algorithmus reduziert sukzessive eine Menge von Primzahlkandidaten, die initial alle natürlichen Zahlen bis zu einer Obergrenze K enthält, also $\{2, 3, ..., K\}$:

- Im ersten Schritt werden alle echten Vielfachen der Basiszahl 2 (also 4, 6, ...) aus der Kandidatenmenge gestrichen, während die Zahl 2 in der Liste verbleibt.
- Dann geschieht iterativ folgendes:
  - Als neue Basis b wird die kleinste Zahl gewählt, welche die beiden folgenden Bedingungen erfüllt:
    b ist größer als die vorherige Basiszahl.
    b ist im bisherigen Verlauf nicht gestrichen worden.
  - Die echten Vielfachen der neuen Basis (also $2 \cdot b$, $3 \cdot b$, ...) werden aus der Kandidatenmenge gestrichen, während die Zahl b in der Liste verbleibt.
- Ist für eine neue Basis b die folgende Ungleichung

$$b > \sqrt{K}$$

  erfüllt, dann kann das Streichverfahren enden, ohne die Vielfachen der neuen Basis auch noch streichen zu müssen.

In der Kandidatenrestmenge befinden sich dann nur noch Primzahlen. Um dies einzusehen, nehmen wir an, es hätte eine Zahl $n \leq K$ mit echtem Teiler das beschriebene Streichverfahren überstanden. Mit zwei positiven Zahlen u, v würde dann gelten:

$$n = u \cdot v \text{ und } u \leq \sqrt{K} \text{ oder } v \leq \sqrt{K} \text{ (wegen } n \leq K\text{)}$$

Wir nehmen ohne Beschränkung der Allgemeinheit $u \leq \sqrt{K}$ an und unterscheiden zwei Fälle:

- u war zuvor als Basis dran.
  Dann wurde n bereits als Vielfaches von u gestrichen.
- u wurde zuvor als Vielfaches einer früheren Basis $\tilde{b}$ ($< b$) gestrichen ($u = k\tilde{b}$).
  Dann wurde auch n bereits als Vielfaches von $\tilde{b}$ gestrichen.

Damit erweist sich die Annahme als falsch, und es ist gezeigt, dass die Kandidatenrestmenge nur noch Primzahlen enthält.

Sollen z. B. alle Primzahlen kleiner oder gleich 18 bestimmt werden, so startet man mit folgender Kandidatenmenge:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Im ersten Schritt werden die echten Vielfachen der Basis 2 gestrichen:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Als neue Basis wird die Zahl 3 gewählt (> 2, nicht gestrichen). Ihre echten Vielfachen werden gestrichen:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Als neue Basis wird die Zahl 5 gewählt (> 3, nicht gestrichen). Allerdings ist 5 größer als $\sqrt{18}$ (≈ 4,24), und der Algorithmus daher bereits beendet. Als Primzahlen kleiner oder gleich 18 erhalten wir also:

2, 3, 5, 7, 11, 13 und 17

[1] Der griechische Gelehrte Eratosthenes lebte laut Wikipedia ca. von 275 bis 194 v. Chr.

4) Definieren Sie eine Klasse für eine zweidimensionale Matrix mit Elementen vom Typ **double** zur Aufnahme von Beobachtungswerten aus einer empirischen Studie. Implementieren Sie …

- eine Methode zum Transponieren der Matrix
- Methoden für elementare statistische Analysen mit den Spalten der Matrix:
  - Eine Methode sollte den eindimensionalen Array mit den Mittelwerten der Spalten als Rückgabe liefern. Der Mittelwert aus den Beobachtungswerten $x_1, x_2, \ldots, x_n$ ist definiert durch

$$\bar{x} := \frac{1}{n}\sum_{i=1}^{n} x_i$$

  - Eine Methode sollte den eindimensionalen Array mit den Varianzen der Spalten als Rückgabe liefern. Der erwartungstreue Schätzer für die Varianz der zu einer Spalte gehörigen Zufallsvariablen mit den Beobachtungswerten $x_1, x_2, \ldots x_n$ ist definiert durch

$$\hat{\sigma}^2 := \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Zur Vereinfachung der Berechnung kann die folgende *Verschiebungsformel* dienen:

$$\sum_{i=1}^{n}(x_i - \bar{x})^2 = \sum_{i=1}^{n} x_i^2 - n\bar{x}^2$$

Sie ermöglicht es, die Varianz einer Variablen bei *einer* einzigen Passage durch die Daten zu berechnen, während die Originalformel eine vorgeschaltete Passage zur Berechnung des Mittelwerts benötigt.

Abschnitt 5.2 (Klassen für Zeichen)

1) Welche der folgenden Aussagen sind richtig bzw. falsch?

1. Mit Hilfe der **for**-Schleife für Arrays und Kollektionen (vgl. Abschnitt 3.7.3.2) kann man bequem über die Zeichen eines **String**-Objekts iterieren.
2. Die Anzahl der Zeichen in einem **String**-Objekt lässt sich mit der Instanzmethode **length()** ermitteln.
3. Auf die Zeichen eines **String**-Objekts kann man wie bei einem Array per Indexoperator zugreifen.
4. Ein **String**-Objekt kann nach dem Erstellen nicht mehr geändert werden.

2) Durch welche Anweisungen des folgenden Programms wird ein **String**-Objekt neu in den internen **String**-Pool aufgenommen?

```
class Prog {
    public static void main(String[] args) {
        String s1 = "abcde";                // (1)
        String s2 = new String("abcde");  // (2)
        String s3 = new String("cdefg");  // (3)
        String s4, s5;
        s4 = s2.intern();                   // (4)
        s5 = s3.intern();                   // (5)
        System.out.print("(s1 == s2) = " + (s1==s2)+
                          "\n(s1 == s4) = " + (s1==s4)+
                          "\n(s1 == s5) = " + (s1==s5));
    }
}
```

3) Erstellen Sie ein Programm zum Berechnen einer persönlichen Glückszahl (zwischen 1 und 100), indem Sie:

- Vor- und Nachnamen als Programmargumente einlesen,
- den Anfangsbuchstaben des Vornamens sowie den letzten Buchstaben des Nachnamens ermitteln (beide in Großschreibung),
- die Nummern der beiden Buchstaben im Unicode-Zeichensatz bestimmen,
- die beiden Buchstabennummern addieren und die Summe als Initialisierungswert für den Pseudozufallszahlengenerator verwenden.

Beenden Sie Ihr Programm mit einer Fehlermeldung, wenn weniger als zwei Programmargumente übergeben werden.

Tipp: Um ein Programm spontan zu beenden, kann man die statische Methode **exit()** der Klasse **System** verwenden.

4) Die Klassen **String** und **StringBuilder** besitzen beide eine Methode namens **equals()**, doch bestehen gravierende Verhaltensunterschiede:

Quellcode:

```
class Prog {
    public static void main(String[] args) {
        StringBuilder sb1 = new StringBuilder("abc");
        StringBuilder sb2 = new StringBuilder("abc");
        System.out.println("sb1 = sb2 = " + sb1);
        System.out.println("StringBuilder-Vergl.: " +
                sb1.equals(sb2));

        String s1 = sb1.toString();
        String s2 = sb1.toString();
        System.out.println("\ns1 = s2 = " + s1);
        System.out.println("String-Vergl.:        " +
                s1.equals(s2));
    }
}
```

Ausgabe:

```
sb1 = sb2 = abc
StringBuilder-Vergl.: false

s1 = s2 = abc
String-Vergl.:        true
```

Ermitteln Sie mit Hilfe der API-Dokumentation die Ursache für das unterschiedliche Verhalten.

5) Erstellen Sie eine Klasse `StringUtil` mit einer statischen Methode `wrapln()`, die einen **String** auf die Konsole schreibt und dabei einen korrekten Zeilenumbruch vornimmt. Anwender Ihrer Methode sollen die gewünschte Zeilenbreite vorgeben und auch die Trennzeichen festlegen dürfen, aber nicht müssen (Methoden überladen!). Am Anfang einer neuen Zeile sollen außerdem keine Leerzeichen stehen.

Im folgenden Programm wird die Verwendung der Methode demonstriert:

```
class StringUtilTest {
   public static void main(String[] args) {
      String s = "Dieser Satz passt nicht in eine Schmal-Zeile, "+
                 "die nur wenige Spalten umfasst.";
      StringUtil.wrapln(s);
      StringUtil.wrapln(s, 40);
      StringUtil.wrapln(s, " ", 40);
   }
}
```

Der zweite `wrapln()` - Methodenaufruf sollte die folgende Ausgabe mit einer auf 40 Zeichen begrenzten Breite erzeugen, weil der Bindestrich zu den voreingestellten Trennzeichen gehört:

```
Dieser Satz passt nicht in eine Schmal-
Zeile, die nur wenige Spalten umfasst.
```

Ein wesentlicher Schritt zur Lösung des Problems ist die Zerlegung der Zeichenfolge in Einzelbestandteile (sogenannte *Tokens*), die nach Möglichkeit nicht durch einen Zeilenumbruch aufgetrennt werden sollten. Diese Zerlegung können Sie einem Objekt der Klasse **StringTokenizer** aus dem Paket **java.util** überlassen. Im folgenden Programm wird demonstriert, wie ein **StringTokenizer** arbeitet:

Quellcode	Ausgabe

<pre>class Prog {
    public static void main(String[] args) {
        String s = "Dies ist der Satz, der zerlegt werden soll.";
        java.util.StringTokenizer stok =
                new java.util.StringTokenizer(s, " ", false);
        while (stok.hasMoreTokens())
            System.out.println(stok.nextToken());
    }
}</pre>
	<pre>Dies
	ist
	der
	Satz,
	der
	zerlegt
	werden
	soll.</pre>

In der verwendeten Überladung des **StringTokenizer** - Konstruktors legt der zweite Parameter (Typ **String**) die Trennzeichen fest. Hat der dritte Parameter (Typ **boolean**) den Wert **true**, dann sind die Trennzeichen im Ergebnis als eigene Tokens (mit Länge 1) enthalten. Anderenfalls werden sie nur zum Separieren verwendet und danach verworfen.

Abschnitt 5.3 (Verpackungsklassen für primitive Datentypen)

1) Ermitteln Sie den kleinsten möglichen Wert des Datentyps **byte**.

2) Ermitteln Sie die maximale natürliche Zahl *k*, für die unter Verwendung des Funktionswertedatentyps **double** die Fakultät *k*! bestimmt werden kann.

3) Entwerfen Sie eine Verpackungsklasse, welche die Aufnahme von **int**-Werten in Container wie **ArrayList** ermöglicht, ohne (wie die Klasse **Integer**) die Werte der Objekte nach der Erzeugung zu fixieren. Ein unvermeidlicher Nachteil der selbstgestrickten Verpackungsklasse im Vergleich zur Klasse **Integer** ist das fehlende Auto(un)boxing.

Anmerkungen:

- Zur Vermeidung von Missverständnissen sei betont, dass im Abschnitt 5.3 vorgestellten Verpackungsklassen nicht etwa versehentlich als unveränderlich konzipiert wurden. Benutzt ein Programm mehrere Ausführungsfäden (Threads), dann sind unveränderliche Objekte von Vorteil.
- Das Java-API enthält im Paket **java.util.concurrent.atomic** Klassen für *veränderliche* Wrapper-Objekte. Ihre mit dem Wort **Atomic** startenden Namen (z. B. **AtomicInteger**) zeigen an, dass diese Klassen die Wertveränderlichkeit kombinieren mit der Synchronisation von Zugriffen durch mehrere Ausführungsfäden (Threads).

Abschnitt 5.4 (Aufzählungstypen)

1) Erstellen und erproben Sie einen Datentyp namens `Wochentag`, der folgende Bedingungen erfüllt:

- Typsicherheit
  Einer Variablen vom Typ `Wochentag` können nur sieben verschiedene Werte zugewiesen werden, die den Wochentagen Sonntag, Montag, usw. entsprechen.
- Ordnungsinformation
  Für zwei Werte des Typs `Wochentag` kann leicht die zeitliche Anordnung festgestellt werden.
- Leicht lesbarer Quellcode
- Verwendbarkeit als Datentyp für den steuernden Ausdruck einer **switch**-Anweisung

6 Pakete und Module

Jede größere Anwendung enthält zahlreiche Klassen, und in der Standardbibliothek von Java-SE sind ca. 6000 Klassen vorhanden.[1] Der Ordnung und Funktionalität halber werden die Klassen der Java-Programme und -Bibliotheken in *Pakete* (engl.: *packages*) eingeordnet. Im Java-SE - API gibt es über 200 Pakete mit zusammengehörigen Klassen. Bei einfachen Anwendungen kann man auf eine explizite Paketierung verzichten und damit das unbenannte Standardpaket (engl.: *default package*) verwenden. Weil jedes Programm aber auch Bibliotheken benötigt, die *stets* in Paketen organisiert sind, kann man sagen, dass *jedes* Java-Programm aus mehreren Paketen besteht. In den Worten der Java-Sprachspezifikation (Gosling et al 2019, S. 179):

> Programs are organized as sets of packages.

Bei größeren und insbesondere bei öffentlich verbreiteten Anwendungen sollte man aus gleich zu erläuternden Gründen die Klassen des Programms in ein benanntes Paket zu stecken oder (je nach Größe des Programms) auf mehrere Pakete zu verteilen.

Neben den Klassen spielen in der objektorientierten Programmierung die sogenannten *Interfaces* (deutsch: *Schnittstellen*) eine wichtige Rolle, und in den meisten Paketen sind sowohl Klassen als auch Interfaces zu finden. Ein Interface taugt wie eine Klasse als Datentyp und enthält in der Regel ebenfalls Methoden, doch fehlt bei den Interface-Methoden meist die Implementation (der Anweisungsblock). Eine typische Interface-Definition listet Methoden auf (definiert durch Rückgabetyp, Name und Parameterliste), die eine Klasse implementieren muss, wenn sie von sich behaupten möchte, dem Interface-Datentyp zu genügen. Als Beispiel betrachten wir die Schnittstelle **Comparable** aus dem API-Paket **java.lang**, die sich auf eine Methode namens **compareTo()** beschränkt:[2]

```
public interface Comparable {
    public int compareTo(Object obj);
}
```

Wie bei einer Klasse erstellt der Java-Compiler auch aus dem Schnittstellen-Quellcode eine Bytecode-Datei mit der Namenserweiterung **class**. Wir werden uns im Kapitel 9 mit den Interfaces und ihrer Rolle bei der objektorientierten Programmierung ausführlich beschäftigen. Im Manuskript ist ab jetzt von *Typen* die Rede, wenn sowohl Klassen als auch Interfaces einbezogen werden sollen.

Während wir bislang (z. B. im Abschnitt 3.1.1) dem jeweiligen Lernfortschritt angemessen festgestellt haben, ein Java-Programm bestehe aus Klassen, kommen wir nun zur genaueren Formulierung, dass ein Java-Programm aus mehreren Paketen besteht, die Typen (Klassen und Schnittstellen) enthalten.

Pakete erfüllen wichtige Aufgaben:

- **Große Projekte strukturieren**
  Wenn in einem Programm oder in einer Bibliothek viele Typen vorhanden sind, sollte man diese nach funktionaler Verwandtschaft auf mehrere Pakete verteilen. In jeder Quellcodedatei, die zum Paket gehörige Typen (Klassen oder Interfaces) definiert, ist eine **package**-Deklaration mit der Paketbezeichnung (siehe Abschnitt 6.1) an den Anfang zu stellen, z. B.:

  ```
  package java.util.concurrent;
  ```

  Es ist ein *hierarchischer* Aufbau über **Unterpakete** möglich. Im Namen eines Unterpakets folgen die Namen aus dem Paketpfad durch Punkte getrennt aufeinander, z. B.:

  java.util.concurrent

---

[1] Hier befindet sich eine Liste mit allen Klassen in Java 10:
https://docs.oracle.com/javase/10/docs/api/allclasses-noframe.html

[2] Damit nicht zu viele Neuerungen gleichzeitig auftauchen, wird hier die veraltete, nicht-generische Variante der Schnittstelle **Comparable** präsentiert.

Bei Ablage in einem Dateisystem wird die Paketstruktur auf einen Dateiverzeichnisbaum abgebildet. Alle **class**-Dateien mit den Klassen und Schnittstellen eines Pakets werden in einem gemeinsamen Ordner abgelegt, dessen Name mit dem Paketnamen übereinstimmt.

- **Namenskonflikte vermeiden**
  Jedes Paket bildet einen eigenen Namensraum für Typen, und der vollqualifizierte Name eines Typs beginnt mit dem Namen des Pakets, in dem er sich befindet. Identische *einfache* Typnamen stellen also kein Problem dar, solange sich die Typen in verschiedenen Paketen befinden.
- **Zugriffskontrolle steuern**
  Per Voreinstellung ist ein Typ nur innerhalb des eigenen Pakets sichtbar. Damit er auch von Typen aus fremden Paketen genutzt werden kann, muss im Kopf der Typdefinition der Zugriffsmodifikator **public** gesetzt werden.
- **Typen gelangen nur als Bestandteile von benannten Paketen in die Öffentlichkeit**
  Für Typen im unbenannten Standardpaket ist der **public**-Modifikator nutzlos, weil diese Typen definitiv nur im Standardpaket sichtbar sind. Nur für Typen in einem benannten Paket hat der **public**-Modifikator die Sichtbarkeit in anderen Paketen (aus berechtigten Modulen) zur Folge. Als wir im Abschnitt 4.9.3 unsere Klasse `Bruch` in ein JavaFX-Programm aufgenommen haben, das ein Assistent unserer Entwicklungsumgebung in ein eigenes Paket gesteckt hatte, sind wir auf das Problem gestoßen.

Bei der Paketierung handelt es sich nicht um eine *Option* für große Projekte, sondern um ein universelles Prinzip: Jeder Typ (Klasse oder Interface) gehört zu einem Paket. Wird ein Typ keinem Paket explizit zugeordnet, gehört er zum unbenannten Standardpaket (siehe Abschnitt 6.1.1.2).

Im Quellcode eines Typs müssen fremde Typen prinzipiell über ein durch Punkt getrenntes Paar aus Paketnamen und Typnamen angesprochen werden, wie Sie es schon in zahlreichen Beispielprogrammen beobachten konnten. Bei manchen Typen ist aber *kein* Paketname erforderlich:

- bei Typen aus **demselben** Paket
  Bei unseren bisherigen Beispielprogrammen befanden sich meist alle selbst erstellten Klassen im Standardpaket, sodass kein Paketname erforderlich war. Im speziellen Fall des Standardpakets existiert auch gar kein Name.
- bei Typen aus **importierten** Paketen
  Importiert man ein Paket per **import**-Deklaration in eine Quellcodedatei (siehe Abschnitt 6.1.2.2), denn können seine Typen *ohne* Paketnamen angesprochen werden. Das Paket **java.lang** mit besonders wichtigen Klassen (z. B. **Object**, **System**, **String**) wird automatisch in jede Quellcodedatei importiert.

Ale zweifellos größte Errungenschaft von Java 9 kann die lange unter dem Projektnamen *Jigsaw* (dt.: *Puzzle*, *Stichsäge*) angestrebte und schließlich unter der offiziellen Bezeichnung **JPMS** (*Java Platform Module System*) realisierte **Zusammenfassung von Paketen zu Modulen** angesehen werden. Wichtige Leistungen des Java-Modulsystems sind:

- Module erlauben eine **bessere Zugriffsregulation** durch eine neue Ebene oberhalb der Pakete, indem zwischen öffentlich zugänglichen und privaten (für andere Module verborgenen) Paketen differenziert wird. Ein als **public** deklarierter Typ ist zunächst nur in den Paketen seines eigenen Moduls sichtbar. Exportiert Modul `A` ein Paket, sind die öffentlichen Typen dieses Pakets in allen anderen Modulen sichtbar, die eine Abhängigkeit von Modul `A` deklariert haben. Die Rolle der Pakete und Module bei der Zugriffsverwaltung wird im Abschnitt 6.3 genauer erläutert.
- Mit dem **JPMS** wurde der **Modulpfad** als Ersatz für den herkömmlichen Klassenpfad (z. B. definiert über die CLASSPATH-Umgebungsvariable) eingeführt, um zur Laufzeit das Laden eines Typs aus einem falschen Paket zu verhindern.

- Um ein selbständig lauffähiges Java-Programm zu erstellen, das auf einem Kundenrechner keine JVM voraussetzt, muss man dank JPMS nur die tatsächlich verwendeten Module der Java-Standardbibliothek integrieren. Das spart Speicherplatz, Übertragungszeit und mindert das Risiko, von einem später entdeckten Sicherheitsproblem im Java-API betroffen zu sein. Allerdings muss sich nun der Programmherausgeber um Updates kümmern, wenn doch Sicherheitsprobleme auftreten.

Pakete sind seit der ersten Java-Version ein wesentliches Konzept zur Strukturierung von Programmen, und sie bleiben es auch in einer Java-Version mit JPMS. Im Abschnitt 6.1 werden Basisbegriffe der Paketierung behandelt sowie die bis Java 8 üblichen Techniken zur Verwendung und Verteilung von Paketen beschrieben. Im Abschnitt 6.2 werden die mit Java 9 eingeführten Neuerungen (speziell die Module) vorgestellt, welche die älteren Begriffe und Techniken ergänzen und teilweise ersetzen.

Es ist für Kompatibilität gesorgt, sodass Programme, die für Java 8 entwickelt wurden (oder werden) und den traditionellen Klassenpfad verwenden, auch von einer Java-Laufzeitumgebung ab Version 9 ausgeführt werden können.

6.1 Pakete

In diesem Abschnitt wird die traditionelle Java-Pakttechnik vorgestellt, die aktuell (Dezember 2019) noch dominiert (auch bei neu entwickelter Software).

6.1.1 Pakete erstellen

6.1.1.1 package-Deklaration und Paketordner

Wir betrachten ein einfaches Paket namens `demopack` mit den Klassen `A`, `B` und `C`. An den Anfang jeder einzubeziehenden Quellcodedatei ist eine **package**-Deklaration mit dem Paketnamen zu setzen, der üblicherweise *komplett klein* geschrieben wird, z. B.:[1]

```
package demopack;

public class A {
   private static int anzahl;
   private int objnr;

   public A() {
      objnr = ++anzahl;
   }

   public void prinr() {
      System.out.println("Klasse A, Objekt Nr. " + objnr);
   }
}
```

Vor der **package**-Deklaration dürfen höchstens Kommentar- oder Leerzeilen stehen.[2]

Sind in einer Quellcodedatei *mehrere* Typdefinitionen vorhanden, was in Java nur unter bestimmten Bedingungen erlaubt und generell nicht zu empfohlen ist, dann werden *alle* Typen (Klassen und Interfaces) dem Paket zugeordnet.

---

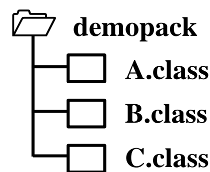
[1] Die Kleinschreibung ist nur eine empfohlene Konvention, und man findet z. B. im Java-API auch Paketnamen mit Großbuchstaben und/oder Ziffern.

[2] Soll das Paket mit einer Annotation versehen werden, hat dies in der Datei **package-info.java** zu geschehen, die im Paketordner abzulegen ist. In der Java-Sprachspezifikation wird empfohlen, in dieser Datei (vor der **package**-Deklaration) auch die (von Werkzeugen wie **javadoc** auszuwertenden) Dokumentationskommentare zum Paket unterzubringen (Gosling et al. 2019, Abschnitt 7.4.1).

Die Typen eines benannten Pakets können von Typen in fremden Paketen nur dann verwendet werden, wenn durch den Modifikator **public** die Genehmigung dazu erteilt wurde. Zusätzlich müssen auch Methoden, Konstruktoren, Felder und andere Mitglieder eines Typs explizit per Zugriffsmodifikator für fremde Pakete freigegeben werden. Steht z. B. in einer als öffentlich deklarierten Klasse kein **public**-Konstruktor zur Verfügung, können fremde Pakete die Klasse zwar „sehen" und auf öffentliche statische Mitglieder zugreifen, aber keine Objekte dieses Typs erzeugen. Mit den Zugriffsrechten für Typen und Typmitglieder werden wir uns im Abschnitt 6.3 beschäftigen.

Bei der Paketablage in einem Dateisystem gehören die **class**-Dateien mit den Klassen und Interfaces eines Pakets in einen gemeinsamen Ordner, dessen Name mit dem Paketnamen identisch ist.[1] In unserem Beispiel mit den Klassen `A`, `B` und `C` im Paket `demopack` muss also folgende Situation hergestellt werden:

- Die drei Bytecodedateien **A.class**, **B.class** und **C.class** befinden sich in einem Ordner namens **demopack**:



- Wo die Quellcodedateien abgelegt werden, ist nicht vorgeschrieben. In der Regel wird man (z. B. im Hinblick auf die Weitergabe eines Programms) die Quellcode- von den Bytecodedateien separieren. Unsere Entwicklungsumgebung IntelliJ verwendet per Voreinstellung im Ordner eines Projekts für die Quellcodedateien den Unterordner **src** und für die Bytecodedateien den Unterordner **out**.
- Der übergeordnete Ordner von **demopack** ist im Suchpfad für **class**-Dateien enthalten, damit die **class**-Dateien in **demopack** vom Compiler und von der JVM gefunden werden (siehe Abschnitt 6.1.2.1).

6.1.1.2 Standardpaket

Ohne **package**-Deklarationen am Beginn einer Quellcodedatei gehören die resultierenden Klassen und Schnittstellen zum unbenannten **Standardpaket** (engl. *default package* oder *unnamed package*). Diese Situation war bei unseren bisherigen Anwendungen meist gegeben und aufgrund der geringen Komplexität dieser Projekte auch angemessen. Eine wesentliche Einschränkung für Typen im Standardpaket besteht darin, dass sie (auch bei einer Dekoration mit dem Zugriffsmodifikator **public**) nur paketintern, d.h. nur für andere Typen im Standardpaket sichtbar sind.

Um vom Compiler und von der JVM gefunden zu werden, müssen die **class**-Dateien mit den Typen des Standardpakets über den Suchpfad für Bytecode-Dateien erreichbar sein (siehe Abschnitt 6.1.2.1). Bei passender CLASSPATH-Definition dürfen sich die Dateien also in verschiedenen Ordnern oder auch in Java-Archiven (in **jar**-Dateien, siehe Abschnitt 6.1.3) befinden. Wir haben z. B. im Kursverlauf die zum Standardpaket gehörige Klasse `Simput` in einem zentralen Ordner oder in einem Java-Archiv abgelegt und für verschiedene Projekte (d.h. die jeweiligen Typen im Standardpaket) nutzbar gemacht. Dazu wurde der Ordner oder das Java-Archiv mit der Datei **Simput.class** per CLASSPATH-Definition oder eine äquivalente Technik unserer Entwicklungsumgebung IntelliJ (vgl. Abschnitt 3.4.2) in den Suchpfad für **class**-Dateien aufgenommen.

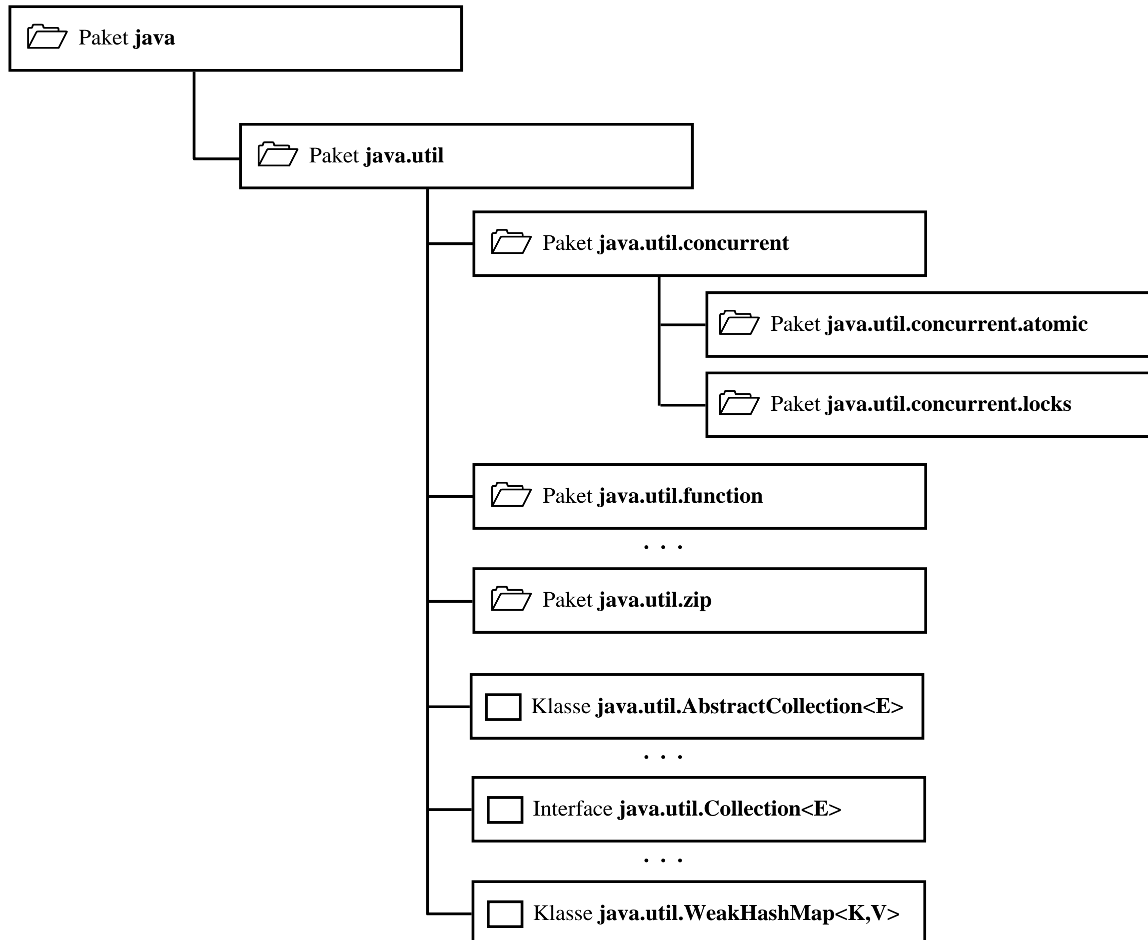
---

[1] Alternative Optionen zur Ablage von Paketen (z. B. in einer Datenbank) spielen keine große Rolle und werden in diesem Manuskript nicht behandelt (siehe Gosling et al. 2019, Abschnitt 7.2).

Während die Typen des Standardpakets trotz **public**-Modifikator in benannten Paketen nicht sichtbar sind, können umgekehrt die Typen im Standardpaket problemlos auf öffentliche Typen aus benannten Paketen zugreifen. Bei Startklassen, die nicht von anderen Typen genutzt werden sollen, werden wir weiterhin der Einfachheit halber das Standardpaket verwenden.

6.1.1.3 Unterpakete

Mit Ausnahme des Standardpakets kann ein Paket **Unterpakete** enthalten, was bei den Paketen im Java-API meist der Fall ist, z. B.:



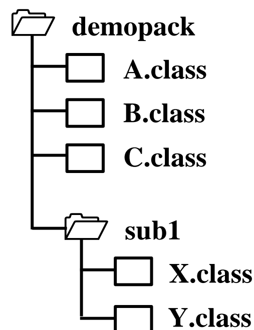
Auf jeder Stufe der Pakethierarchie sind sowohl Typen (Klassen, Interfaces) als auch Unterpakete erlaubt. So enthält z. B. das Paket **java.util** u.a.

- die Klassen **AbstractCollection<E>, Arrays**, **Random**, ...
- die Interfaces **Collection<E>**, **List<E>**, **Map<K,V>**, ...
- die Unterpakete **java.util.concurrent**, **java.util.function**, **java.util.zip**, ...

Soll eine Klasse einem Unterpaket zugeordnet werden, muss in der **package**-Deklaration am Anfang der Quellcodedatei der gesamte Paketpfad angegeben werden, wobei die Namensbestandteile jeweils durch einen Punkt getrennt werden. Es folgt der Quellcode der Klasse `X`, die zusammen mit der analog definierten Klasse `Y` in das Unterpaket `sub1` des `demopack`-Pakets eingeordnet wird:

```
package demopack.sub1;

public class X {
   private static int anzahl;
   private int objnr;

   public X() {
      objnr = ++anzahl;
   }

   public void prinr() {
      System.out.println("Klasse X, Objekt Nr. " + objnr);
   }
}
```

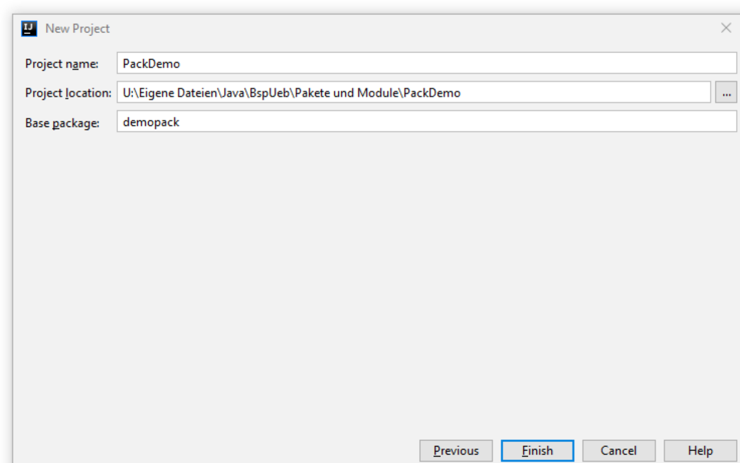
Bei der Paketablage in einem Dateisystem müssen die **class**-Dateien in einem zur Pakethierarchie analog aufgebauten Dateiverzeichnisbaum abgelegt werden, der in unserem Beispiel folgendermaßen auszusehen hat:



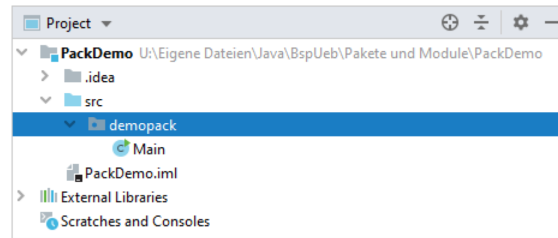
Typen eines Unterpakets gehören *nicht* zum übergeordneten Paket, was beim Importieren von Paketen (siehe Abschnitt 6.1.2.2) zu beachten ist. Außerdem haben gemeinsame Bestandteile im Paketnamen keine Relevanz für die wechselseitigen Zugriffsrechte (vgl. Abschnitt 6.3). Klassen im Paket `demopack.sub1` haben z. B. für Klassen im Paket `demopack` dieselben Rechte wie Klassen in beliebigen anderen Paketen.

6.1.1.4 Paketunterstützung in IntelliJ

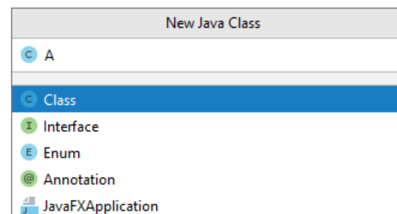
Nachdem schon einige Quellen aus dem Beispielpaket `demopack` zu sehen waren, erstellen wir es nun mit IntelliJ. Wir starten ein neues Java-Projekt mit dem Namen `PackDemo` und verwenden dabei erstmals ein **Base package**:



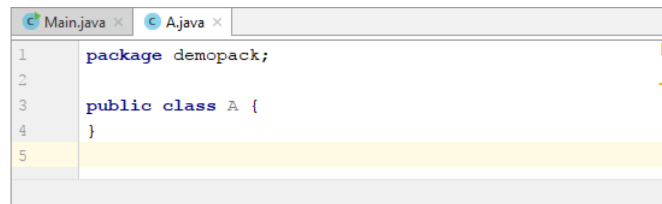
Im **Project**-Fenster zeigt sich die folgende Ausgangssituation mit dem Paket `demopack`, das sich im **src**-Ordner befindet und eine vom Assistenten angelegte Klasse namens `Main` enthält:



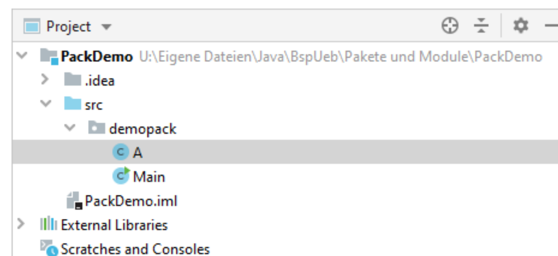
Nun legen wir im Paket `demopack` die Klasse `A` an, z. B. über den Befehl **New > Java Class** aus dem Kontextmenü zum Paket:



Nach dem **OK** startet IntelliJ im Editor eine Klassendefinition mit **package**-Deklaration



und zeigt im **Project**-Fenster den aktuellen Entwicklungsstand:



Wir vervollständigen den Quellcode der Klasse `A` (siehe Abschnitt 6.1.1.1) und legen analog auch die Klassen `B` und `C` im Paket `demopack` an:

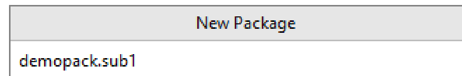
```java
package demopack;

public class B {
   private static int anzahl = 0;
   private int objnr;

   public B() {
      objnr = ++anzahl;
   }

   public void prinr() {
      System.out.println("Klasse B, Objekt Nr. "
                               + objnr);
   }
}
```

```java
package demopack;

public class C {
   private static int anzahl;
   private int objnr;

   public C() {
      objnr = ++anzahl;
   }

   public void prinr() {
      System.out.println("Klasse C, Objekt Nr. "
                               + objnr);
   }
}
```

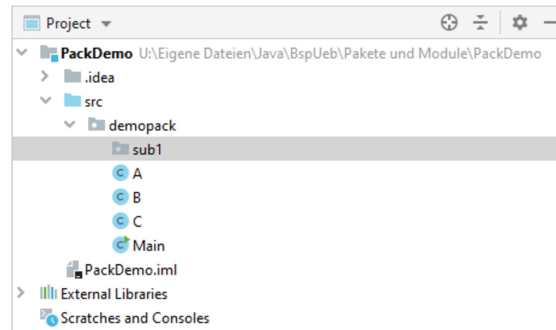
Um das Unterpaket `sub1` zu erstellen, wählen wir im **Project**-Fenster aus dem Kontextmenü zum Paket `demopack` den Befehl

New > Package

und geben in folgender Dialogbox den gewünschten Namen für das Unterpaket an:



Nach dem Quittieren per **Enter**-Taste erzeugt IntelliJ den Unterordner **demopack\sub1** im **src**-Ordner des Projekts:



Im Unterpaket `demopack.sub1` legen wir nun (z. B. über den Kontextmenübefehl **New > Java Class**) die Klasse `X` an, deren Quellcode schon im Abschnitt 6.1.1.3 zu sehen war, und danach die analog aufgebaute Klasse `Y`.

Schließlich komplettieren wir noch die vom Assistenten angelegte Startklasse `Main` und stören uns nicht am überflüssigen Klassen-Modifikator **public**:

```
package demopack;

import demopack.sub1.*;

public class Main {
    public static void main(String[] args) {
        A a1 = new A(), a2 = new A();
        a1.prinr(); a2.prinr();
        B b = new B(); b.prinr();
        C c = new C(); c.prinr();
        X x = new X(); x.prinr();
        Y y = new Y(); y.prinr();
    }
}
```

Das Programm liefert die folgende Ausgabe:

```
Klasse A, Objekt Nr. 1
Klasse A, Objekt Nr. 2
Klasse B, Objekt Nr. 1
Klasse C, Objekt Nr. 1
Klasse X, Objekt Nr. 1
Klasse Y, Objekt Nr. 1
```

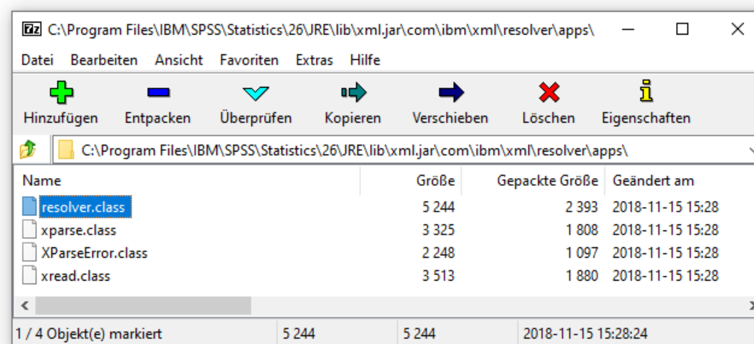
6.1.1.5 Konventionen für weltweit eindeutige Paketnamen

Bei Verwendung einfacher Paketnamen (wie im Beispiel `demopack`) kann es passieren, dass sich zwei Entwickler(teams) für denselben Namen entscheiden. Dies wird zum Problem, wenn irgendwann die beiden gleichnamigen Pakete in *einem* Programm verwendet werden sollen. Einfache Paketnamen sind in einem begrenzten Umfang (z. B. firmenintern) akzeptabel. Ist ein Programm bzw. eine Bibliothek aber für die Öffentlichkeit gedacht, sollte durch Beachtung der folgenden Regeln für weltweit eindeutige Paketnamen gesorgt werden (siehe Gosling et al 2019, Abschnitt 6.1):

- Unter der Voraussetzung, dass eine eigene Internet-Domäne existiert, werden die Bestandteile des Domänennamens in umgekehrter Reihenfolge als führende Bestandteile der Pakethierarchie verwendet. Den restlichen Paketpfad legt eine Firma bzw. Institution nach eigenem Ermessen fest, um Namenskonflikte innerhalb der Domäne zu vermeiden. Die Firma **IBM** mit der Internet-Domäne **ibm.com** kann z. B. den folgenden Paketnamen verwenden:

  com.ibm.xml.resolver.apps

  Unter Beachtung dieser Regel hat die Firma **IBM** zusammen mit dem Programm **SPSS Statistics** (Version 26) in der Java-Archivdatei **xml.jar** (siehe Abschnitt 6.1.3) ein Paket mit dem als Beispiel verwendeten Namen ausgeliefert:



- Bestandteile im Domänenamen, die zu einem ungültigen Java-Bezeichner führen würden, müssen ersetzt werden, z. B. der Bindestrich durch einen Unterstrich (Gosling et al 2019, Abschnitt 6.1). Im ZIMK an der Universität Trier kann z. B. für den klientenseitigen Teil einer Kommunikations-Software der folgende Paketpfad verwendet werden:

  `de.uni_trier.zimk.chat.client`

- Eine Ausnahme von den Konventionen für die Bildung von weltweit eindeutigen Paketnamen bestehen bei den Paketen der Standardbibliothek, die mit **java** oder **javax** beginnen.

6.1.2 Pakete verwenden

Damit man die in fremden Paketen vorhandenen Typen in einem eigenen Programm verwenden kann, …

- müssen die **class**-Dateien für die beteiligten Werkzeuge (Compiler, JVM) auffindbar sein
- und die Klassen bzw. Schnittstellen im eigenen Quellcode korrekt angesprochen werden.

Mit den beiden Themen wurden wir schon konfrontiert, doch sollen die zugehörigen Lösungen gleich noch einmal zusammengestellt werden.

Mit einer weiteren Nutzungsvoraussetzung, den Zugriffsrechten, werden wir uns im Abschnitt 6.3 beschäftigen.

6.1.2.1 Verfügbarkeit der class-Dateien (Klassenpfad)

Damit die in einem Paket vorhandenen Typen beim Übersetzen bzw. bei der Ausführung eines Programms genutzt werden können, muss dem Compiler bzw. der JVM der Aufenthaltsort der Typen bekannt gemacht werden. Die Typen in den API-Paketen werden auf jeden Fall gefunden. Welche Maßnahmen bei anderen Paketen erforderlich sind, wird anschließend beschrieben (vgl. auch Abschnitt 2.2.4). Wir ignorieren vorläufig Pakete in Java-Archiven (siehe Abschnitt 6.1.3) und beschränken uns passend zum Entwicklungsstand des `demopack`-Beispiels auf Pakete in Verzeichnissen. Wir behandeln das Übersetzen und die Ausführung eines paketierten Java-Programms auf einem Windows-Rechner mit installiertem OpenJDK, d.h.:

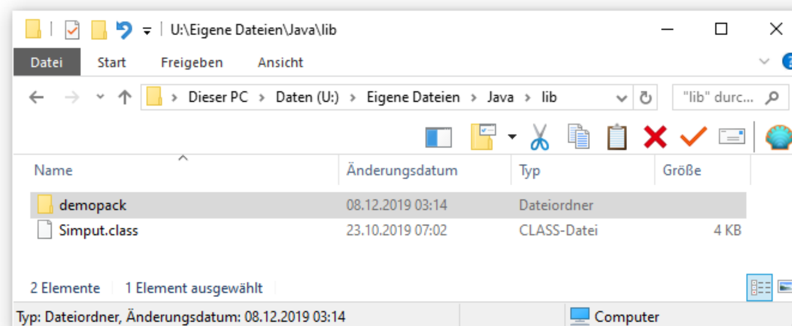
- Der Compiler **javac.exe** und der Starter **java.exe** sind vorhanden
- und befinden sich in einem Ordner, der von Windows dank PATH-Definition nach ausführbaren Programmen durchsucht wird.

Der Stammordner des Pakets (im Beispiel: der Ordner, der das Verzeichnis `demopack` enthält) muss über die **CLASSPATH**-Umgebungsvariable oder per **classpath-**Befehlszeilenoption bekanntgegeben werden:

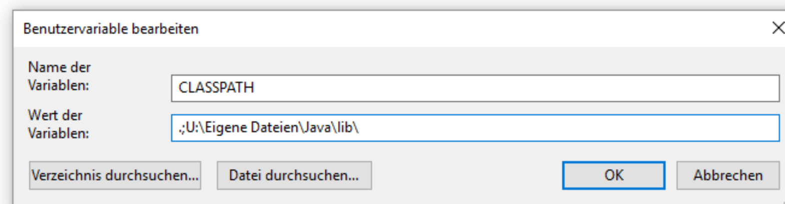
- Stammordner des Pakets in der **CLASSPATH-Umgebungsvariablen** eingetragen
  Über die Betriebssystem-Umgebungsvariable CLASSPATH lässt sich der Suchpfad für **class**-Dateien definieren. Wenn sich z. B. der Ordner zur oberste Paketebene (in unserem Beispiel: **demopack**) im Ordner

  U:\Eigene Dateien\Java\lib

  befindet,

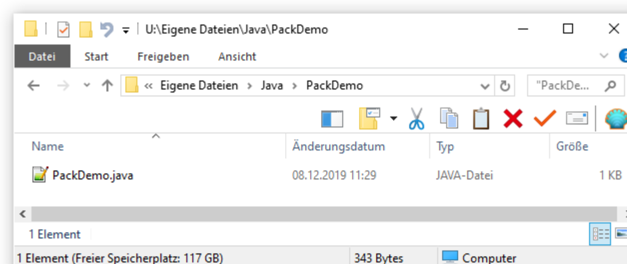


  kann die CLASSPATH-Definition lauten (vgl. Abschnitt 2.2.4):



  Unter Windows werden die Einträge in der CLASSPATH-Definition durch ein Semikolon getrennt. Durch einen Punkt als Listenelement wird der aktuelle Ordner des laufenden Betriebssystem-Prozesses in den Klassenpfad aufgenommen (siehe Beispiel).

  Der OpenJDK-Compiler und die JVM finden z. B. den Bytecode der Klasse `demopack.A`, indem sie jedes Verzeichnis in der CLASSPATH-Definition nach der Datei mit dem relativen Pfad **demopack\A.class** durchsuchen. Befindet sich z. B. im aktuellen Ordner



  eines Konsolenfensters die Datei **PackDemo.java** mit dem folgenden Inhalt,

```
import demopack.*;
import demopack.sub1.*;
```

```
class PackDemo {
    public static void main(String[] args) {
        A a1 = new A(), a2 = new A();
        a1.prinr(); a2.prinr();
        B b = new B(); b.prinr();
        C c = new C(); c.prinr();
        X x = new X(); x.prinr();
        Y y = new Y(); y.prinr();
    }
}
```

dann gelingt aufgrund der obigen CLASSPATH-Definition die Übersetzung mit dem folgenden Kommando:

```
>javac PackDemo.java
```

Der Compiler findet die benötigten **class**-Dateien mit den relativen Pfaden:

- **demopack\A.class**, **demopack\B.class**, **demopack\B.class**
- **demopack\sub1\X.class**, **demopack\sub1\Y.class**

Zum Starten taugt in derselben Situation der folgende Aufruf des Java-Starters:

```
>java PackDemo
```

Es ist zu beachten, dass die eben vorgestellte Startklasse `PackDemo` zum Standardpaket gehört, weil sich am Anfang ihrer Quellcodedatei keine **package**-Deklaration befindet.

Der Java-Starter findet die Datei **PackDemo.class** im aktuellen Verzeichnis, weil die oben beschriebene CLASSPATH-Definition einen Punkt als Vertreter für das aktuelle Verzeichnis enthält. Ohne explizite CLASSPATH-Definition ist ein voreingestellter Klassenpfad wirksam, der ebenfalls das aktuelle Verzeichnis enthält. Zum Problem kann eine unerwartete, z. B. im Rahmen einer Programminstallation automatisch erstellte CLASSPATH-Definition werden, wenn dort der Punkt als Vertreter für das aktuelle Verzeichnis fehlt.

- Stammordner des Pakets in der **classpath-Befehlszeilenoption**
  Beim Aufruf der OpenJDK-Werkzeuge **javac.exe**, **java.exe** und **javaw.exe** lässt sich die CLASSPATH-Umgebungsvariable durch die **-classpath** - Befehlszeilenoption (abzukürzen mit **-cp**) dominieren, z. B.:

```
>javac -cp ".;U:\Eigene Dateien\Java\lib" PackDemo.java
>java -cp ".;U:\Eigene Dateien\Java\lib" PackDemo
```

Im bisherigen Kursverlauf haben wir per CLASSPATH-Umgebungsvariable auch den Ordner mit der Standardpaketklasse **Simput.class** bekanntgegeben (vgl. Abschnitt 2.2.4). Eine Bibliotheksklasse im unbenannten Standardpaket bereitzuhalten, war der Einfachheit halber zu Kursbeginn eine akzeptable Technik. Generell ist jedoch für Bibliothekstypen die Einordnung in ein benanntes Paket strikt zu bevorzugen.

In IntelliJ spielt die CLASSPATH-Umgebungsvariable keine Rolle. Mit den (globalen) Bibliotheken ist eine flexible Lösung vorhanden, die wir im Abschnitt 3.4.2 kennengelernt haben.

6.1.2.2 Typen aus fremden Paketen ansprechen

In einer Quellcodedatei können Typen aus dem eigenen Paket sowie Typen aus dem (automatisch importierten) Paket **java.lang** ohne Paketpräfix angesprochen werden. Auf folgende Weise lassen sich (entsprechende Rechte vorausgesetzt) Typen aus einem anderen fremden Paket ansprechen:

- **Verwendung des vollqualifizierten Namens**
  Dem Klassennamen ist der durch Punkt abgetrennte Paketname voranzustellen. Bei einem hierarchischen Paketaufbau ist der gesamte Pfad anzugeben, wobei die Unterpaketnamen wiederum durch Punkte zu trennen sind. Wir haben bereits mehrfach die Klasse **Random** aus dem Paket **java.util** auf diese Weise angesprochen, z. B.:

```
java.util.Random zzg = new java.util.Random();
```

Bei einem mehrfach benötigten Typ wird es schnell lästig, den vollqualifizierten Namen schreiben zu müssen. Außerdem erschweren zahlreich auftretende Paketnamen die Lesbarkeit des Quellcodes.

- **Import eines einzelnen Typs**
  Um die lästige Angabe von Paketnamen zu vermeiden, kann man eine Klasse oder Schnittstelle in eine Quellcodedatei *importieren*. Anschließend ist der Typ durch seinen einfachen Namen (*ohne* Paket-Präfix) anzusprechen. Die zuständige **import**-Deklaration ist an den Anfang einer Quellcodedatei zu setzen, ggf. aber *hinter* eine **package**-Deklaration (vgl. Abschnitt 6.1.1.1). Im folgenden Programm wird die Klasse **Random** aus dem API-Paket **java.util** importiert und verwendet:

```
import java.util.Random;
class Prog {
    public static void main(String[] args) {
        Random zzg = new Random();
        System.out.println(zzg.nextInt(101));
    }
}
```

- **Import eines kompletten Pakets**
  Um z. B. *alle* Typen aus dem Paket **java.util** zu importieren, setzt man den Joker-Stern ein:

```
import java.util.*;
```

  Es ist zu beachten, dass *Unterpakete* dabei *nicht* einbezogen werden. Für sie ist bei Bedarf eine separate **import**-Deklaration fällig.
  Weil durch die Verwendung des Jokerzeichens *keine* Rechenzeit- oder Speicherressourcen verschwendet werden, ist dieses bequeme Vorgehen im Allgemeinen sinnvoll, wenn aus einem Paket *mehrere* Typen benötigt werden. Eventuelle Namenskollisionen (durch identische Typnamen in verschiedenen Paketen) müssen durch die Verwendung des vollqualifizierten Namens aufgehoben werden.
  Das API-Paket **java.lang** mit wichtigen Klassen wie **System**, **String**, **Math** wird automatisch importiert.

- **Import von statischen Methoden und Feldern**
  Seit Java 5 besteht die Möglichkeit, statische Methoden und Variablen fremder Typen so zu importieren, dass bei der Ansprache weder der Paket- noch der Typname erforderlich ist. Bisher haben wir die statischen Mitglieder der Klasse **Math** aus dem Paket **java.lang** wie im folgenden Beispielprogramm genutzt:

```
class Prog {
    public static void main(String[] args) {
        System.out.println("Sin(Pi/2) = " + Math.sin(Math.PI/2));
    }
}
```

  Seit Java 5 lassen sich die statischen Mitglieder einer Klasse einzeln

```
import static java.lang.Math.sin;
```

  oder insgesamt importieren, z. B.:

```
import static java.lang.Math.*;
class Prog {
    public static void main(String[] args) {
        System.out.println("Sin(Pi/2) = " + sin(PI/2));
    }
}
```

  In der importierenden Quellcodedatei wird im Vergleich zum normalen Paketimport nicht nur der Paket- sondern auch der Klassenname eingespart.

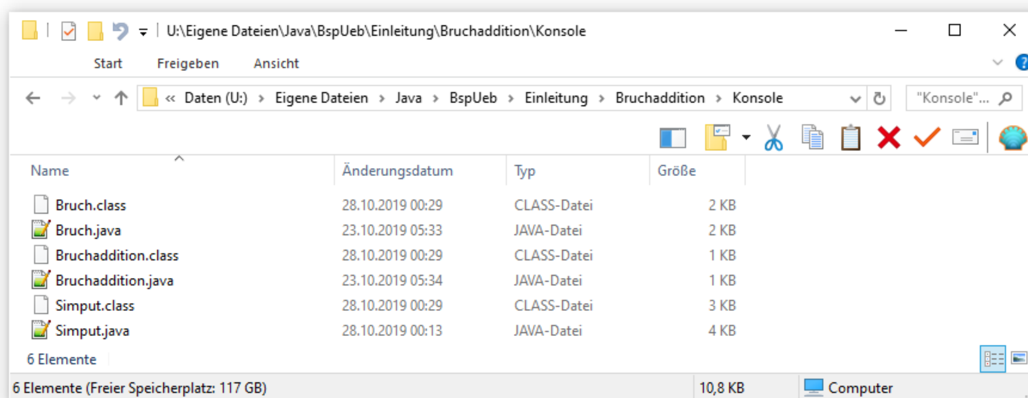
Die Typen im unbenannten Standardpaket sind in anderen Paketen generell (auch bei Verwendung des Typmodifikators **public**) *nicht* verfügbar. Diese Einschränkung haben wir bisher der Einfachheit halber auch bei Klassen in Kauf genommen, die in mehreren Programmen verwendet wurden (z. B. `Bruch` und `Simput`). Während bei Projekten mit Bibliotheks-Charakter eine Paketierung unbedingt zu empfehlen ist, kann sie bei kleineren Projekten, die keine andernorts benötigten Typen enthalten, unterbleiben.

6.1.2.3 Startklassen im Paket

Wir haben uns im bisherigen Kursverlauf nur wenig damit beschäftigt, wie ein Java-Programm außerhalb unserer Entwicklungsumgebung IntelliJ gestartet werden kann (z. B. auf einem Kundenrechner). Bei den wenigen Anwendungsbeispielen lag eine recht simple Situation vor:

- Die zu startende Hauptklasse befand sich im Standardpaket.
- Die **class**-Datei der Startklasse befand sich im aktuellen Verzeichnis des Konsolenfensters, in dem der Java-Starter aufgerufen wurde. In diesem Verzeichnis befanden sich auch:
    - die **class**-Dateien von weiteren benötigten Klassen
    - die Ordner von Paketen mit weiteren benötigten Klassen
- Es war keine explizite CLASSPATH-Definition im Spiel, sodass sich das aktuelle Verzeichnis der Konsole im impliziten Klassensuchpfad befand.

Diese Situation lag z. B. im Einleitungsbeispiel des Kurses vor (siehe Abschnitt 1.2.2):



Der Programmstart gelingt in dieser Situation mit einem simplen Kommando, z. B.:

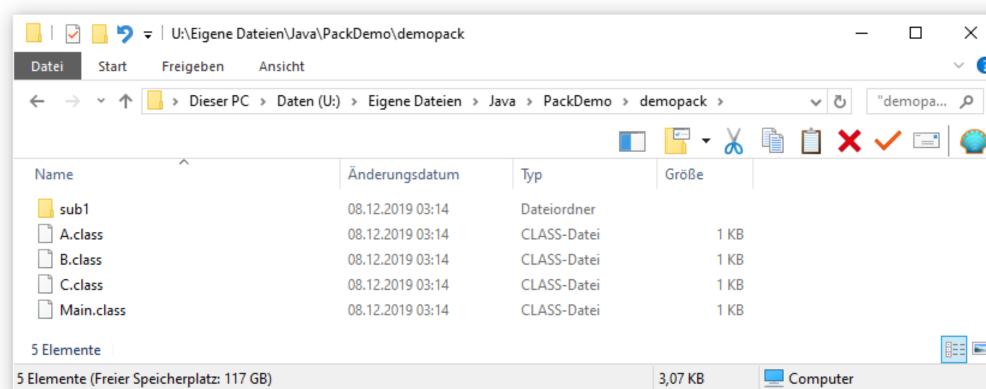
```
Eingabeaufforderung

U:\Eigene Dateien\Java\BspUeb\Einleitung\Bruchaddition\Konsole>echo %classpath%
%classpath%

U:\Eigene Dateien\Java\BspUeb\Einleitung\Bruchaddition\Konsole>java Bruchaddition
1. Bruch
Zaehler: 1
Nenner : 2
   1
 -----
   2

2. Bruch
Zaehler: 4
Nenner : 9
   4
 -----
   9

Summe
   17
 -----
   18

U:\Eigene Dateien\Java\BspUeb\Einleitung\Bruchaddition\Konsole>
```

Wird für ein Programm mit mehreren Klassen ein eigenes Paket verwendet, sollte in der Regel auch die Startklasse dort untergebracht werden, damit sie z. B. auch alle Typen im Paket mit der voreingestellten Schutzstufe (**package**) sehen kann. Bei der Startklasse des `PackDemo`-Beispiels (Variante von Abschnitt 6.1.1.4) spielt die Schutzstufe **package** keine Rolle, weil die Klassen in den Paketen

`demopack` und `demopack.pub1` sowie deren relevante Member alle die Zugriffsstufe **public** besitzen. Die Startklasse befindet sich aber (aufgrund einer Entscheidung des IntelliJ-Assistenten für neue Projekte) im Anwendungspaket:

```
package demopack;
import demopack.sub1.*;
public class Main {
    public static void main(String[] args) {
        A a1 = new A(), a2 = new A();
        a1.prinr(); a2.prinr();
        B b = new B(); b.prinr();
        C c = new C(); c.prinr();
        X x = new X(); x.prinr();
        Y y = new Y(); y.prinr();
    }
}
```

Wenn ein Konsolenfenster auf den folgenden Ordner **demopack** (mit der Datei **Main.class**)



positioniert ist, und keine explizite CLASSPATH-Definition besteht, scheitert ein Startversuch nach dem oben erfolgreich angewendeten Muster

```
>java Main
```

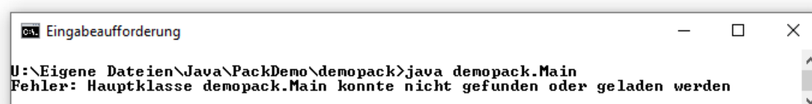
mit der Fehlermeldung:

Wir erinnern uns daran, dass im **java**-Aufruf keine *Datei* anzugeben ist, sondern eine *Klasse*. Im aktuellen Beispiel hat die gewünschte Klasse den Namen `demopack.Main`, sodass der Java-Starter zu Recht reklamiert, es sei keine Hauptklasse mit dem Namen `Main` zu finden.

Den vollständigen Namen der Hauptklasse anzugeben,

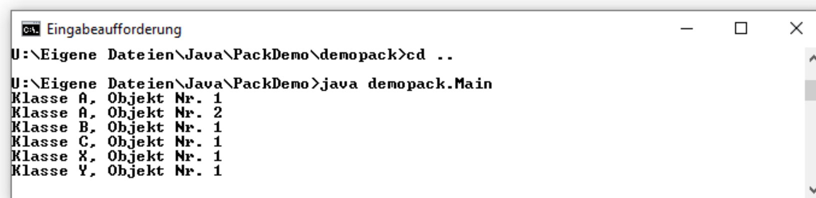
```
>java demopack.Main
```

hilft aber nicht, solange das Konsolenfenster auf den Ordner **demopack** positioniert ist:



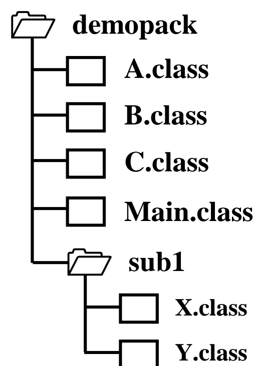
Nun sucht der Java-Starter nämlich ausgehend vom aktuellen Ordner vergeblich nach dem Paketordner **demopack**.

Damit diese Suche gelingt, bewegen wir uns mit dem Konsolenfenster in der Ordnerhierarchie um eine Stufe nach oben und haben schließlich mit dem zuletzt verwendeten Startkommando Erfolg:



Mit den Informationen aus diesem Abschnitt sollte nun klar sein, wie ein Java-Programm in Form von Bytecode-Dateien ausgeliefert und auf einem Kundenrechner installiert werden kann. Im `PackDemo`-Beispiel (mit der Startklasse im Paket `demopack`) muss man ...

- im Installationsordner einen Unterordner namens **demopack** anlegen und die Dateien **A.class**, **B.class**, **C.class** sowie **Main.class** dorthin kopieren,



- zu **demopack** einen Unterordner namens **sub1** anlegen und die Dateien **X.class** sowie **Y.class** dorthin kopieren.

Wenn ...

- ein Konsolenfenster auf den Installationsordner positioniert ist,
- und der Klassensuchpfad den aktuellen Ordner enthält (= Voreinstellung ohne explizite CLASSPATH-Definition),

dann kann die Hauptklasse über ihren vollständigen Namen gestartet werden:

```
>java demopack.Main
```

Soll dieser Start aus einem Konsolenfenster mit einem *beliebigen* aktuellen Verzeichnis möglich sein, dann muss ...

- entweder der Installationsordner in die CLASSPATH-Definition aufgenommen werden
- oder im Startkommando per **-cp** - Argument eine äquivalente Definition des Klassensuchpfads vorgenommen werden, z. B.:

```
>java -cp "U:\Eigene Dateien\Java\PackDemo" demopack.Main
```

Die im aktuellen Abschnitt beschriebenen, ziemlich komplexen und fehleranfälligen Regeln für den Programmstart entfallen, wenn ein Programm als ausführbare **jar**-Datei ausgeliefert wird. Mit dieser Distributionstechnik im Speziellen und mit **jar**-Dateien in Allgemeinen beschäftigt sich der nächste Abschnitt.

6.1.3 Traditionelle jar-Dateien (Java 8)

Wenn zu einem Programm oder zu einer Bibliothek zahlreiche **class**-Dateien (eventuell aufgeteilt in mehrere Pakete) und zusätzliche Hilfsdateien (z. B. mit übersetzten Texten oder Multimedia-

Inhalten) gehören, dann bietet sich die Zusammenfassung zu *einer* Java-Archivdatei (Namenserweiterung **.jar**) an.

In diesem Abschnitt werden *traditionelle* **jar**-Dateien behandelt, die in allen Java-Versionen eingesetzt werden können. Die die Java 9 eingeführten *modularen* **jar**-Dateien werden im Abschnitt 6.2.6 vorgestellt.

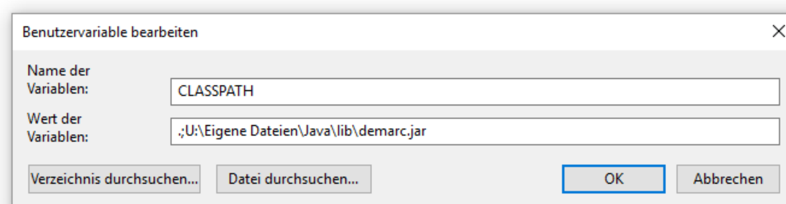
6.1.3.1 Eigenschaften von Archivdateien

Java-Archivdateien bieten viele Vorteile, z. B.:

- **Übersichtlichkeit, Bequemlichkeit**
  Im Vergleich zu zahlreichen Einzeldateien ist ein Archiv für den Anwender deutlich bequemer. Ein per Archiv ausgeliefertes Programm kann sogar direkt über die Archivdatei gestartet werden, bei entsprechender Konfiguration des Betriebssystems auch per Maus(doppel)klick.
- **Sicherheit**
  Bei *signierten* **jar**-Dateien kann sich der Anwender Gewissheit über den Urheber verschaffen und der Software entsprechende Rechte einräumen.[1]
- **Versionsangaben für Pakete**
  In einem Archiv kann man Hersteller- und Versionsangaben zu den enthaltenen Paketen unterbringen.[2]

Mit den beiden zuletzt genannten Vorteilen können wir uns in diesem Manuskript aus Zeitgründen nicht beschäftigen.

Eine traditionelle Archivdatei kann beliebig viele Pakete enthalten. Damit die dortigen **class**-Dateien vom Compiler und von der JVM gefunden werden, muss die Archivdatei analog zu einem Dateiordner mit Paketen in den **Suchpfad** für **class**-Dateien aufgenommen werden (vgl. Abschnitte 3.4.2 und 6.1.2.1), z. B. über die Umgebungsvariable CLASSPATH:



Bei den bis Java 8 üblichen **jar**-Dateien mit API - Paketen ist dies allerdings *nicht* erforderlich.

Der Klassenpfad kann auch ab Java 9 zur Lokalisation von traditionellen Archivdateien verwendet werden. Hier stehen mit dem Modulpfad und den modularen Archivdateien modernere Alternativen zur Verfügung, die allerdings bisher (Dezember 2019) noch relativ selten eingesetzt werden (siehe Abschnitt 6.2).

Weil Java-Archive das ZIP-Dateiformat besitzen, können sie von diversen (De-)Komprimierungsprogrammen geöffnet werden. Das *Erzeugen* von Java-Archiven sollte man aber dem speziell für diesen Zweck entworfenen JDK-Werkzeug **jar.exe** (siehe Abschnitte 6.1.3.2 und 6.1.3.4) oder einer entsprechend ausgestatteten Entwicklungsumgebung überlassen.

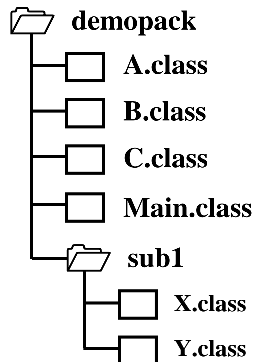
---

[1] https://docs.oracle.com/javase/tutorial/deployment/jar/signindex.html

[2] https://docs.oracle.com/javase/tutorial/deployment/jar/packageman.html

6.1.3.2 Archivdateien mit dem JDK-Werkzeug jar erstellen

Zum Erstellen und Verändern von Java-Archivdateien kann das JDK-Werkzeug **jar.exe** verwendet werden. Wir nutzen es, um eine Archivdatei mit den **class**-Dateien im Paket `demopack` und im Unterpaket `demopack.sub1`



zu erzeugen.

Wir öffnen ein Konsolenfenster und wechseln zum Verzeichnis, das den Ordner **demopack** enthält. Dann lassen wir mit folgendem **jar**-Aufruf das Archiv **`demarc.jar`** mit der gesamten Paket-Hierarchie erstellen:[1]

```
>jar cf0 demarc.jar demopack
```

Darin bedeuten:[2]

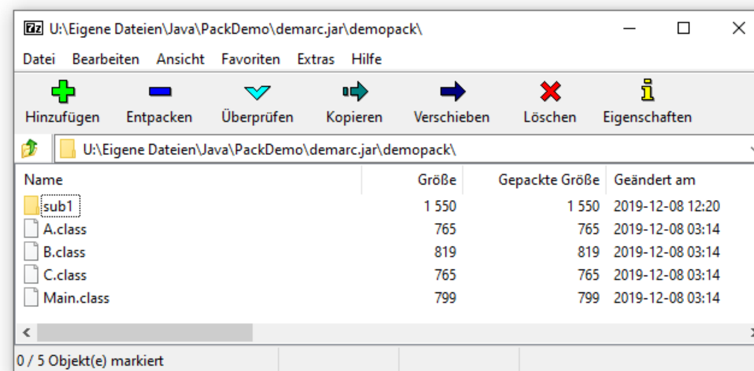
- 1. Parameter: Optionen
  Die Optionen werden durch einzelne Zeichen angefordert, die unmittelbar hintereinander stehen müssen:
  - **c**
    Mit einem **c** (für *create*) wird das Erstellen eines Archivs angefordert.
  - **f**
    Mit **f** (für *file*) wird ein Name für die Archivdatei angekündigt, der als weiteres Kommandozeilenargument auf die Optionen zu folgen hat.
  - **0**
    Mit der Ziffer **0** wird auf die ZIP-Kompression verzichtet.
- 2. Parameter: Archivdatei
  Der Archivdateiname muss einschließlich Extension (üblicherweise **.jar**) geschrieben werden.
- 3. Parameter: Zu archivierende Dateien und Ordner
  Bei einem Ordner wird rekursiv der gesamte Verzeichnisast einbezogen. Ein Ordner kann die **class**-Dateien eines Pakets oder auch sonstige Dateien (z. B. mit Medien) enthalten. Soll eine Archivdatei mehrere Pakete bzw. Ordner aufnehmen, sind die Ordnernamen durch Leerzeichen getrennt anzugeben.

Aus obigem **jar**-Aufruf resultiert die folgende **jar**-Datei (hier angezeigt vom kostenlosen Hilfsprogramm **7-Zip**):

---

[1] Sollte der Aufruf nicht klappen, befindet sich vermutlich das OpenJDK-Unterverzeichnis **bin** (z. B. **C:\Program Files\ojdkbuild\java-1.8.0-openjdk-1.8.0.232-1\bin**) nicht im Suchpfad für ausführbare Programme. In diesem Fall muss das Programm mit kompletter Pfadangabe gestartet werden.

[2] Hier ist die **jar**-Dokumentation der Firma Oracle für Java 8 zu finden:
https://docs.oracle.com/javase/8/docs/technotes/guides/jar/index.html



Es ist erlaubt, dass sich die zu einem Paket gehörigen **class**-Dateien in verschiedenen **jar**-Dateien befinden. Möglicherweise enthalten zwei **jar**-Dateien zufälligerweise ein namensgleiches Paket. Wenn dann Typnamen in den beiden Paketen übereinstimmen, hängt es von der Reihenfolge der **jar**-Dateien in der CLASSPATH-Definition ab, aus welcher Datei ein Typ geladen wird. Man spricht in diesem Zusammenhang von der *JAR-Hölle*. Im JPMS (bei modularen **jar**-Dateien) ist ein solches *Package Splitting* verboten (siehe Abschnitt 6.2.4).

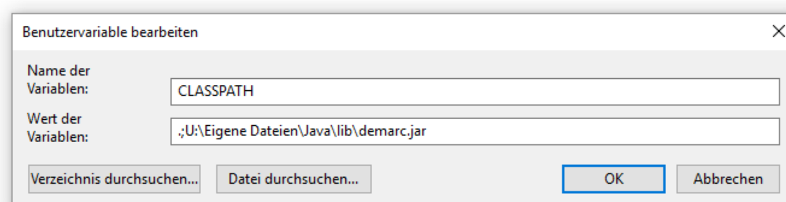
Die Quellcodedateien sind für die *Verwendung* eines Archivs (als Programm oder Klassenbibliothek) *nicht* erforderlich und sollten daher (z. B. aus urheberrechtlichen Gründen) durch die Ablage in einer separaten Ordnerstruktur aus dem Archiv herausgehalten werden.

6.1.3.3 Archivdateien verwenden

Um ein Archiv mit seinen Paketen als Klassenbibliothek in verschiedenen Projekten nutzen zu können, kann es in den Suchpfad des Compilers bzw. Interpreters für **class**-Dateien aufgenommen werden. Befindet z. B. die eben erstellte Archivdatei **demarc.jar** im Ordner **U:\Eigene Dateien\Java\lib**, dann kann die Klasse `demopack.Main` in einem Konsolenfenster mit beliebigem aktuellem Ordner folgendermaßen gestartet werden:

```
>java -cp ".;U:\Eigene Dateien\Java\lib\demarc.jar" demopack.Main
```

Analog zur **-cp** - Option in den Werkzeugaufrufen kann eine Archivdatei in die CLASSPATH-Umgebungsvariable des Betriebssystems aufgenommen werden, z. B.:



Danach lässt sich das obige Startkommando vereinfachen:

```
>java demopack.Main
```

Für die Nutzung von Archivdateien in IntelliJ eignen sich (globale) Bibliotheken (siehe Abschnitt 3.4.2).

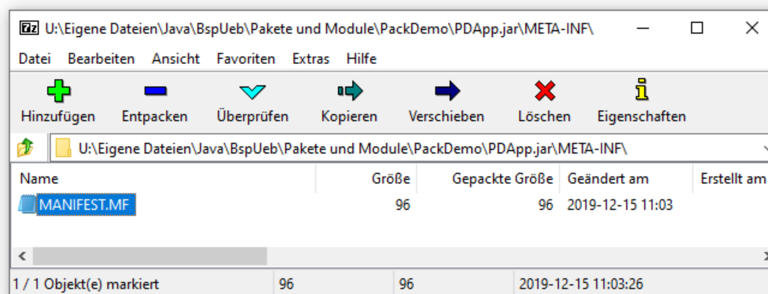
Wie sich eine traditionelle **jar**-Datei als sogenanntes *automatisches Modul* im Modulsystem von Java 9 verwenden lässt, ist im Abschnitt 6.2.9.1 zu erfahren.

6.1.3.4 Ausführbare jar-Dateien

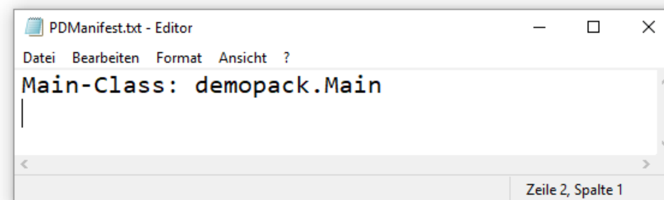
Um eine als Anwendung ausführbare, traditionelle **jar**-Datei zu erstellen, nimmt man die gewünschte Startklasse in das Archiv auf. Diese Klasse muss bekanntlich eine Methode **main()** mit folgendem Definitionskopf besitzen:

```
public static void main(String[] args)
```

Außerdem muss die Klasse im sogenannten **Manifest** des Archivs, dem wir bisher keine Beachtung geschenkt haben, als **Main-Class** eingetragen werden. Das Manifest befindet sich in der Datei **MANIFEST.MF**, die das **jar**-Werkzeug im Archiv-Ordner **META-INF** anlegt, z. B.:



Im **jar**-Aufruf kann man eine Textdatei mit Manifestinformationen übergeben. Um z. B. im `PackDemo`-Projekt (Variante von Abschnitt 6.1.1.4) die Startklasse `Main` im Paket `demopack` auszuzeichnen, legt man eine Textdatei an, die folgende Zeile und eine anschließende Leerzeile (!) enthält:

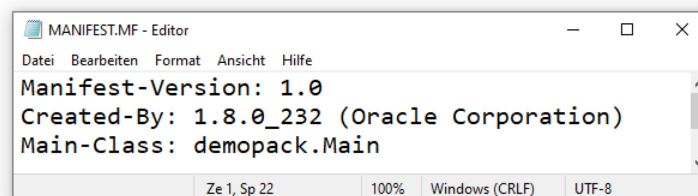


Im **jar**-Aufruf zum Erstellen des Archivs wird über die Option **m** eine Datei mit Manifestinformationen angekündigt, z. B. mit dem Namen **PDManifest.txt**:

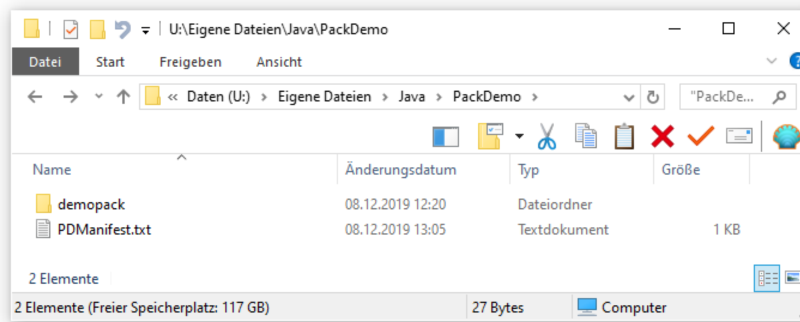
```
>jar cmf0 PDManifest.txt PDApp.jar demopack
```

Beachten Sie bitte, dass die Namen der Manifest- und der Archivdatei in derselben Reihenfolge wie die zugehörigen Optionen auftauchen müssen.

Es resultiert eine **jar**-Datei mit dem folgenden Manifest:



Der obige **jar**-Aufruf klappt, wenn sich die Datei **PDManifest.txt** mit den Manifestinformationen und das Paketverzeichnis **demopack** im aktuellen Ordner befinden, z. B.:



Auf eine Manifestinformationsdatei, die lediglich den Namen der Startklasse verrät, kann man seit Java 6 verzichten und stattdessen im **jar**-Aufruf die Option **e** (für *entry point*) verwenden, z. B.:

```
>jar cef0 demopack.Main PDApp.jar demopack
```

Unter Verwendung der Archivdatei **PDApp.jar** lässt sich das Programm mit der Hauptklasse `demopack.Main` mit dem folgenden Kommando

```
>java -jar PDApp.jar
```

starten. Damit dies auf einem Kundenrechner nach dem Kopieren der Datei **PDApp.jar** sofort möglich ist, muss dort lediglich eine JVM mit geeigneter Version installiert sein. Somit kann die Software-Verteilung und -Nutzung durch eine ausführbare **jar**-Datei erheblich vereinfacht werden:

- Es ist nur eine einzige Datei im Spiel.
- Beim Programmstart ist kein Klassensuchpfad relevant.

Wird ein Java-Programm per **jar**-Datei gestartet, dann legt allein das Manifest den **class**-Suchpfad fest. Weder die Umgebungsvariable CLASSPATH, noch das Kommandozeilenargument **-classpath** sind wirksam. Die Klassen im Java-API werden aber auf jeden Fall gefunden. Über das **jar**-Werkzeug lässt sich der **class**-Suchpfad einer **jar**-Datei so konfigurieren, dass Klassen in anderen Archivdateien gefunden werden.[1] Dabei entsteht in der Manifestdatei ein **Class-Path** - Eintrag.

6.1.3.5 Ausführbare jar-Datei für ein Projekt mit OpenJFX 8

Wenn mit dem Betriebssystem die Behandlung von **jar**-Dateien passend vereinbart wird, klappt bei Java-Programmen mit grafischer Bedienoberfläche sogar der Start per Mausdoppelklick auf die Archivdatei. Unter Windows sind dazu folgende Registry-Einträge geeignet:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.jar]
@="jarfile"
"Content Type"="application/jar"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\jarfile]

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\jarfile\shell]

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\jarfile\shell\open]

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\jarfile\shell\open\command]
@="\"C:\\Program Files\\ojdkbuild\\java-1.8.0-openjdk-1.8.0.232-1\\bin\\javaw.exe\" -jar \"%1\" %*"
```

Zum Starten dient hier das nur für GUI-Anwendungen geeignete OpenJDK-Werkzeug **javaw.exe**, das kein Konsolenfenster anzeigt, sodass der Doppelklick auf die im Abschnitt 6.1.3.4 beschriebenen Datei **PDApp.jar** ohne sichtbare Folgen bleibt.

Wir ergänzen das im Abschnitt 4.9 erstellte Bruchkürzungsprogramm mit JavaFX-GUI um eine ausführbare **jar**-Datei, damit das Programm über eine einzelne Datei verbreitet und auf jedem

---

[1] Siehe: http://docs.oracle.com/javase/tutorial/deployment/jar/downman.html

Rechner mit geeigneter JVM-Installation bequem per Doppelklick gestartet werden kann. Leider klappt das gegenwärtig (Dezember 2019) nur mit der im Abschnitt 1.2.1 beschriebenen OpenJDK 8 - Installation.

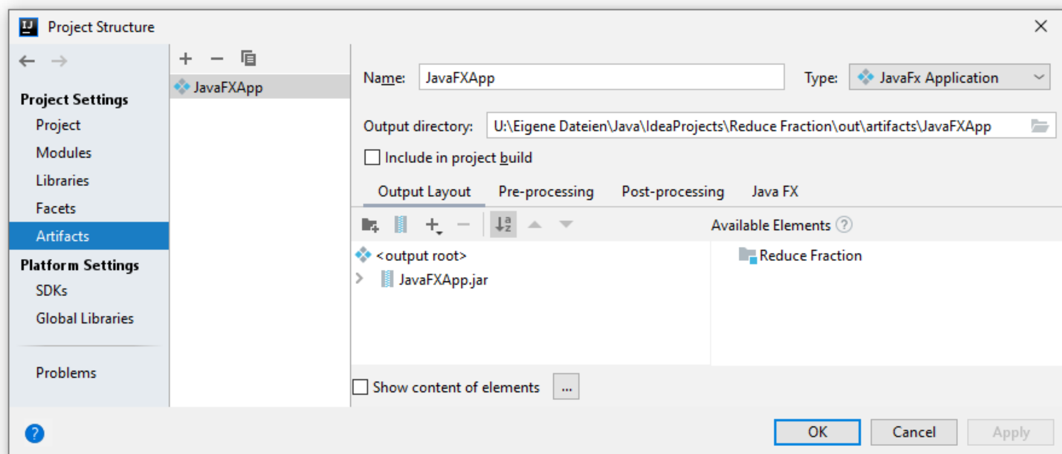
Wir kehren zum Projekt in

U:\Eigene Dateien\Java\IdeaProjects\Reduce Fraction

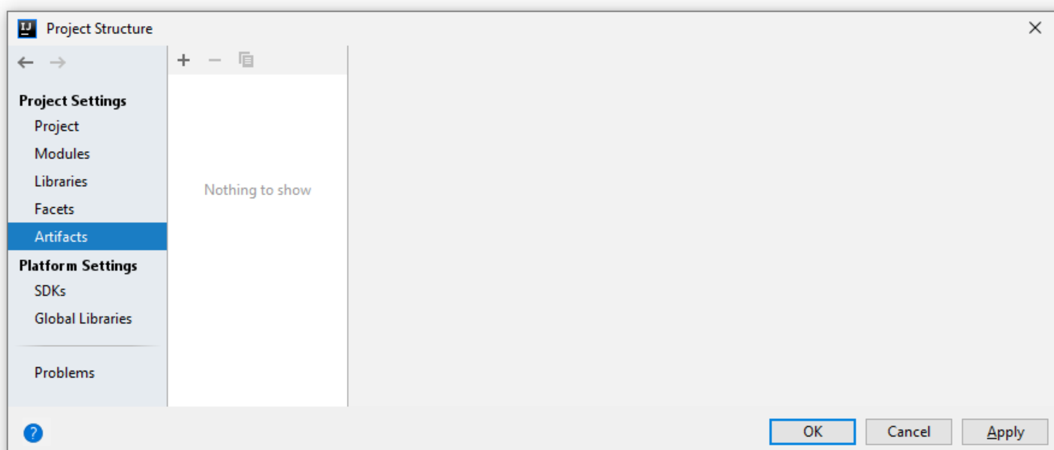
zurück.[1] Nach

File > Project Structure > Artifacts

ist in der Regel (aber nicht zuverlässig) bereits ein sogenanntes **Artifact** vorhanden, und wir können über das Kontrollkästchen **Include in project build** dafür sorgen, dass beim Erstellen des Projekts auch eine ausführbare **jar**-Datei erstellt wird:



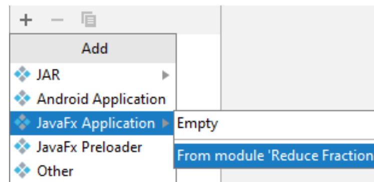
In der folgenden Situation



ist leicht Abhilfe zu schaffen:

[1] Es ist hier zu finden: **...\BspUeb\JavaFX\Reduce Fraction\Reduce Fraction mit Java 8**

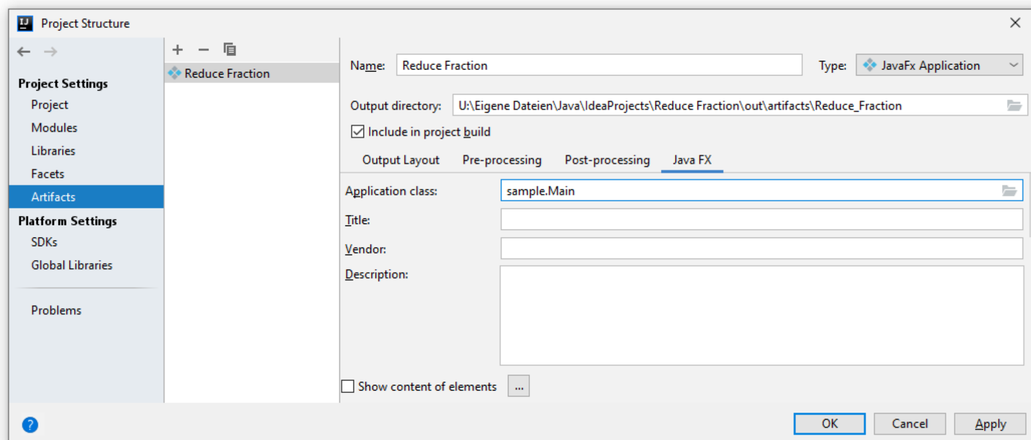
- Wählen Sie nach einem Klick auf das Pluszeichen eine aus dem aktuellen Modul (im Sinn von IntelliJ) zu erstellende JavaFX-Anwendung:



- Sorgen Sie über das Kontrollkästchen **Include in project build** dafür, dass beim Erstellen des Projekts auch eine ausführbare **jar**-Datei erstellt wird. Sie können aber auch darauf verzichten und bei Bedarf die Erstellung der **jar**-Datei mit dem folgenden Menübefehl anfordern:

  Build > Build Artifacts

- Tragen Sie auf der Registerkarte **Java FX** die Anwendungsklasse ein (inkl. Paketname):



- Quittieren Sie mit **OK**.

Wird das Projekt anschließend z. B. über den Menübefehl

Build > Build Project

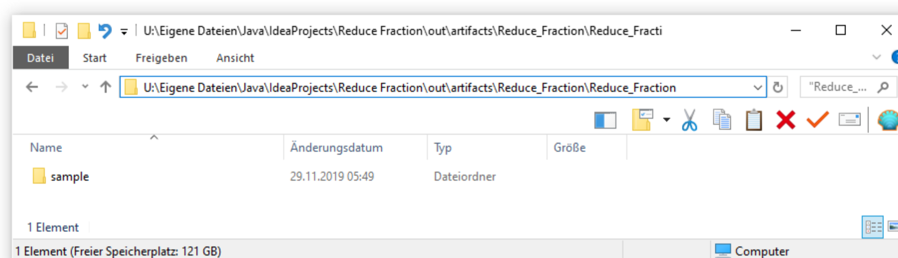
oder den Symbolschalter neu erstellt, erhalten wir im Ordner

...\out\artifacts

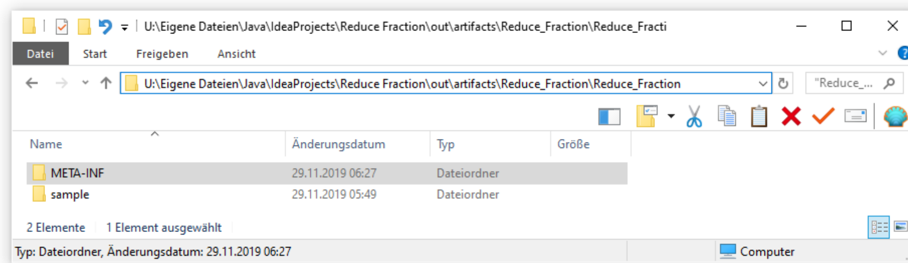
einen Unterordner (im Beispiel: **Reduce_Fraction**) mit der (leider noch nicht ausführbaren) **jar**-Datei.

Weil das Erstellen der **jar**-Datei scheitert, wenn der Projektname deutsche Umlaute enthält, haben wir für das JavaFX-Projekt eine englische Bezeichnung gewählt (siehe Abschnitt 4.9.1).

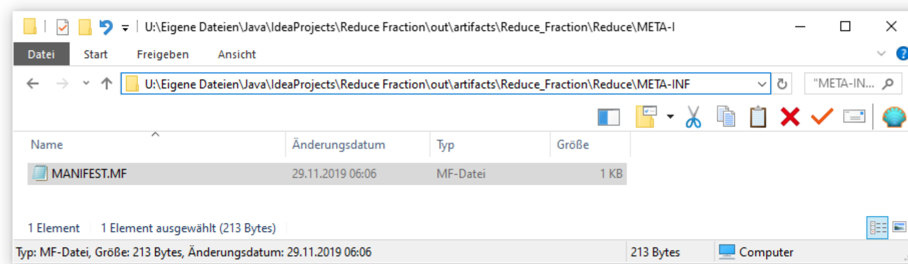
Nach dem Auspacken der **jar**-Datei zeigt sich für die Produktion aus unserem Projekt, dass der essentielle Ordner **META-INF** fehlt (vgl. Abschnitt 6.1.3.4 zum korrekten Inhalt einer **jar**-Datei):



Wir legen ihn an



und erstellen darin eine Textdatei mit dem Namen **MANIFEST.MF**



und mit dem folgenden Inhalt:

```
Manifest-Version: 1.0
Created-By: JavaFX Packager
Implementation-Vendor:
Implementation-Title:
Implementation-Version:
Main-Class: sample.Main
Class-Path:
Permissions: sandbox
JavaFX-Version: 8u202
```

Nun kann die **jar**-Datei auf einem Rechner per Doppelklick gestartet werden, wenn die installierte JVM für **jar**-Dateien zuständig ist, und JavaFX (alias OpenJFX) in der Standardbibliothek enthalten ist. Bei der im Abschnitt 1.2.1 beschriebenen Installation sind diese Bedingungen erfüllt.

Zum Erstellen der **jar**-Datei zu einem JavaFX-Projekt benötigt IntelliJ den sogenannten *Java FX Packager*, der im OpenJDK 13 leider fehlt.[1] Ein mit dem OpenJDK 8 entwickeltes JavaFX-Programm kann aber durchaus von einer JVM auf dem Versionsstand 13 ausgeführt werden, wobei lediglich das (von Ihnen als Profi eingerichtete) Startverfahren etwas aufwändiger ist (siehe Abschnitt 4.9.5). Für die Benutzer bleibt es beim Doppelklick.

6.2 Module

Die mit Java 9 eingeführte Paket-Modularisierung (das *Java Platform Module System*, JPMS) bringt folgende Optimierungen für die Java-Plattform:

- Zuverlässige Konfiguration statt JAR-Hölle
  Befinden sich im Klassenpfad mehrere Ordner oder **jar**-Dateien, die verschiedene Versionen eines Pakets enthalten, dann hängt es von der Reihenfolge im Klassenpfad ab, welche Version einer Klasse geladen wird. Das Modulsystem verwendet statt des Klassenpfads einen sogenannten *Modulpfad* und verhindert, dass ein Paket in mehreren Modulen auf dem Pfad enthalten ist (siehe Abschnitt 6.2.4).

[1] Das Problem sollte bald gelöst sein: https://openjdk.java.net/jeps/311, https://openjdk.java.net/jeps/343

- Zugriffsregulation oberhalb von Paketen
  Bis Java 8 erlaubt der Typ-Modifikator **public** den Vollzugriff für fremde Typen in beliebigen Paketen. Ab Java 9 kann man Pakete zu Modulen zusammenfassen und für jedes Modul festlegen, welche seiner Pakete exportiert oder für den Modul-internen Gebrauch reserviert werden sollen. Eine als **public** deklarierte Klasse in einem *nicht-exportierten* Paket ist nur für andere Pakete im *selben Modul* sichtbar. Befindet sich eine als **public** deklarierte Klasse hingegen in einem *exportierten* Paket, dann kann sie von Klassen in allen Modulen genutzt werden, die entweder ihre Abhängigkeit vom Quellmodul explizit deklariert haben oder implizit vom Quellmodul abhängig sind. Von der neuen Zugriffsabschottung profitiert nicht zuletzt das Java-API, indem API-interne Pakete (z. B. **com.sun.***) in Java 9 für normalen Anwendungscode nicht mehr zugänglich sind.
- Definierte Abhängigkeiten zwischen Programmteilen
  Abgesehen von Kompatibilitätslösungen deklariert ein JPMS-Modul seine Abhängigkeiten von anderen Modulen explizit. Somit wird die Struktur einer komplexen Anwendung im Quellcode klar artikuliert. Die im vorherigen Aufzählungspunkt beschriebene modulbasierte Zugriffsregulation verhindert z. B., dass die Klassen im GUI-Modul direkt auf Klassen im Datenbankmodul zugreifen. Java 9 erleichtert die Erstellung von übersichtlichen und wartungsfreundlichen Anwendungen, während sich traditionelle Großprojekte gelegentlich zu einer großen Matschkugel (engl.: *big ball of mud*) entwickeln.[1] Auf den Installationsrechner der Anwender lässt sich ein nicht-modulares Projekt zwar in **jar**-Dateien ausliefern, die jeweils mehrere Pakete zusammenfassen (siehe Abschnitt 6.1.3), doch befinden sich diese **jar**-Dateien unstrukturiert im Klassenpfad und verwenden sich auf schwer durchschaubare Weise gegenseitig.
- Eigenständige Anwendungen mit bedarfsgerechter Laufzeitumgebung
  Selbstverständlich ist auch die Standardbibliothek seit Java 9 modular aufgebaut. Wie mit dem Kommando

  ```
  java --list-modules
  ```

  zu ermitteln ist, besteht das API von OpenJDK 13 aus 71 Modulen. Das neue JDK-Werkzeug **jlink** macht es möglich, ein Programm mit einer angepassten Laufzeitumgebung bestehend aus den tatsächlich benötigten Modulen zu erstellen. Der Kunde erhält ein selbständig ausführbares Programm mit relativ schlankem Lieferumfang und einem reduzierten Risiko, von zukünftig entdeckten Sicherheitslücken im Java-API betroffen zu sein. Enthält ein Programm mit angepasster modularer Laufzeitumgebung aber doch ein Sicherheitsproblem, dann ist der Programmanbieter für die Update-Versorgung verantwortlich.
- Bessere Performanz beim Laden von Klassen
  Jedes Modul deklariert (explizit oder implizit) seine Abhängigkeiten von anderen Modulen, sodass nur wenige Module nach einem Paket bzw. nach einer Klasse durchsucht werden müssen. Vor Java 9 musste der gesamte Klassenpfad durchsucht werden.

Die am häufigsten formulierte Kritik am JPMS betrifft die fehlende Versionsunterstützung. Bei der Deklaration einer Abhängigkeit von einem anderen Modul ist keine Versionsangabe möglich. Im Moduldeskriptor und im Namen einer modularen Archivdatei (siehe Abschnitte 6.2.3 und 6.2.6) lässt sich eine Version angeben, die aber keine steuernde Wirkung hat. Die Verwaltung von Modul-Versionen wird explizit den Erstellungswerkzeugen (wie z. B. Maven, Gradle) überlassen, was in der quasi-offiziellen JPMS-Dokumentation (Reinhold 2016, Abschnitt 1.1) unmissverständlich zum Ausdruck kommt:

[1] https://de.wikipedia.org/wiki/Big_Ball_of_Mud

> A module's declaration does not include a version string, nor constraints upon the version strings of the modules upon which it depends. This is intentional: It is not a goal of the module system to solve the version-selection problem, which is best left to build tools and container applications.

Ein Modul im JPMS enthält:

- Eine Moduldeklaration
  Die Moduldeklaration befindet sich in einer Java-Quellcodedatei namens **module-info.java**. Sie enthält den Namen des Moduls und deklariert u.a. die exportierten Pakete sowie die Abhängigkeiten von anderen Modulen (siehe Abschnitt 6.2.1).
- Pakete
  Gelegentlich sind Module sinnvoll, die *keine* Pakete enthalten, sondern nur transitive Abhängigkeitsdeklarationen (siehe Abschnitt 6.2.1.2).
- Optional auch Daten (z. B. Medien, Sprachversionen von Beschriftungen)

Joshua Bloch, ein profunder Kenner der Java-Plattform und Mitentwickler der Standardbibliothek, beurteilt in seinem zuletzt 2018 aufgelegten Standardwerk zur Java-Programmierung den aktuellen Nutzen des Modulsystems zurückhaltend (S. 77):

> It is too early to say whether modules will achieve widespread use outside of the JDK itself. In the meantime, it seems best to avoid them unless you have a compelling need.

6.2.1 *Moduldeklarationsdatei module-info.java*

In der Datei **module-info.java** deklariert ein Modul im Wesentlichen ...

- seinen Namen
- seine Anhängigkeiten von anderen Modulen durch **requires**-Deklarationen
- seine exportierten (für andere Module zugänglichen) Pakete durch **exports**-Deklarationen

Die folgenden Deklarationen werden nur selten benötigt:

- Verwendung bzw. Implementation von Diensten durch **uses**- bzw. **provides**-Deklarationen
- Öffnung von Paketen für die Reflexion zur Laufzeit durch **opens**-Deklarationen

In der Moduldeklarationsdatei folgt auf das einleitende Schlüsselwort **module** der Name und ein durch geschweifte Klammern begrenzter Block, z. B.:

```
module de.uni_trier.zimk.matrain {
   requires transitive de.uni_trier.zimk.util;
   exports de.uni_trier.zimk.matrain.br;
}
```

Die Moduldeklarationsdatei **module-info.java** wird vom Compiler in den sogenannten Moduldeskriptor **module-info.class** übersetzt. Dass der Name des Moduldeskriptors wegen des Bindestrichs kein zulässiger Klassenname ist, trägt zur Vermeidung von Namenskollisionen bei.

6.2.1.1 Modulnamen

Für Modulnamen gelten folgende Regeln und Konventionen:

- Die im Abschnitt 3.1.6 beschriebenen Namensregeln sind einzuhalten, sodass z. B. der Bindestrich kein zulässiges Zeichen in Modulnamen ist.
- Durch die empfohlene Beschränkung auf Kleinbuchstaben werden Namenskonflikte mit Klassen und Schnittstellen vermieden.

- Durch die Verwendung von DNS (*Domain Name System*) - Namensbestandteilen in umgekehrter Reihenfolge und durch Punkte separiert als Präfix wird für weltweit eindeutige Modulnamen gesorgt (analog zur entsprechenden Empfehlung für Paketnamen, siehe Abschnitt 6.1.1.5), z. B.:
  ```
  de.uni_trier.zimk.util
  ```
  Wenn ein Modul garantiert nie den Anwendungsbereich einer Organisation/Firma verlässt, kann der Kürze halber auf führende DNS-Bestandteile im Modulnamen verzichtet werden.
- Existiert im Modul ein *herausgehobenes* exportiertes Paket, sollte dessen Name auch als Modulname verwendet werden.
- Die exportierten Pakete eines Moduls sollten den Modulnamen als Präfix verwenden, z. B.:
  ```
  de.uni_trier.zimk.util.conio
  ```

6.2.1.2 requires-Deklaration

Ein normales (sogenanntes *explizites*) Modul deklariert in der Datei **module-info.java** seine Abhängigikeiten von anderen Modulen, legt also das Universum der Typen (Klassen und Schnittstellen) fest, die in seinem eigenen Code benötigt werden (Gosling et al. 2019, Abschnitt 7.7). Für jedes erforderliche andere Modul (außer **java.base**, siehe unten) ist eine **requires**-Deklaration erforderlich, die nach dem einleitenden Schlüsselwort einen Modulnamen nennt und mit einem Semikolon endet, z. B.:

```
module de.uni_trier.zimk.matrain {
   requires de.uni_trier.zimk.util;
    . . .
}
```

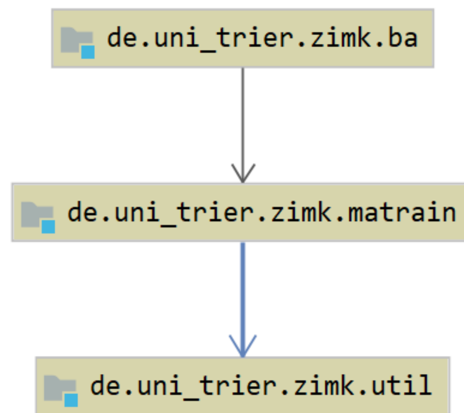
Man sagt, dass im Beispiel die **Lesbarkeit** (engl.: *readability*) des Moduls `de.uni_trier.zimk.util` durch das Modul `de.uni_trier.zimk.matrain` beantragt wird.

Wenn z. B. eine Klasse im Modul `de.uni_trier.zimk.matrain` eine Methode enthält, welche ein Objekt mit einem in `de.uni_trier.zimk.util` definierten Typ abliefert, dann muss jedes von `de.uni_trier.zimk.matrain` abhängige Modul ebenfalls eine Leseberechtigung für `de.uni_trier.zimk.util` besitzen. Damit dazu keine explizite Abhängigkeitsdeklaration erforderlich ist, kann mit dem Zusatz **transitive** hinter **requires** eine Abhängigkeit weitergegeben werden an Module, welche vom deklarierenden Modul abhängen. Wenn auf Basis der folgenden Deklaration

```
module de.uni_trier.zimk.matrain {
   requires transitive de.uni_trier.zimk.util;
    . . .
}
```

ein Modul `de.uni_trier.zimk.ba` seine Abhängigkeit von `de.uni_trier.zimk.matrain` erklärt, dann ist `de.uni_trier.zimk.ba` implizit auch von `de.uni_trier.zimk.util` abhängig. Wer eine Leseberechtigung für `de.uni_trier.zimk.matrain` beantragt, muss sinnvollerweise dessen Abhängigkeiten nicht kennen. Selbstverständlich klappt die transitive Abhängigkeit auch über Zwischenschritte.

Mit der kostenpflichtigen Ultimate-Edition unserer Entwicklungsumgebung IntelliJ IDEA lässt sich ein Abhängigkeitsdiagramm für die Module eines Projekts (**Java Modules Diagram**) erstellen, z. B.:



Für die einfache und die transitive Abhängigkeit werden unterschiedlich formatierte Pfeile verwendet.

Ein sogenanntes *Aggregatormodul* enthält keine Pakete, aber mehrere transitive Abhängigkeitsdeklarationen. So kann ein Bündel von Abhängigkeiten mit geringem Aufwand auf mehrere Module übertragen und an einer Stelle gepflegt werden.

Das Modul **java.base** der Java-Standardbibliothek ist (ohne **requires**-Anweisung) für jedes Modul lesbar.

6.2.1.3 exports-Deklaration

Die in einem Modul enthaltenen Pakete sind per Voreinstellung nur modulintern sichtbar. Eine **public**-Deklaration für eine im Modul enthaltene Klasse wirkt sich also nur auf die anderen Pakete im eigenen Modul aus. Für jedes Paket, das auch außerhalb des Moduls sichtbar sein soll, ist in der Datei **module-info.java** eine **exports**-Deklaration erforderlich, z. B.:

```
module de.uni_trier.zimk.util {
   exports de.uni_trier.zimk.util.conio;
}
```

Unterpakete (also Pakete mit einem durch Anhängen von Segmenten gebildeten) Namen (z. B. `de.uni_trier.zimk.util.conio.impl`) werden durch eine **exports**-Anweisung *nicht* mit einbezogen. Weil keine Joker-Zeichen unterstützt werden, sind eventuell zahlreiche **exports**-Deklarationen fällig.

Damit die von einem Modul `A` exportierten Pakete im Code eines Moduls `B` tatsächlich nutzbar sind, muss für das Modul `B` eine explizit deklarierte oder implizit bestehende Abhängigkeit vom Modul `A` vorliegen (siehe Abschnitt 6.2.1.2).

Durch eine **exports**-Deklaration mit **to**-Klausel kann die Freigabe eines Pakets auf eine Liste von Modulen eingeschränkt werden, z. B.:

```
module de.uni_trier.zimk.util {
   exports de.uni_trier.zimk.util.conio to
      de.uni_trier.zimk.util.ba,
      de.uni_trier.zimk.util.bm;
}
```

Man spricht hier von einer *qualifizierten* **exports**-Deklaration.

Im Java-API wird die neue Option interner (nicht exportierter) Pakete, die für Anwendungen nicht sichtbar sind, intensiv genutzt.

6.2.1.4 uses- und provides-Deklaration

Bei großen Software-Systemen ist eine lose Kopplung von kooperierenden Klassen erwünscht, sodass erst zur Laufzeit (eventuell aufgrund einer Benutzerentscheidung) zwischen mehreren Klassen, die eine bestimmte Dienstleistung erbringen können, gewählt wird. Zur Laufzeit wird die Suche nach einem Service-Provider durch die API-Klasse **java.util.ServiceLoader** unterstützt.

Wenn ein Modul einen Dienst benötigt, der in der Regel über eine bestimmte Schnittstelle (mit abstrakt definierten Handlungskompetenzen, siehe Kapitel 9) definiert ist, dann wird dieser Dienst (diese Schnittstelle) in der Moduldeklaration per **uses**-Deklaration angemeldet. Als Beispiel betrachten wir das Modul **java.sql** aus dem Java-API, das einen Treiber benötigt, um mit einer SQL-Datenbank zu kooperieren:

```
module java.sql {
    requires transitive java.logging;
    requires transitive java.xml;

    exports java.sql;
    exports javax.sql;
    exports javax.transaction.xa;

    uses java.sql.Driver;
}
```

Grundsätzlich darf als Service-Spezifikation an Stelle einer Schnittstelle auch eine Klasse angegeben werden.

Enthält ein Modul eine Implementation für einen Dienst, so wird dies in der Moduldeklaration durch eine **provides**-Deklaration angezeigt. Auf das Schlüsselwort **provides** folgen:

- der Schnittstellen- oder (seltener) der Klassenname
- das Schlüsselwort **with**
- der Name der implementierenden Klasse

Im folgenden Modul **com.mysql.jdbc** (Beispiel aus Reinhold 2016, Abschnitt 4) wird der vom Modul **java.sql** benötigte Dienst **java.sql.Driver** angeboten:

```
module com.mysql.jdbc {
    requires java.sql;
    requires org.slf4j;
    exports com.mysql.jdbc;
    provides java.sql.Driver with com.mysql.jdbc.Driver;
}
```

Beim Anwendungsstart stellt das Java-Modulsystem sicher, dass zu jeder **uses**-Deklaration mindestens *eine* passende **provides**-Deklaration vorhanden ist und verweigert anderenfalls den Start.

6.2.1.5 opens-Deklaration

Einige Bibliotheken bzw. Frameworks verwenden eine bisher im Kurs noch nicht angesprochene Software-Technik namens *Reflexion* zur Erledigung ihrer Aufgaben. Dabei werden zur Laufzeit Klassen geladen, inspiziert und instanziert. Beispiele für solche Frameworks sind:

- das zum Speichern von Objekten in relationalen Datenbanken verwendete JPA (*Java Persistence API*)
- das für Geschäftsanwendungen verbreitete **Spring**-Framework
- Testsysteme

Die meisten Entwickler müssen sich mit der Reflexion (im Zusammenhang mit den JPMS) *nicht* beschäftigen (Reinold 2016) und können daher den Rest dieses Abschnitts überspringen.

Mit der **opens**-Deklaration kann ein Modul für ein Paket die Laufzeit-Reflexion durch beliebige andere Module erlauben, die Sichtbarkeit zur Übersetzungszeit aber auf das eigene Modul beschränken, z. B.:

```
module com.example.foo {
   requires java.logging;
   exports com.example.foo.api;
   opens com.example.foo.impl;
}
```

Wie bei der **exports**-Deklaration kann auch bei der **opens**-Deklaration per **to**-Klausel die Freigabe auf eine Liste von Modulen eingeschränkt werden.

Man kann ein komplettes Modul für die Laufzeit-Reflexion öffnen, indem man die Moduldeklaration mit dem Schlüsselwort **open** einleitet, z. B.:

```
open module com.example.foo {
     . . .
}
```

Damit entsteht ein sogenanntes *offenes Modul*.

6.2.2 Quellcode-Organisation

Empfehlungen für die Quellcode-Organisation bei einem Modul:

- Für ein Modul legt man einen Ordner an, der den Namen des Moduls übernimmt.
- In diesem Ordner erstellt man die Moduldeklarationsdatei **module-info.java**.
- Unterhalb des Modulordners legt man eine Ordnerhierarchie mit den Paketen an (siehe Abschnitt 6.1.1.3).

Im weiteren Verlauf des Abschnitts 6.2 entwickeln wir ein aus den folgenden drei Modulen bestehendes Beispielprogramm, das sich (wieder einmal) mit der Bruchaddition beschäftigt:

- `de.uni_trier.zimk.util`
- `de.uni_trier.zimk.ba`
- `de.uni_trier.zimk.matrain`

Wir erstellen das Modul `de.uni_trier.zimk.util` mit einem Paket namens `de.uni_trier.zimk.util.conio`, das die altbekannte Klasse **Simput.java** enthält. In der Moduldeklarationsdatei wird das Paket exportiert:

```
module de.uni_trier.zimk.util {
   exports de.uni_trier.zimk.util.conio;
}
```

Wir übernehmen den `Simput`-Quellcode (samt **import**-Deklarationen) aus der Datei

...\BspUeb\Einleitung\Bruchaddition\Konsole\Simput.java

und ergänzen am Anfang eine **package**-Deklaration:

```
package de.uni_trier.zimk.util.conio;

import java.util.*;
import java.io.*;

public class Simput {
 . . .
}
```

Der Einfachheit halber sind die Module und Pakte im Demonstrationsbeispiel von untypisch-beschränktem Umfang:

- Jedes Modul enthält nur *ein* Paket.
- Jedes Paket enthält nur *eine* Klasse.

In einer realen modularen Anwendung enthalten die Module mehrere Pakete und die Pakete jeweils mehrere Typen (Klassen und Schnittstellen).

Das Modul `de.uni_trier.zimk.matrain` enthält im Paket `de.uni_trier.zimk.matrain.br` die altbekannte Klasse `Bruch`. Die Datei **module-info.java** wurde in früheren Abschnitten schon als Beispiel verwendet:

```
module de.uni_trier.zimk.matrain {
   requires transitive de.uni_trier.zimk.util;
   exports de.uni_trier.zimk.matrain.br;
}
```

Weil die Abhängigkeit von `de.uni_trier.zimk.util` als transitiv erklärt wird, überträgt sie sich auf Module, die von `de.uni_trier.zimk.matrain` abhängen.

Wir übernehmen den Quellcode der Klasse `Bruch` aus

...\BspUeb\Klassen und Objekte\Bruch\B3 (mit Konstruktoren)\Bruch.java

und ergänzen am Anfang eine **package**-Deklaration sowie eine **import**-Deklaration für das Paket `de.uni_trier.zimk.util.conio` mit der Klasse `Simput`:

```
package de.uni_trier.zimk.matrain.br;

import de.uni_trier.zimk.util.conio.Simput;

public class Bruch {
  . . .
}
```

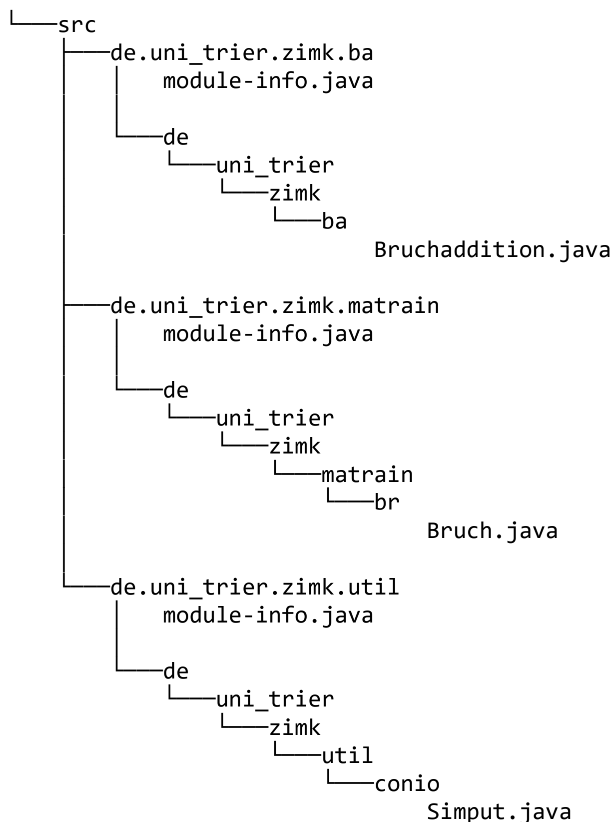
Das Hauptmodul `de.uni_trier.zimk.ba` des Bruchadditionsprogramms enthält ein gleichnamiges Paket mit der Startklasse. In der Modul-Deklarationsdatei wird die Abhängigkeit von `de.uni_trier.zimk.matrain` erklärt:

```
module de.uni_trier.zimk.ba {
   requires de.uni_trier.zimk.matrain;
}
```

In der Quellcodedatei der Startklasse wird die Paketzugehörigkeit deklariert. Danach wird die Klasse `Bruch` aus dem Paket `de.uni_trier.zimk.matrain.br` im Modul `de.uni_trier.zimk.matrain` importiert, von dem das Modul `de.uni_trier.zimk.ba` explizit abhängt. Außerdem wird die Klasse `Simput` aus dem Paket `de.uni_trier.zimk.util.conio` im Modul `de.uni_trier.zimk.util` importiert, von dem das Modul `de.uni_trier.zimk.ba` transitiv abhängt:

```
package de.uni_trier.zimk.ba;

import de.uni_trier.zimk.matrain.br.Bruch;
import de.uni_trier.zimk.util.conio.Simput;

class Bruchaddition {
    public static void main(String[] args) {
        Bruch b1 = new Bruch(), b2 = new Bruch();

        System.out.println("1. Bruch");
        b1.frage();
        b1.kuerze();
        b1.zeige();

        System.out.println("\n2. Bruch");
        b2.frage();
        b2.kuerze();
        b2.zeige();

        System.out.println("\nSumme");
        b1.addiere(b2);
        b1.zeige();

        System.out.print("\nIhre Zufriedenheit mit der Software (1-5): ");
        int zuf = Simput.gint();
        System.out.println("Verstanden: " + zuf);
    }
}
```

Wir sammeln alle Modul-Quellcodedateien in einem mit dem Ordner **src** startenden Verzeichnisast, der folgenden Aufbau besitzt:[1]



[1] Die Baumansicht wurde mit dem Windows-Kommando **tree /f** erstellt.

6.2.3 Übersetzung in ein explodiertes Modul und den Moduldeskriptor

Da wir momentan der Klarheit halber direkt mit den OpenJDK-Werkzeugen arbeiten, rufen wir zum Übersetzen des Quellcodes der drei Module `de.uni_trier.zimk.util`, `de.uni_trier.zimk.matrain` und `de.uni_trier.zimk.ba` den OpenJDK 13 - Compiler **javac.exe** auf.[1] Bei der Übersetzung eines Moduls entstehen:

- Der Moduldeskriptor **module-info.class**
- Die **class**-Dateien zu den Paketklassen

Einen Dateiverzeichnisbaum mit den **class**-Dateien bezeichnet man als *explodiertes Modul*.

Zunächst übersetzen wir die Module einzeln. Im **javac**-Kommando zur Übersetzung des Moduls `de.uni_trier.zimk.util` wird mit der Option **-d** (*destination*) ein Ausgabeordner benannt, der existieren muss. Er wird im Beispiel relativ vom aktuellen Ordner des Konsolenfensters aus adressiert. Danach folgen (ebenfalls relativ adressiert) die zu übersetzenden Quellcodedateien:[2]

```
>javac -d expmods\de.uni_trier.zimk.util  src\de.uni_trier.zimk.util\*.java
   src\de.uni_trier.zimk.util\de\uni_trier\zimk\util\conio\*.java
```

Während die Übersetzung des Moduls `de.uni_trier.zimk.util` klappt, scheitert ein analog aufgebauter **javac**-Aufruf für das Modul `de.uni_trier.zimk.matrain`

```
>javac -d expmods\de.uni_trier.zimk.matrain  src\de.uni_trier.zimk.matrain\*.java
   src\de.uni_trier.zimk.matrain\de\uni_trier\zimk\matrain\br\*.java
```

mit der Fehlermeldung:

```
src\de.uni_trier.zimk.matrain\module-info.java:2: error: module not found:
de.uni_trier.zimk.util
    requires transitive de.uni_trier.zimk.util;
                                             ^
```

Das laut Deklarationsdatei zu lesende Modul `de.uni_trier.zimk.util` wird nicht gefunden.

Da wir das Modul bereits übersetzt haben, besteht eine Lösung des Problems darin, den sogenannten *Modulpfad* (siehe Abschnitt 6.2.4) über die **javac**-Option **-p** (oder die Langform **--module-path**) anzugeben:

```
>javac -d expmods\de.uni_trier.zimk.matrain -p expmods
   src\de.uni_trier.zimk.matrain\*.java
   src\de.uni_trier.zimk.matrain\de\uni_trier\zimk\matrain\br\*.java
```

Alternativ kann über die Option **--module-source-path**, zu der keine Kurzschreibweise existiert, der Ordner mit den Quellcodedateien benötigter Module angegeben werden, wobei keine vorherige Übersetzung vorausgesetzt wird. Im Beispiel befinden sich vom aktuellen Ordner das Konsolenfensters aus gesehen (relativ adressiert) alle Module im Ordner **src**. Es ist ein Zielordner anzugeben, der *beide* Module aufnehmen soll, sodass der folgende **javac**-Aufruf resultiert:

```
>javac -d expmods --module-source-path src src\de.uni_trier.zimk.matrain\*.java
   src\de.uni_trier.zimk.matrain\de\uni_trier\zimk\matrain\br\*.java
```

Schließlich wird noch das Hauptmodul `de.uni_trier.zimk.ba` unter Verwendung des Modulpfads übersetzt:

---

[1] Hier ist die Dokumentation der Firma Oracle zum Java 13 - Compiler zu finden:
https://docs.oracle.com/en/java/javase/13/docs/specs/man/javac.html

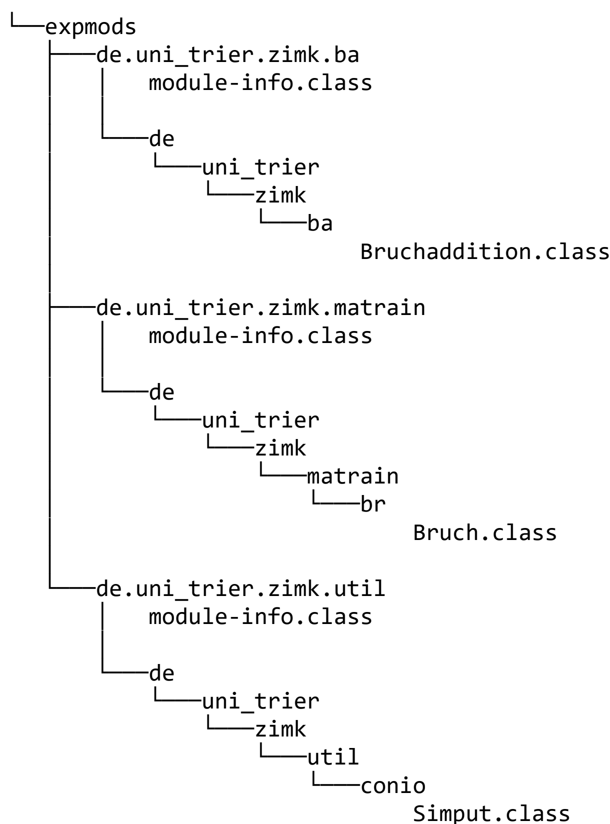
[2] Die **javac**-Aufrufe des aktuellen Abschnitts setzen voraus, dass sich der **bin**-Unterordner einer JDK-Installation mit Modulunterstützung (z. B. **C:\Program Files\Java\OpenJDK-13\bin**) im Windows-Pfad für ausführbare Programme befindet.

```
>javac -d expmods\de.uni_trier.zimk.ba -p expmods src\de.uni_trier.zimk.ba\*.java
   src\de.uni_trier.zimk.ba\de\uni_trier\zimk\ba\*.java
```

Die Übersetzung der drei Module lässt sich auch in *einem* **javac**-Aufruf erledigen. Allerdings wird dazu eine leistungsfähige Betriebssystem-Kommandosprache benötigt. Diese Voraussetzung ist in Linux, in MacOS X und auch in der Windows-PowerShell erfüllt, aber nicht im traditionellen Windows-Konsolenfenster, das wir oben verwendet haben. Ist unter Windows ein PowerShell-Fenster auf den Stammordner von **src** eingestellt, lassen sich die drei in **src**-Unterordnern befindlichen Module `de.uni_trier.zimk.util`, `de.uni_trier.zimk.matrain` und `de.uni_trier.zimk.ba` mit dem folgenden **javac**-Kommando übersetzen:

```
>javac -d expmods  --module-source-path src  $(dir src -r -i "*.java")
```

Als Resultat der **javac**-Aufrufe erhalten wir die folgende Ordnerstruktur mit den explodierten Modulen:



```
└──expmods
    ├──de.uni_trier.zimk.ba
    │   │   module-info.class
    │   │
    │   └──de
    │       └──uni_trier
    │           └──zimk
    │               └──ba
    │                       Bruchaddition.class
    │
    ├──de.uni_trier.zimk.matrain
    │   │   module-info.class
    │   │
    │   └──de
    │       └──uni_trier
    │           └──zimk
    │               └──matrain
    │                   └──br
    │                           Bruch.class
    │
    └──de.uni_trier.zimk.util
        │   module-info.class
        │
        └──de
            └──uni_trier
                └──zimk
                    └──util
                        └──conio
                                Simput.class
```

6.2.4 Modulpfad

Wir haben im Abschnitt 6.2.3 in einem **javac**-Aufruf den mit Java 9 eingeführten Modulpfad benutzt, der den fehleranfälligen traditionellen Klassenpfad ersetzt, z. B.:

```
>javac -d expmods\de.uni_trier.zimk.matrain -p expmods
   src\de.uni_trier.zimk.matrain\*.java
   src\de.uni_trier.zimk.matrain\de\zimk\matrain\br\*.java
```

Zur Spezifikation des Modulpfads bei der Übersetzung oder Ausführung eines Programms dient die Option **--module-path** mit der Kurzform **-p**. Man kann einzelne Module oder Ordner mit Modulen angeben, wobei ein Modul im explodierten Zustand oder als modulare **jar**-Datei vorliegen kann.

Zusammen mit den so zugänglichen Anwendungs- oder Bibliotheksmodulen gehören die Module der Java-Runtime zu den beobachtbaren Modulen (engl.: *observable modules*).

Regeln für den Modulpfad (den Wert zur Option **-p**) sind (vgl. Mak & Bakker 2017):

- Es ist eine Liste von Einträgen erlaubt, die unter Windows jeweils durch ein Semikolon zu trennen sind.
- Als Einträge sind erlaubt:
    - Ein einzelnes Modul
      Es kann sich um den Ordner eines explodierten Moduls oder um eine modulare **jar**-Datei handeln.
    - Ein Ordner mit Modulen
      Es dürfen explodierte Module oder modulare **jar**-Dateien enthalten sein.

Wichtige Details zu den modularen **jar**-Dateien folgen im Abschnitt 6.2.6.

Um ein Wiederaufflammen der JAR-Hölle zu verhindern, darf im JPMS ein Paket nicht über mehrere Module verteilt werden, d.h.:

- Der Compiler protestiert, wenn zwei Module namensgleiche Pakete enthalten.
- Die Laufzeitumgebung verweigert den Programmstart, wenn sich im Modulpfad zwei Module mit namensgleichen Paketen befinden.

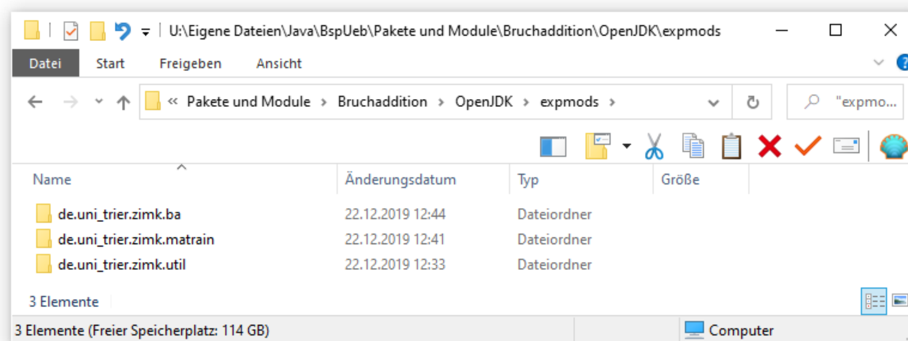
Das *Package Splitting* wird rigoros auch dann verhindert, wenn ...[1]

- die namensgleichen Pakete nicht exportiert werden,
- die namensgleichen Pakete keine namensgleichen Typen enthalten.

Anders als beim herkömmlichen Klassenpfad kann es beim Modulpfad nicht passieren, dass die mehr oder weniger zufällige Reihenfolge der Pfadeinträge darüber entscheidet, aus welchem Paket eine Klasse schließlich geladen wird.

6.2.5 Ausführen

Wir starten das Beispielprogramm in einem Konsolenfenster, dessen aktuelles Verzeichnis u.a. den Ordner **expmods** mit den explodierten Modulen enthält,



über den folgenden Aufruf des Java-Starters[2]

```
>java --module-path expmods --module
     de.uni_trier.zimk.ba/de.uni_trier.zimk.ba.Bruchaddition
```

und verwenden dabei zwei Optionen:

---

[1] Quelle: https://www.informatik-aktuell.de/entwicklung/programmiersprachen/java-9-das-neue-modulsystem-jigsaw-tutorial.html

[2] Der **java**-Aufruf setzt voraus, dass sich der **bin**-Unterordner einer JDK-Installation mit Modulunterstützung (z. B. **C:\Program Files\Java\OpenJDK-13\bin**) im Windows-Pfad für ausführbare Programme befindet.

- Modulpfad
  Die Option **--module-path** mit der Kurzform **-p** zur Spezifikation des Modulpfads haben wir schon beim Übersetzen verwendet.
- Startmodul mit Startklasse
  Über die Option **--module** (mit der Kurzform **-m**) gibt man den Namen des Hauptmoduls und dahinter, durch einen Schrägstrich getrennt, den vollqualifizierten Namen der Startklasse an.

Hier ist die Kurzform des Startkommandos zu sehen:

```
>java -p expmods  -m de.uni_trier.zimk.ba/de.uni_trier.zimk.ba.Bruchaddition
```

6.2.6 Modulare jar-Dateien

Aus einem explodierten Modul wird in aller Regel eine modulare **jar**-Datei erstellt. Sie unterscheidet sich von einer herkömmlichen **jar**-Datei (siehe Abschnitt 6.1.3) nur durch die Anwesenheit des Moduldeskriptors **module-info.class** im Wurzelverzeichnis.

Die Erstellung der modularen **jar**-Datei zu einem Modul erledigt man mit dem OpenJDK-Hilfsprogramm **jar**.[1] Für uns sind folgende Programmoptionen relevant:

- **--create**
  Es soll eine modulare **jar**-Datei angelegt werden.
- **--file**
  Pfad und Dateiname für die anzulegende **jar**-Datei werden festgelegt. Der Ausgabeordner muss existieren. Während ein Modulname keine Version enthalten darf, ist er im Namen der modularen Archivdatei erlaubt.
- **--module-version**
  Das Modul erhält eine Version, die im Moduldeskriptor **module-info.class** eingetragen wird, aber lediglich als beschreibendes Attribut fungiert.
- **--main-class**
  Enthält das Modul eine zu startende Hauptklasse, dann muss diese über ihren vollqualifizierten Namen bekanntgegeben werden. Es wird ein Attribut im Moduldeskriptor **module-info.class** eingetragen. Außerdem wird ein Eintrag in der Datei **Manifest.MF** vorgenommen.
- **-C**
  Hier ist das Wurzelverzeichnis des Moduls anzugeben (mit der Datei **module-info.class**). Die Option muss groß geschrieben werden. Wenn sie fehlt, wird der aktuelle Ordner angenommen.
- **.**
  Am Ende des Kommandos ist unbedingt ein Punkt erforderlich.

Im Bruchadditionsbeispiel arbeiten wir mit einem Konsolenfenster, dessen aktuelles Verzeichnis u.a. den Ordner **expmods** mit den explodierten Modulen enthält:

```
Eingabeaufforderung

Verzeichnis von U:\Eigene Dateien\Java\BspUeb\Pakete und Module\Bruchaddition\OpenJDK

22.12.2019  13:04    <DIR>          .
22.12.2019  13:04    <DIR>          ..
22.12.2019  12:44    <DIR>          expmods
22.12.2019  12:28    <DIR>          src
               0 Datei(en),              0 Bytes
               4 Verzeichnis(se), 122.480.091.136 Bytes frei
```

[1] Hier ist eine aktuelle Dokumentation der Firma Oracle zu finden:
https://docs.oracle.com/en/java/javase/12/tools/jar.html

Hier legen wir zunächst einen Ordner namens `mods` für die zu erstellenden modularen **jar**-Dateien an:

```
>mkdir mods
```

Im folgenden **jar**-Aufruf wird das Modul `de.uni_trier.zimk.util` verpackt:

```
>jar --create --file mods/de.uni_trier.zimk.util-1.0.jar --module-version 1.0
   -C expmods/de.uni_trier.zimk.util .
```

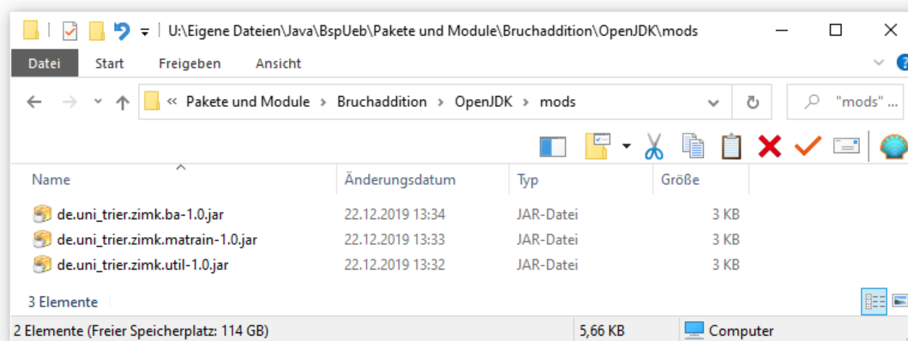
Es folgt die Verpackung des Moduls `de.uni_trier.zimk.matrain`:

```
>jar --create --file mods/de.uni_trier.zimk.matrain-1.0.jar --module-version 1.0
   -C expmods/de.uni_trier.zimk.matrain .
```

Schließlich wird das Moduls `de.uni_trier.zimk.ba` (mit der Startklasse) verpackt:

```
>jar --create --file mods/de.uni_trier.zimk.ba-1.0.jar --module-version 1.0
   --main-class de.uni_trier.zimk.ba.Bruchaddition -C expmods/de.uni_trier.zimk.ba .
```

Das Ergebnis im Explorer-Fenster:



Zum Starten eines in modularen **jar**-Dateien vorliegenden Programms per **java.exe** sind die folgenden Optionen relevant:

Option	Kurzform	Beschreibung
--module-path *pfad*	**-p**	Modulpfad Alle im Modulpfad befindlichen **jar**-Dateien müssen vom modularen Typ sein, wobei auch die sogenannten *automatischen Module* erlaubt sind. Dies sind traditionelle **jar**-Dateien mit einem Dateinamen, der sich als Modulname in das JPMS einfügt (siehe Abschnitt 6.2.9.1).
--module *modul/hauptklasse*	**-m**	Zu startende Klasse Es ist der Modulname anzugeben, also nicht etwa der Name der modularen **jar**-Datei, der zusätzliche Versionsangaben enthalten kann. Wenn beim Erstellen der modularen **jar**-Datei über die Option **--main-class** eine Hauptklasse deklariert wurde, muss diese im Startkommando *nicht* genannt werden.

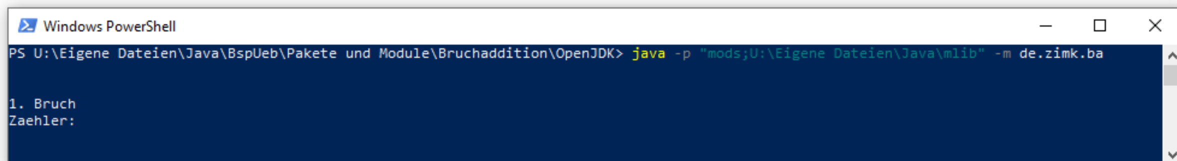
In einem Konsolenfenster, dessen aktueller Ordner u.a. den Wurzelordner **mods** mit den modularen **jar**-Dateien enthält, kann das Beispielprogramm folgendermaßen gestartet werden:

- Langform:
```
>java --module-path mods --module
de.uni_trier.zimk.ba/de.uni_trier.zimk.ba.Bruchaddition
```
- Kurzform (ohne Nennung der Hauptklasse):
```
>java -p mods -m de.uni_trier.zimk.ba
```

Wird z. B. **de.uni_trier.zimk.util-1.0.jar** nach **U:\Eigene Dateien\Java\mlib** verschoben, muss der Modulpfad im Startkommando angepasst werden:

```
>java -p mods;"U:\Eigene Dateien\Java\mlib" -m de.uni_trier.zimk.ba
```

In einem PowerShell-Fenster unter Windows muss ein mehrelementiger Modulpfad durch Anführungszeichen begrenzt werden, damit das Semikolon nicht als Kommandoterminator missverstanden wird, z. B.:



Ist zu einer modularen **jar**-Datei kein Quellcode vorhanden, kann man den Inhalt des Moduldeskriptors **module-info.class** vom JDK-Werkzeug **jar** über die Option **--describe-module** anzeigen lassen, z. B.:

```
>jar --describe-module --file=mods\de.uni_trier.zimk.matrain-1.0.jar
```

Für das modulare Archiv **de.uni_trier.zimk.matrain-1.0.jar**, das wir eben erstellt haben, resultiert die folgende Ausgabe:

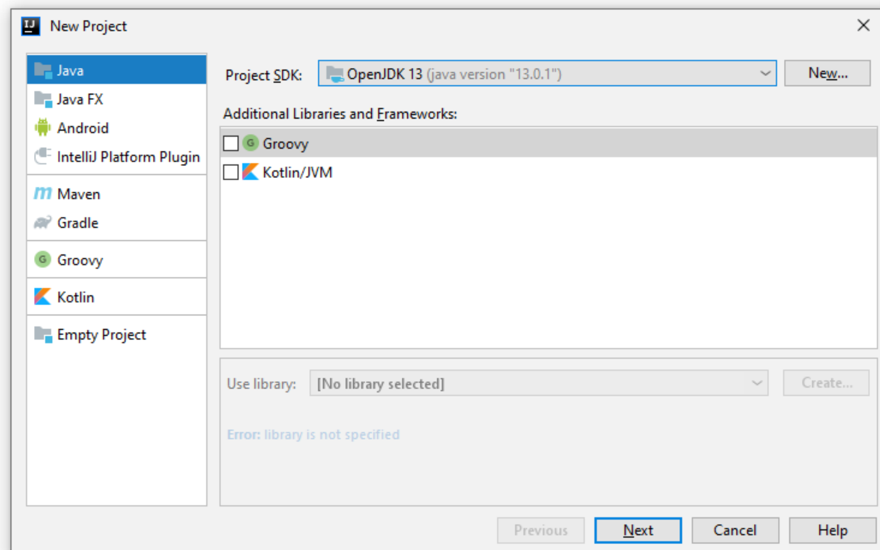
```
de.uni_trier.zimk.matrain@1.0
jar:file:///U:/Eigene%20Dateien/Java/BspUeb/Pakete%20und%20Module/Bruchaddition/OpenJ
DK/mods/de.uni_trier.zimk.matrain-1.0.jar/!module-info.class
exports de.uni_trier.zimk.matrain.br
requires de.uni_trier.zimk.util transitive
requires java.base mandated
```

Eine modulare **jar**-Datei kann auch als herkömmliche **jar**-Datei genutzt werden:

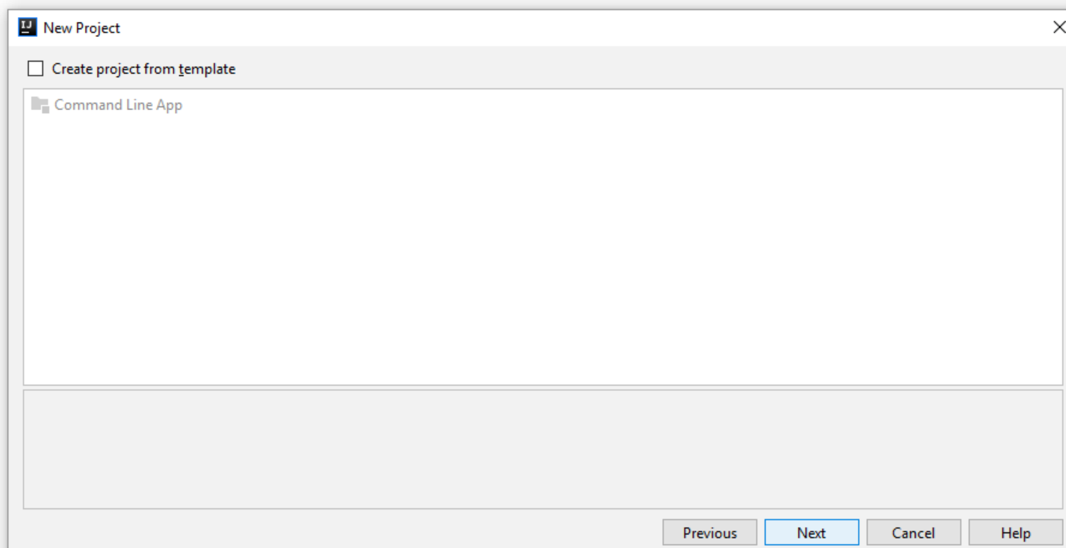
- Sie wird dann über den Klassenpfad angesprochen.
- Der Moduldeskriptor wird ignoriert.

6.2.7 Modulunterstützung in IntelliJ IDEA

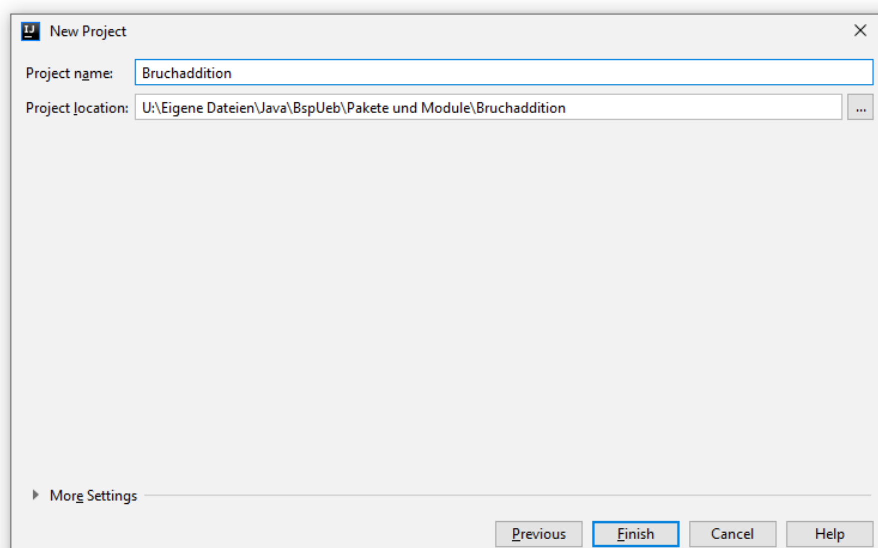
Um das im bisherigen Verlauf von Abschnitt 6.2 als Beispiel verwendete Bruchadditionsprogramm bestehend aus den Modulen `de.uni_trier.zimk.util`, `de.uni_trier.zimk.matrain` und `de.uni_trier.zimk.ba` mit IntelliJ zu erstellen, legen wir ein neues Java-Projekt mit dem OpenJDK 13 als **Project SDK** an:



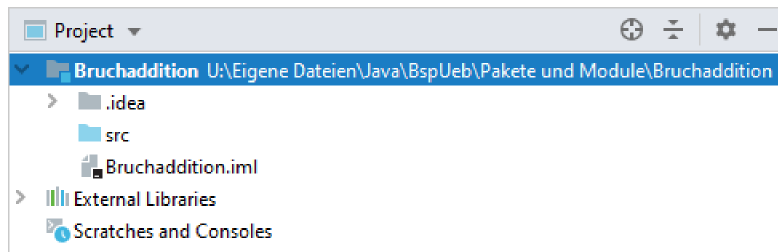
Diesmal verzichten wir auf eine Projekt-Vorlage:



Weil wir auf eine Vorlage verzichtet haben, wird *kein* **Base package** erfragt:



Im **Project**-Fenster resultiert der folgende Ausgangszustand:



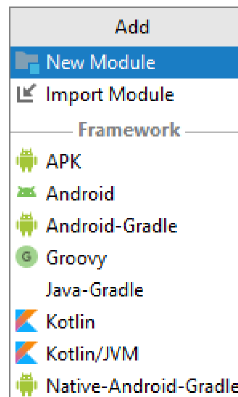
6.2.7.1 *Modul* `de.uni_trier.zimk.util`

Nach

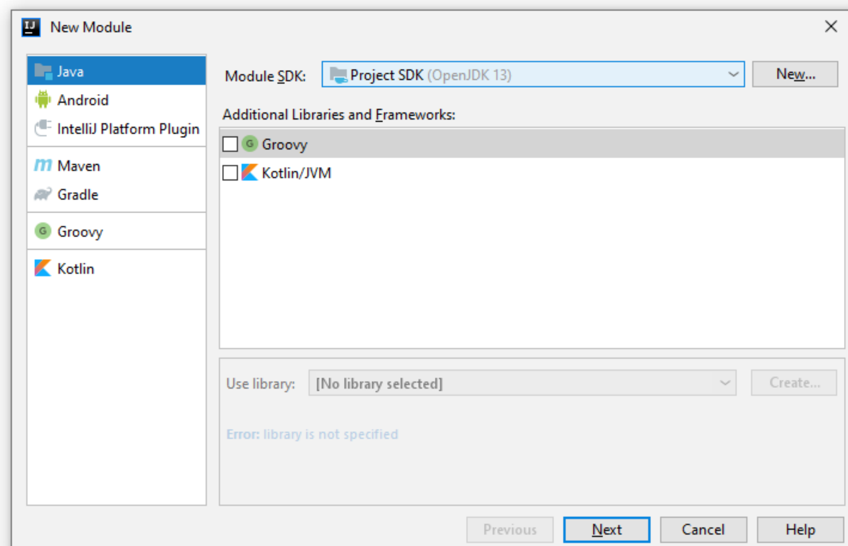
File > Project Structure > Modules

legen wir ein neues Modul im Sinne von IntelliJ an, aus dem letztlich das JPMS - Modul `de.uni_trier.zimk.util` entstehen soll:

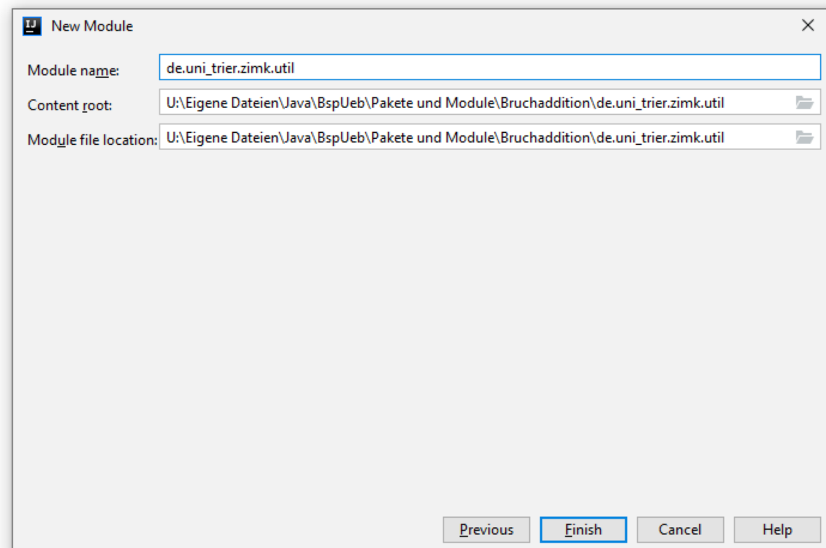
- Wir klicken auf das Pluszeichen über dem Projektnamen `Bruchaddition`
- und wählen aus dem Pop-Up-Menü das Item **New Module**:



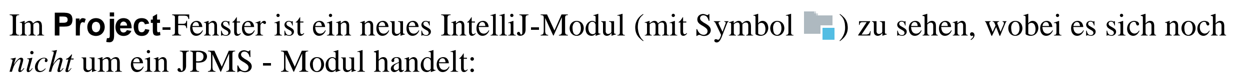
- Weil ein IntelliJ-Modul entsteht, ist ein **Modul SDK** zu wählen:



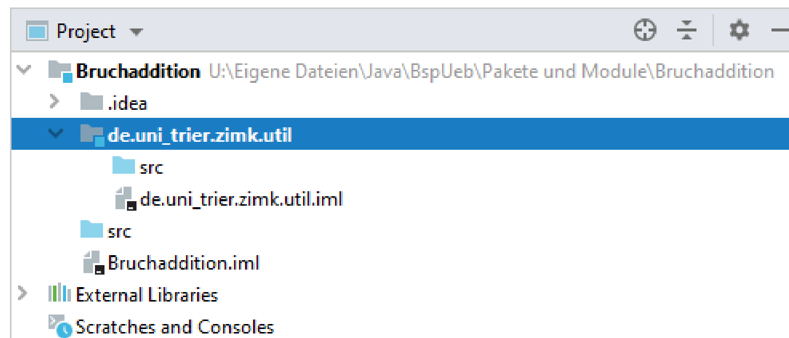
- Wir starten mit dem Modul `de.uni_trier.zimk.util`



und beenden anschließend den Dialog **Project Structure**.



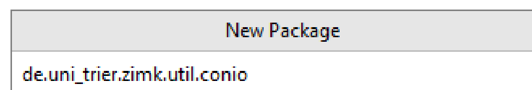
Im **Project**-Fenster ist ein neues IntelliJ-Modul (mit Symbol) zu sehen, wobei es sich noch *nicht* um ein JPMS - Modul handelt:



Um ein Paket im neuen Modul anzulegen, wählen wir aus dem Kontextmenü zum **src**-Ordner im neuen Modul den Befehl

New > Package

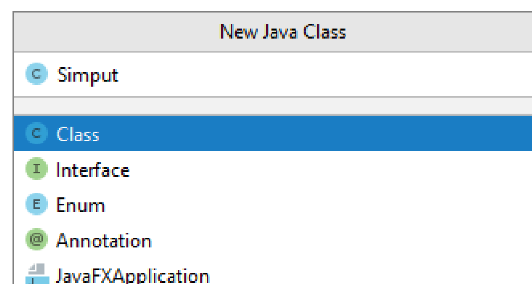
und vergeben im folgenden Fenster den Namen `de.uni_trier.zimk.util.conio`:



Um die Klasse `Simput` im neuen Paket anzulegen, wählen wir aus dem Kontextmenü zum neuen Paket den Befehl

New > Java Class

und tragen im folgenden Fenster den Namen ein:



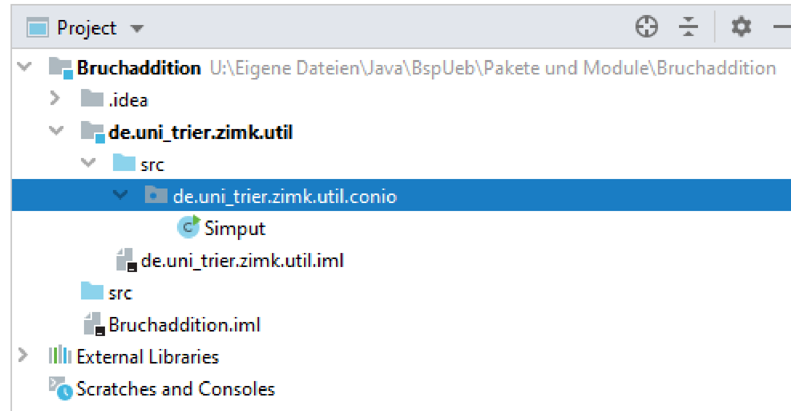
Am Anfang der `Simput`-Quellcodedatei fügt IntelliJ die passende **package**-Deklaration ein:

```
package de.uni_trier.zimk.util.conio;

public class Simput {
}
```

Wir übernehmen den im Abschnitt 6.2.2 ausgehend von

...\BspUeb\Einleitung\Bruchaddition\Konsole\Simput.java

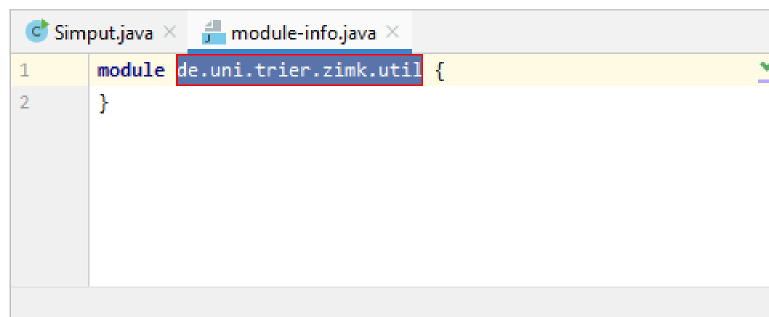
erstellten `Simput`-Quellcode. Neuer Stand im **Project**-Fenster:



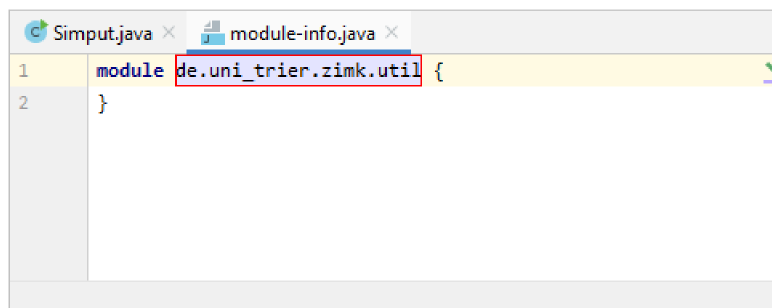
Nun machen wir aus dem IntelliJ-Modul `de.uni_trier.zimk.util` endlich ein Java - Modul, indem wir aus dem Kontextmenü zum **src**-Ordner des Moduls den Befehl

New > module-info.java

wählen. Die Moduldeklarationsdatei wird im Editor geöffnet mit dem Angebot, per Refaktorieren den Modulnamen zu ändern. Weil IntelliJ aus nicht bekannten Gründen den Unterstrich im Namensbestanteil `uni_trier` durch einen Punkt ersetzt hat,



korrigieren wir den Vorschlag



und quittieren mit **OK**. Schließlich ergänzen wir eine **exports**-Deklaration für das Paket `de.uni_trier.zimk.util.conio` (vgl. Abschnitt 6.2.1.3):

```
module de.uni_trier.zimk.util {
    exports de.uni_trier.zimk.util.conio;
}
```

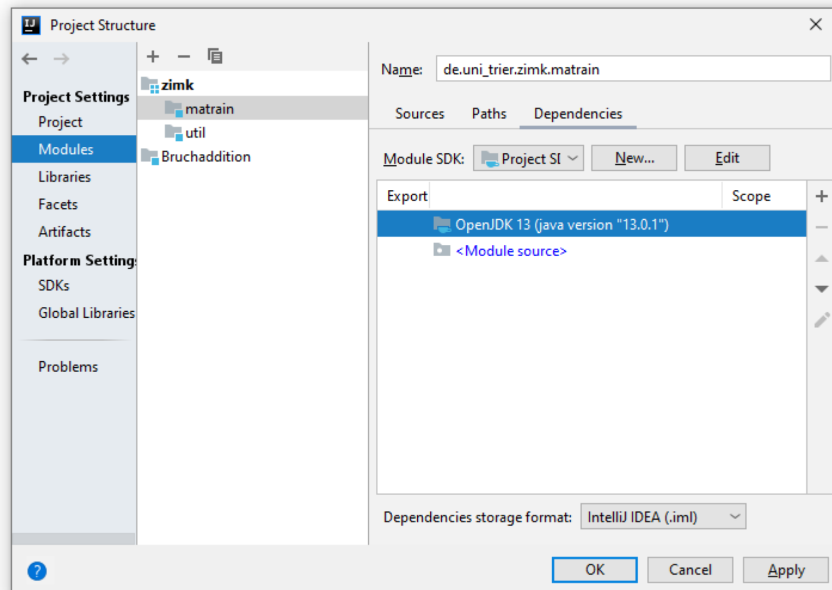
6.2.7.2 Modul `de.uni_trier.zimk.matrain`

Nun legen wir analog zu Abschnitt 6.2.7.1 über

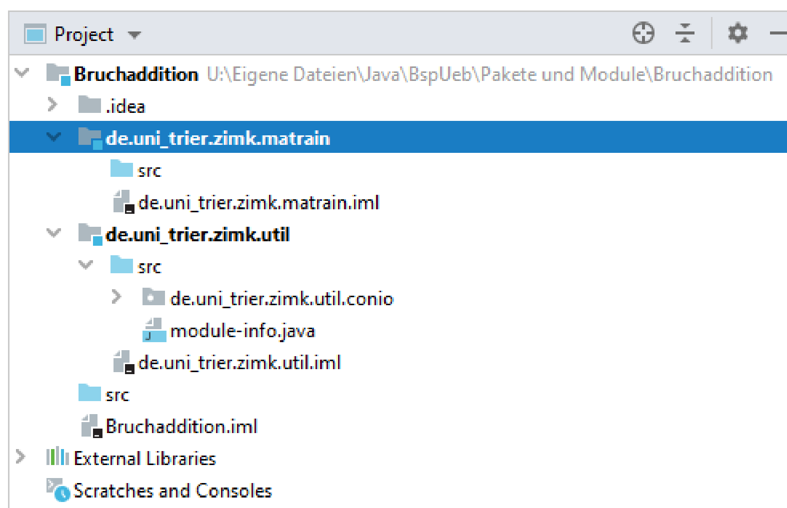
File > Project Structure > Modules

ein neues Modul im Sinne von IntelliJ an, aus dem letztlich das Java - Modul `de.uni_trier.zimk.matrain` entstehen soll.

Im Fenster **Project Structure** zeigt IntelliJ eine Gruppe bestehend aus den beiden Modulen mit `zimk` im Namen an:



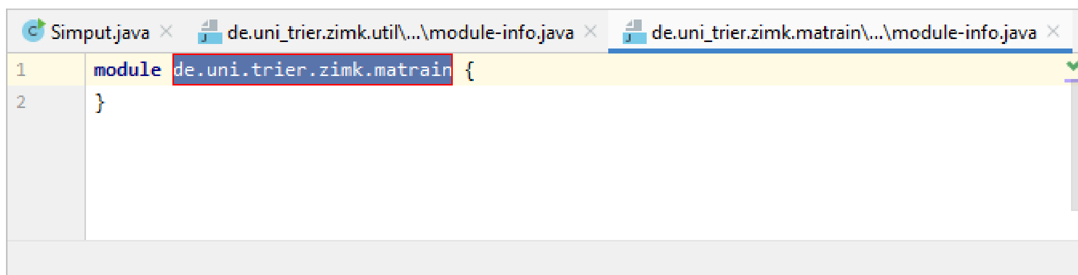
Im **Project**-Fenster resultiert der folgende Zwischenstand:



Wir machen aus dem neuen IntelliJ-Modul ein Java - Modul, indem wir aus dem Kontextmenü zum **src**-Ordner des Moduls den Befehl

New > module-info.java

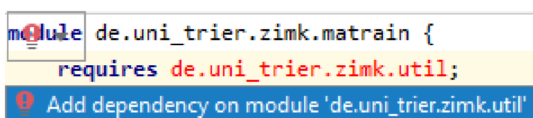
wählen. Die Moduldeklarationsdatei wird im Editor geöffnet mit dem Angebot, per Refaktorieren den Modulnamen zu ändern. Das ist auch nötig, weil IntelliJ erwartungsgemäß erneut den Unterstrich im Namensbestanteil `uni_trier` durch einen Punkt ersetzt hat:



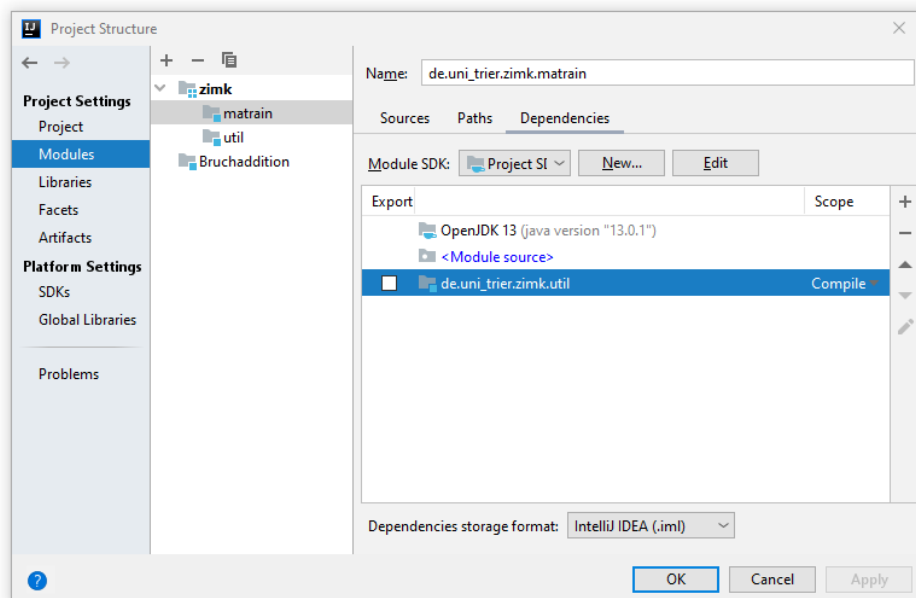
Wir korrigieren den Fehler und ergänzen eine **requires**-Deklaration für das Paket `de.uni_trier.zimk.util`:

```
module de.uni_trier.zimk.matrain {
   requires de.uni_trier.zimk.util;
}
```

Der von IntelliJ durch rote Farbe signalisierte Fehler besteht darin, dass sich das IntelliJ-Modul `de.uni_trier.zimk.util` nicht in der IntelliJ-Abhängigkeitsliste des Moduls `de.uni_trier.zimk.matrain` befindet. Wir lassen das Problem von IntelliJ per QuickFix beheben:



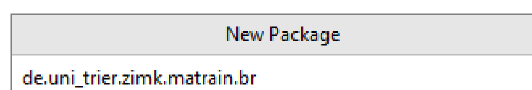
Es ist zu beachten, dass es sich bei der Abhängigkeitsliste um ein Konzept der Entwicklungsumgebung IntelliJ handelt, *nicht* um ein JPMS-Konzept:



Um das Paket `de.uni_trier.zimk.matrain.br` im Modul `de.uni_trier.zimk.matrain` anzulegen, wählen wir aus dem Kontextmenü zum **src**-Ordner des Moduls den Befehl

New > Package

und vergeben im folgenden Fenster den gewünschten Namen:



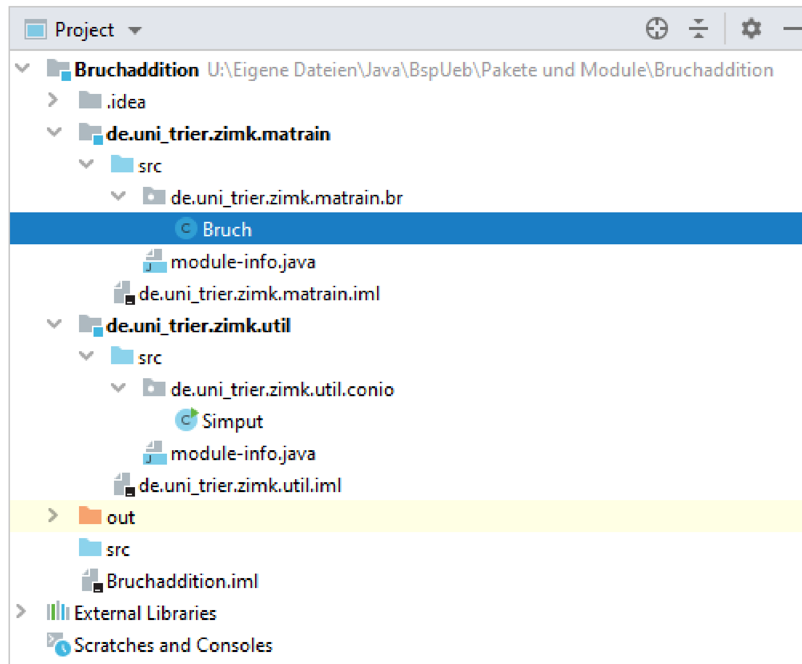
Im Paket `de.uni_trier.zimk.matrain.br` erstellen wir die Klasse `Bruch` und übernehmen den im Abschnitt 6.2.2 ausgehend von

...\BspUeb\Klassen und Objekte\Bruch\B3 (mit Konstruktoren)\Bruch.java

erstellten Quellcode. Dabei gelangt hinter die von IntelliJ an den Anfang des Quellcodes gesetzte **package**-Deklaration eine **import**-Deklaration für die Klasse `Simput` aus dem Paket `de.uni_trier.zimk.util.conio`:

```
import de.uni_trier.zimk.util.conio.Simput;
```

Neuer Zwischenstand im **Project**-Fenster:



Nun ergänzen wir in der Deklarationsdatei zum Modul `de.uni_trier.zimk.matrain` eine **exports**-Deklaration zum Paket `de.uni_trier.zimk.matrain.br`:

```
module de.uni_trier.zimk.matrain {
    requires de.uni_trier.zimk.util;
    exports de.uni_trier.zimk.matrain.br;
}
```

6.2.7.3 Hauptmodul `de.uni_trier.zimk.ba`

Wir legen analog zu Abschnitt 6.2.7.1 über

File > Project Structure > Modules

ein neues Modul im Sinne von IntelliJ an, aus dem letztlich das Java Hauptmodul `de.uni_trier.zimk.ba` der Anwendung entstehen soll.

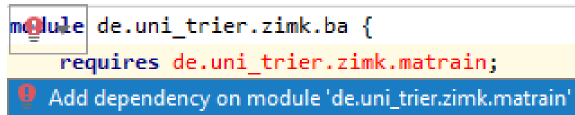
Wir machen aus dem neuen IntelliJ-Modul ein Java - Modul, indem wir aus dem Kontextmenü zum **src**-Ordner des Moduls den Befehl

New > module-info.java

wählen. Die Moduldeklarationsdatei wird im Editor geöffnet mit dem Angebot, per Refaktorieren den Modulnamen zu ändern. Das ist auch nötig, weil IntelliJ erneut den Unterstrich im Namensbestanteil `uni_trier` durch einen Punkt ersetzt hat. Wir korrigieren den Fehler und ergänzen eine **requires**-Deklaration für das Paket `de.uni_trier.zimk.matrain`:

```
module de.uni_trier.zimk.ba {
    requires de.uni_trier.zimk.matrain;
}
```

Der von IntelliJ durch rote Farbe signalisierte Fehler besteht darin, dass sich das IntelliJ-Modul `de.uni_trier.zimk.matrain` nicht in der IntelliJ-Abhängigkeitsliste des Moduls `de.uni_trier.zimk.ba` befindet. Wir lassen das Problem von IntelliJ per QuickFix beheben:



Analog verfahren wir mit einer **requires**-Deklaration für das Paket `de.uni_trier.zimk.util`. Nachdem die IntelliJ-Abhängigkeitsliste des Moduls `de.uni_trier.zimk.ba` komplettiert ist, können wir die Moduldeklarationsdateien optimieren:

- In der Deklaration zum Modul `de.uni_trier.zimk.matrain` formulieren wir eine *transitive* Abhängigkeit vom Modul `de.uni_trier.zimk.util`:

```
module de.uni_trier.zimk.matrain {
   requires transitive de.uni_trier.zimk.util;
   exports de.uni_trier.zimk.matrain.br;
}
```

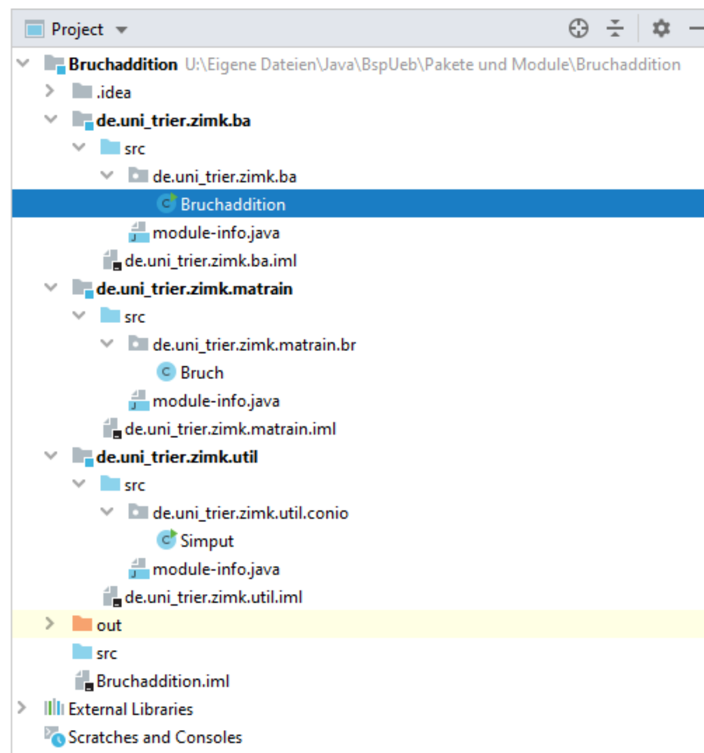
- Daraufhin übernimmt das von `de.uni_trier.zimk.matrain` abhängige Hauptmodul `de.uni_trier.zimk.ba` automatisch die Abhängigkeit vom Modul `de.uni_trier.zimk.util`, sodass seine Moduldeklaration vereinfacht werden kann:

```
module de.uni_trier.zimk.ba {
   requires de.uni_trier.zimk.matrain;
}
```

Wir legen im Modul `de.uni_trier.zimk.ba` ein gleichnamiges Paket an (über **New > Package** aus dem Kontextmenü zum **src**-Ordner des Moduls). Im Paket `de.uni_trier.zimk.ba` erstellen wir die folgende, aus Abschnitt 6.2.2 bekannte Klasse `Bruchaddition`:

```
package de.uni_trier.zimk.ba;

import de.uni_trier.zimk.matrain.br.Bruch;
import de.uni_trier.zimk.util.conio.Simput;

class Bruchaddition {
   public static void main(String[] args) {
      Bruch b1 = new Bruch(), b2 = new Bruch();

      System.out.println("1. Bruch");
      b1.frage();
      b1.kuerze();
      b1.zeige();

      System.out.println("\n2. Bruch");
      b2.frage();
      b2.kuerze();
      b2.zeige();

      System.out.println("\nSumme");
      b1.addiere(b2);
      b1.zeige();

      System.out.print("\nIhre Zufriedenheit mit der Software (1-5): ");
      int zuf = Simput.gint();
      System.out.println("Verstanden: " + zuf);
   }
}
```

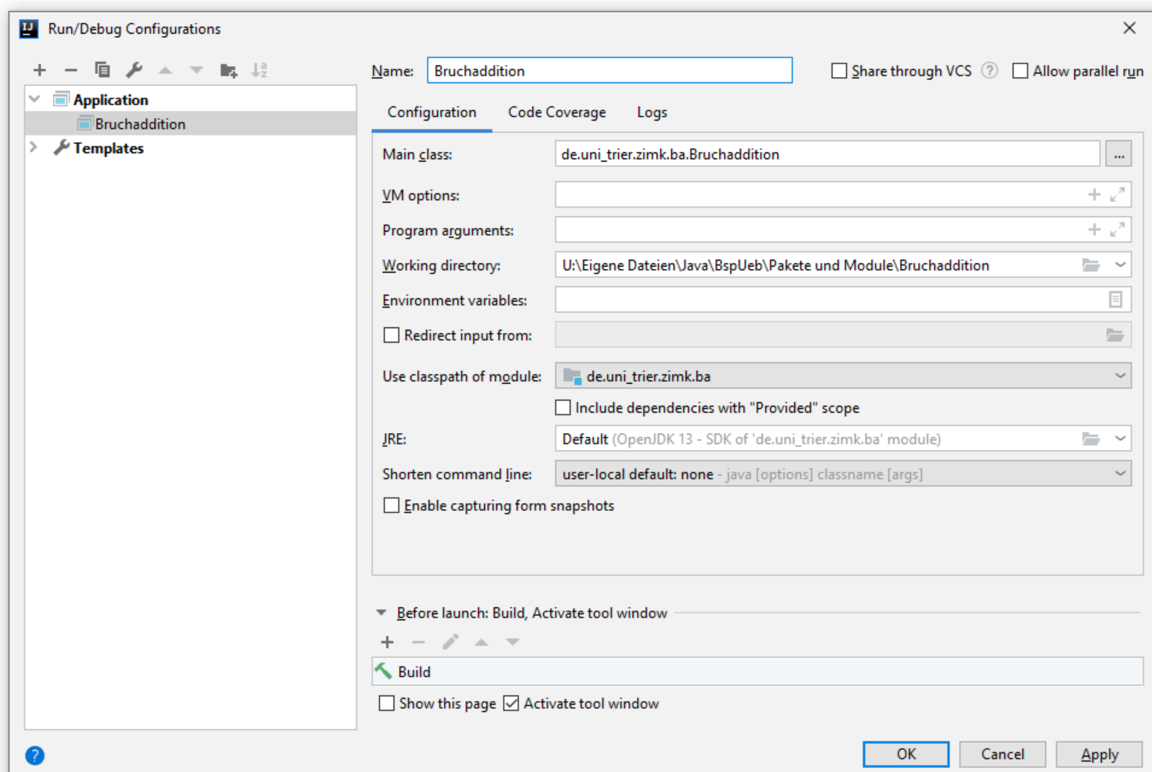
Im **Project**-Fenster zeigt sich nun das folgende Bild:



Wir ergänzen über

Add Configuration > + > Application

noch eine **Run**-Konfiguration



und können das Programm endlich starten:

```
Run: Bruchaddition
"C:\Program Files\Java\OpenJDK-13\bin\java.exe" "-javaagent:C:\Program Files\
1. Bruch
Zähler: 2
Nenner: 7
    2
  -----
    7


2. Bruch
Zähler: 7
Nenner: 9
    7
  -----
    9


Summe
    67
  -----
    63


Ihre Zufriedenheit mit der Software (1-5): 1
Verstanden: 1

Process finished with exit code 0
```

Das komplette IntelliJ-Projekt befindet sich im Ordner:

...\BspUeb\Pakete und Module\Bruchaddition\IntelliJ

6.2.8 Eigenständige Anwendungen mit maßgeschneiderter Laufzeitumgebung

Ab Java 9 gesellt sich als neue Phase der Anwendungserstellung zur Übersetzungsphase (basierend auf dem Compiler **javac.exe**) und der Ausführungsphase (basierend auf der JVM) eine optionale **Bindungs- bzw. Link-Phase** (basierend auf dem neuen JDK-Werkzeug **jlink**). Dabei wird ein Programm mit einer **angepassten modularen Laufzeitumgebung** (engl. *custom modular runtime image*) bestehend aus den tatsächlich benötigten API-Modulen verbunden. Der Kunde erhält ein selbständig ausführbares Programm, das auf seinem Rechner keine JVM voraussetzt. Dank der maßgeschneiderten (und somit reduzierten) modularen Laufzeitumgebung hält sich der Lieferumfang in Grenzen.

Außerdem besteht für das ausgelieferte Programm eine reduzierte Gefahr, von zukünftig entdeckten Sicherheitslücken in der Java- Laufzeitumgebung betroffen zu sein. Wenn es doch passiert, muss allerdings der Programmanbieter ein Update ausliefern. Während für die von Oracle oder von einem OpenJDK-Distributor stammende komplette Java-Laufzeitumgebung auf einem Kundenrechner in der Regel der jeweilige Anbieter mehr oder weniger zeitnah und kundenfreundlich Updates zur Verfügung stellt, ist für eine Anwendung mit eigenständiger Laufzeitumgebung der Programmanbieter verantwortlich. Auf diese Verantwortungsübertragung weist die Firma Oracle in der Online-Dokumentation zum **jlink**-Werkzeug nachdrücklich hin:[1]

> Developers are responsible for updating their custom runtime images.

Wir erproben die Erstellung einer eigenständigen Distribution anhand des Bruchadditionsbeispiels bestehend aus den Modulen `de.uni_trier.zimk.ba`, `de.uni_trier.zimk.matrain` und `de.uni_trier.zimk.util`, die sich als modulare **jar**-Dateien im Ordner

U:\Eigene Dateien\Java\BspUeb\Pakete und Module\Bruchaddition\OpenJDK\mods

befinden. Das folgende **jlink**-Kommando zum Erstellen des eigenständigen Programms mit angepasster Laufzeitumgebung

```
>jlink -p "C:\Program Files\Java\OpenJDK-13\jmods;mods"
   --add-modules de.uni_trier.zimk.ba --output distrib
```

---

[1] https://docs.oracle.com/en/java/javase/13/docs/specs/man/jlink.html

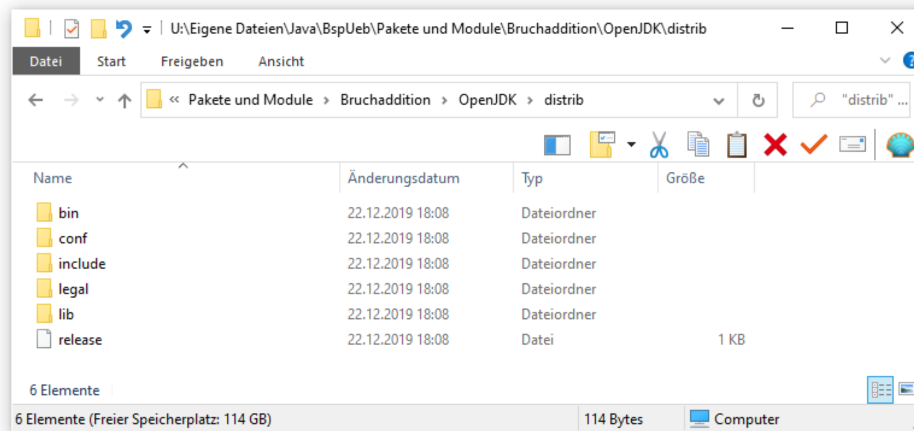
setzt voraus, ...

- dass die OpenJDK-Version 13 in **C:\Program Files\Java\OpenJDK-13** installiert ist,
- dass sich im aktuellen Verzeichnis des verwendeten Konsolenfensters der Ordner **mods** mit den drei modularen **jar**-Dateien des Bruchadditionsbeispiels befindet,
- dass der **bin**-Unterordner der OpenJDK 13 - Installation im Windows-Pfad für ausführbare Programme eingetragen ist.

Das Kommando verwendet drei Optionen:

- **-p**
  Der Modulpfad enthält die API-Module im OpenJDK-Installationsordner und die anwendungsspezifischen Module im Unterordner **mods**.
- **--add-modules**
  Damit durch Auflösung der Abhängigkeiten der Graph mit den benötigten Modulen erstellt werden kann, ist über die Option **--add-modules** ein Wurzelmodul (engl.: *root module*) zu benennen.
- **--output**
  Das selbständig ausführbare Programm soll im Unterordner **distrib** erstellt werden.

Der resultierende Ordner mit der angepassten modularen Laufzeitumgebung ist ca. 40 MB groß:



Durch das folgende Kommando mit Platzspartipps von Thorsten Horn[1]

```
>jlink -p "C:\Program Files\Java\OpenJDK-13\jmods;mods"
   --add-modules de.uni_trier.zimk.ba --output distrib
   --exclude-files *.diz --strip-debug --compress=2
```

reduziert sich der Platzbedarf auf ca. 25 MB. Ein kompletter OpenJDK-Ordner auf demselben Versionsstand ist ca. 300 MB groß.

Die angepasste modulare Laufzeitumgebung enthält auch den Starter **java.exe**, der als Argument nur das Startmodul (aber keinen Modulpfad) benötigt:

```
>distrib\bin\java -m de.uni_trier.zimk.ba
```

6.2.9 Kompatibilität und Migration

Java kann seit der Version 9 beim Übersetzen und beim Ausführen von Programmen sowohl mit dem Klassenpfad als auch mit dem Modulpfad umgehen, sodass kein Zwang für eine schnelle Umstellung bestehender Anwendungen auf das JPMS besteht. Über die anschließenden Erläuterungen hinausgehende Hinweise zur Umstellung von vorhandenen Anwendungen auf das JPMS finden sich

[1] http://www.torsten-horn.de/techdocs/Jigsaw.html

im *Java Platform, Standard Edition Oracle JDK Migration Guide* (Oracle 2019b)[1] und bei Reinhold (2016).

6.2.9.1 Automatische Module

Noch wichtiger als die im Abschnitt 6.2.6 erwähnte Option, eine modulare **jar**-Datei in den traditionellen Klassenpfad aufzunehmen, ist die „umgekehrte“ Möglichkeit, eine traditionelle **jar**-Datei (ohne Moduldeskriptor!) in den Modulpfad aufzunehmen, also ohne nennenswerten Aufwand im JPMS zu nutzen. Dabei entsteht ein sogenanntes *automatisches Modul*.

Das JPMS generiert den Modulnamen aus dem **jar**-Dateinamen, wobei die Namenserweiterung und etwaige Versionsangaben entfernt werden, sodass z. B. aus dem Dateinamen **mylib-3.01.jar** der Modulname `mylib` entsteht. Während ein traditioneller **jar**-Dateiname im Java-Quellcode nicht auftaucht, kann der Name eines automatischen Moduls potentiell in den **requires**-Deklarationen expliziter Module verwendet werden.

Eigenschaften eines automatischen Moduls:

- Es ist ein *benanntes* Modul und unterscheidet sich diesbezüglich vom dem im Abschnitt 6.2.9.2 zu beschreibenden *unbenannten* Modul, das alle per Klassenpfad zugänglichen Typen enthält. Ein automatisches Modul gehört aber nicht zu den sogenannten *expliziten* Modulen, die einen Moduldeskriptor besitzen.
- Alle Pakete in einem automatischen Modul gelten als exportiert. Sie können also von jedem anderen Modul mit bestehender Abhängigkeitsbeziehung genutzt werden. Diese Abhängigkeitsbeziehung besteht implizit bei anderen automatischen Modulen und beim unbenannten Modul. Bei einem expliziten Modul muss sie per **requires**-Deklaration erklärt werden. Außerdem sind alle Pakete eines automatischen Moduls für die Laufzeit-Reflexion geöffnet.
- Ein automatisches Modul besitzt implizit eine Abhängigkeitsbeziehung zu allen anderen benannten Modulen (explizit oder automatisch).
- Außerdem besitzt ein automatisches Modul im Gegensatz zu den expliziten Modulen eine Leseberechtigung für das unbenannte Modul.

Vom Package Splitting - Verbot (siehe Abschnitt 6.2.4) sind auch automatische Module betroffen, sodass zwei **jar**-Dateien mit identisch benannten Paketen nicht gleichzeitig aus dem chaotischen Klassenpfad als automatische Module in den Modulpfad übernommen werden können. In dieser Lage hilft es, dass automatische Module neben benannten Modulen auch das unbenannte Modul lesen dürfen.

6.2.9.2 Das unbenannte Modul

Auf dem Modulpfad befindliche traditionelle **jar**-Dateien werden zu automatischen Modulen und können im Unterschied zu den expliziten Modulen (mit Moduldeskriptor) alle Typen auf dem traditionellen Klassenpfad sehen. Wird durch ein automatisches Modul ein Typ angefordert, der in keinem benannten Modul zu finden ist, dann wird dieser Typ auf dem Klassenpfad gesucht. Die dortigen Typen werden als Mitglieder des sogenannten *unbenannten Moduls* betrachtet, das eine ähnliche Rolle spielt wie das unbenannte Standardpaket.

Eigenschaften des unbenannten Moduls:

---

[1] https://docs.oracle.com/en/java/javase/11/migrate/index.html

- Das unbenannte Modul besitzt eine implizite Abhängigkeitsbeziehung zu allen Modulen auf dem Modulpfad, also zu jedem expliziten und zu jedem automatischen Modul, wobei auch die API-Module einbezogen sind. Es kann also auf exportierte Pakete in benannten Modulen zugreifen.
- Es exportiert sämtliche Pakete.
- Explizite Module können die Pakete des unbenannten Moduls allerdings *nicht* sehen. Automatische Module können das unbekannte Modul hingegen nutzen.
- Die Typen im unbenannten Modul können sich gegenseitig uneingeschränkt sehen.
- Enthalten ein benanntes und das unbenannte Modul ein gleichnamiges Paket, dann wird das Paket im unbenannten Modul ignoriert.
- Zum unbenannten Modul gehört auch das Standardpaket mit den Klassen, die keinem Paket zugeordnet wurden.
- Sollte sich ein explizites Modul auf den Klassenpfad verirren, wird sein Moduldeskriptor ignoriert.

6.2.9.3 Notlösung

Benötigt ein Programm Zugriffe auf interne (nicht exportierte) API-Pakete, dann ist die Übersetzung mit dem Compiler einer modularen Java-Version (ab 9) trotzdem möglich mit Hilfe der neuen Option **--add-exports**. Damit lassen sich interne API-Pakete für ein Modul zugänglich machen (siehe Oracle 2019b).

6.2.9.4 Moduldeklaration zu vorhandenem Quellcode erstellen

Ist zu einer Menge zusammengehöriger Pakete der Quellcode vorhanden, bietet sich die Erstellung eines expliziten Moduls an, wobei sich das JDK-Werkzeug **jdeps** nützlich macht. Als vertrautes, wenn auch nicht sonderlich realistisches Beispiel verwenden wir die traditionelle Archivdatei Archiv **`demarc.jar`** (vgl. Abschnitt 6.1.3.2) mit den Paketen `demopack` und `demopack.sub1`. Wir schmuggeln eine Abhängigkeit vom API-Paket **javax.swing** in eine Klasse des Pakets `demopack` ein (durch Verwendung der Klasse **JOptionPane**):

```
package demopack;

import demopack.sub1.*;
import javax.swing.*;

public class Main {
    public static void main(String[] args) {
        A a1 = new A(), a2 = new A();
         . . .
        JOptionPane.showMessageDialog(null, "jdeps-Demo");
    }
}
```

Das Ergebnis (in der traditionellen Archivdatei **demarcjop.jar**) lassen wir durch den folgenden **jdeps**-Aufruf auf Modul-Abhängigkeiten untersuchen:

```
>jdeps -s demarcjop.jar
```

Erwartungsgemäß ist **demarcjop.jar** (wie jede andere Java-Anwendung bzw. -Bibliothek) vom API-Modul **java.base** abhängig. Außerdem besteht eine Abhängigkeit vom API-Modul **java.desktop**, das u.a. das Paket **javax.swing** enthält:

```
demarcjop.jar -> java.base
demarcjop.jar -> java.desktop
```

Über die Option **--generate-module-info** kann man **jdeps** auffordern, die zur Erstellung eines expliziten Moduls benötigte Deklarationsdatei **module-ionfo.java** zu erstellen, z. B.:

```
>jdeps --generate-module-info . demarcjop.jar
```

Während **jdeps** genau die benötigten Abhängigkeiten in **requires**-Deklarationen umsetzt, kann das Werkzeug natürlich nicht zwischen zu exportierenden und internen Paketen unterscheiden. Folglich werden *alle* Pakete exportiert, und die Liste der **exports**-Deklarationen muss eventuell modifiziert werden:

```
module demarcjop {
    requires java.desktop;
    exports demopack;
    exports demopack.sub1;
}
```

Für den praxisgerechten Einsatz von **jdeps** sind weitere Informationen erforderlich, die z. B. das Oracle Help Center anbietet.[1]

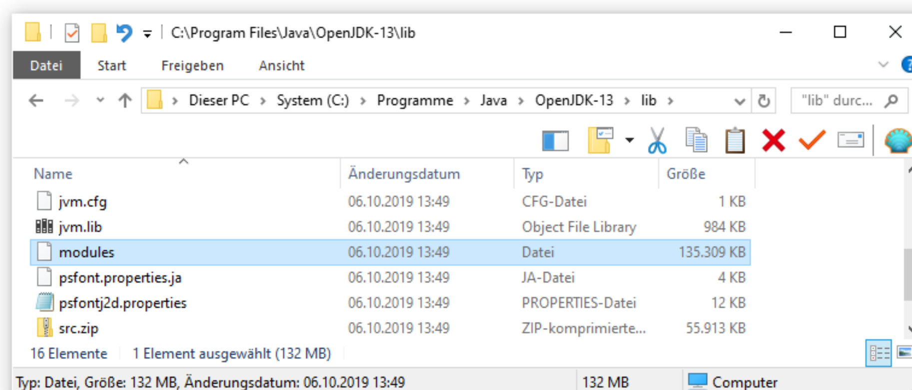
6.2.10 Das modulare API der Java Standard Edition

Wie Sie längst wissen, gehören zur Java-Plattform zahlreiche Pakete, die Klassen und Schnittstellen für wichtige Aufgaben der Programmentwicklung (z. B. Zeichenkettenverarbeitung, Netzwerkverbindungen, Datenbankzugriffe) enthalten. Die Zusammenfassung dieser Pakete bezeichnen wir als *Java-Standardbibliothek* oder *Java-API*. Allerdings kann man eigentlich nicht von *dem* Java-API sprechen, denn neben der **Java Standard Edition (JSE)**, auf die wir uns im Kurs beschränken, existieren noch die **Java Enterprise Edition (JEE)** mit großer Bedeutung für die Entwicklung von Server-Anwendungen[2] und die **Java Micro Edition (JME)** mit etwas unsicheren Zukunftsaussichten (siehe Abschnitt 1.3.4).

Seit Java 9 ist die Standardbibliothek modular aufgebaut. Wie mit dem Kommando

```
java --list-modules
```

zu ermitteln ist, besteht das API der OpenJDK-Version 13 aus 71 Modulen. Man bezeichnet sie als *Platform Explicit Modules* zur Abgrenzung von den im Abschnitt 6.2 beschriebenen *Application Explicit Modules* mit Programmen und Bibliotheken. Während letztere als explodierte Module oder modulare **jar**-Dateien vorliegen, befinden sich die Plattform-Module gemeinsam in einer speziell formatierten Datei namens **modules** im **lib**-Unterordner einer JDK-Installation, z. B.:



---

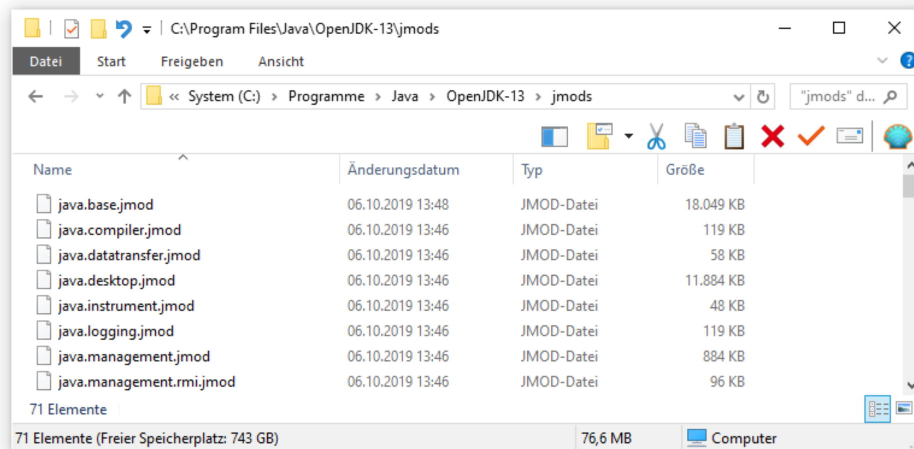
[1] https://docs.oracle.com/en/java/javase/13/docs/specs/man/jdeps.html

[2] Seite 2018 wird die JEE mit dem neuen Namen **Jakarta EE** unter dem Dach der Open-Source-Organisation **Eclipse Foundation** weitergeführt. Davon wird vermutlich die Dynamik der Entwicklung profitieren. Zu Details siehe z. B.: https://www.heise.de/developer/meldung/Jakarta-EE-Eclipse-Foundation-uebernimmt-die-Verantwortung-fuer-Enterprise-Java-4030557.html

Das Modul **java.base** enthält die wichtigsten Pakete (z. B. **java.lang**, **java.math** und **java.io**) und wird von jeder Java-Software benötigt. Es hat im JPMS eine herausgehobene Stellung:

- Jedes andere Modul hängt implizit von **java.base** ab.
- **java.base** hängt von keinem anderen Modul ab.

In einem JDK - Installationsordner sind die Platform-Module zweimal vorhanden. Sie befinden sich nicht nur in der eben beschriebenen Datei **modules** im Unterordner **lib**, sondern auch im Unterordner **jmods**, den man z. B. als Modulpfad-Bestandteil für die Erstellung einer individuellen Laufzeitumgebung benötigt (vgl. Abschnitt 6.2.8). Diesmal stecken die Plattform-Module in individuellen Dateien mit der Namenserweiterung **jmod**, z. B.:



Neben den JRE-Modulen mit dem initialen Namensbestandteil **java**, befinden sich im Ordner **jmods** auch JDK-spezifische Module, die am initialen Namensbestandteil **jdk** zu erkennen sind.

Mit Hilfe des JDK-Werkzeugs **jmod** kann man sich über eine **jmod**-Datei informieren, z. B.:

```
C:\Program Files\Java\OpenJDK-13\jmods>jmod describe java.base.jmod
```

Man kann die **jmod**-Dateien in der Übersetzungs- und in der Bindungs- bzw. Link-Phase verwenden, aber nicht zur Laufzeit:[1]

> JMOD files can be used at compile time and link time, but not at run time.

6.2.11 Modul-Taxonomie

Weil im Verlauf von Abschnitt 6.2 von zahlreichen Modulsorten die Rede war, sollen diese noch einmal im Überblick präsentiert werden:

- Explizites Modul
  Dies ist ein normales JPMS-Modul und besitzt einen Moduldeskriptor. Man unterscheidet:
  - *Application Explicit Module*
    Es stammt von einem Programmentwickler und wird entweder als modulare **jar**-Datei oder als Verzeichnisbaum (explodiertes Modul) ausgeliefert.
  - *Platform Explicit Module*
    Es gehört zur Java-Standardbibliothek. Ein besonders wichtiges Plattform-Modul ist **java.base**, von dem alle anderen Module implizit (ohne **requires**-Deklaration im Moduldeskriptor) abhängig sind.

  Explizite Module können die Pakete des unbenannten Moduls *nicht* sehen.

[1] http://openjdk.java.net/jeps/261#Packaging:-JMOD-files

- Offenes Modul (siehe Abschnitt 6.2.1.5)
  Wird die Moduldeklaration eines expliziten Moduls mit dem Schlüsselwort **open** eingeleitet, öffnet es *alle* Pakete für die Laufzeit-Reflexion und wird zum offenen Modul.
- Automatisches Modul (siehe Abschnitt 6.2.9.1)
  Wird eine traditionelle **jar**-Datei in den Modulpfad aufgenommen, entsteht ein benanntes Modul mit einem Modulnamen, der aus dem **jar**-Dateinamen abgeleitet wird. Ein automatisches Modul ...
  - besitzt implizit eine Abhängigkeitsbeziehung zu allen anderen benannten Modulen (explizit oder automatisch),
  - exportiert alle Pakete
  - und kann - im Unterschied zu den expliziten Modulen (mit Moduldeskriptor) - auch auf das unbenannte Modul zugreifen.
- Benanntes Modul
  Zu den benannten Modulen gehören die expliziten Module und die automatischen Module.
- Unbenanntes Modul (siehe Abschnitt 6.2.9.2)
  Es sammelt alle per Klassenpfad zugänglichen Typen und besitzt eine implizite Abhängigkeitsbeziehung zu allen Modulen auf dem Modulpfad, also zu jedem expliziten und zu jedem automatischen Modul, wobei auch die API-Module einbezogen sind. Es exportiert alle benannten Pakete, die allerdings nur in automatischen Modulen sichtbar sind.

6.3 Zugriffsschutz

Nach der Beschäftigung mit Paketen und Modulen lässt sich endlich präzise erläutern, wie in Java die Zugriffsrechte für Typen, Felder, Methoden, Konstruktoren und andere Member (z. B. innere Klassen) geregelt sind. Dabei wird vorausgesetzt, dass für den aktuell angemeldeten Entwickler bzw. Benutzer auf der Ebene des Betriebs- bzw. Dateisystems Leserechte für die beteiligten Dateien bestehen.

6.3.1 Sichtbarkeit von Top-Level - Typen

Bisher haben wir uns überwiegend mit *Top-Level - Typen* beschäftigt, die *nicht* innerhalb des Quellcodes anderer Typen definiert werden. In diesem Abschnitt geht es um den Zugriffsschutz für solche Typen.[1]

Bei den Top-Level - Typen ist nur der Zugriffsmodifikator **public** erlaubt, sodass zwei Schutzstufen möglich sind:

- Ohne Zugriffsmodifikator ist der Typ nur innerhalb des eigenen Pakets verwendbar.
- Durch den Zugriffsmodifikator **public** wird die Verwendung in allen *berechtigten Modulen* erlaubt, z. B.:

```
package demopack;
public class A {
  . . .
}
```

Für eine Klasse in einem Paket des Moduls `moda` ist ein als **public** dekorierter Typ aus dem Paket `modb.pack` im Modul `modb` genau dann sichtbar, ...

[1] Die *Mitgliedsklassen* (synonym: *eingeschachtelten Klassen*), die innerhalb einer Top-Level - Klasse, aber außerhalb von Methoden definiert werden (siehe Abschnitt 4.8.1), sind beim Zugriffsschutz wie andere Klassenmitglieder (z. B. Felder und Methoden) zu behandeln (siehe Abschnitt 6.3.2). Bei den innerhalb von Methoden definierten *lokalen Klassen* (siehe Abschnitt 4.8.2) und den *anonymen Klassen* sind Zugriffsmodifikatoren irrelevant und verboten.

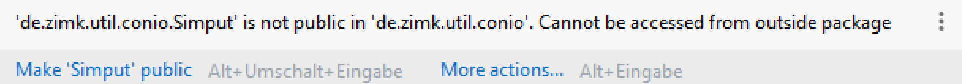
- wenn sich das Modul `moda` in seiner Moduldeklarationsdatei **modul-info.java** per **requires**-Deklaration als abhängig von Modul `modb` erklärt hat,
- und wenn außerdem das Modul `modb` in seiner Moduldeklarationsdatei das Paket `modb.pack` per **exports**-Deklaration freigegeben hat (generell oder speziell für das Modul `moda`).

Ob die Member (z. B. Methoden und Felder) eines sichtbaren Typs verwendbar sind, hängt von deren speziellem Zugriffsschutz ab (siehe Abschnitt 6.3.2).

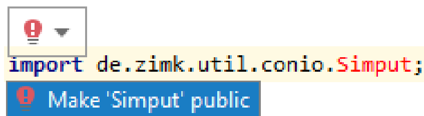
Im Rahmen der im Kurs bisher nicht behandelten Laufzeit-Reflexion kann auf einen (nicht unbedingt als **public** deklarierten Typ) zugegriffen werden, wenn für sein Paket in der Moduldeklaration per **opens**-Deklaration die Reflexion erlaubt wurde, oder wenn das gesamte Modul mit dem **open**-Modifikator dekoriert wurde.

Das im Abschnitt 6.2.7 mit IntelliJ erstellte Bruchadditionsbeispiel enthält die Klasse `Simput` im Paket `de.uni_trier.zimk.util.conio`, das sich im Modul `de.uni_trier.zimk.util` befindet. Die Klasse `Simput` soll von der Klasse `Bruch` im Paket `de.uni_trier.zimk.matrain.br` verwendet werden, das sich im Modul `de.uni_trier.zimk.matrain` befindet. Abhängigkeits- und Exporterklärung sind in den Moduldeklarationen vorhanden. Wenn nun die Klasse `Simput` *nicht* als **public** deklariert ist, beschwert sich IntelliJ:

```
package de.zimk.matrain.br;
import de.zimk.util.conio.Simput;
```

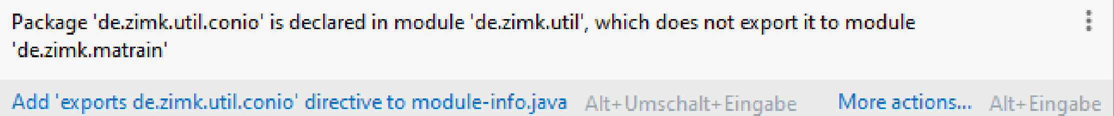


Es wird eine sinnvolle QuickFix-Reparatur empfohlen:



Auch eine fehlende Exportdeklaration im Modul `de.uni_trier.zimk.util` wird von IntelliJ erkannt und unmissverständlich als Problemursache benannt:

```
import de.zimk.util.conio.Simput;
```



Gemeinsame Bestandteile im Paketpfad haben keine Relevanz für die wechselseitigen Zugriffsrechte von Klassen. Folglich haben z. B. die Klassen im Paket `demopack.sub1` für Klassen im Paket `demopack` dieselben Rechte wie Klassen aus beliebigen anderen Paketen.

Bei aufmerksamer Lektüre der (z. B. im Internet) zahlreich vorhandenen Java-Beschreibungen stellt man fest, dass bei Hauptklassen neben der Startmethode **main()** oft auch die Klasse selbst als **public** definiert wird, z. B.:

```
public class Hallo {
    public static void main(String[] args) {
        System.out.println("Hallo allerseits!");
    }
}
```

Diese Praxis erscheint plausibel, jedoch verlangt die JVM lediglich bei der Startmethode den Modifikator **public**. Bei der Wahl einer Regel für dieses Manuskript habe ich mich an den Java-Urhebern orientiert: Gosling et at. (2019) lassen bei Hauptklassen den Modifikator **public** systematisch weg.

6.3.2 Sichtbarkeit von Typmitgliedern

Zunächst einmal soll in Erinnerung gerufen werden, dass in Java der Zugriffsschutz nicht objekt-, sondern **klassenbezogen** organisiert ist (Goll et al. 2000, S. 322). Ist z. B. eine Klasse `A` als **public** definiert, können Objekte dieses Typs durch beliebige andere Klassen in berechtigten Modulen genutzt werden. Typischerweise sind die Felder der `A`-Klasse durch den Modifikator **private** geschützt (Datenkapselung), während die Methoden der `A`-Klasse durch den Modifikator **public** der Öffentlichkeit zur Verfügung gestellt werden.

Methoden einer Klasse `B` aus einem berechtigten Modul (ausgeführt von einem beliebigen `B`-Objekt oder der `B`-Klasse selbst) ...

- können bei vorhandener Referenz ein `A`-Objekt `a1` auffordern, eine Methode auszuführen,
- haben aber keinen Zugriff auf die Felder des `A`-Objekts.

Methoden der eigenen Klasse `A` (z. B. ausgeführt von einem `A`-Objekt `a2`) ...

- können bei vorhandener Referenz nicht nur das `A`-Objekt `a1` auffordern, eine Methode auszuführen,
- sondern haben auch vollen Zugriff auf die Felder von `a1`.

Das Objekt `a1` ist also nicht vor anderen `A`-Objekten geschützt (außer durch die Klugheit des `A`-Programmierers), sondern vor der Klasse `B`, deren Programmierer in der Regel nur beschränktes Wissen von der `A`-Klasse hat.

Bei der Deklaration bzw. Definition von Feldern, Methoden, Konstruktoren und Mitgliedstypen können die Modifikatoren **private**, **protected** und **public** angegeben werden, um die Zugriffsrechte festzulegen.[1] In der folgenden Tabelle sind die Effekte der Zugriffsmodifikatoren für Mitglieder eines Top-Level - Typs beschrieben, der selbst als **public** definiert ist. Bei den „Zugriffsbewerbern" soll es sich um Top-Level - Typen[2] in berechtigten Modulen handeln.[3]

---

[1] Unter dem Begriff *Mitgliedstypen* (vgl. Abschnitt 4.8.1) sind hier Klassen und Schnittstellen zu verstehen, die innerhalb eines Top-Level-Typs außerhalb von Methoden definiert werden, z. B.:

```
public class Top {
  . . .
  private class MemberClass {
   . . .
  }
  . . .
}
```

[2] Mitgliedsklassen und lokale Klassen haben erweiterte Rechte zum Zugriff auf die Mitglieder der umgebenden Klasse (siehe Abschnitt 4.8).

[3] Von einem *berechtigten Modul* `modb` in Bezug auf das Paket `pina` des Moduls `moda` sprechen wir dann, wenn ...

- das Modul `moda` in seiner Deklarationsdatei das Paket `pina` für das Modul `modb` exportiert (z. B. im Rahmen einer generellen **exports**-Deklaration),
- das Modul `modb` sich in seiner Deklarationsdatei per **requires**- Deklaration als abhängig vom Modul `moda` erklärt.

Modifikator	Der Zugriff ist erlaubt für ...			
	den eigenen Typ	**andere Typen im eigenen Paket**	**abgeleitete Typen in fremden Paketen**	**sonstige Typen in fremden Paketen**
ohne[1]	ja	ja	nein	nein
private	ja	nein	nein	nein
protected	ja	ja	nur geerbte Elemente	nein
public	ja	ja	ja	ja

Mit abgeleiteten Klassen und dem nur dort relevanten Zugriffsmodifikator **protected** werden wir uns im Kapitel 7 beschäftigen.

Wird im Bruchadditionsbeispiel (siehe Abschnitt 6.2.7) die Klasse `Simput`, die sich im Paket `de.uni_trier.zimk.util.conio` befindet, mit **public**-Zugriffsmodifikator versehen, ihre `gint()` - Methode jedoch *nicht*, dann kann die Klasse `Bruch`, die sich im Paket `de.uni_trier.zimk.matrain.br` befindet, die Methode **`gint()`** nicht verwenden, z. B.:

```
public void frage() {
    int n;
    do {
        System.out.print("Zähler: ");
        setzeZaehler(Simput.gint());
```

'gint()' is not public in 'de.zimk.util.conio.Simput'. Cannot be accessed from outside package
Make 'Simput.gint' public Alt+Umschalt+Eingabe More actions... Alt+Eingabe

Für Konstruktoren gilt:

- Bei expliziten Konstruktoren sind wie bei anderen Klassenmitgliedern die Modifikatoren **public**, **private** und **protected** erlaubt. Ein als **protected** deklarierter Konstruktor darf im eigenen Paket und von abgeleiteten Klassen in beliebigen Paketen aus berechtigten Modulen genutzt werden.
- Der vom Compiler bereitgestellte Standardkonstruktor (vgl. Abschnitt 4.4.3) hat denselben Zugriffsschutz wie die Klasse.

6.4 Übungsaufgaben zum Kapitel 6

1) Im folgenden Programm bestehend aus zwei in derselben Quellcodedatei implementierten Klassen

```
class Worker {
    void work() {
        System.out.println("Geschafft!");
    }
}

class Prog {
    public static void main(String[] args) {
        Worker w = new Worker();
        w.work();
    }
}
```

[1] Für die voreingestellte Sichtbarkeit (nur das eigene Paket darf zugreifen) wird gelegentlich die Bezeichnung *package* access verwendet.

erzeugt und verwendet die **main()** - Methode der Klasse `Prog` ein Objekt der fremden Klasse `Worker`, obwohl die Klasse `Worker` und ihre Methode `work()` *nicht* als **public** deklariert wurden. Wieso ist dies möglich?

2) Welche der folgenden Aussagen sind richtig bzw. falsch?

1. Die Namen von Paketen in Java - Modulen müssen mit dem Modulnamen beginnen.
2. Traditionelle **jar**-Dateien können als sogenannte *automatische Module* in den Modulpfad aufgenommen werden.
3. Pakete eines Moduls sind per Voreinstellung (ohne **exports**-Deklaration) nur Modul-intern sichtbar.
4. Explizite Java - Module können auf **class**-Dateien im traditionellen Klassenpfad zugreifen.

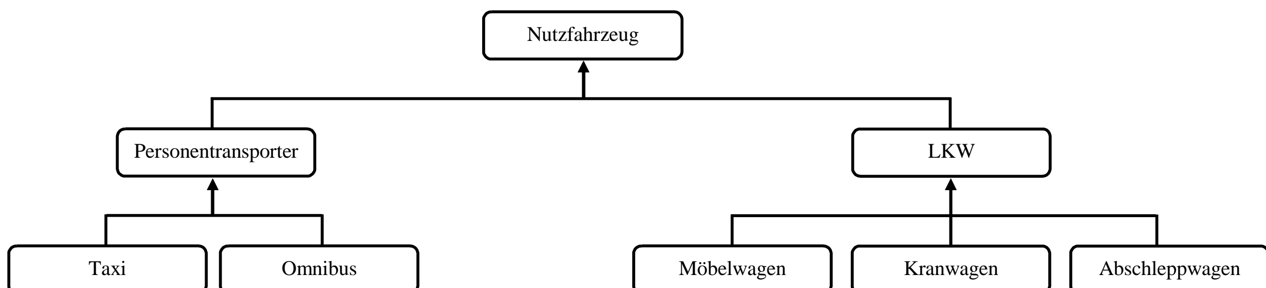
7 Vererbung und Polymorphie

Im Manuskript war schon mehrfach davon die Rede, dass sich die Java-Klassen nicht auf *einer* Ebene befinden, sondern in eine strenge Abstammungshierarchie eingeordnet sind. Nun betrachten wir die Vererbungsbeziehung zwischen Klassen und die damit verbundenen Vorteile für die Software-Entwicklung im Detail, wobei vor allem rationelle Wiederverwendung von vorhandener Software zur Lösung neuer Aufgaben zu nennen ist.

Die bequeme Implementation einer abgeleiteten Klasse geht oft einher mit dem Nachteil einer Abhängigkeit von der Basisklasse, was insbesondere beim Beerben einer „fremden“ (nicht im selben Paket befindlichen) Klasse beachtet werden muss, deren mögliche Veränderungen man nicht unter Kontrolle hat. Am Ende des Kapitels werden wir uns mit unerwünschten Abhängigkeiten zwischen Klassen und Möglichkeiten zur Vermeidung von daraus resultierenden Problemen beschäftigen.

Modellierung realer Klassenhierarchien

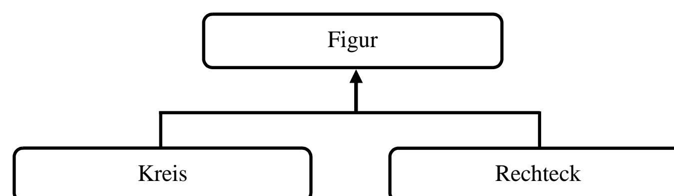
Beim Modellieren eines Gegenstandbereiches durch Klassen, die durch Eigenschaften (Instanz- und Klassenvariablen) sowie Handlungskompetenzen (Instanz- und Klassenmethoden) gekennzeichnet sind, müssen auch die Spezialisierungs- bzw. Generalisierungsbeziehungen zwischen real existierenden Objektsorten abgebildet werden. Eine Firma für Transportaufgaben aller Art mag ihre Nutzfahrzeuge folgendermaßen klassifizieren:



Einige Eigenschaften sind für alle Nutzfahrzeuge relevant (z. B. Anschaffungspreis, momentane Position), andere betreffen nur spezielle Klassen (z. B. maximale Anzahl von Fahrgästen, maximale Traglast). Ebenso sind einige Handlungsmöglichkeiten bei allen Nutzfahrzeugen vorhanden (z. B. eigene Position melden, ein Ziel ansteuern), während andere Handlungskompetenzen speziellen Fahrzeugen vorbehalten sind (z. B. Fahrgäste befördern, Lasten anheben). Ein Programm zur Verwaltung der Fahrzeuge und ihrer Einsätze muss diese reale Klassenhierarchie abbilden.

Übungsbeispiel

Bei unseren Beispielprogrammen bewegen wir uns in einem bescheideneren Rahmen und betrachten eine einfache Hierarchie mit Klassen für geometrische Figuren:



Vielleicht haben manche Leser als Gegenstück zum Rechteck (auf derselben Hierarchieebene) die *Ellipse* erwartet, die ebenfalls zwei ungleiche lange Hauptachsen besitzt. Weiterhin liegt es auf den ersten Blick nahe, den Kreis als Spezialisierung der Ellipse und das Quadrat als Spezialisierung des Rechtecks zu betrachten. Wir werden aber im Abschnitt 7.9 über das *Liskovsche Substitutionsprinzip* genau diese Ableitungen (von Kreis aus Ellipse bzw. von Quadrat aus Rechteck) kritisieren.

Man spricht hier vom *Kreis-Ellipse* - oder *Quadrat-Rechteck - Problem.*[1] Es ist wohl akzeptabel, an Stelle der Ellipse den Kreis neben das Rechteck zu stellen, um das Erlernen der neuen Konzepte durch ein möglichst einfaches Beispiel ohne Verstoß gegen das Liskovsche Substitutionsprinzip zu erleichtern.

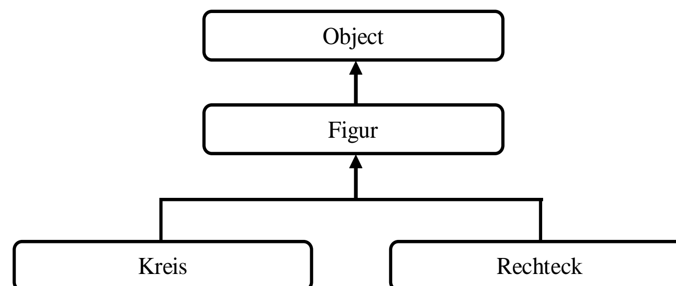
Die Vererbungstechnik in Java

In objektorientierten Programmiersprachen wie Java ist es weder sinnvoll noch erforderlich, jede Klasse einer Hierarchie komplett neu zu definieren. Es steht eine mächtige und zugleich einfach handhabbare Vererbungstechnik zur Verfügung: Man geht von der allgemeinsten Klasse aus und leitet durch Spezialisierung neue Klassen ab, nach Bedarf in beliebig vielen Stufen. Eine abgeleitete Klasse erbt alle Felder, Methoden und Mitgliedstypen ihrer **Basis**- oder **Superklasse** (jedoch keine Konstruktoren) und kann nach Bedarf Anpassungen bzw. Erweiterungen zur Lösung spezieller Aufgaben vornehmen, z. B.:

- zusätzliche Felder deklarieren
- zusätzliche Methoden definieren
- geerbte Methoden überschreiben, d.h. unter Beibehaltung der Signatur umgestalten

Ihre Konstruktoren muss eine abgeleitete Klasse neu definieren, wobei es aber leicht möglich ist, einen Basisklassenkonstruktor zur Initialisierung von geerbten Instanzvariablen einzuspannen (siehe Abschnitt 7.4).

In Java stammen *alle* Klassen von der Klasse **Object** aus dem Paket **java.lang** ab. Wird (wie bei den meisten bisherigen Beispielen im Manuskript) in der Definition einer Klasse *keine* Basisklasse angegeben, dann stammt sie auf direktem Wege von **Object** ab, anderenfalls indirekt. Bei der Implementation in Java wird die oben dargestellte Figuren-Klassenhierarchie so eingehängt:



In Java ist die in anderen objektorientierten Programmiersprachen (wie z. B. C++) erlaubte **Mehrfachvererbung ausgeschlossen**, sodass jede Klasse (mit Ausnahme von **Object**) genau *eine* Basisklasse hat (siehe Abschnitt 7.2.2).

Mit ihrem Vererbungsmechanismus bietet die objektorientierte Programmierung ideale Voraussetzungen dafür, vorhandene Software auf rationelle Weise zur Lösung neuer Aufgaben wiederzuverwenden. Dabei können allmählich umfangreiche Softwaresysteme entstehen, die gleichzeitig stabil und innovationsoffen sind (vgl. Abschnitt 4.1.1.3 zum Open-Closed - Prinzip). Die nicht selten anzutreffende Praxis, vorhandenen Code per *Copy & Paste* in neuen Projekten bzw. Klassen zu verwenden, hat gegenüber einer sorgfältig geplanten Klassenhierarchie offensichtliche Nachteile. Natürlich kann Java nicht garantieren, dass jede Klassenhierarchie exzellent entworfen ist und langfristig von einer stetig wachsenden Entwicklergemeinde eingesetzt wird.

[1] Siehe z. B. https://en.wikipedia.org/wiki/Circle-ellipse_problem

7.1 Definition einer abgeleiteten Klasse

Wir definieren im angekündigten Beispiel zunächst die Basisklasse `Figur`, die Instanzvariablen für die X- und die Y-Position der linken oberen Ecke einer zweidimensionalen Figur sowie zwei Konstruktoren besitzt:

```
package de.uni_trier.zimk.figuren;

public class Figur {
   private double xpos = 100.0, ypos = 100.0;

   public Figur(double x, double y) {
      if (x >= 0 && y >= 0) {
         xpos = x;
         ypos = y;
      }
      System.out.println("Figur-Konstruktor");
   }

   public Figur() {
      System.out.println("Figur-Konstruktor");
   }
}
```

Mit Hilfe des Schlüsselwortes **extends** wird nun die Klasse `Kreis` als Spezialisierung der Klasse `Figur` definiert. Sie erbt die beiden Positionsvariablen und ergänzt eine zusätzliche Instanzvariable für den Radius:

```
package de.uni_trier.zimk.figuren;

public class Kreis extends Figur {
   private double radius = 50.0;

   public Kreis(double x, double y, double rad) {
      super(x, y);
      if (rad >= 0)
         radius = rad;
      System.out.println("Kreis-Konstruktor");
   }

   public Kreis() {
      System.out.println("Kreis-Konstruktor");
   }
}
```

Es wird ein parametrisierter `Kreis`-Konstruktor definiert, der über das Schlüsselwort **super** den parametrisierten Konstruktor der Basisklasse aufruft. Ein direkter Zugriff auf die privaten (!) Instanzvariablen `xpos` und `ypos` der Klasse `Figur` wäre dem Konstruktor der Klasse `Kreis` auch nicht erlaubt. Das Schlüsselwort **super** hat übrigens den oben eingeführten Begriff *Superklasse* motiviert.

In der Klasse `Kreis` wird (wie in der Basisklasse `Figur`) auch ein parameter*freier* Konstruktor definiert. Vielleicht hat jemand gehofft, die `Kreis`-Klasse würde den parameterfreien Konstruktor ihrer Basisklasse (bei automatischer Anpassung des Namens) übernehmen. Konstruktoren werden jedoch grundsätzlich *nicht* vererbt. Der parameterfreie `Kreis`-Konstruktor tut nichts anderes, als per Konsolenausgabe einen Existenznachweis zu liefern und (insgeheim) den parameterfreien Konstruktor der Basisklasse aufzurufen. Im Abschnitt 7.4 werden wir uns mit einigen Regeln für die Mitwirkung von Basisklassenkonstruktoren bei der Objektkreation beschäftigen.

Das folgende Programm erzeugt ein Objekt aus der Basisklasse `Figur` und ein Objekt aus der abgeleiteten Klasse `Kreis`:

Quellcode	Ausgabe

`import de.uni_trier.zimk.figuren.*;`
 	
`class FigurenDemo {`
`   public static void main(String[] args) {`
`      Figur fig = new Figur(50.0, 50.0);`
`      System.out.println();`
`      Kreis krs = new Kreis(10.0, 10.0, 5.0);`
`   }`
`}`
	`Figur-Konstruktor`
	
	`Figur-Konstruktor`
	`Kreis-Konstruktor`

7.2 Optionale und erzwungene Beschränkungen beim (Ver)erben

In der Java-Vererbungstechnik bestehen optionale und erzwungene Einschränkungen.

7.2.1 Finale Klassen

Gelegentlich gibt es Gründe dafür, eine Klasse mit dem Modifikator **final** zu deklarieren, sodass sie zwar verwendet (z. B. instanziert), aber nicht beerbt werden kann. Bei der Klasse **String** im API-Paket **java.lang**

```
public final class String { ... }
```

ist das Finalisieren z. B. erforderlich, damit keine abgeleitete Klasse die Unveränderlichkeit von **String**-Objekten (vgl. Abschnitt 5.2.1) unterlaufen kann.

7.2.2 Keine Mehrfachvererbung

In Java ist generell **keine Mehrfachvererbung** möglich: Man kann also in einer Klassendefinition hinter dem Schlüsselwort **extends** nur *eine* Basisklasse angeben. Im Sinne einer realitätsnahen Modellierung wäre eine Mehrfachvererbung gelegentlich durchaus wünschenswert. So könnte z. B. die Klasse `Receiver` von den Klassen `Tuner` und `Amplifier` erben. Man hat auf die in anderen Programmiersprachen (z. B. C++) erlaubte Mehrfachvererbung bewusst verzichtet, um von vornherein den kritischen Fall auszuschließen, dass eine abgeleitete Klasse gleichnamige *Instanzvariablen* von mehreren Klassen erbt, woraus leicht Mehrdeutigkeiten und Fehler resultieren können (siehe Kreft & Langer 2014 zum sogenannten *Deadly Diamond of Death* bei der Mehrfachvererbung).

Einen gewissen Ersatz bieten die im Kapitel 9 behandelten Schnittstellen (Interfaces), weil ...

- bei Schnittstellen die Mehrfachvererbung erlaubt ist,
- und außerdem eine Klasse mehrere Schnittstellen implementieren darf.

7.3 Der Zugriffsmodifikator protected

In diesem Abschnitt wird anhand einer Variante des Figurenbeispiels der Effekt des bei Klassenmitgliedern erlaubten Zugriffsmodifikators **protected** demonstriert (vgl. Abschnitt 6.3.2). Wenn die Basisklasse `Figur` die Instanzvariablen `xpos` und `ypos` als **protected** deklariert,

```
protected double xpos = 100.0, ypos = 100.0;
```

können Instanzmethoden abgeleiteter Klassen unabhängig von ihrer Paketzugehörigkeit direkt darauf zugreifen. Dies geschieht in der neuen `Kreis`-Methode `abstand()`, die für einen beliebigen Punkt im zweidimensionalen Koordinatensystem über den Satz von Pythagoras den Abstand zum

Kreismittelpunkt berechnet.[1] Weil *innerhalb* eines Pakets die abgeleiteten Klassen dieselben Zugriffsrechte haben wie beliebige andere Klassen (vgl. Abschnitt 6.3), sorgen wir zu Demonstrationszwecken dafür, dass die Basisklasse `Figur` und die abgeleitete Klasse `Kreis` zu verschiedenen Paketen gehören.

```
package de.uni_trier.zimk.figuren.kreis;

import de.uni_trier.zimk.figuren.Figur;

public class Kreis extends Figur {
   protected double radius = 50.0;

   public Kreis(double x, double y, double rad) {
      super(x, y);
      if (rad >= 0)
         radius = rad;
   }

   public Kreis() {}

   public double abstand(double x, double y) {
      return Math.sqrt(Math.pow(xpos+radius-x, 2) + Math.pow(ypos+radius-y, 2));
   }
}
```

Es ist zu beachten, dass die `Kreis`-Methode `abstand()` auf *geerbte Instanzvariablen* von `Kreis`-Objekten zugreift. Es ist auch erlaubt, dass ein handelndes `Kreis`-Objekt auf die geerbten Instanzvariablen eines *anderen* `Kreis`-Objekts direkt zugreift. Auf das `xpos`-Feld eines `Figur`-Objekts könnte eine Methode der `Kreis`-Klasse hingegen *nicht* direkt zugreifen.

Während Objekte aus abgeleiteten Klassen ihre geerbten **protected**-Elemente direkt ansprechen können, haben paketfremde Klassen auf Elemente mit dieser Sichtbarkeit *keinen* Zugriff, z. B.:

```
import de.uni_trier.zimk.figuren.kreis.Kreis;

class FigurenDemo {
   public static void main(String[] args) {
      Kreis  k1 = new Kreis(50.0, 50.0, 30.0);
      System.out.println("Abstand von (100, 100): " + k1.abstand(100.0, 100.0));
      //klappt nicht:    System.out.println(k1.xpos);
   }
}
```

7.4 Basisklassenkonstruktoren und Initialisierungsmaßnahmen

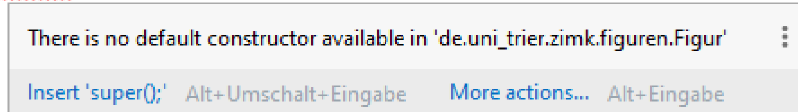
Abgeleitete Klassen erben die Basisklassenkonstruktoren *nicht*, können diese aber in eigenen Konstruktoren über das Schlüsselwort **super** aufrufen. Dieser Aufruf muss am Anfang eines Konstruktors stehen. Dadurch ist es z. B. möglich, geerbte Instanzvariablen zu initialisieren, die in der Basisklasse als **private** deklariert wurden. Diese Konstellation war in der ursprünglichen Version des Figurenbeispiels gegeben (siehe Abschnitt 7.1).

Wird in einem Konstruktor einer abgeleiteten Klasse kein Basisklassenkonstruktor explizit aufgerufen, dann ruft der Compiler implizit den *parameterfreien* Konstruktor der Basisklasse auf. Fehlt ein solcher, weil der Basisklassenprogrammierer einen eigenen, parametrisierten Konstruktor erstellt und nicht durch einen expliziten parameterfreien Konstruktor ergänzt hat, dann protestiert der

[1] Falls Sie sich über die Berechnung des Kreismittelpunkts wundern: In der Computergrafik ist die Position (0, 0) in der *oberen* linken Ecke des Bildschirms bzw. des aktuellen Fensters angesiedelt. Die X-Koordinaten wachsen (wie aus der Mathematik gewohnt) von links nach rechts, während die Y-Koordinaten von oben nach unten wachsen.

Compiler. Um die folgende Reklamation von IntelliJ zu provozieren, wurde in der Klasse `Figur` der parameterfreie Konstruktor auskommentiert:

```
public Kreis() {}
```



Es gibt zwei offensichtliche Möglichkeiten, das Problem zu lösen:

- Im Konstruktor der abgeleiteten Klasse über das Schlüsselwort **super** einen parametrisierten Basisklassenkonstruktor aufrufen.
- In der Basisklasse einen parameterfreien Konstruktor definieren.

Der parameterfreie Basisklassenkonstruktor wird auch vom Standardkonstruktor einer abgeleiteten Klasse aufgerufen, sodass jede potentiell als Erblasser in Frage kommende Klasse einen parameterfreien Konstruktor haben sollte.

Beim Erzeugen eines Unterklassenobjekts laufen folgende Initialisierungsmaßnahmen ab (vgl. Gosling et al. 2019, Abschnitt 12.5):

- Das Objekt wird mit allen Instanzvariablen (auch den geerbten) auf dem Heap angelegt, und die Instanzvariablen werden mit den typspezifischen Nullwerten initialisiert.
- Der Unterklassenkonstruktor beginnt seine Tätigkeit mit dem (impliziten oder expliziten) Aufruf eines Basisklassenkonstruktors. Falls in der Vererbungshierarchie die Urahnklasse **Object** noch nicht erreicht ist, wird am Anfang des Basisklassenkonstruktors ein Konstruktor der Super-Superklasse aufgerufen, bis diese Sequenz schließlich mit dem Aufruf eines **Object**-Konstruktors endet.
  Auf jeder Hierarchieebene (beginnend bei **Object**) laufen zwei Teilschritte ab:
  - Die Instanzvariablen der Klasse werden initialisiert, wobei die Deklaration und eventuell vorhandene Instanzinitialisierer (siehe Abschnitt 4.4.4) zu berücksichtigen sind.
  - Der Rumpf des Konstruktors wird ausgeführt.

Betrachten wir beispielhaft das Geschehen bei einem `Kreis`-Objekt, das mit dem Konstruktoraufruf

```
Kreis(150.0, 200.0, 30.0)
```

erzeugt wird:

- Das `Kreis`-Objekt wird mit seinen Instanzvariablen (`xpos`, `ypos`, `radius`) auf dem Heap angelegt, und die Instanzvariablen werden mit Nullen initialisiert.
- Aktionen für die Klasse **Object**:
  - Mangels Existenz sind keine Instanzvariablen der Klasse **Object** auf den deklarierten Initialisierungswert zu setzen.
  - Der Rumpf des parameterfreien **Object**-Konstruktors wird ausgeführt.
- Aktionen für die Klasse `Figur`:
  - Die Instanzvariablen `xpos` und `ypos` erhalten den Initialisierungswert laut Deklaration (jeweils 100,0).
  - Der Rumpf des Konstruktoraufrufs
    ```
    super(x, y)
    ```
    wird ausgeführt, wobei `xpos` und `ypos` die Werte 150,0 bzw. 200,0 erhalten.

- Aktionen für die Klasse `Kreis`:
  - Die Instanzvariable `radius` erhält den Initialisierungswert 50,0 aus der Deklaration.
  - Der Rumpf des Konstruktoraufrufs
    `Kreis(150.0, 200.0, 30.0)`
    wird ausgeführt, wobei `radius` den Wert 30,0 erhält.

7.5 Überschreiben und Überdecken

7.5.1 Überschreiben von Instanzmethoden

Eine Basisklassenmethode darf in einer abgeleiteten Klasse durch eine Methode mit gleichem Namen und gleicher Parameterliste (also mit gleicher Signatur, vgl. Abschnitt 4.3.4) überschrieben werden, um ein spezialisiertes Verhalten zu realisieren. Es liegt übrigens *keine* Überschreibung vor, wenn in der abgeleiteten Klasse eine Methode mit gleichem Namen, aber abweichender Parameterliste definiert wird. In diesem Fall sind die beiden Signaturen verschieden, und es handelt sich um eine Über*ladung* (siehe Abschnitt 4.3.4).

Um das Überschreiben von Instanzmethoden demonstrieren zu können, erweitern wir die `Figur`-Basisklasse um eine Methode namens `wo()`, welche die Position der linken oberen Ecke ausgibt:

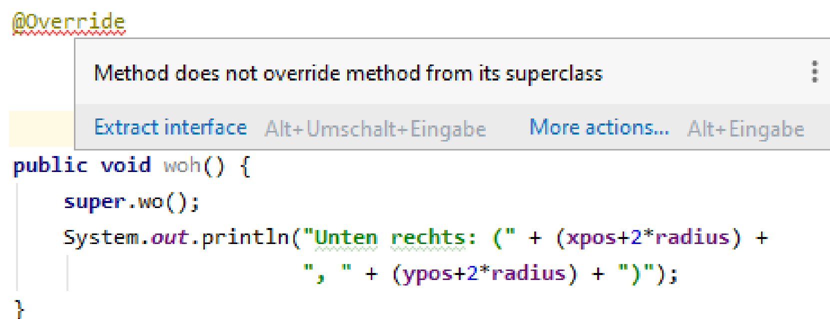
```
package de.uni_trier.zimk.figuren;

public class Figur {
   protected double xpos = 100.0, ypos = 100.0;

   public Figur(double x, double y) {
      if (x >= 0 && y >= 0) {
         xpos = x;
         ypos = y;
      }
   }

   public Figur() {}

   public void wo() {
      System.out.println("\nOben links: (" + xpos + ", " + ypos + ") ");
   }
}
```

In der `Kreis`-Klasse ist eine bessere Ortsangabenmethode realisierbar, weil hier auch die rechte untere Ecke definiert ist:

```
package de.uni_trier.zimk.figuren;

public class Kreis extends Figur {
   protected double radius = 50.0;

   public Kreis(double x, double y, double rad) {
      super(x, y);
      if (rad >= 0)
         radius = rad;
   }

   public Kreis() {}

   public double abstand(double x, double y) {
      return Math.sqrt(Math.pow(xpos+radius-x, 2) + Math.pow(ypos+radius-y, 2));
   }
```

```
    @Override
    public void wo() {
       super.wo();
       System.out.println("Unten rechts: (" + (xpos+2*radius) +
                              ", " + (ypos+2*radius) + ")");
    }
 }
```

Mit der Marker-Annotation @**Override** (vgl. Abschnitt 9.5) kann man seine Absicht bekunden, bei einer Methodendefinition eine Basisklassenvariante zu überschreiben. Misslingt dieser Plan z. B. aufgrund eines Tippfehlers, protestiert unsere Entwicklungsumgebung:



In der überschreibenden Methode kann man sich oft durch Rückgriff auf die überschriebene Methode die Arbeit erleichtern, wobei wieder das Schlüsselwort **super** zum Einsatz kommt.

Das folgende Programm schickt an eine `Figur` und an einen `Kreis` jeweils die Nachricht `wo()`, und beide zeigen ihr artspezifisches Verhalten:

Quellcode	Ausgabe

`import de.uni_trier.zimk.figuren.*;`
 	
`class FigurenDemo {`
`   public static void main(String[] ars) {`
`      Figur f = new Figur(10.0, 20.0);`
`      f.wo();`
`      Kreis k = new Kreis(50.0, 100.0, 25.0);`
`      k.wo();`
`   }`
`}`
	`Oben links: (10.0, 20.0)`
	
	`Oben links: (50.0, 100.0)`
	`Unten rechts: (100.0, 150.0)`

Auch bei den vom Urahntyp **Object** geerbten Methoden kommt ein Überschreiben in Frage. Die **Object**-Methode **toString()** liefert neben dem Klassennamen den (meist aus der Speicheradresse abgeleiteten) Hashcode des Objekts. Sie wird z. B. von der **String**-Methode **println()** automatisch genutzt, um eine Zeichenfolgendarstellung zu einem Objekt zu ermitteln, z. B.:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`        Prog tst1 = new Prog(), tst2 = new Prog();`
`        System.out.println(tst1 + "\n"+ tst2);`
`    }`
`}`
	`Prog@15e8f2a0`
	`Prog@7090f19c`

In der API-Dokumentation zur Klasse **Object** wird das Überschreiben der Methode **toString()** explizit für alle Klassen empfohlen. Diese Methode wird vom Designer und von den Anwendern einer Klasse durch expliziten oder impliziten Aufruf potentiell oft genutzt. Ein impliziter **toString()** - Aufruf findet z. B. immer dann statt, wenn ein Objekt als Parameter an **print()**, **println()**, **printf()** oder an den Zeichenkettenverknüpfungsoperator übergeben wird. Bloch (2018, S. 55ff) empfiehlt,

als **toString()** - Rückgabe interessante Informationen über das angesprochene Objekt zu liefern, und gibt noch einige Tipps zum Implementieren der Methode.

In der folgenden Klasse `Mint` (ein **int**-Wrapper, siehe Übungsaufgabe zu Abschnitt 5.3) liefert die **toString()** - Überladung den verpackten Wert:

```
public class Mint {
    public int value;

    public Mint(int value) {
        this.value = value;
    }

    public Mint() {}

   @Override
    public String toString() {
       return String.valueOf(value);
    }
}
```

Ein `Mint`-Objekt antwortet auf die **toString()** - Botschaft mit der Zeichenfolgendarstellung des gekapselten **int**-Werts:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class MintDemo {
   public static void main(String[] args) {
      Mint zahl = new Mint(4711);
      System.out.println(zahl);
   }
}</pre></td><td><pre>4711</pre></td></tr>
</table>

Den Versuch, eine Instanzmethode der Basisklasse durch eine *statische* Methode der abgeleiteten Klasse zu überschreiben, verhindert der Compiler.

Wie sich gleich im Abschnitt 7.7 über die Polymorphie zeigen wird, besteht der Clou bei überschriebenen Instanzmethoden darin, dass erst zur Laufzeit in Abhängigkeit vom tatsächlichen Typ eines handelnden, über eine Basisklassenreferenz angesprochenen Objekts entschieden wird, ob die Basisklassen- oder die Unterklassenmethode zum Einsatz kommt. Der Typ des handelnden Objekts ist in vielen Fällen zur Übersetzungszeit noch nicht bekannt, weil:

- über eine Basisklassenreferenzvariable durchaus auch ein Unterklassenobjekt verwaltet werden kann (siehe Abschnitt 7.6),
- und sich der konkrete Typ oft erst zur Laufzeit entscheidet, z. B. in Abhängigkeit von einer Benutzerentscheidung.

7.5.2 Überdecken von statischen Methoden

Es ist erlaubt, in einer abgeleiteten Klasse eine statische Methode zu definieren, welche die Signatur einer statischen Basisklassenmethode besitzt (selber Name und selbe Parameterliste). Zur der im letzten Abschnitt beschriebenen späten Entscheidung über die auszuführende Methode kommt es aber auch dann nicht, wenn die statische Methode über eine Objektreferenz angesprochen wird, was nicht empfehlenswert ist, aber erlaubt. In diesem Fall entscheidet der deklarierte Datentyp, nicht der Laufzeittyp, z. B.:

Quellcode	Ausgabe

<pre>package de.uni_trier.zimk.figuren;
public class Figur {
   public static void sm() {
      System.out.println("Statische Figur-Methode");
   }
   . . .
}
 	
package de.uni_trier.zimk.figuren;
public class Kreis extends Figur {
   public static void sm() {
      System.out.println("Statische Kreis-Methode");
   }
   . . .
}
 	
import de.uni_trier.zimk.figuren.*;
class FigurenDemo {
   public static void main(String[] ars) {
      Figur kr = new Kreis();
      kr.sm();
   }
}</pre>
	Statische Figur-Methode

Die auszuführende statische Methode steht also grundsätzlich schon zur Übersetzungszeit fest, und man spricht hier vom *Überdecken* oder *Verstecken* der Basisklassenmethode. Die überdeckte Basisklassenvariante einer statischen Methode ist natürlich durch Voranstellen des Klassennamens in den Methoden der abgeleiteten Klasse ansprechbar.

Den Versuch, eine statische Methode der Basisklasse durch eine Instanzmethode der abgeleiteten Klasse zu überdecken, verhindert der Compiler.

7.5.3 Finalisierte Methoden

Gelegentlich ist es sinnvoll, die Flexibilität der objektorientierten Vererbungstechnik gezielt einzuschränken, um das Auftreten von Unterklassenobjekten zu verhindern, die essentielles Basisklassenverhalten auf unerwünschte Weise neu definieren. Wie Sie aus Abschnitt 7.2.1 wissen, kann man für eine Klasse generell verbieten, abgeleitete Klassen zu definieren. Finalisiert man statt der gesamten Klasse nur eine Methode, darf zwar eine abgeleitete Klasse definiert, dabei aber die finalisierte Methode nicht überschrieben bzw. überdeckt werden. Dient etwa die Methode `passwd()` einer Klasse `Acl` zum Abfragen eines Passwortes, will ihr Programmierer eventuell verhindern, dass `passwd()` in einer von `Acl` abstammenden Klasse `Bcl` überschrieben wird. Ein guter Grund zum Finalisieren besteht meist auch bei Methoden, die in einer initialisierenden Deklarationsanweisung, in einem Instanzinitialisierer oder von einem Konstruktor aufgerufen werden.

Um das Überscheiben einer Instanzmethode oder das Überdecken einer statischen Methode zu verbieten, setzt man bei der Definition den Modifikator **final**. Unsere Klasse `Figur` (siehe Abschnitt 7.5.1) könnte z. B. eine Methode `oleck()` zur Ausgabe der oberen linken Ecke erhalten, die von abgeleiteten Klassen nicht geändert werden soll und daher als **final** (endgültig) deklariert wird:

```
final public void oleck() {
   System.out.print("\nOben links: (" + xpos + ", " + ypos + ") ");
}
```

Neben der beschriebenen Anwendungssicherheit bringt das Finalisieren einer Instanzmethode noch einen kleinen Performanzvorteil: Während bei nicht-finalisierten Instanzmethoden *das Laufzeitsys-*

tem feststellen muss, welche Variante in Abhängigkeit von der faktischen Klassenzugehörigkeit des angesprochenen Objekts tatsächlich ausgeführt werden soll (vgl. Abschnitt 7.7 über Polymorphie), steht eine **final**-Methode schon beim Übersetzen fest.

7.5.4 Felder überdecken

Wird in der abgeleiteten Klasse `Spezial` für eine Instanz- oder Klassenvariable ein Name verwendet, der bereits eine Variable der beerbten Klasse `General` bezeichnet, dann wird die Basisklassenvariable überdeckt. Sie ist jedoch weiterhin vorhanden und kommt in folgenden Situationen zum Einsatz:

- Von `General` geerbte Methoden verwenden weiterhin die `General`-Variable. In der `Spezial`-Klasse implementierte Methoden (zusätzliche, überschreibende oder überdeckende) greifen auf die `Spezial`-Variable zu.
- In der `Spezial`-Klasse implementierte Methoden *können* auf die `General`-Variable zugreifen:
    - auf eine überdeckte Instanzvariable durch das vorangestellte Schlüsselwort **super**
    - auf eine überdeckte statische Variable durch den vorangestellten Klassennamen.

Im folgenden Beispielprogramm führt ein `Spezial`-Objekt eine `General`- und eine `Spezial`-Methode aus, um den Zugriff auf eine überdeckte Instanzvariable zu demonstrieren:

Quellcode

```
// Datei General.java
class General {
   String x = "x-Gen";
   static String xs = "sx-Gen";
   void gm() {
      System.out.println("x in gm():\t\t "+x);
      System.out.println("xs in gm():\t\t "+xs);
   }
   static void gms() {
      System.out.println("xs in gms():\t\t "+xs);
   }
}
// Datei Spezial.java
class Spezial extends General {
   int x = 333;
   static int xs = 555;
   void sm() {
      System.out.println("x in sm():\t\t "+x);
      System.out.println("xs in sm():\t\t "+xs);
      System.out.println("\nsuper-x in sm():\t "+super.x);
      System.out.println("Basis-xs in sm():\t "+General.xs);
   }
   static void sms() {
      System.out.println("xs in sms():\t\t "+xs);
   }
}
// Datei Test.java
class Test {
   public static void main(String[] args) {
      Spezial sp = new Spezial();
      sp.gm(); System.out.println(""); sp.sm();
      System.out.println("");
      Spezial.gms(); Spezial.sms();
   }
}
```

Ausgabe

```
x in gm():              x-Gen
xs in gm():             sx-Gen

x in sm():              333
xs in sm():             555

super-x in sm():       x-Gen
Basis-xs in sm():      sx-Gen

xs in gms():            sx-Gen
xs in sms():            555
```

Während das Überschreiben von Methoden oft von entscheidender Bedeutung bei der Entwicklung einer guten Lösung ist, finden sich für das potentiell verwirrende Überdecken von Feldern nur wenige sinnvolle Einsatzzwecke.

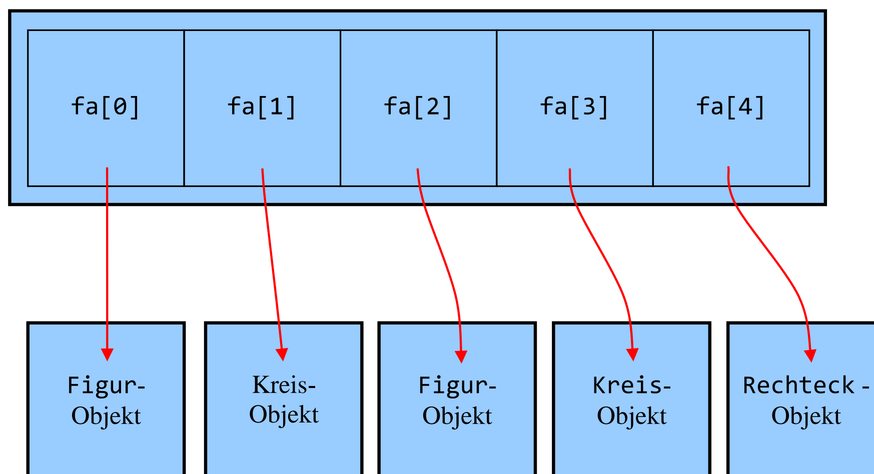
7.6 Verwaltung von Objekten über Basisklassenreferenzen

Eine Basisklassenreferenzvariable darf die Adresse eines beliebigen Unterklassenobjektes aufnehmen. Schließlich besitzt Letzteres die komplette Ausstattung der Basisklasse und kann z. B. dort definierte Methoden ausführen. Ein Objekt steht nicht nur zur eigenen Klasse in der „ist-ein"-Beziehung, sondern erfüllt diese Relation auch in Bezug auf die direkte Basisklasse sowie in Bezug auf alle indirekten Basisklassen in der Ahnenreihe.

Andererseits verfügt ein Basisklassenobjekt in der Regel *nicht* über die Ausstattung von abgeleiteten (erweiterten bzw. spezialisierten) Klassen. Daher ist es sinnlos und verboten, die Adresse eines Basisklassenobjektes in einer Unterklassen-Referenzvariablen abzulegen.

Über Referenzvariablen vom Typ einer gemeinsamen Basisklasse lassen sich also Objekte aus unterschiedlichen Klassen verwalten. Im Rahmen eines Grafikprogramms kommt vielleicht ein Array mit dem Elementtyp `Figur` zum Einsatz, dessen Elemente auf Objekte aus der Basisklasse oder aus einer abgeleiteten Klasse wie `Kreis` oder `Rechteck` zeigen:

Array `fa` mit Elementtyp `Figur`



Das folgende Programm verwaltet Referenzen auf Figuren und Kreise in einem Array vom Typ `Figur`. Weil wir die Klasse `Rechteck` noch nicht definiert haben, ist der Array `fa` im Vergleich zur obigen Abbildung um ein Element gekürzt:

Quellcode	Ausgabe

<pre>import de.uni_trier.zimk.figuren.*;
class FigurenDemo {
   public static void main(String[] args) {
      Figur[] fa = new Figur[4];
      fa[0] = new Figur(10.0, 20.0);
      fa[1] = new Kreis(50.0, 50.0, 25.0);
      fa[2] = new Figur(0.0, 30.0);
      fa[3] = new Kreis(100.0, 100.0, 10.0);
      for (int i = 0; i < fa.length; i++)
         if (fa[i] instanceof Kreis)
            System.out.println("Figur "+i+": Radius = "+
                              ((Kreis)fa[i]).gibRadius());
         else
            System.out.println("Figur "+i+": kein Kreis");
   }
}</pre>
	<pre>Figur 0: kein Kreis
	Figur 1: Radius = 25.0
	Figur 2: kein Kreis
	Figur 3: Radius = 10.0</pre>

Über eine Figur-Referenzvariable, die auf ein Kreis-Objekt zeigt, sind Erweiterungen der Kreis-Klasse (zusätzliche Felder und Methoden) *nicht* unmittelbar zugänglich. Wenn (auf eigene Verantwortung des Programmierers) eine Basisklassenreferenz als Unterklassenreferenz behandelt werden soll, um eine unterklassenspezifische Methode oder Variable anzusprechen, dann muss eine explizite Typumwandlung vorgenommen werden, z. B.:

```
((Kreis)fa[i]).gibRadius())
```

Im Zweifelsfall sollte man sich über den **instanceof**-Operator vergewissern, ob das referenzierte Objekt tatsächlich zur vermuteten Klasse gehört.

```
if (fa[i] instanceof Kreis)
    System.out.println("Figur "+i+": Radius = " + ((Kreis)fa[i]).gibRadius());
```

Um den Zugriff auf Unterklassenerweiterungen demonstrieren zu können, hat die Klasse Kreis im Vergleich zur Version im Abschnitt 7.5.1 die zusätzliche Methode gibRadius() erhalten:

```
public double gibRadius() {
   return radius;
}
```

7.7 Polymorphie

Werden Objekte aus verschiedenen Klassen über Referenzvariablen eines gemeinsamen Basistyps verwaltet, dann sind nur Methoden nutzbar, die schon in der Basisklasse definiert sind. Bei überschriebenen Methoden reagieren die Objekte jedoch unterschiedlich (jeweils unterklassentypisch) auf dieselbe Botschaft. Genau dieses Phänomen bezeichnet man als **Polymorphie**. Wer sich hier mit einem exotischen und nutzlosen Detail konfrontiert glaubt, sei an die Auffassung von Alan Kay erinnert, der wesentlich zur Entwicklung der objektorientierten Programmierung beigetragen hat. Er zählt die Polymorphie neben der Datenkapselung und der Vererbung zu den Grundelementen dieser Softwaretechnologie (siehe Abschnitt 4.1.1).

Gegen die unvermeidlichen Gewöhnungsprobleme mit dem Konzept der Polymorphie hilft am besten praktische Erfahrung. In welchem Ausmaß durch Polymorphie die Programmierpraxis erleichtert wird, kann leider durch die notwendigerweise kurzen Demonstrationsbeispiele nur ansatzweise vermittelt werden.

Das Figurenprojekt besitzt bereits alle Voraussetzungen zur Demonstration der Polymorphie im folgenden Beispielprogramm:[1]

```
import de.uni_trier.zimk.figuren.*;
import de.uni_trier.zimk.util.conio.Simput;

class FigurenDemo {
   public static void main(String[] ars) {
      Figur[] fa = new Figur[3];
      fa[0] = new Figur(10.0, 20.0);
      fa[1] = new Kreis(50.0, 50.0, 25.0);
      fa[0].wo();
      fa[1].wo();
```

[1] Im Beispielprogramm wird die Klasse Simput aus dem Paket de.uni_trier.zimk.util.conio bezogen (siehe Abschnitt 6.2.7.1 zur Erstellung). Allerdings nutzen wir derzeit keine Modultechnik und behandeln die modulare **jar**-Datei **de.uni_trier.zimk.util-1.0.jar** mit dem Modul de.uni_trier.zimk.util, das u.a. das Paket de.uni_trier.zimk.util.conio enthält, wie eine traditionelle **jar**-Datei. In IntelliJ wird diese **jar**-Datei über eine globale Bibliothek (siehe Abschnitt 3.4.2) in den traditionellen Klassenpfad aufgenommen. Weil die Klasse Simput mit Java 13 übersetzt wurde, muss das **Project SDK** auf die Version 13 eingestellt werden.

```
        System.out.print("\nWollen Sie zum Abschluss noch eine"+
         " Figur oder einen Kreis erleben?"+
         "\nWählen Sie durch Abschicken von \"f\" oder \"k\": ");
        if (Character.toUpperCase(Simput.gchar()) == 'F') {
           fa[2] = new Figur();
           fa[2].wo();
        }
        else {
           fa[2] = new Kreis();
           fa[2].wo();
           System.out.println("Radius: "+((Kreis)fa[2]).gibRadius());
        }
     }
  }
```

Hier werden Referenzen auf `Figur`- und `Kreis`-Objekte in einem Array vom gemeinsamen Basistyp `Figur` verwaltet (vgl. Abschnitt 7.6). Beim Ausführen der `wo()` - Methode, stellt das Laufzeitsystem die tatsächliche Klassenzugehörigkeit fest und wählt die passende Methode aus (spätes bzw. dynamisches Binden):

```
Oben Links: (10.0, 20.0)

Oben Links: (50.0, 50.0)
Unten Rechts: (100.0, 100.0)

Wollen Sie zum Abschluss noch eine Figur oder einen Kreis erleben?
Waehlen Sie durch Abschicken von "f" oder "k": k

Oben Links: (100.0, 100.0)
Unten Rechts: (200.0, 200.0)
Radius: 50.0
```

Zum „Beweis", dass tatsächlich eine späte Bindung stattfindet, darf im Beispielprogramm der Laufzeittyp des Array-Elements `fa[2]` vom Benutzer festgelegt werden.

Wird in einem Programm zur Verwendung von geometrischen Objekten der allgemeine Datentyp `Figur` genutzt, dann führen die zu diversen `Figur`-Unterklassen gehörigen Objekte beim Aufruf einer Basisklassenmethode ihr artspezifisches Verhalten aus. Später können neu entwickelte `Figur`-Ableitungen einbezogen werden, ohne den Quellcode der bereits vorhandenen Klassen ändern zu müssen. So sorgen Vererbung und Polymorphie für produktives Software-Recycling im Sinn des Open-Closed - Prinzips (vgl. Abschnitt 4.1.1.3).

Eng verwandt mit der eben beschriebenen *Basisklassen - Polymorphie* ist die *Interface - Polymorphie*, wobei als Datentyp für die flexiblen Referenzen an Stelle einer gemeinsamen Basisklasse ein Interface steht, das alle beteiligten Klassen implementieren (siehe Abschnitt 9.4).

7.8 Abstrakte Methoden und Klassen

Um die eben beschriebene gemeinsame Verwaltung von Objekten aus diversen abgeleiteten Klassen über Referenzvariablen von einem gemeinsamen Basisklassentyp nutzen und dabei artspezifisch realisierte Methodenaufrufe realisieren zu können, müssen die betroffenen Methoden in der Basisklasse vorhanden sein. Wenn es für eine Methode in der Basisklasse keine sinnvolle Implementierung gibt, erstellt man dort eine sogenannte **abstrakte** Methode:

- Man beschränkt sich auf den Methodenkopf und setzt dort den Modifikator **abstract**.
- Den Methodenrumpf ersetzt man durch ein Semikolon.

Im Figurenbeispiel erweitern wir die Klasse `Kreis` um eine Methode namens `meldeInhalt()` zum Ermitteln des Flächeninhalts:

```
   public double meldeInhalt() {
      return Math.PI * radius*radius;
   }
```

Außerdem erstellen wir die Klasse `Rechteck` und definieren auch hier eine Methode namens `meldeInhalt()`:

```
package de.uni_trier.zimk.figuren;

public class Rechteck extends Figur {
   protected double breite = 50.0, hoehe = 50.0;

   public Rechteck(double x, double y, double b, double h) {
      super(x, y);
      if (b >= 0 && h >= 0) {
         breite = b;
         hoehe = h;
      }
   }

   public Rechteck() {}

   @Override
   public void wo() {
      super.wo();
      System.out.println("Unten rechts: (" + (xpos+breite) +
                        ", " + (ypos+hoehe) + ")");
   }

   @Override
   public double meldeInhalt() {
      return breite * hoehe;
   }
}
```

Weil die Methode zum Ermitteln des Flächeninhalts in der Basisklasse `Figur` nicht sinnvoll realisierbar ist, wird sie hier abstrakt definiert:

```
package de.uni_trier.zimk.figuren;

public abstract class Figur {
    . . .
   public abstract double meldeInhalt();
      . . .
}
```

Enthält eine Klasse mindestens eine abstrakte Methode, dann handelt es sich um eine **abstrakte Klasse**, und bei der Klassendefinition muss der Modifikator **abstract** vergeben werden.

Aus einer abstrakten Klasse kann man zwar keine Objekte erzeugen, aber andere Klassen ableiten. Implementiert eine abgeleitete Klasse die abstrakten Methoden, lassen sich Objekte daraus herstellen; anderenfalls ist sie ebenfalls abstrakt. Im Beispiel werden aus der nunmehr abstrakten Klasse `Figur` die beiden konkreten Klassen `Kreis` und `Rechteck` abgeleitet.

Eine abstrakte Klasse lässt sich beerben, und sie eignet sich bestens als Datentyp. Referenzen dieses Typs sind ja auch unverzichtbar, wenn Objekte diverser Unterklassen polymorph verwaltet werden sollen. Das folgende Programm:

```
import de.uni_trier.zimk.figuren.*;

class FigurenDemo {
   public static void main(String[] ars) {
      Figur[] fa = new Figur[2];
      fa[0] = new Kreis(50.0, 50.0, 25.0);
      fa[1] = new Rechteck(10.0, 10.0, 100.0, 200.0);
      double ges = 0.0;
      for (int i = 0;  i < fa.length; i++) {
         System.out.printf("Fläche Figur %d (%-34s): %15.2f\n",
               i, fa[i].getClass().getName(), fa[i].meldeInhalt());
         ges += fa[i].meldeInhalt();
      }
      System.out.printf("\nGesamtfläche:  %10.2f",ges);
   }
}
```

liefert die Ausgabe:

```
Fläche Figur 0 (de.uni_trier.zimk.figuren.Kreis   ):         1963,50
Fläche Figur 1 (de.uni_trier.zimk.figuren.Rechteck):        20000,00

Gesamtfläche:    21963,50
```

Die Methode `meldeInhalt()` eignet sich dazu, den Nutzen der Polymorphie zu demonstrieren. Ein Programm für das Malerhandwerk könnte zur Planung der benötigten Farbmenge seinem Benutzer erlauben, beliebig viele Objekte aus diversen `Figur`-Unterklassen anzulegen, und dann die gesamte Oberfläche in einer Schleife durch polymorphe Methodenaufrufe ermitteln.

Statische Methoden dürfen *nicht* abstrakt definiert werden.

7.9 Das Liskovsche Substitutionsprinzip

In diesem Abschnitt geht es um eine auf den ersten Blick theoretisch wirkende, aber durchaus praxisrelevante Klärung zur objektorientierten Vererbungsbeziehung. Das nach Barbara Liskov benannte Substitutionsprinzip (dt.: *Ersetzbarkeitsprinzip*) verlangt von einer Klassenhierarchie (Liskov & Wing 1999, S. 1):

> Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Wird beim Entwurf einer Klassenhierarchie das Liskovsche Substitutionsprinzip (LSP) beachtet, dann können Objekte einer abgeleiteten Klasse stets die Rolle von Basisklassenobjekten perfekt übernehmen, d.h. u.a.:

- Das „vertraglich“ zugesicherte Verhalten der Basisklassenmethoden wird auch von den (eventuell überschreibenden) Unterklassenvarianten eingehalten.
- Unterklassenobjekte werden bei Verwendung in der Rolle von Basisklassenobjekten nicht beschädigt.

Eine Verletzung der Ersetzbarkeitsregel kann auch bei einfachen Beispielen auftreten, wobei oft eine aus dem Anwendungsbereich stammende Plausibilität zum fehlerhaften Design verleitet. So ist z. B. ein Quadrat aus mathematischer Sicht ein spezielles Rechteck. Definiert man in einer Klasse für Rechtecke die Methoden `skaliereX()` und `skaliereY()` zur Änderung der Länge in X- bzw. - Y-Richtung, so gehört zum „vertraglich“ zugesicherten Verhalten dieser Methoden:

- Bei einem Zuwachs in X-Richtung bleibt die Y-Ausdehnung unverändert.
- Verdoppelt man die Breite eines Objekts, verdoppelt sich auch der Flächeninhalt.

Die simple Tatsache, dass aus mathematischer Sicht jedes Quadrat ein Rechteck ist, rät offenbar dazu, eine Klasse für Quadrate aus der Klasse für Rechtecke abzuleiten. In der neuen Klasse ist allerdings die Konsistenzbedingung zu ergänzen, dass bei einem Quadrat stets alle Seiten gleich lang bleiben. Um das Auftreten irregulärer (defekter) Objekte der Klasse `Quadrat` zu verhindern, wird man z. B. die Methode `skaliereX()` so überschreiben, dass bei einer X-Modifikation automatisch auch die Y-Ausdehnung angepasst wird. Damit ist aber der `skaliereX()` - Vertrag verletzt, wenn ein Quadrat die Rechteckrolle übernimmt. Eine verdoppelte X-Länge führt etwa nicht zur doppelten, sondern zur vierfachen Fläche. Verzichtet man andererseits in der Klasse `Quadrat` auf das Überschreiben der Methode `skaliereX()`, ist bei den Objekten dieser Klasse die Konsistenzbedingung identischer Seitenlängen massiv gefährdet. Offenbar haben Plausibilitätsüberlegungen zu einer schlecht entworfenen Klassenhierarchie geführt.

Eine exakte Verhaltensanalyse zeigt, dass ein Quadrat in funktionaler Hinsicht eben doch kein Rechteck ist. Es fehlt die für Rechtecke typische Option, die Ausdehnung in X- bzw. Y-Richtung separat zu verändern. Diese Option könnte in einem Algorithmus, der den Datentyp `Rechteck` voraussetzt, von Bedeutung sein. Es muss damit gerechnet werden, dass der Algorithmus irgendwann (bei einer Erweiterung der Software) auf Objekte mit einem von `Rechteck` abstammenden Datentyp trifft. Passiert dies mit der Klasse `Quadrat` könnte es zu Problemen kommen, weil nach einer Verdopplung der X-Ausdehnung der Flächeninhalt entgegen der Erwartung nicht auf das Doppelte, sondern auf das Vierfache wächst.

Um die Einhaltung des Substitutionsprinzips beurteilen zu können, bedarf es einer sorgfältigen Analyse. Wenn etwa Objekte der Klasse `Rechteck` *unveränderlich* wären, wenn also die Methoden `skaliereX()` und `skaliereY()` in der Klassendefinition von `Rechteck` fehlen würden, dann könnte die Klasse `Quadrat` sehr wohl als Spezialisierung von `Rechteck` definiert werden.

Java bietet gute Voraussetzungen für eine erfolgreiche objektorientierte Programmierung, kann aber z. B. eine Verletzung des Substitutionsprinzips nicht verhindern.

7.10 Unerwünschte Abhängigkeiten durch Vererbung

Unter Software-Entwicklern hat es sich herumgesprochen, dass die Vererbung einerseits ein Segen ist (Erleichterung der Implementation durch rationelle Wiederverwendung von Software), andererseits aber auch ein Fluch, weil unerwünschte Abhängigkeiten zwischen Klassen resultieren.

7.10.1 Risiken für abgeleitete Klassen

Eine abgeleitete Klasse erbt oftmals viele Methoden der Basisklasse, die im günstigsten Fall überflüssig sind, eventuell aber auch Schadpotential besitzen. Ein bekanntes Negativbeispiel aus dem Java-API ist die von **Hashtable<Object, Object>** abgeleitete Klasse **Properties** im Paket **java.util**:

```
public class Properties extends Hashtable<Object,Object> {
 . . .
  public synchronized Object setProperty(String key, String value) {
      return put(key, value);
  }
. . .
}
```

Um korrekt zu funktionieren, darf eine Kollektion aus der Klasse **Properties** als Einträge nur Paare mit einem Namen und einem Wert vom Typ **String** enthalten (siehe Definition der Methode **setProperty**()).[1] Über die von der Klasse **Hashtable<Object, Object>** geerbte Methode **put()** kann

[1] Der Methodenmodifikator **synchronized** wird im Kapitel 15 über die Multithread-Programmierung behandelt.

aber auch ein (**Object**, **Object**) - Paar eingeschmuggelt werden. In der Dokumentation zur Klasse **Properties** wird auf die Gefahr hingewiesen:

> Because `Properties` inherits from `Hashtable`, the `put` and `putAll` methods can be applied to a `Properties` object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not Strings. The `setProperty` method should be used instead. If the `store` or `save` method is called on a "compromised" `Properties` object that contains a non-`String` key or value, the call will fail.

Im Beispiel bestand das Problem schon bei der Definition der abgeleiteten Klasse **Properties**. Es kann aber auch durch eine spätere Erweiterung der Basisklasse auftreten.

Neben den unerwünschten Erbstücken besteht eine weitere Problemkonstellation für abgeleitete Klassen in der Überschreibung von Methoden. Diese ist möglich bei Basisklassenmethoden mit der Sichtbarkeit **public** oder **protected**, die nicht als **final** deklariert worden sind (vgl. Abschnitt 7.5.3). Auch hier liegen wieder Segen (in Form der Polymorphie, siehe Abschnitt 7.7) und Fluch (in Form einer Abhängigkeit von Implementierungsdetails der Basisklasse) nah beieinander. Bloch (2018, S. 87ff) beschreibt eine Ableitung der zur Mengenverwaltung dienenden Kollektionsklasse **HasSet<E>** aus dem Java Collections Framework.[1] Blochs Klasse soll die eingefügten (und eventuell teilweise später wieder gelöschten) Elemente zählen. Zu diesem Zweck werden die geerbten Methoden **add()** und **addAll()** so überschrieben, dass ein privates Feld geeignet inkrementiert wird. Weil die überschriebene Basisklassenmethode Methode **addAll()** intern die ebenfalls überschriebene Methode **add()** aufruft, stimmt das Zählergebnis in der Ableitung aber nicht. Jedes per **addAll()** eingefügte Element wird doppelt gezählt. Der Fehler ist aufgetreten, weil der Programmierer der abgeleiteten Klasse ein wichtiges Implementierungsdetail der Basisklasse nicht kannte. Um den Fehler zu beseitigen, muss lediglich in der abgeleiteten Klasse auf das Überschreiben der Methode **addAll()** verzichtet werden. Das geht solange gut, bis die Implementierung der Basisklasse sich ändert. Die abgeleitete Klasse ist von der Basisklasse abhängig und damit fragil.

Die beiden Klassen bilden (in Blochs Worten) ein Tandem, das nur gemeinsam gepflegt werden kann. Solange sich die beiden Klassen im selben Paket befinden und vom selben (gut organisierten) Team gepflegt werden, sollten sich die Probleme vermeiden lassen. Riskant ist es jedoch, eine fremde Klasse (aus einem anderen Paket) als Basisklasse zu verwenden.

Das riskante Beerben einer fremden Klasse mit unbekannten Implementationsdetails lässt sich effektiv durch eine Lösung aus den beiden folgenden Bestandteilen ersetzen, wobei sogar eine Refaktorierung (also die automatische Transformation des Quellcodes) möglich ist (Hegel & Steimann 2008):

- Komposition
  Statt eine Klasse zu beerben, verwendet man ein Member-Objekt vom Typ dieser Klasse. Man kann hier von einer *Komposition* (vgl. Abschnitt 4.7) oder von einer *Verpackung* sprechen.
- Delegation
  In der neu erstellten Klasse müssen die benötigten Methoden alle definiert werden, weil keine Vererbung stattfindet. Zur Implementation ist aber in der Regel nur ein Aufruf der entsprechenden Methode des Member-Objekts erforderlich.

Bei der vorgeschlagenen Lösung aus Komposition und Delegation ist keine Basisklasse im Spiel, die (aktuell oder in Zukunft) ...

- durch unerwünschte Methoden die neue Klasse stören könnte,
- oder unerwartete Aufrufe von überschreibenden Methoden ausführt.

Weitere Details und ein Beispiel finden sich bei Bloch (2018, S. 89ff).

[1] Das Java Collections Framework ist ein wichtiger Bestandteil im Java-API und wird im Kapitel 10 ausführlich behandelt.

Abschließend ist festzuhalten, dass die Definition einer abgeleiteten Klasse eine sinnvolle Technik zur rationellen Wiederverwendung vorhandener Software ist, sofern dabei keine Paketgrenzen überschritten werden, sodass die Abhängigkeiten zwischen der Basisklasse und der abgeleiteten Klasse bekannt und unter Kontrolle sind. Eine fremde Klasse sollte nur dann beerbt werden, wenn in deren Dokumentation die Verwendung als Basisklasse explizit unterstützt wird (siehe Abschnitt 7.10.2). Im Zweifelsfall sollte statt einer Ableitung trotz des höheren Aufwands eine Lösung aus Komposition und Delegation zum Einsatz kommen.

7.10.2 Nachteile für potentielle Basisklassen

Im bisherigen Verlauf von Abschnitt 7.10 wurden Risiken für abgeleitete Klasse behandelt und Empfehlungen zur Vermeidung gegeben. Aber auch für den Entwickler einer potentiellen Basisklasse entstehen Nachteile aus seiner Entscheidung, die Ableitung *nicht* zu verhindern:

- Um Fehler beim Ableiten (insbesondere beim Überschreiben von Methoden) zu vermeiden, müssen Implementationsdetails offengelegt werden, die nach der reinen objektorientierten Lehre eigentlich verborgen bleiben sollten, um spätere Weiterentwicklungen zu ermöglichen (Bloch 2018, S. 93f). Eine ableitbare Klasse muss in ihrer Dokumentation jede Verwendung einer überschreibbaren Methode dokumentieren. Eine offengelegte Implementation kann nicht mehr geändert werden.
- Eventuell müssen Methoden (oder sogar Felder) mit der Sichtbarkeit **protected** definiert und dokumentiert werden, um einer abgeleiteten Klasse performante Lösungen zu ermöglichen. Erneut werden Implementierungsdetails offengelegt, sodass die Weiterentwicklungsflexibilität leidet.
- Im Konstruktor der Basisklasse dürfen keine überschreibbaren Methoden aufgerufen werden, weil der Konstruktor der Basisklasse *vor* dem Konstruktor der abgeleiteten Klasse ausgeführt wird (siehe Abschnitt 7.4). Folglich würde die überschreibende Methode der abgeleiteten Klasse *vor* dem Konstruktor der abgeleiteten Klasse aufgerufen. Ein Objekt der abgeleiteten Klasse befindet sich aber erst nach dem Konstruktoraufruf in einsatzfähigem Zustand. Analoge Probleme sind möglich, wenn die Basisklasse die Schnittstellen **Cloneable** oder **Serializable** (siehe Kapitel 9) implementiert, weil es auch in diesem Zusammenhang zu Objektekreationen kommt (Bloch 2018, S. 95f).

Eine risikofrei, ohne sorgfältig zu erstellende und zu studierende Dokumentation verwendbare Basisklasse ist dadurch zu realisieren, dass auf jede interne Verwendung von überschreibbaren Methoden verzichtet wird.

Wir kennen schon die beiden in Frage kommenden Techniken, um das Ableiten einer Klasse generell zu verhindern:

- Man kann eine Klasse als **final** deklarieren (siehe Abschnitt 7.2.1)
- Man kann alle Konstruktoren als **private** deklarieren (siehe Abschnitt 7.4).

Über Konstruktoren mit der voreingestellten Sichtbarkeit (Paket) sorgt man dafür, dass sich abgeleitete Klassen nur im eigenen Paket definieren lassen.

7.11 Übungsaufgaben zum Kapitel 7

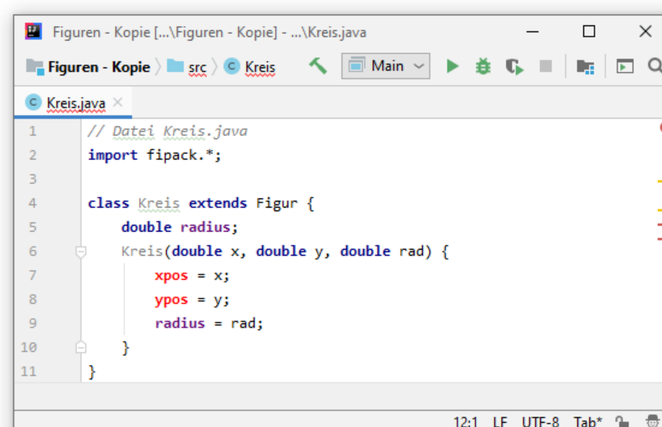
1) Warum kann der folgende Quellcode (mit zwei Klassen im Standardpaket) nicht übersetzt werden?

```
// Datei General.java
class General {
    int ig;
    General(int igp) {
        ig = igp;
    }
    void hallo() {
        System.out.println("hallo-Methode der Klasse General");
    }
}
// Datei Spezial.java
class Spezial extends General {
    int is = 3;
    void hallo() {
        System.out.println("hallo-Methode der Klasse Spezial");
    }
}
```

2) Im folgenden Beispiel wird die Klasse `Kreis` aus der Klasse `Figur` abgeleitet:

```
// Datei Figur.java
package fipack;

public class Figur {
   double xpos, ypos;
}

// Datei Kreis.java
import fipack.*;

class Kreis extends Figur {
   double radius;
   Kreis(double x, double y, double rad) {
      xpos = x;
      ypos = y;
      radius = rad;
   }
}
```

Trotzdem erlaubt der Compiler dem `Kreis`-Konstruktor keinen Zugriff auf die geerbten Instanzvariablen `xpos` und `ypos` eines neuen `Kreis`-Objekts:



Wie ist das Problem zu erklären und zu lösen?

3) Welche der folgenden Aussagen sind richtig bzw. falsch?

1. Aus einer abstrakten Klasse lassen sich keine Objekte erzeugen.
2. Aus einer abstrakten Klasse lassen sich keine Klassen ableiten.
3. In einer abstrakten Klasse müssen alle Methoden abstrakt sein.
4. Wird eine abstrakte Basisklasse beerbt, muss die abgeleitete Klasse alle abstrakten Methoden implementieren.
5. Für ein per Basisklassenreferenz ansprechbares Objekt kann zur Laufzeit über den **instanceof -** Operator festgestellt werden, ob es zu einer bestimmten abgeleiteten Klasse gehört.

4) Im Ordner

...\BspUeb\Vererbung und Polymorphie\abstract

finden Sie das Figurenbeispiel auf dem Entwicklungsstand von Abschnitt 7.8. Neben der im Manuskript diskutierten `Kreis`-Klasse ist die ebenfalls von `Figur` abgeleitete Klasse `Rechteck` vorhanden mit …

- zusätzlichen Instanzvariablen für Breite und Höhe,
- einer `wo()` - Methode, welche die geerbte `Figur`-Version überschreibt und
- einer `meldeInhalt()` - Methode, welche die abstrakte `Figur`-Version implementiert.

In der `Kreis`-Klasse ist seit Abschnitt 7.3 die Methode `abstand()` vorhanden, welche die Entfernung einer bestimmten Position vom Kreismittelpunkt liefert. Implementieren Sie diese Methode analog auch in der Klasse `Rechteck`. Damit die Methode polymorph verwendbar ist, muss sie in der Basisklasse `Figur` vorhanden sein, wobei eine Implementation aber wohl nicht sinnvoll ist. Erstellen Sie ein Testprogramm, das eine polymorphe Objektverwaltung und entsprechende Methodenaufrufe demonstriert.

5) Wird in einer Basisklasse die Implementation einer Methode verbessert, dann profitieren auch alle abgeleiteten Klassen. Was muss geschehen, damit die Objekte einer abgeleiteten Klasse bei einer geerbten Methode die verbesserte Variante benutzen?

a) Es genügt, die Basisklasse neu zu übersetzen und per Klassen- oder Modulpfad dafür zu sorgen, dass die aktualisierte Basisklasse von der JVM geladen wird.
b) Man muss sowohl die Basisklasse als auch die abgeleitete Klasse neu übersetzen.

8 Generische Klassen und Methoden

In Java haben Variablen und Methodenparameter einen festen Datentyp, sodass der Compiler die Typsicherheit garantieren, d.h. die Zuweisung von ungeeigneten Werten bzw. Objekten verhindern kann. So sorgt der Compiler für Software-Stabilität und Kunden-Zufriedenheit. Allerdings werden oft für unterschiedliche Datentypen völlig analog arbeitende Klassen oder Methoden benötigt, z. B. eine Klasse zur Verwaltung einer geordneten Liste mit Elementen eines bestimmten (bei allen Elementen identischen) Typs. Statt die Definition für jeden in Frage kommenden Elementdatentyp zu wiederholen, kann man die Klassendefinition seit Java 5 *typgenerisch* formulieren. Wird ein Objekt einer generischen Listenklasse erzeugt, ist der Elementtyp konkret festzulegen. Im Ergebnis erhält man durch *eine* Definition zahlreiche konkrete Klassen, wobei die Typsicherheit durch den Compiler überwacht wird.

Wir werden in diesem Kapitel erste Erfahrungen mit typgenerischen Klassen und Methoden sammeln. Wegen der starken Verschränkung mit noch unbehandelten Themen (z. B. Interfaces, siehe Kapitel 9) folgen später noch Ergänzungen zur Generizität.

Ein besonders erfolgreiches Anwendungsfeld für Typgenerizität sind die Klassen und Schnittstellen zur Verwaltung von Listen, Mengen oder Schlüssel-Wert - Tabellen (Abbildungen) im Java Collections Framework, das im Kapitel 10 vorgestellt wird. Auf Beispiele aus dem Bereich der Kollektionsverwaltung kann auch das aktuelle Kapitel nicht verzichten.

Weitere Details zu generischen Typen und Methoden in Java finden Sie z. B. bei Bloch[1] (2018, Kapitel 5), Bracha (2004) sowie Naftalin & Wadler (2007).

8.1 Generische Klassen

Aus der Entwicklerperspektive besteht der wesentliche Vorteil einer generischen Klasse darin, dass mit *einer* Definition beliebig viele konkrete Klassen zur Verwendung mit speziellen Datentypen geschaffen werden. Dieses Konstruktionsprinzip ist speziell bei den Kollektionsklassen sehr verbreitet (siehe Kapitel 10), aber keinesfalls auf Kollektionen mit ihrer weitgehend inhaltstypunabhängigen Verwaltungslogik beschränkt.

8.1.1 Vorzüge und Verwendung generischer Klassen

8.1.1.1 Veraltete Technik mit Risiken und Umständlichkeiten

Im Abschnitt 5.3.2 haben Sie die Klasse **ArrayList** aus dem Paket **java.util** als Container für Objekte beliebigen Typs kennengelernt, z. B.:

```
java.util.ArrayList al = new java.util.ArrayList();
al.add("Otto");
al.add(13);
al.add(23.77);
al.add('x');
```

Dabei wurde der aus Kompatibilitätsgründen auch in aktuellen Java-Versionen noch unterstützte, sogenannte *Rohtyp* der generischen Klasse **ArrayList** genutzt. Diese veraltete und verbesserungsbedürftige Praxis ist hier noch einmal zu sehen, damit gleich im Kontrast die Vorteile der korrekten Nutzung generischer Klassen deutlich werden.

Im Unterschied zu einem Java-Array (siehe Abschnitt 5.1) bietet die Klasse **ArrayList** bei der eben vorgeführten Verwendungsart:

---

[1] Joshua Bloch hat nicht nur ein lesenswertes Buch über Java verfasst (2018, *Effective Java*), sondern auch viele Klassen im Java-API programmiert und insbesondere das Java Collections Framework entworfen.

- eine automatische Größenanpassung
- Typflexibilität bzw. -beliebigkeit

In der Praxis ist aber in der Regel ein Container mit automatischer Größenanpassung für Objekte eines bestimmten, *identischen* Typs gefragt (z. B. zur Verwaltung von **String**-Objekten). Bei diesem Verwendungszweck stören zwei Nachteile der Typbeliebigkeit:

- Weil beliebige Objekte zugelassen sind, kann der Compiler keine **Typsicherheit** garantieren. Er kann nicht sicherstellen, dass ausschließlich Objekte der gewünschten Klasse in den Container eingefüllt werden. Viele Programmierfehler werden erst zur Laufzeit (womöglich vom Kunden) entdeckt.
- Aus dem Container entnommene Objekte können erst nach einer **expliziten Typumwandlung** die Methoden ihrer Klasse ausführen. Die häufig benötigten Typanpassungen sind lästig und fehleranfällig.

Im folgenden Beispielprogramm sollen **String**-Objekte in einem Container mit dem **ArrayList**-Rohtyp verwaltet werden:

```
import java.util.ArrayList;
class RawArrayList {
   public static void main(String[] args) {
      ArrayList al = new ArrayList();
      // Bitte nur String-Objekte einfüllen!
      al.add("Otto");
      al.add("Rempremerding");
      al.add('.');
      int i = 0;
      for (Object s: al)
         System.out.printf("Länge von String %d: %d\n",++i,((String)s).length());
   }
}
```

Bevor ein mutmaßliches **String**-Element des Containers nach seiner Länge befragt werden kann, ist eine lästige Typanpassung fällig, weil der Compiler nur die Typzugehörigkeit **Object** kennt, z. B.:

```
((String)s).length()
```

Beim dritten **add() -** Aufruf des Beispielprogramms wird ein **Character**-Objekt (per Autoboxing!) in den Container befördert. Weil der Container eigentlich zur Aufbewahrung von **String**-Objekten gedacht war, liegt hier ein Programmierfehler vor, den der Compiler aber wegen der mangelhaften Typsicherheit nicht verhindern kann. Beim Versuch, das **Character**-Objekt als **String**-Objekt zu behandeln, scheitert das Programm am folgenden Ausnahmefehler vom Typ **ClassCastException**:

```
Exception in thread "main" java.lang.ClassCastException:
java.base/java.lang.Character cannot be cast to java.base/java.lang.String
    at RawArrayList.main(RawArrayList.java:11)
```

8.1.1.2 Generische Klassen bringen Typsicherheit und Bequemlichkeit

Es wäre nicht schwer, eine spezielle Container-Klasse zur Verwaltung von **String**-Objekten zu definieren, welche die im letzten Abschnitt beschriebenen Probleme (mangelnde Typsicherheit, syntaktische Umständlichkeit) vermeidet. Analog funktionierende Behälter werden aber auch für andere Elementtypen benötigt, und entsprechend viele Klassen zu definieren, die sich nur durch den Inhaltstyp unterscheiden, ist nicht rationell. Für eine solche Aufgabenstellung bietet Java seit der Version 5 die *generischen* Klassen. Durch Verwendung von Typformalparametern bei der Definition wird die gesamte Handlungskompetenz einer Klasse typunabhängig formuliert. Bei jedem Instanzieren (z. B. beim Erstellen eines Container-Objekts) sind jedoch konkrete Typen anzugeben. Weil der Compiler die konkreten Typen kennt, kann er bei Zuweisungen und Methodenaufrufen für Typ-

sicherheit sorgen. Im Quellcode sind keine lästigen Typumwandlungen erforderlich; die fügt der Compiler automatisch in den Bytecode ein.

Wie ein Blick in die API-Dokumentation zeigt, ist die Klasse **ArrayList<E>** selbstverständlich generisch definiert und verwendet den Typformalparameter **E**:

Module `java.base`

Package `java.util`

Class ArrayList<E>

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>
```

Type Parameters:

E - the type of elements in this list

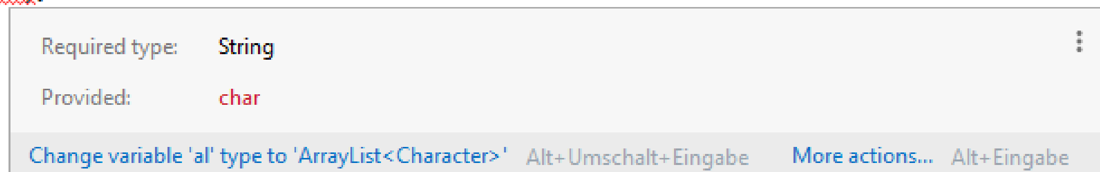
Der im Abschnitt 8.1.1.1 verwendetet Rohtyp der generischen Klasse **ArrayList<E>** ist wenig geeignet, wenn ein *sortenreiner* Container (mit *identischem* Typ für alle Elemente) benötigt wird. Die beiden Nachteile dieser Konstellation (Typunsicherheit, lästige Typanpassungen) wurden oben beschrieben.

Wird ein **ArrayList<String>** - Objekt mit Angabe des gewünschten Elementtyps (Typaktualparameters) **String** verwendet,

```
import java.util.ArrayList;
class GenArrayList {
   public static void main(String[] args) {
      ArrayList<String> al = new ArrayList<String>();
      al.add("Otto");
      al.add("Rempremerding");
      // al.add('.');  // führt zum Übersetzungsfehler
      int i = 0;
      for (String s: al)
         System.out.printf("Länge von String %d: %d\n", ++i, s.length());
   }
}
```

dann verhindert der Compiler die Aufnahme eines Elements mit unpassendem Datentyp:

```
al.add('.');
```



Außerdem beherrschen die **String**-Elemente des **ArrayList<String>** - Containers ohne Typanpassung die Methoden ihrer Klasse, z. B.:

```
s.length()
```

Durch die Verwendung generischer Klassen sind Programme ...

- robuster aufgrund der Typüberwachung durch den Compiler,
- leichter zu erstellen, weil die Typumwandlungen entfallen,
- leichter zu lesen.

Bei der Verwendung eines generischen Typs durch Wahl konkreter Datentypen an Stelle der Typformalparameter entsteht ein sogenannter **parametrisierter Typ**, z. B. **ArrayList<String>**.

Als Konkretisierung für einen Typformalparameter ist ein *Referenztyp* vorgeschrieben. Den Grund für diese Einschränkung erfahren Sie im Abschnitt 8.1.2.1. Zwar werden über Wrapper-Klassen und Auto(un)boxing auch primitive Typen unterstützt, doch ist bei einer sehr hohen Anzahl von Auto(un)boxing-Operationen mit einem relativ hohen Zeitaufwand zu rechnen.

Seit Java 7 ist es beim Instanzieren parametrisierter Typen nicht mehr erforderlich, den Typaktualparameter in der Bezeichnung des Konstruktors zu wiederholen, sodass man bei der Deklaration mit Initialisierung

```
ArrayList<String> als = new ArrayList<String>();
```

etwas Schreibaufwand sparen kann:

```
ArrayList<String> als = new ArrayList<>();
```

Aus dem deklarierten Datentyp lässt sich der Typaktualparameter sicher ableiten, und seit Java 7 können schreibfaule Programmierer von dieser Typinferenz profitieren.

Manche Autoren bezeichnen das Paar unmittelbar aufeinander folgender spitzer Klammern in laxer Redeweise als *diamond operator*, obwohl es sich *nicht* um einen Operator handelt.

8.1.2 Technische Details und Komplikationen

8.1.2.1 Typlöschung und Rohtyp

Java-Compiler erzeugen für eine generische Klasse unabhängig von der Anzahl der im Quellcode vorhandenen Konkretisierungen (parametrisierten Typen) ausschließlich den sogenannten **Rohtyp**. Hier sind die Typformalparameter jeweils durch den generellsten zulässigen Datentyp ersetzt. Bei unrestringierten Parametern ist diese *obere Begrenzung* (engl.: *upper bound*) der Urahntyp **Object**, bei restringierten Parametern ist sie entsprechend enger (siehe Abschnitt 8.1.3.2). Man spricht hier von **Typlöschung** (engl.: *type erasure*). Im Bytecode existieren also keine parametrisierten Typen, sondern nur der Rohtyp.

Während die Entwickler seit Java 5 generische Klassen erstellen und verwenden können, weiß die JVM nichts von dieser Technik. Die damit zahlreich erforderlichen Typumwandlungen fügt der Compiler automatisch in den Bytecode ein.

Weil Typformalparameter im Bytecode durch den generellsten zulässigen Datentyp, der stets ein Referenztyp ist, ersetzt werden, muss für konkretisierende Typen Zuweisungskompatibilität zu diesem Datentyp bestehen. Aus einem unrestringierten Typformalparameter resultiert im Bytecode der Typ **Object**, z. B. bei der Deklaration von Variablen. Solche Variablen können also keinen Wert mit primitivem Datentyp aufnehmen. Dies hat zur Folge, dass zur Konkretisierung von Typformalparametern nur Referenztypen erlaubt sind. Ein primitiver Typ ist also durch die zugehörige Wrapper-Klasse zu ersetzen.

Auf ihre Klassenzugehörigkeit befragt, nennen Objekte eines parametrisierten Typs stets den zugehörigen Rohtyp, z. B.:

Quellcode	Ausgabe

`import java.util.ArrayList;`
`class Prog {`
`    public static void main(String[] args) {`
`        ArrayList<String> al = new ArrayList<>();`
`        System.out.println(al.getClass());`
`    }`
`}`
	`class java.util.ArrayList`

Ist eine generische Klasse im Quellcode über ein sogenanntes *Klassenliteral* anzusprechen (Klassenname mit Suffix **.class**), dann muss der Rohtyp verwendet werden, z. B.:

```
if (al.getClass() ==  ArrayList.class)
    System.out.println("Der Rohtyp von al ist ArrayList.");
```

Die Typlöschung ist auch bei Verwendung des im Abschnitt 7.6 vorgestellten **instanceof**-Operators zu berücksichtigen, der die Zugehörigkeit eines Objekts zu einer bestimmten Klasse prüft. Er akzeptiert keine parametrisierten Typen, sodass z. B. die folgende Anweisung *nicht* übersetzt werden kann:

```
System.out.println(al instanceof ArrayList<String>);
```

Illegal generic type for instanceof

Anstelle des Rohtyps kann man auch auf den unrestringierten Wildcard-Datentyp (siehe Abschnitt 8.3.2) prüfen, was aber den Informationsgehalt der Abfrage nicht verändert:

```
System.out.println(al instanceof ArrayList<?>);
```

Von der strikten Empfehlung, den Rohtyp einer generischen Klasse im Quellcode eines Programms zu vermeiden, müssen die folgenden Ausnahmen gemacht werden:

- **import**-Deklaration, z. B.:
  `import java.util.ArrayList;`
- Klassenliteral, z. B.:
  `if (al.getClass() ==  ArrayList.class) ...`
- **instanceof**-Operator, z. B.:
  `al instanceof ArrayList`
- Namen von Konstruktoren (siehe Abschnitt 8.1.3).
- Besitzt eine generische Klasse eine statische Methode, ist beim Aufruf dieser Methode der Rohtypname zu verwenden (siehe Abschnitt 8.4.2.1).

Schließlich taucht der Rohtyp einer Klasse noch im Dateinamen mit dem Quellcode auf. Der Quellcode der Klasse **ArrayList<E>** steckt also in der Datei **ArrayList.java**.

Als die Generizität in Java eingeführt wurde, existierte die Programmiersprache bereits ca. 10 Jahre, sodass die Kooperation mit alten Java-Typen einen sehr hohen Stellenwert besaß und die aus heutiger Sicht suboptimale Designentscheidung mit Typlöschung und Rohtyp erzwungen hat. Es war z. B. unbedingt erforderlich, ein Objekt eines neu erstellten, parametrisierten Typs an eine alte Methode übergeben zu können.

Sofern man den Quellcode einer nicht-generischen Klasse besitzt, ist die Transformation in eine generische Variante ohne großen Aufwand möglich. So werden die Vorteile der generischen Programmierung genutzt, ohne die Interoperabilität mit älteren Lösungen zu verlieren.

Java-Programmierer müssen lernen, mit der „latenten Gefahr" des Rohtyps zu leben. Vor allem ist die Deklaration einer Referenzvariablen vom Rohtyp (z. B. **ArrayList**) zu unterlassen, weil ihr (versehentlich) ein Objekt eines parametrisierten Typs (z. B. **ArrayList<String>**) zugewiesen werden könnte:

```java
public static void main(String[] args) {
    ArrayList<String> alString = new ArrayList<>(5);
    ArrayList alObject = alString;
    alObject.add(13);
    System.out.println(alString.get(0).length());
}
```

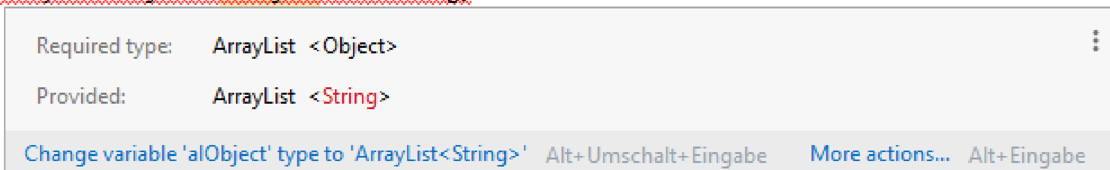
Ein Aufruf dieser **main()** - Methode führt zu einer **ClassCastException**, weil das eingeschmuggelte **Integer**-Objekt (Autoboxing!) keine **length()** - Methode beherrscht:

```
Exception in thread "main" java.lang.ClassCastException: java.base/java.lang.Integer
cannot be cast to java.base/java.lang.String
    at Prog.main(Prog.java:7)
```

Ist ausnahmsweise ein „Gemischtwaren" - Container gewünscht, sollte trotzdem *nicht der Rohtyp* verwendet werden, sondern eine Konkretisierung mit dem Elementtyp **Object**, z. B.:

```java
ArrayList<Object> alObject = new ArrayList<>();
```

Bei einer Referenzvariablen vom Typ **ArrayList<Object>** kann der eben beschriebene Fehler nicht auftreten:



Warum der parametrisierte Typ **ArrayList<String>** *keine* Spezialisierung des parametrisierten Typs **ArrayList<Object>** ist, wird gleich im Abschnitt 8.1.2.2 erläutert.

Weil man es nicht oft genug sagen kann, steht am Ende dieses Abschnitts noch einmal in den Worten von Joshua Bloch (2018, S. 117) der dringende Rat:

Don't use raw types.

8.1.2.2 Spezialisierungsbeziehungen bei parametrisierten Klassen und bei Arrays

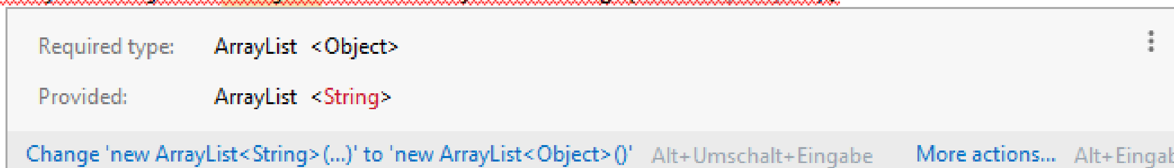
Bei der ersten Beschäftigung mit generischen Klassen könnte man z. B. den parametrisierten Typ

ArrayList<String>

für eine Spezialisierung des parametrisierten Typs

ArrayList<Object>

halten, weil schließlich die Klasse **String** eine Spezialisierung der Urahnklasse **Object** ist. Wie Sie im Kapitel 7 über Vererbung gelernt haben, können Objekte einer abgeleiteten Klasse über Referenzvariablen der Basisklasse angesprochen werden. Der Compiler verbietet jedoch, ein Objekt der Klasse **ArrayList<String>** über eine Referenzvariable vom Typ **ArrayList<Object>** anzusprechen, z. B.:



Ein Objekt der Klasse **ArrayList<Object>** kann als „Gemischtwarenladen" Objekte von beliebigem Typ aufnehmen, während in einem Objekt vom Typ **ArrayList<String>** nur **String**-Objekte zugelassen sind. Ein Objekt vom Typ **ArrayList<String>** ist also *nicht* in der Lage, den Job eines

Objekts vom Typ **ArrayList<Object>** zu übernehmen. Dies ist aber von einer abgeleiteten Klasse zu fordern (siehe Abschnitt 7.9). Die oben formulierte naive Spezialisierungsvermutung ist also *falsch*.

Hinsichtlich der Zuweisungskompatibilität in Abhängigkeit vom Elementtyp (und damit bei der Typsicherheit) besteht ein wichtiger Unterschied zwischen generischen Klassen und Arrays. Während der Compiler die Zuweisung

```
ArrayList<Object> alObject = new ArrayList<String>(5); //verboten
```

ablehnt, erlaubt er das analoge Vorgehen bei einem Array:

```
Object[] arrObject = new String[5]; // leider erlaubt
```

Für die beiden gravierend abweichenden Regeln bei der Übertragung der Spezialisierungsrelation von der Element- auf die Container-Ebene haben sich die folgenden Begriffe eingebürgert:

- Generischen Typen sind **invariant**.
- Arrays sind **kovariant**.
  Warum der statistische Terminus *Kovarianz* hier zum Einsatz kommt, wurde im Abschnitt 5.1.3 veranschaulicht.

Aufgrund der Kovarianz-Eigenschaft von Arrays übersetzt der Compiler z. B. die folgenden Anweisungen ohne jede Kritik:

```
Object[] arrObject = new String[5];
arrObject[0] = 13;
```

Zur Laufzeit kommt es jedoch zu einem Ausnahmefehler vom Typ **ArrayStoreException**:

```
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer
 at Prog.main(Prog.java:6)
```

Der per

```
new String[5]
```

erzeugte Array kennt zur Laufzeit seinen Elementtyp (**String**) und lehnt die Aufnahme eines **Integer**-Objekts ab. Weil Programmierfehler nicht zur Laufzeit, sondern vom Compiler entdeckt werden sollten, ist die bei Arrays realisierte kovariante Zuweisungskompatibilität als Mangel einzuschätzen, von dem neben Java auch andere Programmiersprachen betroffen sind (z. B. C#).

8.1.2.3 Keine Array-Kreation mit einem nicht-reifizierbaren Elementtyp

Wegen der Typlöschung bei generischen Klassen und der Kovarianz von Arrays passen in Java die Generizität und Arrays schlecht zusammen (Bloch 2018, S. 126ff). Z. B. lässt sich kein Array mit einer konkretisierten Variante des generischen Typs **ArrayList<E>** als Elementtyp erstellen:

```
ArrayList<String>[] aals = new ArrayList<String>[100];
```

Generic array creation

Genauso wenig kann ein Typformalparameter bei der Array-Kreation als Elementtyp verwendet werden, z. B.:

```
private E[] elements = new E[DEF_INIT_SIZE];
```

Type parameter 'E' cannot be instantiated directly

Bloch (2018, S. 129) empfiehlt, bei Schwierigkeiten mit der Kombination von Arrays und generischer Programmierung die Arrays durch Listen (z. B. durch Objekte aus der API-Klasse **ArrayList<E>**) zu ersetzen:

> As a rule, arrays and generics don't mix well. If you find yourself mixing them and getting compile-time errors or warnings, your first impulse should be to replace the arrays with lists.

Wie ein Blick in den Quellcode der Klasse **ArrayList<E>** zeigt, wird bei der Anwendung von Blochs Empfehlung das Problem auf die Programmierer der API-Klasse aus dem Java Collection Framework übertragen. Die Klasse **ArrayList<E>** speichert ihre Listenelemente nämlich intern in einem Array vom Typ **Object[]**.

Die im aktuellen Abschnitt beschriebene Einschränkung stört nicht bei der *Verwendung* der im Java-API und in anderen Bibliotheken zahlreich vorhandenen generischen Klassen. Bei der *Definition* einer eigenen generischen Klasse lässt sich das Problem ohne allzu großen Aufwand lösen. Die eben erwähnte Lösung aus der API-Klasse **ArrayList<E>** werden wir im Abschnitt 8.1.3 auch für ein eigenes Beispiel verwenden.

Wer momentan nicht daran interessiert ist, warum in Java die generische Programmierung und die Arrays mit etwas Aufwand zur Kooperation gebracht werden müssen, kann den aktuellen Abschnitt an dieser Stelle verlassen.

Wie das folgende Beispiel zeigt, könnte der Compiler bei einem Array mit einem parametrisierten Elementtyp nicht für Typsicherheit sorgen.[1] Er verhindert daher die Objektkreation:

```
1  import java.util.ArrayList;
2  class Prog {
3      public static void main(String[] args) {
4          ArrayList<String>[] aals = new ArrayList<String>[3];
5                                        Generic array creation
6
7          ArrayList<Integer> ali = new ArrayList<>();
8          ali.add(13);
9
10         Object[] ao = aals;
11         ao[0] = ali;
12         String s = aals[0].get(0);
13     }
14 }
```

Würde der Compiler die Objektkreation in Zeile 4 erlauben, käme es zur Laufzeit zu einer **ClassCastException**:

- Die parametrisierten Typen **ArrayList<String>** und **ArrayList<Integer>** werden zur Laufzeit durch den Rohtyp **ArrayList** ersetzt.
- Wegen der Kovarianz von Arrays ist **ArrayList[]** eine Spezialisierung des Typs **Object[]**, sodass der Compiler die Zeile 10 nicht beanstandet.
- In Zeile 11 wird ein **ArrayList<Integer>** - Objekt als Element 0 in den **Object**-Array `ao` aufgenommen, was der Compiler erlauben muss, weil hier beliebige Objekte erlaubt sind.
- Auch zur Laufzeit würde die Zeile 11 kein Problem machen (keine **ArrayStoreException** verursachen), obwohl der per `ao` angesprochene Array sehr wohl wüsste, dass seine Elemente vom Rohtyp **ArrayList** sind. Schließlich hat das eingefügte Element `ali` ja genau diesen Rohtyp.

[1] Das Beispiel wurde übernommen von Bloch (2018, S. 127) bzw. Flanagan (2005, S. 166), wo es in weitgehend identischer Form zu finden ist.

- In der Zeile 12 wird ausgenutzt, dass ein **ArrayList<String>** - Container nur Objekte vom Typ **String** enthalten kann. Genau hier käme es zur **ClassCastException**, weil das Element 0 von `aals` *kein* **ArrayList<String>** - Container, sondern ein **ArrayList<Integer>** - Container wäre.

Wegen der Kovarianz von Arrays muss ihr Elementtyp **reifizierbar** sein, d.h. zur Laufzeit darf nicht weniger Information über den Typ zur Verfügung stehen als zur Übersetzungszeit (Bloch 2018, S. 127; siehe auch Gosling et al. 2019, Abschnitt 4.7). Bei parametrisierten Typen wie **ArrayList<String>** sorgt aber die Typlöschung für eine solche Informationsreduktion. Folglich verbietet der Compiler die Array-Kreation mit einem parametrisierten Elementtyp. Generell gilt, dass kein Array mit einem nicht-reifizierbaren Elementtyp erstellt werden kann.

8.1.2.4 Serienparameter mit einem parametrisierten Typ

Mit Hilfe eines Serienparameters (siehe Abschnitt 4.3.1.4.3) gelingt es doch, einen Array mit einem parametrisierten Elementtyp anzulegen. Im folgenden Beispiel, das im Wesentlichen aus der Java-API - Online-Dokumentation[1] stammt, verwendet die Methode `genVarargs()` einen Serienparameter (also letztlich einen Array-Parameter) vom Typ **ArrayList<String>**:

```
import java.util.ArrayList;
class Prog {
    private void genVarargs(ArrayList<String>... stringLists) {
        ArrayList[] array = stringLists;
        ArrayList<Integer> iList = new ArrayList<>();
        iList.add(13);
        array[0] = iList;
        String s = stringLists[0].get(0);
    }
    public static void main(String[] args) {
        Prog p = new Prog();
        p.genVarargs(new ArrayList<String>());
    }
}
```

In der Methode **main()** wird beim Aufruf an `genVarargs()` ein Element mit dem vereinbarten parametrisierten Typ übergeben. In Methodenrumpf wird der Array mit **ArrayList<String>** - Elementen über eine **ArrayList[]** - Referenz dazu gebracht, ein Element vom Typ **ArrayList<Integer>** aufzunehmen, das den passenden Rohtyp **ArrayList** besitzt. Das in die **ArrayList<Integer>** eingefügte **Integer**-Objekt wird mit Hilfe des Referenzparameters als **String**-Objekt behandelt, was eine **ClassCastException** zur Folge hat.

Der Java-Compiler im OpenJDK übersetzt das Programm, warnt aber wegen des Serienparameters mit parametrisiertem Typ:

```
> javac Prog.java
Note: Prog.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Der kapitale Semantikfehler im Programm spielt bei der Compiler-Warnung *keine* Rolle. Die Warnung erfolgt auch bei komplett leerem Rumpf.

Weil auch nützliche und sichere Methoden möglich sind, die einen Serienparameter mit parametrisiertem Typ verwenden, haben die Java-Designer an dieser Stelle die Array-Kreation mit einem nicht-reifizierbaren Elementtyp zugelassen, die ansonsten unterbunden wird (siehe Abschnitt 8.1.2.3). Mit der seit Java 7 verfügbaren Annotation **@SafeVarargs** (siehe Abschnitt 9.5.4) versi-

[1] https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/SafeVarargs.html

chert man dem Compiler, dass eine Methode mit ihren Serienparameter von einem konkretisierten generischen Typ *keine* unsicheren Operationen ausführt, sodass der Compiler auf die Warnung verzichtet. In Java 7 und 8 ist die Annotation **@SafeVarargs** nur erlaubt bei

- finalen Methoden
- statischen Methoden
- Konstruktoren

In dieser Liste befinden sich nur Methoden, die beim Vererben nicht überschrieben werden können, denn für überschreibbare Methoden könnte kein Programmierer eine Garantie geben. Ein Überschreiben scheidet aber auch bei privaten Methoden aus, und seit Java 9 ist daher die Annotation **@SafeVarargs** auch bei privaten Methoden erlaubt. In der folgenden sinn- und harmlosen Variante des Beispielprogramms wird mit der **@SafeVarargs** - Annotation eine unbegründete Compiler-Warnung verhindert:

```
import java.util.ArrayList;
class Prog {
    // Kein Schreibzugriff auf den varargs-Array, keine Weitergabe der Adresse
    @SafeVarargs
    private void genVarargs(ArrayList<String>... stringLists) {
        System.out.println(stringLists.length);
    }
    public static void main(String[] args) {
        Prog p = new Prog();
        p.genVarargs(new ArrayList<String>());
    }
}
```

Nach Bloch (2018, S. 147) kann und sollte die **@SafeVarargs** - Annotation zur Unterdrückung von irrelevanten Warnungen verwendet werden, wenn ...

- in der Methode keine Schreibzugriffe auf den Serienparameter stattfinden,
- keine Referenz auf das Array-Objekt des Serienparameters weitergegeben wird (z. B. per Rückgabe).

8.1.3 Definition von generischen Klassen

Wie Sie im Abschnitt 8.1.1.2 feststellen konnten, ist die *Verwendung* von generischen Typen aus der Standardbibliothek sehr empfehlenswert und einfach. Aber auch die Definition von eigenen generischen Klassen ist kein großes Problem, wenngleich man dabei mit einigen technischen Details und Komplikationen der in Java realisierten Generizitätslösung in näheren Kontakt kommt.

Im Namen der Datei mit dem Quellcode einer generischen Klasse tauchen *keine* Typformalparameter auf. Z. B. steckt die Klasse **ArrayList<E>** der Java-Standardbibliothek in einer Datei mit dem Namen **ArrayList.java**.

8.1.3.1 Unbeschränkte Typformalparameter

Bei der generischen Klassendefinition verwendet man **Typformalparameter**, die im Definitionskopf hinter dem Klassennamen zwischen spitzen Klammern angegeben werden. Verwendet eine Klassendefinition *mehrere* Typformalparameter, sind diese durch Kommata voneinander zu trennen. Wir erstellen nun als einfaches Beispiel eine generische Klasse namens `SimpleList<E>`, die hinsichtlich Einsatzzweck und Konstruktion den Listenverwaltungsklassen aus dem Java Collections Framework ähnelt (z. B. **ArrayList<E>**, siehe oben und Abschnitt 10.3), aber nicht annähernd denselben Funktionsumfang bietet:

```
package de.uni_trier.zimk.util.coll;
import java.util.Arrays;

public class SimpleList<E> {
   private Object[] elements;
   private final static int DEF_INIT_SIZE = 16;
   private int initSize;
   private int size;

   public SimpleList(int len) {
      if (len > 0) {
         initSize = len;
         elements = new Object[len];
      } else {
         initSize = DEF_INIT_SIZE;
         elements = new Object[DEF_INIT_SIZE];
      }
   }

   public SimpleList() {
      initSize = DEF_INIT_SIZE;
      elements = new Object[DEF_INIT_SIZE];
   }
   public void add(E element) {
      if (size ==  elements.length)
         elements = Arrays.copyOf(elements, elements.length + initSize);
      elements[size++] = element;
   }

   public E get(int index) {
      if (index >= 0 && index < size) {
         return (E) elements[index];
      } else
         return null;
   }

   public int size() {return size;}

   public int capacity() {return elements.length;}
}
```

Innerhalb der Klassendefinition kann der Typformalparameter **E** in vielen Situationen wie ein konkreter Referenzdatentyp verwendet werden (als Typ von Instanzvariablen, lokalen Variablen, Methodenparametern und Rückgabewerten). Wie gleich zu sehen ist, bestehen allerdings Einschränkungen bei der Verwendung eines Typformalparameters.

Es wird empfohlen, für Typformalparameter einzelne Großbuchstaben zu verwenden (Bloch 2018, S. 290):

T Type
Treten *mehrere* Typen auf, können diese mit **T**, **U**, **V**, ... oder mit **T1**, **T2**, **T3**, ... benannt werden.
E Element
R Return Type
K Key
V Value
X Exception

In den Namen der Konstruktoren einer generischen Klasse werden die Typformalparameter *nicht* wiederholt, z. B.:

```
public class SimpleList<E> {
  . . .
   public SimpleList() {
      initSize = DEF_INIT_SIZE;
      elements = new Object[DEF_INIT_SIZE];
   }
  . . .
}
```

Die generische Klasse `SimpleList<E>` verwendet intern zur Ablage ihrer Elemente einen Array namens `elements`. Aufgrund der im Abschnitt 8.1.2.1 erläuterten Typlöschung kann jedoch kein Array mit Elementen vom Typ **E** erzeugt werden. Ein entsprechender Versuch führt zu einer Fehlermeldung wie im folgenden Beispiel:

```
elements = new E[DEF_INIT_SIZE];
```

Type parameter 'E' cannot be instantiated directly

Der Elementtyp des Arrays kann also leider *nicht* über den Typformalparameter bestimmt werden. Auf dieses Problem stößt man regelmäßig bei der Definition einer Kollektionsklasse, die im Hintergrund einen Array zur Datenverwaltung verwendet.

In der Definition einer generischen Klasse mit dem Typformalparameter **E** ist es aufgrund der Typlöschung auch nicht möglich, ein einzelnes Objekt vom Typ **E** zu erstellen, z. B.:

```
private E el = new E();
```

Type parameter 'E' cannot be instantiated directly

Die JVM weiß nicht, durch welche Klasse der Typformalparater **E** konkretisiert ist, und kann folglich den zugehörigen Konstruktor nicht aufrufen.

Wir müssen bei der Array-Kreation als Elementtyp den generellsten zulässigen (nichtgenerischen) Datentyp für Konkretisierungen von **E** verwenden: den Urahntyp **Object**.

Beim Datentyp für die Referenzvariable `elements` haben wir zwei, letztlich äquivalente Alternativen, die an unterschiedlichen Stellen in der Klassendefinition explizite Typumwandlungen erfordern, deren Korrektheit der Compiler *nicht* sicherstellen kann:

- **Object[]**
- **E[]**

Bei der Klasse `SimpleList<E>` wählen wir den ersten Weg. Weil also `elements` vom deklarierten Typ **Object[]** ist, muss in der Methode `get()`, die ihren Rückgabetyp per Typparameter definiert, eine Typumwandlung vorgenommen werden:

```
return (E) elements[index];
```

IntelliJ übermittelt die Warnung des Compilers vor einer ungeprüften Umwandlung:

```
return (E) elements[index];
```

Unchecked cast: 'java.lang.Object' to 'E'
Try to generify 'SimpleList.java' Alt+Umschalt+Eingabe More actions... Alt+Eingabe

Im aktuellen Beispiel kann ausgeschlossen werden, dass ein Element in den privaten Array `elements` gelangt, das nicht vom Typ **E** ist, weil die einzige Möglichkeit zum Einfügen eines Elements in der Verwendung der Methode `add()` besteht:

```
public void add(E element) {...}
```

Folglich kann bei der Typwandlung in `get()` nichts schiefgehen.

Nachweislich irrelevante Compiler-Warnungen sollten unterdrückt werden, damit wir uns nicht durch häufige unbegründete Warnungen angewöhnen, Warnungen zu ignorieren. Um den Compiler anzuweisen, eine Warnung zu unterlassen, fügt man eine sogenannte **Annotation** vom Typ **SuppressWarnings** in den Quellcode ein und gibt in Klammern den Namen der zu ignorierenden Warnung an (siehe Abschnitt 9.5). Eine Annotation vom Typ **SuppressWarnings** kann sich u.a. auf eine Variable, Methode oder Klasse beziehen, z. B.:

```
// Casting erforderlich, weil kein Array vom Typ E erstellt werden kann.
// elements kann nur Objekte vom Typ E enthalten.
@SuppressWarnings("unchecked")
public E get(int index) {
   if (index >= 0 && index < size)
      return (E) elements[index];
   else
      return null;
}
```

Das Unterdrücken von Warnungen sollte mit einem möglichst begrenzten Gültigkeitsbereich erfolgen und außerdem kommentiert werden. In der anschließend vorgestellten Lösung wird es über eine Hilfsvariable vermieden, die Unterdrückung auf die gesamte Methode zu beziehen:

```
public E get(int index) {
   if (index >= 0 && index < size) {
      // Casting erforderlich, weil kein Array vom Typ E erstellt werden kann.
      // elements kann nur Objekte vom Typ E enthalten.
      @SuppressWarnings("unchecked")
      E result = (E) elements[index];
      return result;
   } else
      return null;
}
```

Wie oben erwähnt, ist es durchaus möglich, für die Referenzvariable `elements` den Datentyp `E[]` zu verwenden und so die Typwandlung in der Methode `get()` zu vermeiden:

```
private E[] elements;
```

Allerdings muss man trotzdem einen Array vom Typ **Object[]** erzeugen, und die Typwandlung ist nun an anderer Stelle fällig, z. B.:

```
elements = (E[]) new Object[DEF_INIT_SIZE];
```

Die eben beschriebene, letztlich gleichwertige Technik wird im Abschnitt 8.1.3.2 bei einem vergleichbaren Beispiel demonstriert.

Den Rohtyp zur Klasse `SimpleList<E>` kann man sich ungefähr so vorstellen:

```
package de.uni_trier.zimk.util.coll;
import java.util.Arrays;

public class SimpleList {
   private Object[] elements;
   private static final int DEF_INIT_SIZE = 16;
   private int initSize;
   private int size;

   // Konstruktoren wie beim generischen Typ

   public void add(Object element) {
      if (size ==  elements.length)
         elements = Arrays.copyOf(elements, elements.length + initSize);
      elements[size++] = element;
   }
```

```
    public Object get(int index) {
       if (index >= 0 && index < size)
          return elements[index];
       else
          return null;
    }

    public int size() {return size;}

    public int capacity() {return elements.length;}
 }
```

Nachdem wir uns zuletzt mit Komplikationen der Generizitätslösung in Java herumschlagen mussten, können wir uns nun bei der Beschäftigung mit einigen Details der Klasse `SimpleList<E>` entspannen. Für den intern zur Datenspeicherung verwendeten Array wird als Länge der Voreinstellungswert `DEF_INIT_SIZE` oder die per Konstruktorparameter festgelegte initiale Listenlänge verwendet. In der Methode `add()` wird bei Bedarf mit Hilfe der statischen **Arrays**-Methode **copyOf()** ein größerer Array erzeugt, der die Elemente des Vorgängers übernimmt. Solange die Klasse `SimpleList<E>` keine Methode zum Löschen von Elementen bietet, müssen wir uns um eine automatische Größen*reduktion* keine Gedanken machen. Das folgende Testprogramm demonstriert u.a. die automatische Vergrößerung des privaten Arrays:

Quellcode

```
import de.uni_trier.zimk.util.coll.SimpleList;

class SimpleListTest {
   public static void main(String[] args) {
      SimpleList<String> sls = new SimpleList<>(3);
      sls.add("Otto");
      sls.add("Rempremmerding");
      System.out.println("Länge: " + sls.size() +
            ", Kapazität: " + sls.capacity());
      sls.add("Hans");
      sls.add("Brgl");
      System.out.println("Länge: "+sls.size() +
            ", Kapazität: " + sls.capacity());
      for (int i = 0; i < sls.size(); i++)
         System.out.println(sls.get(i));
   }
}
```

Ausgabe

```
Länge: 2, Kapazität: 3
Länge: 4, Kapazität: 6
Otto
Rempremmerding
Hans
Brgl
```

Die API-Klasse **HashMap<K,V>** (siehe Abschnitt 10.6), die eine Tabelle mit Schüssel-Wert - Paaren verwaltet, ist ein Beispiel für eine generische Klasse mit *zwei* Typformalparametern:

Module `java.base`

Package `java.util`

Class HashMap<K,V>

```
java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.HashMap<K,V>
```

Type Parameters:

```
K - the type of keys maintained by this map
V - the type of mapped values
```

8.1.3.2 Beschränkte Typformalparameter

Häufig muss eine generische Klasse oder Methode (siehe Abschnitt 8.2) bei den Klassen, welche einen Typparameter konkretisieren dürfen, gewisse Handlungskompetenzen voraussetzen. Soll z. B. ein generischer Container mit dem Typformalparameter **E** seine Elemente *sortieren*, muss für jede konkrete Elementklasse gefordert werden, dass sie das Interface **Comparable<E>** implementiert. Wir begegnen hier erneut dem wichtigen Begriff *Interface*, mit dem wir uns im Kapitel 9 ausführlich beschäftigen werden. Allerdings stellt der Vorgriff kein didaktisches Problem dar, weil die Forderung an eine zulässige Konkretisierungsklasse leicht mit vertrauten Begriffen zu formulieren ist: Diese muss eine Instanzmethode namens **compareTo()** mit dem folgenden Definitionskopf besitzen (hier beschrieben unter Verwendung des Typformalparameters **E**):

public int compareTo(E *vergl***)**

Das angesprochene Objekt vergleicht sich mit dem Parameterobjekt vom selben Typ.

Im Abschnitt 5.2.1.4.2 haben Sie erfahren, dass die Klasse **String** eine solche Methode besitzt, und wie **compareTo()** das Prüfergebnis über den Rückgabewert signalisiert. Damit sollte klar sein, was die Schnittstelle (das Interface) **Comparable<E>** von einer implementierenden Klasse verlangt: Objekte dieser Klasse müssen sich mit Artgenossen vergleichen können. Wie das Beispiel **Comparable<E>** zeigt, sind auch bei Schnittstellen typgenerische Varianten von großer Bedeutung.

Wir erstellen nun eine Variante der simplen Listenklasse aus Abschnitt 8.1.2.1, die neue Elemente automatisch einsortiert und daher ihren Typformalparameter auf den Datentyp **Comparable<E>** einschränkt:

```
package de.uni_trier.zimk.util.coll;
import java.util.Arrays;

public class SimpleSortedList<E extends Comparable<E>> {
   private E[] elements;
   private final static int DEF_INIT_SIZE = 16;
   private int initSize;
   private int size;

   @SuppressWarnings("unchecked")
   // Casting erforderlich, weil kein Array vom Typ E erstellt werden kann.
   // elements kann nur Objekte vom Typ E enthalten.
   public SimpleSortedList(int len) {
      if (len > 0) {
         initSize = len;
         elements = (E[]) new Comparable[len];
      } else {
         initSize = DEF_INIT_SIZE;
         elements = (E[]) new Comparable[DEF_INIT_SIZE];
      }
   }

   @SuppressWarnings("unchecked")
   // Casting erforderlich, weil kein Array vom Typ E erstellt werden kann.
   // elements kann nur Objekte vom Typ E enthalten.
   public SimpleSortedList() {
      initSize = DEF_INIT_SIZE;
      elements = (E[]) new Comparable[DEF_INIT_SIZE];
   }
```

```
      public void add(E element) {
         if (size == elements.length)
            elements = Arrays.copyOf(elements, elements.length + initSize);
         boolean inserted = false;
         for (int i = 0; i < size; i++) {
            if (element.compareTo(elements[i]) <= 0) {
               for (int j = size; j > i; j--)
                  elements[j] = elements[j-1];
               elements[i] = element;
               inserted = true;
               break;
            }
         }
         if (!inserted)
            elements[size] = element;
         size++;
      }

      public E get(int index) {
         if (index >= 0 && index < size)
            return elements[index];
         else
            return null;
      }

      public int size() {return size;}

      public int capacity() {return elements.length;}
   }
```

Bei der Formulierung von Beschränkungen für Typformalparameter wird das Schlüsselwort **extends** verwendet, das wir im Zusammenhang mit Vererbungsbeziehungen zwischen Klassen kennengelernt haben. Zum Zwecke der Typrestriktion kann hinter dem Schlüsselwort eine Basisklasse oder (wie im Beispiel) eine zu implementierende Schnittstelle angegeben werden.

Wenn (wie im Beispiel) der Typformalparameter **E** in der Beschränkungsdefinition selbst auftaucht,

```
E extends Comparable<E>
```

spricht Bloch (2018, S. 137) von einer *rekursiven Typeinschränkung*.

Wie Sie bereits wissen, kann der intern zur Datenspeicherung verwendete Array *nicht* mit dem Elementtyp **E** erzeugt werden. Zur Lösung des Problems verwaltet man die Elemente durch ein Array-Objekt mit dem generellsten zulässigen Elementtyp. Im aktuellen Beispiel der generischen Klasse `SimpleSortedList<E extends Comparable<E>>` ist dies der Interface-Datentyp **Comparable**. Im folgenden Ausdruck

```
new Comparable[DEF_INIT_SIZE]
```

wird ein **Array**-Objekt erzeugt, das Elemente aus jeder beliebigen Klasse aufnehmen kann, die das Interface **Comparable** erfüllt.

Für den restlichen Lösungsweg gibt es zwei im Wesentlichen äquivalente Techniken:

- Instanzvariable vom Typ **Comparable[]** deklarieren und in Methoden eine Typwandlung in Richtung Typformalparameter vornehmen, z. B.:

```
public class SimpleSortedList<E extends Comparable<E>> {
   private Comparable[] elements;
     .  .  .
   public SimpleSortedList() {
      initSize = DEF_INIT_SIZE;
      elements = new Comparable[DEF_INIT_SIZE];
   }
```

```
    public void add(E element) {
       if (size == elements.length)
          elements = Arrays.copyOf(elements, elements.length + initSize);
       boolean inserted = false;
       for (int i = 0; i < size; i++) {
          if (element.compareTo((E) elements[i]) <= 0) {
             for (int j = size; j > i; j--)
                elements[j] = elements[j-1];
             elements[i] = element;
             inserted = true;
             break;
          }
       }
       if (!inserted)
          elements[size] = element;
       size++;
    }

    public E get(int index) {
       if (index >= 0 && index < size)
          return (E) elements[index];
       else
          return null;
    }
 }
```

- Bei der Instanzvariablen für den internen Array den Typ des Formalparameters verwenden und auf den **Comparable**-Array eine Typwandlung anwenden, z. B.:

```
public class SimpleSortedList<E extends Comparable<E>> {
   private E[] elements;
      .  .  .
   public SimpleSortedList() {
      initSize = DEF_INIT_SIZE;
      elements = (E[]) new Comparable[DEF_INIT_SIZE];
   }
      .  .  .
   public E get(int index) {
      if (index >= 0 && index < size)
         return elements[index];
      else
         return null;
   }
}
```

Während im Abschnitt 8.1.3.1 (bei der Klasse `SimpleList<E>`) die erste Technik zum Einsatz kam (mit dem Typ **Object[]** für den internen Array), wird im aktuellen Beispiel die zweite Technik verwendet.

Wie das folgende Testprogramm zeigt, hält ein Objekt einer Konkretisierung der Klasse `SimpleSortedList<E extends Comparable<E>>` seine Elemente stets in sortiertem Zustand:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>
import de.uni_trier.zimk.util.coll.SimpleSortedList;

class SimpleSortedListTest {
  public static void main(String[] args) {
    SimpleSortedList<Integer> si = new SimpleSortedList<>(3);
    si.add(4); si.add(11); si.add(1); si.add(2);
    System.out.println("Länge: " + si.size()+
                       " Kapazität: " + si.capacity());
    for (int i = 0; i < si.size(); i++)
      System.out.println(si.get(i));
  }
}
</pre></td><td><pre>
Länge: 4 Kapazität: 6
1
2
4
11
</pre></td></tr>
</table>

Der Compiler stellt sicher, dass die Liste sortenrein bleibt:

```
si.add("Verboten!");
```

Required type: Integer
Provided: String
Change 1st parameter of method 'add' from 'E' to 'String' Alt+Umschalt+Eingabe More actions... Alt+Eingabe

Außerdem verhindert er das Konkretisieren des Typparameters durch eine Klasse, welche die Typrestriktion nicht erfüllt, z. B.:

```
SimpleSortedList<Object> s0 = new SimpleSortedList<>(3);
```

Type parameter 'java.lang.Object' is not within its bound; should implement 'java.lang.Comparable<java.lang.Object>'

Man kann für einen Typformalparameter auch *mehrere* Beschränkungen (Restriktionen) definieren, die mit dem **&**-Zeichen verknüpft werden. Im folgenden Beispiel

```
public class MultiRest<E extends SuperKlasse & Comparable<E>> {...}
```

steht `E` für einen Datentyp, der …

- von `SuperKlasse` abstammt und
- die Schnittstelle **Comparable<E>** implementiert.

In Bezug auf die Typlöschung (vgl. Abschnitt 8.1.2.1) ist zu beachten, dass sich die obere Schranke bei multiplen Restriktionen ausschließlich an der *ersten* Restriktion orientiert, sodass im letzten Beispiel der Typ `SuperKlasse` resultiert (siehe Naftalin & Wadler 2007, S. 55).

8.2 Generische Methoden

Im Vergleich zu mehreren überladenen Methoden (vgl. Abschnitt 4.3.4), die analoge Operationen mit verschiedenen Datentypen ausführen, ist *eine* generische Methode oft die bessere Lösung. Im folgenden Beispiel liefert eine statische und generische Methode das Maximum von zwei Argumenten, wobei der gemeinsame Datentyp der Argumente die Schnittstelle **Comparable<T>** (vgl. Abschnitt 8.1.3.2) erfüllen, also eine Methode **compareTo(T** *vergl***)** besitzen muss:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre><code>class Prog {
    static <T extends Comparable<T>> T max(T x, T y) {
        return x.compareTo(y) >= 0 ? x : y;
    }

    public static void main(String[] args) {
        System.out.println("String-max:\t" + max("abc", "def"));
        System.out.println("int-max:\t" + max(12, 4711));
    }
}</code></pre></td><td><pre><code>String-max: def
int-max: 4711</code></pre></td></tr>
</table>

In der Definition einer generischen Methode befindet sich unmittelbar vor dem Rückgabetyp zwischen spitzen Klammern mindestens ein Typformalparameter. Mehrere Typparameter werden durch Kommata getrennt. Sie sind als Datentypen für den Rückgabewert, für Parameter und für lokale Variablen erlaubt. Zur Formulierung von Typrestriktionen verwendet man wie bei den generischen Klassen das Schlüsselwort **extends** (siehe Beispiel, vgl. Abschnitt 8.1.3.2).

Verwendet eine Methode einer generischen Klasse einen Typparameter der Klasse als Formalparameter- oder Rückgabetyp, spricht man *nicht* von einer generischen Methode, weil keine eigenen Typparameter definiert werden, z. B. bei der Methode `add()` der im Abschnitt 8.1.3 beschriebenen Klasse `SimpleList<E>`:

```
public void add(E element) {
  . . .
}
```

Wie bei generischen Klassen sind auch bei generischen Methoden als Konkretisierung für einen Typformalparameter nur *Referenztypen* zugelassen. Zwar werden über Wrapper-Klassen und Auto(un)boxing auch primitive Typen unterstützt (siehe obiges Beispiel), doch sollte eine große Zahl von Auto(un)boxing-Operationen wegen des damit verbundenen Zeitaufwandes vermieden werden (siehe unten).

Beim Aufruf einer generischen Methode kann der Compiler fast immer aus den Datentypen der Aktualparameter die passende Konkretisierung ermitteln (Typinferenz). Daher konnte im obigen Beispiel an Stelle der kompletten Syntax

```
System.out.println("int-max:\t" + Prog.<Integer>max(12, 4711));
```

die folgende Kurzschreibweise verwendet werden:

```
System.out.println("int-max:\t" + max(12, 4711));
```

Bei seiner Bytecode-Produktion erstellt der Compiler *eine* Methode und ersetzt dabei die Typparameter jeweils durch den generellsten erlaubten Typ (z. B. **Comparable**). Eine im Quellcode mehrfach konkretisierte generische Methode landet also nur einmal im Bytecode. Die gelöschten Typkonkretisierungen werden vom Compiler durch Typumwandlungen ersetzt.

Bei generischen Methoden sind Überladungen erlaubt, auch unter Beteiligung von gewöhnlichen Methoden, z. B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Prog {
    static <T extends Comparable<T>> T max(T x, T y) {
        return x.compareTo(y) > 0 ? x : y;
    }

    static int max(int x, int y) {
        return x > y ? x : y;
    }

    public static void main(String[] args) {
        System.out.println("String-max:\t" + max("abc", "def"));
        System.out.println("int-max:\t" + max(12, 4711));
    }
}
```

</td><td>

```
String-max:    def
int-max:       4711
```

</td></tr>
</table>

Der Compiler ermittelt zu einem konkreten Aufruf die am besten passende Methode und beschwert sich in Zweifelsfällen.

Wie oben erwähnt, kann es sich lohnen, eine generische Methode durch eine Überladungsfamilie von Methoden zur Unterstützung primitiver Typen zu ergänzen, um den Zeitaufwand von Auto(un)boxing-Operationen zu vermeiden. Im folgenden Programm finden jeweils 1 Million Aufrufe einer generischen und einer regulären Methode zur Bestimmung des Maximums von zwei **int**-Werten statt:

```
class Prog {
  static final int VERGL = 1_000_000;

  static <T extends Comparable<T>> T max(T x, T y) {
    return x.compareTo(y) > 0 ? x : y;
  }

  static int imax(int x, int y) {
    return x > y ? x : y;
  }

  public static void main(String[] args) {
    long start = System.currentTimeMillis();
    for (int i = 0; i < VERGL; i++)
      max(12, 4711);
    System.out.println("Zeit für " + VERGL + " Aufrufe der generischen Methode: \t" +
                       (System.currentTimeMillis()-start));

    start = System.currentTimeMillis();
    for (int i = 0; i < VERGL; i++)
      imax(12, 4711);
    System.out.println("Zeit für "+VERGL + " Aufrufe der traditionellen Methode: \t"+
                       (System.currentTimeMillis()-start));
  }
}
```

Dabei verursacht die generische Methode einen deutlich höheren Zeitaufwand (in Millisekunden):

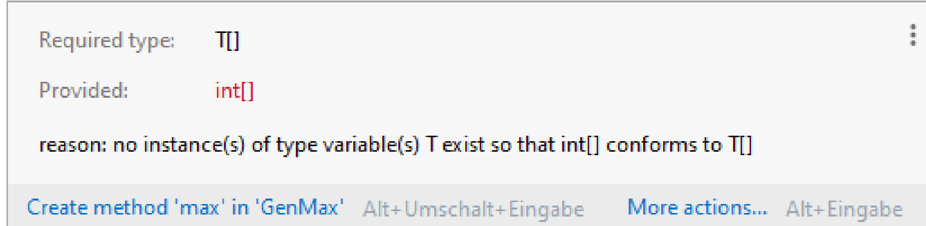
```
Zeit für 1000000 Aufrufe der generischen Methode:     16
Zeit für 1000000 Aufrufe der traditionellen Methode:  4
```

Während die generische `max()` - Methode für *zwei einzelne* Argumente dank Autoboxing auch mit primitiven Konkretisierungen arbeitet, lässt sich eine analoge generische Methode zur Bestimmung eines maximalen Array-Elements

```
static <T extends Comparable<T>> T max(T[] ar) {
    . . .
}
```

nicht für Arrays mit einem primitiven Typ nutzen. Z. B. lässt sich der folgende Aufruf mit einem Aktualparameter vom Typ **int[]** *nicht* übersetzen:

```
System.out.println("Max. von int-Serie: " + max(new int[] {4, 777, 11, 81}));
```



Der Java-Compiler nimmt *kein* Autoboxing auf Array-Ebene vor, ersetzt also z. B. keinesfalls **int[]** durch **Integer[]**. Genau das wäre zur Nutzung der generischen Methode aber erforderlich, weil der Typformalparameter nur durch Referenztypen konkretisiert werden darf. Soll eine Array-Max - Methode auch Arrays mit primitivem Elementtypen unterstützen, muss man entsprechende Überladungen erstellen.

Im letzten Beispiel kann man sich durch die explizite Verwendung eines **Integer[]** -Arrays helfen:

```
System.out.println("Max. von int-Serie: " + max(new Integer[] {4, 777, 11, 81}));
```

Bei einer generischen Methode, die das maximale Element zu einer beliebig langen Serie von Argumenten zurückgibt,

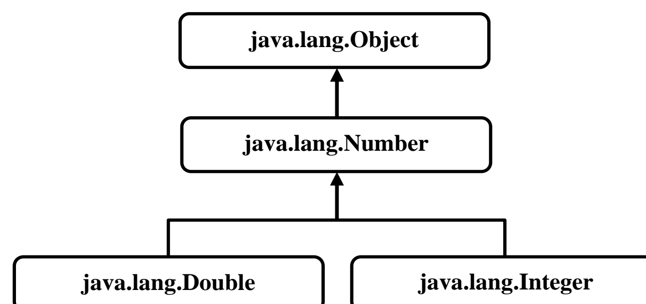
```
public static <T extends Comparable<T>> T max(T... ar) {
  . . .
}
```

klappt das Autoboxing und damit die Nutzung durch Argumente mit primitivem Typ, z. B.:

```
System.out.println("Max. von int-Serie: " + max(4, 777, 11, 81));
```

8.3 Wildcard-Datentypen

Generische Klassen sind invariant (vgl. Abschnitt 8.1.2.2), sodass z. B. der parametrisierte Datentyp `SimpleList<Integer>` *keine* Spezialisierung des parametrisierten Typs `SimpleList<Number>` ist, obwohl die numerischen Verpackungsklassen **Integer**, **Double** etc. (vgl. Abschnitt 5.3) von der Klasse **Number** abstammen:



Folglich ist z. B. bei einem Methodenformalparameter vom Typ `SimpleList<Number>` als Aktualparameter keine Referenz auf ein Objekt vom Typ `SimpleList<Integer>` zugelassen.

Es ist jedoch oft wünschenswert, für einen Methodenparameter einen generischen Datentyp zu vereinbaren und dabei *unterschiedliche* (geeignet restringierte) Konkretisierungen der Typformalparameter zu erlauben. Genau dies ermöglicht Java über die mit Hilfe eines Fragezeichens definierten Wildcard-Datentypen.

Dem folgenden *unbeschränkten* Wildcard-Typ

```
SimpleList<?>
```

genügt *jede* Konkretisierung der generischen Klasse `SimpleList<E>`. Verwendet eine Methode diesen Wildcard-Typ für einen Formalparameter, kann als Aktualparameter ein Objekt aus einer beliebigen Konkretisierung von `SimpleList<E>` übergeben werden. Weil der Compiler den Typ der Elemente nicht kennt, kann man allerdings über einen Parameter mit dem unbeschränkten Wildcard-Typ mit den Elementen nur das tun, was mit *jedem* Objekt geht (siehe Abschnitt 8.3.2).

Häufiger als der unbeschränkte Wildcard-Datentyp wird die *beschränkte* Variante benötigt, wobei z. B. als Konkretisierungen für einen Typformalparameter eine Basisklasse und deren Ableitungen erlaubt sind. Mit diesem praxisrelevanten Fall werden wir uns zuerst beschäftigen.

Wir halten fest, dass es sich bei den Wildcard-Typen um spezielle, partiell offene parametrisierte Datentypen handelt, die hauptsächlich bei Methodendefinitionen (aber nicht nur dort) Verwendung finden.

8.3.1 Beschränkte Wildcard-Typen

8.3.1.1 Beschränkung nach oben

Unsere generische Beispielklasse `SimpleList<E>` aus Abschnitt 8.1.2.1 soll um eine Methode `addList()` erweitert werden, sodass die angesprochene Liste alle Elemente einer zweiten, typkompatiblen Liste übernehmen kann. Wir starten mit der folgenden Definition, die sich bald als unpraktisch einschränkend herausstellen wird:

```
public void addList(SimpleList<E> list) {
   if (size + list.size > elements.length)
      elements = Arrays.copyOf(elements, size + list.size + initSize);
   for (int i = 0; i < list.size; i++)
      elements[size++] = list.get(i);
}
```

In einem Testprogramm erzeugen wir ein Listenobjekt mit dem parametrisierten Typ `SimpleList<Number>`:

```
SimpleList<Number> sln = new SimpleList<>(3);
```

Bei der Einzelelementaufnahme über die Methode

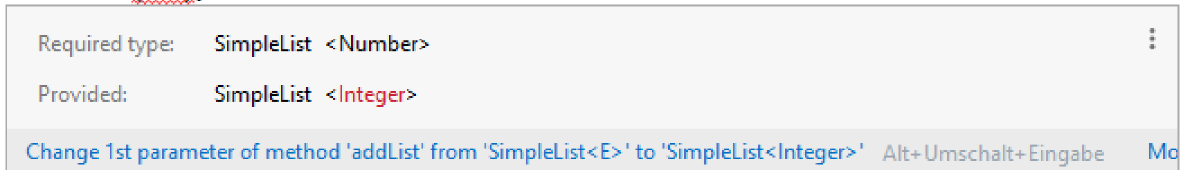
```
public void add(E element) {
  . . .
}
```

sind Objekte der Klasse **Number** und Objekte einer beliebigen abgeleiteten Klasse erlaubt, z. B.:

```
sln.add(13); sln.add(1.13);
```

Demgegenüber scheitert der Versuch, über die eben definierte Methode `addList()` alle Elemente eines `SimpleList<Integer>` - Objekts aufzunehmen:

```
SimpleList<Number> sln = new SimpleList<>(3);
sln.add(13); sln.add(1.13);
SimpleList<Integer> sli = new SimpleList<>(3);
sli.add(101); sli.add(102); sli.add(103);
sln.addList(sli);
```



Aufgrund der Invarianz generischer Klassen ist `SimpleList<Integer>` *keine* Spezialisierung von `SimpleList<Number>`.

Das Problem ist mit einem durch den Typformalparameter *nach oben beschränkten Wildcard-Datentyp* (engl.: *upper bound*) für den Parameter der Methode `addList()` zu lösen, wobei `SimpleList<E>` - Konkretisierungen mit dem Typ **E** oder einer Ableitung von **E** erlaubt werden:

```
public void addList(SimpleList<? extends E> list) {
   . . .
}
```

Mit der verbesserten Methode kann eine **Integer**-Liste komplett in eine **Number**-Liste aufgenommen werden, was im folgenden Programm demonstriert wird:

Quellcode

```
import de.uni_trier.zimk.util.coll.SimpleList;

class SimpleListWildcardTest {
   public static void main(String[] args) {
      SimpleList<Number> sln = new SimpleList<>(3);
      sln.add(13); sln.add(1.13);
      SimpleList<Integer> sli = new SimpleList<>(3);
      sli.add(101); sli.add(102); sli.add(103);
      sln.addList(sli);

      System.out.println("Element\tTyp");
      for (int i=0; i < sln.size(); i++)
         System.out.println(sln.get(i)+
                        "  \t"+sln.get(i).getClass().getName());
   }
}
```

Ausgabe

```
Element  Typ
13       java.lang.Integer
1.13     java.lang.Double
101      java.lang.Integer
102      java.lang.Integer
103      java.lang.Integer
```

8.3.1.2 Beschränkung nach unten

Neben der eben vorgestellten Beschränkung nach oben über das Schlüsselwort **extends** erlaubt Java auch die (seltener benötigte) Beschränkung *nach unten* über das Schlüsselwort **super**, wobei zur Typparameter-Konkretisierung eine bestimmte Klasse und ihre sämtlichen (auf verschiedene Ebenen angesiedelten) Basisklassen bis hinauf zur Urahnklasse **Object** zugelassen sind (engl.: *lower bound*). Zur Illustration der Beschränkung nach oben erweitern wir die generische Klasse `SimpleList<E>` um eine Methode namens `copyElements()`, welche die angesprochene Liste auffordert, ihre Elemente in eine per Parameter benannte Liste zu kopieren. Die folgende Definition

```
public void copyElements(SimpleList<E> list) {
   for (int i = 0; i < size; i++)
      list.add((E) elements[i]);
}
```

ist wenig nützlich, weil die Abnehmerliste exakt vom selben Typ wie die Lieferantenliste sein muss. Es kann z. B. aber durchaus sinnvoll sein, die Elemente eines `SimpleList<Integer>` - Objekts in ein `SimpleList<Number>` - Objekt zu kopieren. Selbst der Abnehmertyp `SimpleList<Object>` kommt in Frage. So sieht die sinnvolle Implementierung der Methode `copyElements()` aus:

```
public void copyElements(SimpleList<? super E> list) {
   for (int i = 0; i < size; i++)
      list.add((E) elements[i]);
}
```

Nach einer von Bloch (2018, S. 141) angegebenen Merkregel ...

- ist ein Wildcard-Typ mit **extends** - Restriktion für Eingabeparameter zu verwenden, die einen *Produzenten* repräsentieren,
- ist ein Wildcard-Typ mit **super** - Restriktion für Ausgabeparameter zu verwenden, die einen *Konsumenten* repräsentieren,
- ist ein einfacher Typformalparameter (ohne Wildcard) zu verwenden, wenn das durch einen Parameter repräsentierte Objekt sowohl Produzent als auch Konsument sein kann.

8.3.2 Unbeschränkte Wildcard-Typen

Um bei einer Variablen- oder Parameterdeklaration unter Verwendung einer generischen Klasse für einen Typformalparameter beliebige Konkretisierungen zu erlauben, verwendet man den ungebundenen Wildcard-Typ. Als Beispiel betrachten wird die statische Methode **reverse()** der API-Klasse **Collections** im Paket **java.util** (siehe Abschnitt 10.9), welche für die per Aktualparameter angegebene Liste die Reihenfolge der Elemente umkehrt:

```
public static void reverse(List<?> list) {
  . . .
}
```

Als Datentyp für den Aktualparameter ist *jede* Konkretisierung von **List<E>** erlaubt, z. B. **List<String>**, **List<Object>**, usw. Weil der Compiler über den Typ der Elemente nichts weiß, kann man über den Parameter mit dem unbeschränkten Wildcard-Typ **List<?>** nicht allzu viel anstellen und insbesondere keine Elemente (außer **null**) in die Liste einfügen.

8.3.3 Verwendungszwecke für Wildcard-Datentypen

Bisher sind uns (beschränkte) Wildcard-Datentypen als Methodenformalparameter begegnet, und in dieser Rolle werden sie auch am häufigsten benötigt. Sie bewähren sich aber auch bei der Deklaration von Typformalparametern. In der API-Klasse **Collections** aus dem Paket **java.util** (siehe Abschnitt 10.9) findet sich z. B. die generische und statische Methode **max()**, die für eine Kollektion mit geordneten Elementen das größte Element ermittelt:

public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> *coll*)

Wozu die erste, scheinbar überflüssige Restriktion (**T extends Object**) für den Typformalparameter **T** dient, wird später im Zusammenhang mit der Klasse **Collections** erklärt (siehe Abschnitt 10.9). Mit der zweiten Restriktion (**T extends Comparable<? super T>**) wird vom Typ **T** eine Methode **compareTo()** verlangt, wobei **T** selbst oder eine Basisklasse von **T** als Parametertyp erlaubt sind. Damit ist insgesamt als **T**-Konkretisierung auch eine Klasse möglich, welche die Methode **compareTo()** nicht selbst implementiert, sondern von einer Basisklasse erbt. Schließlich ist die Vergleichbarkeit mit Artgenossen auf diese Weise sichergestellt.

Nur selten verwendet man Wildcard-Datentypen für lokale Variablen und Felder. Als Rückgabetyp von Methoden sind sie zwar erlaubt, aber *nicht* empfehlenswert, weil die Benutzung einer derartigen Methode zur Verwendung eines Wildcard-Datentyps zwingen würde (Bloch 2018, S. 142).

8.4 Einschränkungen der Generizitätslösung in Java

Generische Klassen, Schnittstellen und Methoden sind für die professionelle Software-Entwicklung in Java unverzichtbare Bestandteile, doch enthält die Java-Generizitätslösung (hauptsächlich bedingt durch das Bemühen um Kompatibilität mit älteren Lösungen) einige Einschränkungen, die abschließend noch einmal angesprochen werden sollen.

8.4.1 Konkretisierung von Typformalparametern nur durch Referenztypen

Als Konkretisierung für einen Typformalparameter kommt nur ein *Referenztyp* in Frage. Dank Wrapper-Klassen und Auto(un)boxing lassen sich zwar auch primitive Werte ohne großen syntaktischen Aufwand versorgen, doch ist bei einer großen Zahl von Auto(un)boxing-Operationen mit einem relativ hohen Zeitaufwand zu rechnen. Sollen z. B. ganze Zahlen in einem Objekt der Klasse `SimpleList<E>` (vgl. Abschnitt 8.1.3.1) abgelegt werden, scheidet der folgenden Ansatz aus:

~~`SimpleList<int> si = new SimpleList<>();`~~

Als Ersatzlösung ist zu verwenden:

```
SimpleList<Integer> si = new SimpleList<>();
```

Im Abschnitt 8.2 wurde durch ein Beispielprogramm demonstriert, welchen zeitlichen Mehraufwand die Unterstützung primitiver Aktualparameter durch eine generische Methode im Vergleich zu einer für den primitiven Typ speziell erstellten Methodenüberladung zur Folge hat. Um 12 Millisekunden bei der Unterstützung des Parametertyps **int** durch eine generische Methode zu vergeuden, waren allerdings 1 Million Methodenaufrufe erforderlich. Es lohnt sich also nur in Ausnahmefällen, spezifische Lösungen für primitive Typen (ergänzend zu einer generischen Lösung) zu erstellen, um die Performanz einer Anwendung zu verbessern.

8.4.2 Typlöschung und die Folgen

Eine generische Klasse ist unabhängig von der Anzahl der im Quellcode vorhandenen Konkretisierungen (parametrisierten Typen) im Bytecode nur durch ihren Rohtyp vertreten.

8.4.2.1 Keine Typparameter bei der Definition von statischen Mitgliedern

Weil alle parametrisierten Typen dieselben statischen Variablen und Methoden der Klasse verwenden, darf bei der Deklaration von *statischen* Feldern oder der Definition von *statischen* Methoden kein Typparameter verwendet werden. Eigenständig generische Methoden sind in generischen Klassen hingegen erlaubt, z. B.:

```
class Gent<E> {
    static <T extends Comparable<T>> T max(T x, T y) {
        return x.compareTo(y) >= 0 ? x : y;
    }
  . . .
}
class Prog {
    public static void main(String[] args) {
        System.out.println("String-max:\t" + Gent.max("abc", "def"));
    }
}
```

Wie das Beispiel zeigt, richtet man sich beim Aufruf einer statischen Methode in einer generischen Klasse an den Rohtyp.

8.4.2.2 *Keine Kreation von Objekten aus einer per Typformalparameter bestimmten Klasse*

Weil zur Laufzeit alle Typformalparameter durch ihre obere Schranke (z. B. **Object**) ersetzt sind, kann der Typ eines zu erzeugenden Objekts nicht über Typformalparameter festgelegt werden. Wird z. B. eine generische Klasse unter Verwendung des Typformalparameters **E** definiert, lässt sich in der Klassendefinition kein Objekt vom Typ **E** erzeugen:

```
private E el = new E();
```

```
Type parameter 'E' cannot be instantiated directly
```

Die JVM weiß schlicht nicht, welchen Konstruktor sie aufrufen soll.

Damit lässt sich natürlich auch kein Array mit einem generisch bestimmten Elementtyp erstellen, was bei Kollektionsklassen mit interner Array-Datenablage zu Lücken in der vom Compiler garantierten Typsicherheit führt. Davon ist aber nur die *Definition* einer generischen Klasse betroffen, nicht ihre *Verwendung*. Bei der Definition kann und muss der Entwickler durch gerechtfertigte Typumwandlungen für *generische und stabile* Klassen sorgen. Im Abschnitt 8.1.3.1 haben wir die generische Kollektionsklasse `SimpleList<E>` definiert und zur internen Verwaltung der Listenelemente einen **Object**-Array verwendet:

```
private Object[] elements;
private final static int DEF_INIT_SIZE = 16;
  . . .
public SimpleList() {
   initSize = DEF_INIT_SIZE;
   elements = new Object[DEF_INIT_SIZE];
}
```

In der `SimpleList<E>` - Methode `get()`, die ihren Rückgabetyp per Typparameter definiert, war daher eine explizite Typumwandlung erforderlich:

```
public E get(int index) {
   if (index >= 0 && index < size) {
      // Casting erforderlich, weil kein Array vom Typ E erstellt werden kann.
      // elements kann nur Objekte vom Typ E enthalten.
      @SuppressWarnings("unchecked")
      E result = (E) elements[index];
      return result;
   } else
      return null;
}
```

Weil im Rohtyp der Typformalparameter durch die obere Schranke **Object** ersetzt ist, muss durch eine explizite Typumwandlung unter der Verantwortung des Klassendesigners dafür gesorgt werden, dass die Methode `get()` eine Referenz vom erwarteten Typ abliefert. Der Compiler macht mit einer **unchecked**-Warnung darauf aufmerksam, dass er keine Kontrollmöglichkeit hat. Im Beispiel `SimpleList<E>` haben wir die Typsicherheit durch Sorgfalt beim Klassendesign hergestellt und die somit irrelevante Warnung unterdrückt (siehe Abschnitt 8.1.3.1).

Die bei `SimpleList<E>` benutzte Lösung wird übrigens auch bei der API-Klasse **ArrayList<E>** verwendet, z. B.:[1]

---

[1] Mit dem per **implements**-Schlüsselwort zum Ausdruck gebrachten Implementieren von Schnittstellen beschäftigen wir uns im Kapitel 9, und den Modifikator **transient** lernen wir im Zusammenhang mit der Serialisierung kennen. Warum die bis Java 7 in der **ArrayList<E>** - Klassendefinition verwendete strikte Datenkapselung

```
private transient Object[] elementData;
```

seit Java 8 durch die voreingestellte Sichtbarkeit *Paket* ersetzt werden konnte, bleibt noch zu klären.

```
public class ArrayList<E> extends AbstractList<E>
      implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

   transient Object[] elementData;
   private int size;
  . . .
   @SuppressWarnings("unchecked")
   E elementData(int index) {
      return (E) elementData[index];
   }

   public E get(int index) {
      Objects.checkIndex(index, size);
      return elementData(index);
   }
  . . .
}
```

In der Methode `elementData()`, die denselben Namen trägt wie der intern zum Speichern der Elemente verwendete **Object[]** - Array, findet eine explizite Typwandlung statt, und die Compiler-Warnung wird per Annotation unterdrückt.

Die aus Kompatibilitätsgründen gewählte Typlöschung ist als Schwachstelle bei der Generizitätslösung in Java zu kritisieren. Bei der *Verwendung* generischer Klassen überwacht der Java-Compiler die Typsicherheit. Beim *Klassendesign* ist der Programmierer für die Typsicherheit verantwortlich.

8.4.2.3 Verwendung einer per Typformalparameter konkretisierten generischen Klasse

In einer generischen Klasse oder Methode kann kein Objekt einer per Typformalparameter bestimmten Klasse erstellt werden, weil die JVM die Klasse nicht kennt und folglich nicht weiß, welcher Konstruktor aufzurufen ist. Es ist aber möglich, ein Objekt einer per Typformalparameter konkretisierten generischen Klasse zu erstellen. Zu dieser Leistung ist die JVM trotz Typlöschung fähig, weil sie nur ein Objekt des Rohtyps zu erstellen hat.

Im folgenden Beispiel verwendet die generische Klasse `Genni<E>` intern ein Objekt der Klasse **ArrayList<E>**, das sie problemlos unter Verwendung des Typformalparameters erzeugen kann:

```
import java.util.ArrayList;

public class Genni<E> {
   private ArrayList<E> ale = new ArrayList<E>();

   public void add(E e) {
      ale.add(e);
   }

   public ArrayList<E> get() {
      return ale;
   }
}
```

Im folgenden Testprogramm wird ein Objekt vom parametrisierten Typ `Genni<String>` erstellt:

Quellcode

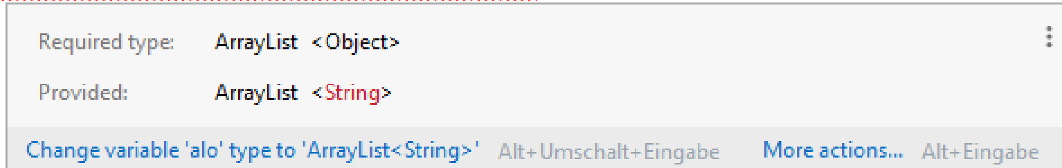
```
public class GenniTest {
   public static void main(String[] args) {
      Genni<String> gs = new Genni<>();
      gs.add("Otto"); gs.add("Rempremerding");
//    gs.add(13); // wird vom Compiler abgelehnt
      System.out.println(gs.get());
   }
}
```

Ausgabe

```
[Otto, Rempremerding]
```

Der Compiler kann die Typsicherheit garantieren, z. B.:

```
java.util.ArrayList<Object> alo = gs.get();
```



8.5 *Übungsaufgaben zum Kapitel 8*

1) In der folgenden Klassendefinition ist eine statische Methode namens `printAll()` vorhanden, die für Listen mit beliebigem festen Elementtyp alle Elemente ausgibt und dazu einen Formalparameter mit Wildcard-Datentyp verwendet:

Quellcode

```
import java.util.*;
class Prog {
    static void printAll(List<?> list) {
        for (Object e : list)
            System.out.println(e);
    }

    public static void main(String[] args) {
        ArrayList<String> als = new ArrayList<>();
        als.add("Otto"); als.add("Ludwig"); als.add("Karl");
        printAll(als);
    }
}
```

Ausgabe

```
Otto
Ludwig
Karl
```

Erstellen Sie eine funktionsgleiche generische Methode mit einem Typformalparameter.

2) Das folgende, bei Bloch (2018, S. 119) gefundene, Programm wird fehlerfrei übersetzt:

```
import java.util.ArrayList;
class Prog {
    private static void addElement(ArrayList list, Object o) {
        list.add(o);
    }

    public static void main(String[] args) {
        ArrayList<String> sl = new ArrayList<>();
        addElement(sl, 42);
        System.out.println(sl.get(0));
    }
}
```

Zur Laufzeit scheitert es mit der Meldung:

```
Exception in thread "main" java.lang.ClassCastException: java.base/java.lang.Integer
cannot be cast to java.base/java.lang.String
    at Prog.main(Prog.java:10)
```

Wie ist der Fehler zu erklären?

3) Warum sollte man als Datentyp für „Gemischtwaren“ - Kollektionsobjekte die Parametrisierung mit dem Elementtyp **Object** (z. B. **ArrayList<Object>**) gegenüber dem Rohtyp (z. B. **ArrayList**) bevorzugen?

9 Interfaces

Ein Interface (dt.: eine *Schnittstelle*) kann in erster Näherung als Referenzdatentyp mit ausschließlich abstrakten Methodendefinitionen beschrieben werden. Wenn eine instanzierbare (nicht abstrakte) Klasse von sich behaupten möchte, ein Interface zu implementieren, muss sie für jede abstrakte Methoden im Interface eine konkrete Realisation (mit Methodenrumpf) besitzen. Objekte einer implementierenden Klasse werden vom Compiler überall dort akzeptiert, wo für eine Referenzvariable oder für einen Referenzparameter das Interface als Datentyp vorgeschrieben ist. Auf diese Weise erhöhen Interface-Datentypen die Flexibilität bei der Software-Entwicklung:

- Die lose Kopplung von Komponenten wird unterstützt (siehe z. B. Abschnitt 6.2.1.4).
- Polymorphie ist nicht mehr von einer gemeinsamen Basisklasse abhängig, sondern kann sehr viel flexibler über ein gemeinsam implementiertes Interface realisiert werden.

Mehr als 1/3 aller Klassen im Java SE - API implementieren mindestens ein Interface.

9.1 Überblick

Zunächst wird an einem Beispiel erläutert, was wir über eine Klasse durch die Liste der von ihr implementierten Interfaces erfahren. Dann beschäftigen wir uns mit dem primären Verwendungszweck von Schnittstellen und mit den möglichen Bestandteilen einer Schnittstellendefinition.

9.1.1 Beispiel

Wer das Manuskript mit seinen zahlreichen, meist unvermeidlichen Vorgriffen auf das aktuelle Kapitel aufmerksam gelesen hat, wird sich wohl kaum noch fragen müssen, was mit den *Implemented Interfaces* gemeint ist, die in der Dokumentation zu zahlreichen API-Klassen an prominenter Stelle angegeben werden, z. B. zur Wrapper-Klasse **java.lang.Double** (vgl. Abschnitt 5.3):

Module java.base

Package java.lang

Class Double

```
java.lang.Object
    java.lang.Number
        java.lang.Double
```

All Implemented Interfaces:

Serializable, Comparable<Double>

Im konkreten Fall erfährt man, dass die Klasse **Double** zwei Interfaces implementiert:

- **Serializable**
  Weil die Klasse **Double** das Interface **Serializable** im Paket **java.io** implementiert, können **Double**-Objekte auf bequeme Weise in eine Datei gespeichert und von dort eingelesen werden. Diese (bei komplexeren Klassen beeindruckende) Option werden wir im Kapitel 14 über die Ein- und Ausgabe kennenlernen.
- **Comparable<Double>**
  Analog zu generischen Klassen (vgl. Abschnitt 8.1) unterstützt Java seit der Version 5 auch Interfaces mit Typparametern (z. B. **Comparable<T>**). Weil die Klasse **java.lang.Double** das parametrisierte Interface **Comparable<Double>** im Paket **java.lang** implementiert, ist für ihre Objekte eine Anordnung definiert. Das hat z. B. zur Folge, dass die Objekte in einem **Double**-Array mit der statischen Methode **java.util.Arrays.sort()** sortiert werden können, z. B.:

```
Double[] da = {15.3, 4.0, 78.1, 12.9};
java.util.Arrays.sort(da);
```

Um das parametrisierte Interface **Comparable<Double>** zu implementieren, muss die Klasse **Double** eine Methode mit folgendem Definitionskopf besitzen:

public int compareTo(Double *d*)

Wie Sie aus dem Abschnitt 5.2.1 wissen, beherrscht auch die Klasse **String** eine analoge Methode. Das Beispiel der Klasse **String** lehrt uns, dass eine „vernünftige“ **compareTo()** -Realisation keinen beliebigen **int**-Wert abliefern darf, sondern das Vergleichsergebnis folgendermaßen mitteilen muss:

- Wenn das angesprochene Objekt in der Anordnung *vor* dem Parameterobjekt steht (kleiner ist), wird ein negativer Rückgabewert geliefert (meist -1).
- Wenn beide hinsichtlich der Anordnung gleich sind, wird die Rückgabe 0 geliefert.
- Wenn das angesprochene Objekt in der Anordnung *hinter* dem Parameterobjekt steht (größer ist), wird ein positiver Rückgabewert geliefert (meist 1).

Weitere Details zum Vertrag der Methode **compareTo()** liefert Bloch (2018, S. 66ff).

9.1.2 Primärer Verwendungszweck

Ein Interface (dt.: eine *Schnittstelle*) dient in der Regel dazu, Verhaltenskompetenzen von Objekten über eine Liste von abstrakten Instanzmethoden zu definieren. Seit Java 8 können Interface-Designer zu einer Instanzmethode aber auch eine **default**-Implementierung mitliefern. Wenn sich eine Klasse zu einem Interface bekennt, gibt sie eine **Verpflichtungserklärung** ab und muss alle im Interface beschriebenen Instanzmethoden implementieren, falls kein glücklicher Umstand die eigene Methodendefinition erübrigt:

- Im Interface ist eine aus Sicht der Klasse akzeptable **default**-Implementierung vorhanden.
- Es wird eine Implementierung von einer Basisklasse geerbt.

Die mit einer Schnittstelle verbundene Verpflichtungserklärung ist in der Regel durch die Definitionsköpfe der abstrakten Methoden *nicht* erschöpfend definiert. Meist beschreibt der Schnittstellendesigner in der begleitenden Dokumentation das geforderte Verhalten der Methoden (siehe Beispiel **Comparable<T>** im letzten Abschnitt).

Wenn sich eine Klasse zu einem Interface bekennt und die daraus resultierenden Verpflichtungen erfüllt, wird ihr vom Compiler die Eignung für den **Datentyp der Schnittstelle** zuerkannt. Es lassen sich zwar keine Objekte von einem Interface-Datentyp erzeugen, aber *Referenzvariablen* von diesem Typ sind erlaubt und als Abstraktionsmittel sehr nützlich. Sie dürfen auf Objekte aus allen Klassen zeigen, welche die Schnittstelle implementieren. Somit können Objekte unabhängig von den Vererbungsbeziehungen ihrer Typen gemeinsam verwaltet werden, wobei Methodenaufrufe polymorph erfolgen (d.h. mit später bzw. dynamischer Bindung, siehe Abschnitt 7.7).

Implementiert eine Klasse ein Interface, dann …

- muss sie die im Interface enthaltenen und nicht mit einer **default**-Implementierung ausgestatteten Instanzmethoden definieren (oder erben), wenn keine abstrakte Klasse entstehen soll (vgl. Abschnitt 7.8),
- werden Variablen mit dem Typ dieser Klasse vom Compiler überall dort akzeptiert, wo der Interface-Datentyp vorgeschrieben ist.

Im Programmieralltag kommen wir auf unterschiedliche Weise mit Schnittstellen in Kontakt, z. B.:

- Verwendung von vorhandenen Schnittstellen als Datentypen
  In einer Methodendefinition kann es sinnvoll sein, Parameterdatentypen über Schnittstellen festzulegen. In den Anweisungen der Methode werden Verhaltenskompetenzen der Parameterobjekte genutzt, die durch Schnittstellenverpflichtungen garantiert sind. Damit wird die Typsicherheit ohne überflüssige Einengung erreicht.
  Beispiel: Wenn man als Datentyp für eine Zeichenfolge das Interface **CharSequence** angibt, kann der Methode beim Aufruf alternativ ein Objekt aus den implementierenden Klassen **String**, **StringBuilder** oder **StringBuffer** übergeben werden (siehe Abschnitt 9.4).
  Sind bei der Definition einer generischen Klasse für einen beschränkten Typformalparameter bestimmte Verhaltenskompetenzen zu fordern, gelingt das oft am besten per Schnittstellendatentyp (siehe Abschnitt 8.1.3.2).
- Implementierung von vorhandenen Schnittstellen in einer eigenen Klassendefinition
  Damit werden Variablen dieses Typs vom Compiler überall dort akzeptiert (z. B. als Aktualparameter), wo die jeweiligen Schnittstellenkompetenzen gefordert sind.
  Beispiel: Wenn unser Klasse `Bruch` (siehe z. B. Abschnitt 4.1.3) das Interface **Comparable<`Bruch`>** implementiert, dann können wir die bequeme Methode **Arrays.sort()** verwenden, um einen Array mit `Bruch`-Objekten zu sortieren.
- Definition von eigenen Schnittstellen
  Beim Entwurf eines Softwaresystems, das als Halbfertigprodukt (oder Programmgerüst) für verschiedene Aufgabenstellungen durch spezielle Klassen mit bestimmten Verhaltenskompetenzen zu einem lauffähigen Programm komplettiert werden soll, definiert man eigene Schnittstellen, um die Interoperabilität der Klassen sicherzustellen. In diesem Fall spricht man von einem **Framework** (z. B. Java Collections Framework, Java Persistence Framework). Auch bei einem **Entwurfsmuster** (engl.: **design pattern**), das für eine konkrete Aufgabe bewährte Lösungsverfahren vorschreibt, spielen Schnittstellen oft eine wichtige Rolle.

9.1.3 Mögliche Bestandteile

Ein Interface kann folgende Bestandteile (Mitglieder) enthalten:

- Instanzmethoden
  Öffentliche Instanzmethoden können abstrakt sein (vgl. Abschnitt 9.2.3.1), oder eine **default**-Implementierung besitzen (vgl. Abschnitt 9.2.3.2). Die seit Java 9 erlaubten privaten Instanzmethoden *müssen* eine Implementierung besitzen (vgl. Abschnitt 9.2.3.4).
- Statische Methoden (vgl. Abschnitt 9.2.3.3)
  Seit Java 8 sind in einem Interface auch statische Methoden erlaubt. Im Unterschied zu den Instanzmethoden einer Schnittstelle, die in der Regel abstrakt definiert sind, müssen die statischen Methoden in der Schnittstelle implementiert werden. Seit Java 9 können statische Methoden als **private** deklariert werden.
- Konstanten (vgl. Abschnitt 9.2.4)
  Manchmal werden Schnittstellen definiert, die ausschließlich als Aufbewahrungsort für Konstanten dienen. Diese Praxis hat allerdings keinen guten Ruf.
- Statische Mitgliedstypen (vgl. Abschnitt 9.2.5)
  Die in einem Interface definierten Typen (Klassen, Enumerationen, Schnittstellen) sind implizit als **static** deklariert (vgl. Abschnitt 4.8.1.2). Sie werden also wie Top-Level - Typen behandelt, doch ist bei ihrer Verwendung durch fremde Typen ein „Doppelname“ anzugeben, z. B. `WinterFace.SchneeEnum.PULVER` bei dem im Interface `WinterFace` definierten Aufzählungstyp `SchneeEnum` mit der Enumerationskonstanten `PULVER`.

Diese Interface-Bestandteile sind implizit als **public** deklariert und können von jeder Klasse genutzt werden, welche Zugriffsrechte für das Interface besitzt (per Voreinstellung von allen Klassen im selben Paket). Der Modifikator **public** kann bei Interface-Bestandteilen weggelassen werden, ist aber erlaubt. Seit Java 9 können Interface-Methoden als **private** definiert werden.

9.2 Interfaces definieren

Wir behandelt zuerst das im Programmieralltag vergleichsweise seltene Definieren einer Schnittstelle, weil man dabei einen guten Eindruck von den Bestandteilen einer Schnittstelle und von ihrer Rolle bei der objektorientierten Programmierung gewinnt. Allerdings verzichten wir vorläufig auf ein eigenes Beispiel und betrachten stattdessen die angenehm einfach aufgebaute und außerordentlich wichtige API-Schnittstelle **Comparable<T>** im Paket **java.lang**:[1]

```
public interface Comparable<T> {
   public int compareTo(T o);
}
```

Seit Java 5 können dem Interface-Namen begrenzt durch ein Paar spitzer Klammern Typformalparameter angehängt werden, sodass für konkrete Typen jeweils eine eigene Interface-Definition entsteht (vgl. Kapitel 8 zu generischen Typen). Im Abschnitt 9.1.1 ist uns mit dem parametrisierten Interface **Comparable<Double>** schon eine Konkretisierung der generischen Schnittstelle **Comparable<T>** begegnet. Im Abschnitt 10.6 über Kollektionen mit (Schlüssel-Wert) - Elementen werden Sie das Interface **Map<K,V>** mit *zwei* Typformalparametern (für *Key* und *Value*) kennenlernen.

Im Schnittstellenrumpf werden in der Regel abstrakte Instanzmethoden aufgeführt, deren Rumpf durch ein Semikolon ersetzt ist. Dabei werden die Typformalparameter wie gewöhnliche Typbezeichner verwendet. Mit einer Schnittstelle wird festgelegt, dass Objekte einer implementierenden Klasse bestimmte Methodenaufrufe beherrschen müssen.

Meist beschreibt der Schnittstellendesigner in der begleitenden Dokumentation das erwünschte Verhalten der Methoden. In der API-Dokumentation zum Interface **Comparable<T>** wird die Methode **compareTo()** so erläutert:

> Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Der Compiler kann aber bei einer implementierenden Klasse nur die Einhaltung syntaktischer Regeln sicherstellen, sodass er z. B. auch die folgende **compareTo()** - Realisation einer Klasse **Joke** akzeptieren würde:

```
public int compareTo(Joke a) {return 1;}
```

Hinsichtlich der Dateiverwaltung gilt analog zu Klassen, dass ein **public**-Interface in einer eigenen Datei gespeichert werden muss, wobei der Schnittstellenname übernommen und die Namenserweiterung **.java** angehängt wird. In der Regel wendet man diese Praxis bei *allen* Schnittstellen an, die nicht in andere Typen eingeschachtelt sind.

Analog zu Mitgliedsklassen (vgl. Abschnitt 4.8.1) können Schnittstellen nicht nur auf Paketebene (in einer eigenen Quellcodedatei) definiert werden, sondern auch innerhalb von Klassen oder anderen Schnittstellen.

[1] Sie finden diese Definition in der Datei **Comparable.java**, die wiederum im Archiv **src.zip** mit den API-Quelltexten steckt. Das Quelltextarchiv landet bei der OpenJDK-Installation auf die Festplatte Ihres PCs (siehe Abschnitt 3.1.5).

9.2.1 Kopf einer Schnittstellen-Definitionen

Regeln für den Kopf einer Schnittstellen-Definitionen:

- Erlaubte Modifikatoren für Schnittstellen
  - **public**
    Ohne den Modifikator **public** ist die Schnittstelle nur innerhalb ihres Pakets verwendbar. Ein Interface mit Schutzstufe **public** muss prinzipiell in einer eigenen Datei (ohne weitere Top-Level - Typen mit **public**-Sichtbarkeit) definiert werden.
  - **abstract**
    Weil Schnittstellen grundsätzlich **abstract** sind, muss der Modifikator nicht angegeben werden. Er ist erlaubt, aber unüblich.
- Schlüsselwort **interface**
  Das obligatorische Schlüsselwort dient zur Unterscheidung zwischen Klassen- und Schnittstellendefinitionen.
- Schnittstellenname
  Wie bei Klassennamen sollte man den ersten Buchstaben groß schreiben. Seit Java 5 kann man dem Interface-Namen zwischen spitzen Klammern einen oder mehrere (jeweils durch ein Komma getrennte) Typformalparameter folgen lassen (analog zu den im Kapitel 8 beschriebenen generischen Klassen).

9.2.2 Vererbung bzw. Erweiterung bei Schnittstellen

Ein Interface kann ein anderes beerben bzw. erweitern, wobei eine solche Generalisierungsbeziehung wie bei Klassen unter Verwendung des Schlüsselworts **extends** deklariert wird, z. B.:

```
interface SortedSet<E> extends Set<E> {
  . . .
}
```

Während bei Java-Klassen die (z. B. von C++ bekannte) *Mehrfachvererbung nicht* unterstützt wird (siehe Abschnitt 7.2.2), ist sie bei Java-Schnittstellen möglich (und oft auch sinnvoll), z. B.:

```
public interface Transform extends XMLStructure, AlgorithmMethod {
   . . .
}
```

Eine mit der Urahnklasse **Object** vergleichbare Urahnschnittstelle gibt es in Java *nicht*.

Bei einer Schnittstelle mit (direkten und indirekten) Basisschnittstellen muss eine implementierende (nicht abstrakte) Klasse die abstrakten Methoden aller Schnittstellen definieren.

Statische Interface-Methoden werden *nicht* vererbt, weder an erweiternde Schnittstellen, noch an implementierende Klassen (siehe Abschnitt 9.2.3.3).

9.2.3 Schnittstellen-Methoden

Die Methoden einer Schnittstelle sind per Voreinstellung öffentlich, und das Schlüsselwort **public** kann weggelassen werden. In der *Java Language Specification* findet sich die folgende Empfehlung (Gosling et al. 2019, Abschnitt 9.4):

> It is permitted, but discouraged as a matter of style, to redundantly specify the public modifier for a method declared in an interface.

Im Quellcode der wichtigen Java-API - Schnittstelle **Comparable<T>** findet sich allerdings zur einzigen Methode **compareTo()** diese Definition:

```
public int compareTo(T o);
```

Seit Java 9 dürfen Schnittstellenmethoden auch als **private** deklariert werden.

Zwar dienen die meisten Schnittstellen dazu, Verhaltenskompetenzen von Klassen über abstrakte Methodendefinitionen vorzuschreiben, doch sind für spezielle Zwecke auch Schnittstellen *ohne* Methoden erlaubt (siehe unten).

Während bis Java 7 in Schnittstellen ausschließlich abstrakte Instanzmethoden erlaubt waren, sind seit Java 8 auch möglich:

- Instanzmethoden mit **default**-Implementierung
- statische Methoden, wobei hier eine Implementierung vorgeschrieben ist.

Während bis Java 8 alle Schnittstellenmethoden öffentlich waren, sind in Java 9 private Methoden dazugekommen. In den nächsten Abschnitten werden die verschiedenen Varianten beschrieben.

9.2.3.1 Abstrakte Instanzmethoden

Durch abstrakte Instanzmethoden werden Verhaltenskompetenzen beschrieben, die implementierende Klassen besitzen müssen. Auf den Methodendefinitionskopf folgt an Stelle des durch geschweifte Klammern begrenzten Rumpfes ein Semikolon, z. B.:

```
public int compareTo(T o);
```

Eine abstrakte Instanzmethode ist implizit öffentlich. Die Modifikatoren **public** und **abstract** sind überflüssig, aber erlaubt.

9.2.3.2 Instanzmethoden mit default-Implementierung

Seit Java 8 ist es möglich, Instanzmethoden *mit* Implementierung (also mit einen ausführbaren Methodenrumpf) in eine Schnittstelle aufzunehmen, wobei der Modifikator **default** zu verwenden ist. Das erleichtert das Implementieren einer Schnittstelle und ist bei neuen Schnittstellen uneingeschränkt zu begrüßen. Ist die **default**-Lösung für eine implementierende Klasse nicht akzeptabel, kann sie problemlos durch eine eigene Implementation ersetzt werden. Ein wesentliches Motiv für die Einführung der **default**-Methoden bestand darin, die nachträgliche Erweiterung von Schnittstellen um neue Methoden ohne Nachteil für vorhandene Implementationen zu ermöglichen. Das ist gut (aber nicht perfekt) gelungen.

9.2.3.2.1 Erweiterung von Schnittstellen um neue Instanzmethoden

Bis zur Version 7 wurde in Java eine Möglichkeit vermisst, vorhandene Interfaces um neue Instanzmethoden zu erweitern, ohne die Binär-Kompatibilität mit implementierenden Altklassen zu verlieren. Seit Java 8 ist das Problem durch die Erweiterung von Schnittstellen um **default**-Methoden gelöst. Altklassen erfüllen auch das erweiterte Interface, weil ihnen die **default**-Implementierung zur Verfügung steht.

Wir betrachten ein einfaches Beispiel mit einem Interface `WinterFace1`

```
interface WinterFace1 {
   void sagA();
}
```

und einer implementierenden Klasse `Impl1`:

```
class Impl1 implements WinterFace1 {
   public void sagA() {
      System.out.println("A");
   }

   static public void main(String[] args) {
      Impl1 ob = new Impl1();
      ob.sagA();
   }
}
```

Nun soll die Schnittstelle `WinterFace1` um eine Instanzmethode `sagB()` erweitert werden, ohne alte Klassen (z. B. `Impl1`) ändern zu müssen. Man ergänzt eine Instanzmethode mit dem Modifikator **default** und einer kompletten Implementierung:

```
interface WinterFace1 {
   void sagA();

   default void sagB() {
      sagA();
      System.out.println("B");
   }
}
```

In einer **default**-Methode dürfen andere Schnittstellenmethoden verwendet werden (instanzbezogene und statische, öffentliche und private).

In einer abgeleiteten (erweiternden) Schnittstelle kann eine geerbte **default**-Methode ...

- durch eine eigene **default**-Implementierung überschrieben
- oder durch eine abstrakte Definition ersetzt werden.

Implementiert eine bestehende Klasse eine neuerdings um eine **default**-Methode erweiterte Schnittstelle, dann bleibt die Klasse binärkompatibel zum erweiterten Interface. Trotzdem sind unerwünschte Effekte der Schnittstellenerweiterung möglich (siehe Abschnitt 9.2.3.2.2).

Eine neue bzw. aktualisierte Klasse, die das Interface implementiert, kann die **default**-Methode unverändert nutzen oder durch eine eigene Implementierung überschreiben. Im folgenden Beispiel wird die *erste* Option verwendet:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Impl2 implements WinterFace1 {
   public void sagA() {
      System.out.println("A");
   }

   static public void main(String[] args) {
      Impl2 ob = new Impl2();
      ob.sagB();
   }
}</pre></td><td><pre>A
B</pre></td></tr>
</table>

Wenn eine Klasse mehrere Interfaces implementiert (siehe Abschnitt 9.3 zum Implementieren) und dabei ein Konflikt mit Signatur-gleichen **default**-Methoden auftritt, verweigert der Compiler die Übersetzung, z. B.:

```
class Impl3 implements WinterFace1, WinterFace2 {
```

Impl3 inherits unrelated defaults for sagB() from types WinterFace1 and WinterFace2

Implement methods Alt+Umschalt+Eingabe   More actions... Alt+Eingabe

Das Problem ist dadurch zu lösen, dass die betroffene Klasse die kritische Methode implementiert oder als **abstract** definiert.

Ein analoges Problem tritt auf, wenn eine Schnittstelle von zwei anderen Schnittstellen Signatur-gleiche **default**-Methoden erbt, z. B.:

```
interface WinterFace3 extends WinterFace1, WinterFace2 {
```

```
WinterFace3 inherits unrelated defaults for sagB() from types WinterFace1 and WinterFace2
Implement methods  Alt+Umschalt+Eingabe     More actions...  Alt+Eingabe
```

Um das Problem zu lösen, muss die abgeleitete Schnittstelle die kritische Methode entweder (implizit) als **abstract** deklarieren oder eine **default**-Implementierung vornehmen.

Eine **default**-Methode wird grundsätzlich ignoriert, wenn in einer implementierenden Klasse eine Signatur-identische Methode vorhanden ist. Gegen eine in der Urahnklasse **Object** definierte Methode (z. B. **equals()**, **hashCode()**, **toString()**) kann eine **default**-Methode also nie gewinnen, weil sie sich im (geerbten) Handlungsrepertoire aller Klassen befindet. Folglich verhindert der Compiler die Definition einer solchen **default**-Methode, z. B.:

```
default String toString() {
    // Verboten
}
```

```
Default method 'toString' overrides a member of 'java.lang.Object'
```

9.2.3.2.2 Unerwünschte Effekte auf bestehende Klassen

Von Kreft & Langer (2014) wird gezeigt, dass die Erweiterung einer Schnittstelle um eine statische Methode das Verhalten vorhandener Klassen ändern könnte, wenn eine Vererbung von statischen Interface-Methoden stattfinden würde (vgl. Abschnitt 9.2.3.3). Anschließend wird unter Verwendung des Beispiels aus Kreft & Langer (2014) demonstriert, dass ein analoger Effekt bei der Erweiterung einer Schnittstelle um **default**-Instanzmethoden tatsächlich auftreten kann, wenn eine vorhandene Klasse nach der Schnittstellenerweiterung neu übersetzt wird.

In der Klasse `Nest`, die das Interface `Kuckuck` implementiert, existiert die Instanzmethode `tuWas()` mit einem Parameter vom Typ **long**. In der **main()** - Methode von `Nest` wird die Methode mit einem **int**-Argument aufgerufen, das der Compiler implizit erweiternd in den Typ **long** wandelt:

```
interface Kuckuck {
   void sagWas();
}

class Nest implements Kuckuck {
   public void sagWas() {
      System.out.println("Was");
   }

   void tuWas(long par) {
      System.out.println("Methode in Nest: " + par);
   }

   static public void main(String[] args) {
      Nest ob = new Nest();
      ob.tuWas(3);
   }
}
```

Nun wird das implementierte Interface um eine **default**-Methode namens `tuWas()` mit einem Parameter vom Typ **int** erweitert:

```
interface Kuckuck {
   void sagWas();
   default void tuWas(int par) {
      System.out.println("default-Methode in Kuckuck: " + par*par);
   }
}
```

Wird nur `Kuckuck` neu übersetzt, `Nest` hingegen nicht, bleibt das Verhalten der Klasse unverändert. Insofern wird das folgende Versprechen aus der Sprachbeschreibung von Java 8 eingehalten (Gosling et al. 2019, Abschnitt 13.5.6):

> Adding a default method, or changing a method from abstract to default, does not break compatibility with pre-existing binaries.

Wenn aber auch `Nest` neu übersetzt wird, bevorzugt der Compiler die besser zum Aktualparameter passende **default**-Methode aus dem Interface an Stelle der klasseneigenen Methode. Anschließend zeigt `Nest` ein abweichendes Verhalten, das hoffentlich bei der nach jeder Änderung des Programms durchgeführten Testprozedur auffällt.

Das im aktuellen Abschnitt beschriebene Risiko ist aber *nicht* auf die in Java 8 eingeführten **default**-Methoden beschränkt, sondern besteht bei jeder Erweiterung einer Klasse um eine neue Methode, sofern abgeleitete Klassen vorhanden sind. Dementsprechend war das Risiko immer schon in Java und vergleichbaren objektorientierten Programmiersprachen vorhanden.

Bloch (2018, S. 104f) beschreibt den Fall einer Klasse namens **SynchronizedCollection**, die das Interface **Collection<E>** (siehe Abschnitt 10.2.2) implementiert, das in Java 8 die **default**-Methode **removeIf()** zum bedingungsabhängigen Entfernen eines Elements erhalten hat. Die Klasse **SynchronizedCollection** verspricht Thread-Sicherheit für ihre Methoden, besitzt aber nun eine zusätzliche Methode, die *nicht* Thread-sicher ist. Man kann erwarten, dass wichtige Bibliotheken ihre Klassen an veränderte Schnittstellen anpassen, die kritische **default**-Methoden erhalten haben, so wie es im Java-API geschehen ist. Generell ist die nachträgliche Erweiterung einer Schnittstelle um **default**-Methoden ein Notbehelf, der möglichst selten genutzt werden sollte, denn Bloch warnt zu Recht (2018, S. 194):

> But it is not always possible to write a default method that maintains all invariants of every conceivable implementation.

9.2.3.3 Statische Methoden

Seit Java 8 sind in Schnittstellen auch statische Methoden erlaubt, wobei eine Implementation erforderlich ist. Die im Abschnitt 9.2.3.2 vorgestellte Schnittstelle `WinterFace1` soll eine statische Methode erhalten, welche in der **default**-Instanzmethode `sagB()` derselben Schnittstelle genutzt wird:

```
interface WinterFace1 {
   static void achtung() {
      System.out.println("Achtung Durchsage:");
   }

   void sagA();

   default void sagB() {
      achtung();
      System.out.println("B");
   }
}
```

Im Unterschied zu den statischen Methoden von Klassen werden die statischen Interface-Methoden *nicht* vererbt, weder an erweiternde Schnittstellen, noch an implementierende Klassen. Wenn eine

Klasse ein Interface mit statischer Methode implementiert, gelangt diese Methode also nicht in das statische Handlungsrepertoire der Klasse. Im Rahmen bestehender Zugriffsrechte kann die statische Schnittstellenmethode jedoch wie eine statische Methode einer fremden Klasse genutzt werden, z. B.:

```
class Impl1 implements WinterFace1 {
   public void sagA() {
      System.out.println("A");
   }
    public static void main(String[] args) {
      // achtung(); klappt nicht!
      WinterFace1.achtung();
    }
}
```

Auf diese Weise wird verhindert, dass sich durch die Aufnahme von statischen Methoden in ein Interface das Verhalten von Klassen ändert, welche das Interface implementieren (Kreft & Langer 2014).

9.2.3.4 Private Interface-Methoden

Seit Java 9 können Schnittstellenmethoden mit Implementation als **private** deklariert werden. Indem mehrfach benötigte Implementierungsdetails in eine private Schnittstellenmethode ausgelagert werden, ...

- vermeidet man Code-Wiederholungen
- und verhindert gleichzeitig Zugriffe durch fremde Typen.

Eine private *statische* Methode kann sowohl von anderen statischen Methoden als auch von **default**-Methoden der Schnittstelle aufgerufen werden. Eine private *Instanzmethode* kann hingegen nur von **default**-Methoden der Schnittstelle aufgerufen werden. Im folgenden Beispiel wird eine private statische Methode von zwei öffentlichen Instanzmethoden mit **default**-Implementation benutzt:

```
interface WinterFace1 {
   private static void achtung() {
      System.out.print("Achtung Durchsage:");
   }

   default void sagA() {
      achtung();
      System.out.println("A");
   }

   default void sagB() {
      achtung();
      System.out.println("B");
   }
}
```

9.2.4 Konstanten

Neben Methoden sind in einer Schnittstellendefinition auch Felder erlaubt, wobei diese implizit als **public**, **final** und **static** deklariert sind, also initialisiert werden müssen. Die `Demo`-Schnittstelle im folgenden Beispiel enthält eine **int**-Konstante namens `ONE` und verlangt das Implementieren einer Methode namens `say1()`:

```
public interface Demo {
   int ONE = 1;
   int say1();
}
```

Implementierende Klassen können auf eine Konstante ohne Angabe des Schnittstellennamens zugreifen. Auch nicht implementierende Klassen dürfen eine Interface-Konstante verwenden, müssen aber den Interface-Namen samt Punkt voranstellen. Implementiert eine Klasse *zwei* Schnittstellen mit namensgleichen Konstanten, dann muss beim Zugriff zur Beseitigung der Zweideutigkeit der Schnittstellenname vorangestellt werden.

Eine früher verbreitete, auch im Java-API anzutreffende, heute aber kritisch beurteilte Praxis besteht darin, Schnittstellen mit dem einzigen Zweck der Aufbewahrung von Konstanten zu definieren, z. B.:

```
public interface DiesUndDas {
   int KW = 4711;
   double PIHALBE = 1.5707963267948966;
}
```

Bloch (2018, S. 107f) plädiert dafür, Schnittstellen ausschließlich als Datentypen zu verwenden und nur eng mit diesem Zweck gekoppelte Konstanten in die Definition aufzunehmen. Als (nicht allzu dramatische) Nachteile der Verwendung von Schnittstellen als Konstanten-Container nennt Bloch:

- Dass eine Klasse bestimmte Konstanten verwendet, ist ein Implementierungsdetail, das nicht in die Öffentlichkeit gehört. Welche Schnittstellen eine Klasse implementiert, ist aber öffentlich zu dokumentieren.
- Eine Klasse vererbt ihre Schnittstellen-Implementationen, sodass die vererbten Schnittstellen-Konstanten auch den Namensraum einer abgeleiteten Klasse belasten.

Als Konstanten-Container sollten anstelle von Schnittstellen besser Klassen verwendet werden. Wenn dort ausschließlich statische Mitglieder vorhanden sind, sollte das Instanzieren verhindert werden (z. B. durch die Schutzstufe **private** für alle Konstruktoren). Wenn ein Klassenname allzu oft in Kombination mit den Namen von Konstanten im Quellcode auftaucht, kann über den statischen Import für eine Vereinfachung gesorgt werden (siehe Abschnitt 6.1.2.2).

9.2.5 Statische Mitgliedstypen

In einer Interface-Definition können Mitgliedstypen (Klassen oder Schnittstellen) definiert werden. Diese sind generell öffentlich und statisch, wobei die überflüssigen Modifikatoren **public** und **static** erlaubt sind, aber weggelassen werden sollten Gosling et al. 2019, Abschnitt 9.5). Statische Mitgliedstypen verhalten sich wie Top-Level - Typen, müssen jedoch über einen „Doppelnamen" angesprochen werden, wobei auf den Namen der Schnittstelle ein Punkt und der Name des Mitgliedstyps folgt.

Als Beispiel betrachten wir das generische API-Interface **Map<K,V>**, das Methoden für Container zur Verwaltung von (Schlüssel-Wert) - Paaren festlegt (siehe Abschnitt 10.6.1). Es enthält das innere Interface **Map.Entry<K,V>**, das die Kompetenzen eines einzelnen (Schlüssel-Wert) - Paares beschreibt:

```
public interface Map<K,V> {
   int size();
   boolean isEmpty();
  . . .
   interface Entry<K,V> {
      K getKey();
      V getValue();
       . . .
   }
  . . .
}
```

Verwendung findet **Map.Entry<K,V>** z. B. als Rückgabetyp für die im Interface **NavigableMap<K,V>**, das (via **SortedMap<K,V>**) von **Map<K,V>** abstammt (siehe Abschnitt 10.6.3), definierte Methode **firstEntry()**:

Map.Entry<K,V> firstEntry()

Das folgende Programm demonstriert die Verwendung eines Objekts, das die generische Schnittstelle **Map.Entry<K,V>** erfüllt, wobei ein Objekt der generischen Klasse **TreeMap<K,V>** (siehe Abschnitt 10.6.4) zum Einsatz kommt.

Quellcode

```
import java.util.*;
class Prog {
    public static void main(String[] args) {
        NavigableMap<Integer, String> m = new TreeMap<>();
        m.put(1,"AAA");
        Map.Entry<Integer, String> me = m.firstEntry();
        System.out.println(me.getValue());
    }
}
```

Ausgabe

```
AAA
```

Mitglieds-Schnittstellen können auch in Klassen definiert werden und sind dann ebenfalls implizit statisch (siehe Gosling et al. 2019, Abschnitt 8.5.1).

9.2.6 Zugriffsschutz bei Schnittstellen

Bei den Top-Level - Schnittstellen ist nur der Zugriffsmodifikator **public** erlaubt, sodass zwei Schutzstufen möglich sind:

- Ohne Zugriffsmodifikator ist das Interface nur innerhalb des eigenen Pakts verwendbar.
- Durch den Zugriffsmodifikator **public** wird die Verwendung in berechtigten Paketen erlaubt (siehe Abschnitt 6.2 über die mit Java 9 eingeführten Module).

Für die Mitglieder von Schnittstellen ist die öffentliche Verfügbarkeit voreingestellt, also insbesondere auch für die Mitgliedstypen (vgl. Abschnitt 9.2.5), z. B.:

```
public interface Map<K,V> {
 . . .
   interface Entry<K,V> {
      . . .
   }
}
```

Der Modifikator **public** ist überflüssig und wird in der Regel weggelassen.

Seit Java 9 ist für (statische) Methoden in Schnittstellen auch der Zugriffsmodifikator **private** erlaubt (siehe Abschnitt 9.2.3.4).

9.2.7 Marker - Interfaces

Es sind auch Schnittstellen erlaubt, die weder Methoden noch sonstige Bestandteile enthalten, also nur aus einem Namen bestehen und gelegentlich als *marker interfaces* bezeichnet werden. Ein besonders wichtiges Beispiel ist die beim Sichern und bei der Netzwerkübertragung kompletter Objekte (siehe Abschnitt 14.6) relevante API-Schnittstelle **java.io.Serializable**, die z. B. von der Klasse **java.lang.Double** implementiert wird (siehe oben):

```
public interface Serializable {
}
```

Durch das Implementieren dieser Schnittstelle teilt eine Klasse mit, dass sie gegen das Serialisieren ihrer Objekte nichts einzuwenden hat.

9.3 Interfaces implementieren

Soll für die Objekte einer Klasse angezeigt werden, dass sie den Datentyp einer bestimmten Schnittstelle erfüllen, muss diese Schnittstelle im Kopf der Klassendefinition nach dem Schlüsselwort **implements** aufgeführt werden. Als Beispiel dient eine Klasse namens `Figur`, die nur partielle Ähnlichkeit mit einem gleichnamigen Beispiel aus Kapitel 7 besitzt und der Einfachheit halber die Datenkapselung sträflich vernachlässigt. Sie implementiert das Interface **Comparable**<`Figur`>, damit für `Figur`-Objekte eine Anordnung definiert ist:

```
package fimpack;

public class Figur implements Comparable<Figur> {
   public int xpos, ypos;
   public String name;

   public Figur(String name_, int xpos_, int ypos_) {
      name = name_; xpos = xpos_; ypos = ypos_;
   }

   public int compareTo(Figur fig) {
      if (xpos < fig.xpos)
         return -1;
      else if (xpos == fig.xpos)
         return 0;
      else
         return 1;
   }
}
```

Alle abstrakten Methoden einer im Klassenkopf angemeldeten Schnittstelle, die nicht von einer Basisklasse geerbt werden, müssen im Rumpf der Klassendefinition implementiert werden, wenn keine abstrakte Klasse entstehen soll. Nach der im Abschnitt 9.2 wiedergegebenen **Comparable<T>** - Definition ist also im aktuellen Beispiel eine Methode mit dem folgenden Definitionskopf erforderlich:[1]

```
public int compareTo(Figur fig)
```

[1] Es ist erlaubt und sinnvoll, aber nicht strikt empfohlen, beim Implementieren von Interface-Methoden wie beim Überschreiben von Instanzmethoden (siehe Abschnitt 7.5.1) die Absicht gegenüber dem Compiler durch die Marker-Annotation **@Override** zu bekunden. Das ist zum frühzeitigen Entdecken von Fehler meist nicht erforderlich, weil eine durch Tippfehler gescheiterte Implementation von Compiler als fehlend reklamiert wird. Es kann aber z. B. passieren, dass eine Implementation (unbeachtet) geerbt wird, und für eine vermeintliche Implementation (mit anderen Parametertypen) der Fehlschlag erst durch die Annotation **@Override** entlarvt wird. IntelliJ dekoriert jedenfalls per Voreinstellung die als QuickFix eingefügten implementierenden Methoden mit der Annotation **@Override**.

In semantischer Hinsicht soll sie eine `Figur` beauftragen, sich mit dem per Aktualparameter bestimmten Artgenossen zu vergleichen. Bei obiger Realisation werden Figuren nach der X-Koordinate ihrer Position verglichen:

- Hat die angesprochene Figur eine kleinere X-Koordinate als der Vergleichspartner, dann wird der Wert -1 zurückgemeldet.
- Haben beide Figuren dieselbe X-Koordinate, lautet die Antwort 0.
- Ansonsten wird der Wert 1 gemeldet.

Damit ist eine Anordnung der `Figur`-Objekte definiert, und einem erfolgreichen Sortieren (z. B. per **java.util.Arrays.sort()**) steht nichts mehr im Weg.

Weil die zu implementierenden Methoden einer Schnittstelle grundsätzlich als **public** definiert sind, und beim Implementieren eine Einschränkungen der Schutzstufe verboten ist, muss beim Definieren von implementierenden Methoden die Schutzstufe **public** verwendet werden, wobei der Modifikator wie bei jeder Methodendefinition explizit anzugeben ist.

Wenn eine implementierende Klasse eine abstrakte Schnittstellenmethode weglässt (oder abstrakt implementiert), dann entsteht eine abstrakte Klasse, die auch als solche deklariert werden muss (vgl. Abschnitt 7.8).

Über den **instanceof**-Operator kann man nicht nur prüfen, ob ein Objekt zu einer Klasse gehört, sondern auch feststellen, ob seine Klasse ein bestimmtes Interface implementiert. Dabei ist bei einem generischen Interface der Rohtyp (vgl. Abschnitt 8.1.2.1) anzugeben, z. B.:

```
System.out.println(fig instanceof Comparable);
```

9.3.1 Mehrere Schnittstellen implementieren

Während eine Klasse nur *eine direkte Basisklasse* besitzt, kann sie *beliebig viele Schnittstellen* implementieren, was z. B. die Klasse **TreeSet<E>** aus dem Java Collections Framework tut:

```
public class TreeSet<E> extends AbstractSet<E>
      implements NavigableSet<E>, Cloneable, Serializable {
  . . .
}
```

Wenn dabei ein Konflikt mit Signatur-gleichen **default**-Methoden aus verschiedenen Schnittstellen auftritt, verweigert der Compiler die Übersetzung. Das Problem ist dadurch zu lösen, dass die Klasse die kritische Methode selbst implementiert.

Es ist *kein* Problem, wenn zwei implementierte Schnittstellen über *abstrakte* Methoden mit identischem Definitionskopf verfügen, weil keine konkurrierenden Realisationen geerbt werden, sondern von der implementierenden Klasse eine eigene Realisation erstellt werden muss.

Implementiert eine Klasse eine Schnittstelle mit (direkten und indirekten) Basisschnittstellen, dann muss sie die Methoden aller Schnittstellen im Stammbaum realisieren. Weil z. B. die Klasse **TreeSet<E>** aus dem Java Collections Framework (siehe Abschnitt 10.5.4) neben den Schnittstellen **Cloneable** und **Serializable** auch die Schnittstelle **NavigatableSet<E>** implementiert (siehe oben), sammelt sich einiges an Lasten an, denn **NavigatableSet<E>** erweitert die Schnittstelle **SortedSet<E>**,

```
public interface NavigableSet<E> extends SortedSet<E> { . . . }
```

die ihrerseits auf **Set<E>** basiert:

```
public interface SortedSet<E> extends Set<E> { . . . }
```

Das Interface **Set<E>** basiert auf dem Interface **Collection<E>**,

```
public interface Set<E> extends Collection<E> { . . . }
```

das wiederum die Schnittstelle **Iterable<E>** erweitert:

```
public interface Collection<E> extends Iterable<E> { . . . }
```

Wer als Programmierer wissen möchte, welche Datentypen eine API-Klasse direkt oder indirekt erfüllt, muss aber keine Ahnenforschung betreiben, sondern wird in der API-Dokumentation zur Klasse komplett informiert, z. B. bei der Klasse **TreeSet<E>**:

Module java.base

Package java.util

Class TreeSet<E>

```
java.lang.Object
  java.util.AbstractCollection<E>
      java.util.AbstractSet<E>
          java.util.TreeSet<E>
```

Type Parameters:
E - the type of elements in this list

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, NavigableSet<E>, Set<E>, SortedSet<E>

Wenn es im Beispiel für den **TreeSet<E>** - Programmierer gut gelaufen ist, hat der **AbstractSet<E>** - Programmierer bereits einige Schnittstellenmethoden implementiert (siehe Abschnitt 9.3.2 zu geerbten Interface-Implementationen).

Auch Schnittstellen ändern nichts daran, dass für Java-*Klassen* eine *Mehrfachvererbung* (vgl. Abschnitt 7) ausgeschlossen ist. Allerdings erlauben Schnittstellen in vielen Fällen eine Ersatzlösung, denn:

- Eine Klasse darf beliebig viele Schnittstellen implementieren.
- Bei Schnittstellen ist die Mehrfachvererbung erlaubt.

Die mit einer Mehrfachvererbung verbundenen Risiken, die beim Java-Design bewusst vermieden wurden, bleiben aber ausgeschlossen: In Schnittstellen sind Felder generell statisch (siehe Abschnitt 9.2.4). Folglich können *Instanzvariablen* nur von der Basisklasse (also nur von *einer* Klasse) übernommen werden, und der sogenannte *Deadly Diamond of Death* ist ausgeschlossen (siehe Kreft & Langer 2014).

9.3.2 Geerbte Interface-Implementationen

Im Zusammenhang mit dem Thema *Vererbung* ist von Bedeutung, dass eine abgeleitete Klasse die in Basisklassen implementierten Schnittstellen erbt. Wird z. B. eine Klasse `Kreis` unter Verwendung des Schlüsselworts **extends** von der oben vorgestellten Klasse `Figur` abgeleitet,

```
package fimpack;

public class Kreis extends Figur {
   public int radius;
   public Kreis(String name_, int xpos_, int ypos_, int rad_) {
      super(name_, xpos_, ypos_);
      radius = rad_;
   }
}
```

dann übernimmt sie auch die Schnittstelle **Comparable**<Figur>, und die statische **sort()** - Methode der Klasse **java.util.Arrays** kann auf Felder mit Kreis-Elementen angewendet werden, z. B.:

Quellcode	Ausgabe

<pre>import fimpack.Kreis;
 	
class ImpleDemo {
   public static void main(String[] args) {
      Kreis[] ka = new Kreis[3];
      ka[0] = new Kreis("C", 70, 20, 10);
      ka[1] = new Kreis("B", 60, 20, 30);
      ka[2] = new Kreis("A", 50, 20, 50);
      java.util.Arrays.sort(ka);
      for (Kreis ko : ka) System.out.print(ko.name + " ");
   }
}</pre>
	A B C

Die Schnittstelle **Comparable**<Kreis> befindet sich weder im Erbe der Kreis-Klasse noch darf sie hier zusätzlich implementiert werden. Es ist erlaubt, in der Kreis-Klasse die folgende Überladung der Methode **compareTo()** zu definieren:

```
public int compareTo(Kreis kr) {
   if (xpos + radius < kr.xpos + kr.radius)
      return -1;
   else if (xpos + radius == kr.xpos + kr.radius)
      return 0;
   else
      return 1;
}
```

Die kommt aber im Beispielprogramm zum Sortieren der Kreis-Objekte *nicht* zum Einsatz, weil die Klasse Kreis die Schnittstelle **Comparable**<Figur> implementiert und daher von **Arrays.sort()** die zugehörige Methode verwendet wird, nämlich:

public compareTo(Figur *fig*)

Man könnte in der Klasse Kreis die Methode **compareTo()** (mit dem Parameter vom Datentyp Figur) z. B. so überschreiben, dass beim Vergleich von zwei Kreisen der Radius berücksichtigt wird, beim Vergleich eines Kreises mit einer Figur jedoch nicht. Diese „Lösung" soll aber nicht vorgeführt werden, weil sie beim Sortieren eines Arrays mit Kreisen und Figuren ein kaum brauchbares Ergebnis liefern würde.

Bei der folgenden Lösung kommt eine Überladung der **Arrays**-Methode **sort()** zum Einsatz, die zum Sortieren nicht die Methode **compareTo()** der Klasse Kreis verwendet, sondern ein Objekt aus einer Klasse engagiert, die das Interface **Comparator<Kreis>** implementiert und daher die Methode **compare**(Kreis *o1*, Kreis *o2*) beherrscht. Als zweiter Aktualparameter der **sort()** - Methode wird per **new**-Operator ein Objekt aus einer anonymen Klasse (siehe Abschnitt 12.1.1.2) mit der erforderlichen Instanzmethode erzeugt:

Quellcode	Ausgabe

`import fimpack.Kreis;`
`import java.util.Comparator;`
 	
`class ImpleDemo {`
`   public static void main(String[] args) {`
`      Kreis[] ka = new Kreis[3];`
`      ka[0] = new Kreis("C", 70, 20, 10);`
`      ka[1] = new Kreis("B", 60, 20, 30);`
`      ka[2] = new Kreis("A", 50, 20, 50);`
 	
`      java.util.Arrays.sort(ka, new Comparator<Kreis>() {`
`         @Override`
`         public int compare(Kreis o1, Kreis o2) {`
`            if (o1.xpos + o1.radius < o2.xpos + o2.radius)`
`               return -1;`
`            else if (o1.xpos + o1.radius == o2.xpos + o2.radius)`
`               return 0;`
`            else`
`               return 1;`
`         }`
`      });`
`      for (Kreis ko : ka) System.out.print(ko.name + " ");`
`   }`
`}`
	`C B A`

9.3.3 Implementieren von Schnittstellen und Sichtbarkeit für Klassen

Durch das Implementieren einer öffentlichen Schnittstelle wird eine mit dem voreingestellten paketinternen Zugriff definierte Klasse im Rahmen der Interface-Implementation **public**, d.h. die per Schnittstellenreferenzvariable ansprechbaren Methoden können nicht nur im eigenen Paket, sondern in allen berechtigten Paketen genutzt werden, bei Verzicht auf die ab Java 9 mögliche Modultechnik also in *allen* Paketen. Zur Demonstration sind einige Bestandteile erforderlich.

Wir definieren in einem Paket eine Schnittstelle `IFace` mit **public**-Zugriff:

```
package meinpaket;

public interface IFace {
   void tell();
}
```

Im selben Paket wird die Klasse `InTell` definiert, die zwar für die paketinterne Verwendung gedacht ist, aber die Schnittstelle `IFace` implementiert:

```
package meinpaket;

class InTell implements IFace {
   @Override
   public void tell() {
      System.out.println("Klasse InTell");
   }
}
```

Die `IFace`-Schnittstellenmethoden haben den voreingestellten **public**-Zugriff, und diese Zugriffstufe darf beim Implementieren nicht eingeschränkt werden (siehe Abschnitt 9.3).

Im Paket existiert weiterhin eine öffentliche Klasse, die als Objektfabrik über die statische Methode `make()` u.a. `InTell`-Objekte liefern kann:

```
package meinpaket;

public class Faktory {
   public static IFace make(int sorte) {
      if (sorte == 1)
         return new ExTell();
      else
         return new InTell();
   }
}
```

Nach diesen Vorbereitungen lässt sich die Klasse `InTell` auch außerhalb ihres Pakets verwenden, z. B.:

```
package sichtbarkeit;

public class Main {
    public static void main(String[] args) {
        meinpaket.IFace myTell1 = meinpaket.Faktory.make(1);
        myTell1.tell();
    }
}
```

9.4 Interfaces als Referenzdatentypen verwenden

Mit der Definition einer Schnittstelle wird ein neuer Referenzdatentyp vereinbart, der anschließend in Variablen- und Parameterdeklarationen verwendbar ist. Eine Referenzvariable des neuen Typs kann auf Objekte jeder Klasse zeigen, welche die Schnittstelle implementiert, z. B.:

Quellcode

```
interface Quatsch {
   void sagWas();
}

class Ritter implements Quatsch {
   void ritterlichesVerhalten() { ... }
   public void sagWas() {
      System.out.println("Bin ein Ritter.");
   }
}

class Wolf implements Quatsch {
   public void jagdausflug() { ... }
   public void sagWas() {
      System.out.println("Bin ein Wolf.");
   }
}

class Intereferenz {
   public static void main(String[] args) {
      Quatsch[] demintiar = {new Ritter(), new Wolf()};
      for (Quatsch di : demintiar)
         di.sagWas();
   }
}
```

Ausgabe

```
Bin ein Ritter.
Bin ein Wolf.
```

Damit wird es z. B. möglich, Objekte aus beliebigen Klassen (z. B. `Ritter` und `Wolf`) in einem Array gemeinsam zu verwalten, sofern alle Klassen dasselbe Interface implementieren. Zwar lässt sich derselbe Zweck auch mit **Object**-Referenzen erreichen, doch leidet unter so viel Liberalität die Typsicherheit. Mit einem Interface als Elementdatentyp ist sichergestellt, dass alle Elemente be-

stimmte Verhaltenskompetenzen besitzen (im Beispiel: die Methode `sagWas()`). Folglich kann diese Funktionalität ohne lästige und fehleranfällige Typwandlungen abgerufen werden.

Im Beispiel werden ein Ritter und ein Wolf über den Datentyp einer gemeinsam implementierten Schnittstelle angesprochen. Sie führen die Schnittstellenmethode `sagWas()` auf ihre klasseneigene Art aus, zeigen also polymorphes Verhalten (vgl. Abschnitt 7.7).

Nach dem etwas verspielten Beispiel für die Verwendung eines Schnittstellendatentyps folgt noch ein sehr praxisrelevantes. Implementiert eine Klasse das Interface **CharSequence**, taugen ihre Objekte zur Repräsentation einer geordneten Folge von Zeichen und beherrschen entsprechende Methoden, z. B. die Methode **charAt()** mit dem folgenden Definitionskopf:

public char charAt(**int** *index*)

Sie liefert das Zeichen an der angegebenen Indexposition. Das Interface **CharSequence** erlaubt bei Verwendung als Formalparameterdatentyp die Definition von Methoden, die als Aktualparameterdatentyp sowohl die Klasse **String** (optimiert für konstante Zeichenfolgen, vgl. Abschnitt 5.2.1) als auch die Klassen **StringBuilder** und **StringBuffer** (optimiert für veränderliche Zeichenketten, vgl. Abschnitt 5.2.2) akzeptieren.

9.5 Annotationen

An Pakete, Typen (Klassen, Schnittstellen), Methoden, Konstruktoren, Parameter und lokale Variablen lassen sich Annotationen anheften, um zusätzliche **Metainformationen** bereit zu stellen, die …

- zur Entwicklungs- bzw. Übersetzungszeit
- und/oder zur Laufzeit

berücksichtigt werden können.[1] Sie ergänzen die im Java - Sprachumfang verankerten *Modifikatoren* für Typen, Methoden etc. und bieten dabei eine enorme Flexibilität. Bei einfachen Annotationen besteht die Information über den Träger in der An- bzw. Abwesenheit einer Eigenschaft (z. B. Deklaration einer Methode als überschreibend), jedoch kann eine Annotation auch Detailinformationen enthalten.

Annotationen mit Relevanz für die *Entwicklungs- bzw. Übersetzungszeit* beeinflussen das Verhalten des Compilers, der z. B. durch die Annotation **Deprecated** zur Ausgabe einer Warnung veranlasst wird. Neben dem Compiler kommen aber auch Entwicklungswerkzeuge als Adressaten für Quellcode-Annotationen in Frage. Diese können z. B. den Quellcode analysieren und aufgrund von Annotationen zusätzlichen Code generieren, um dem Programmierer lästige und fehleranfällige Routinearbeiten abzunehmen. So bieten die Annotationen eine Option zur *deklarativen Programmierung*.

Annotationen mit Relevanz für die *Laufzeit* beeinflussen ein Programm über ihre Signalwirkung auf Methoden, welche sich über die Existenz bzw. Ausgestaltung der Annotation informieren und ihr Verhalten daran orientieren (siehe Abschnitt 9.5.3). Wir lernen hier eine weitere Technik zur Kommunikation zwischen Programmbestandteilen kennen. In komplexen objektorientierten Softwaresystemen spielt die als *Reflexion* (engl.: *reflection*) bezeichnete Ermittlung von Informationen über Typen zur Laufzeit eine wichtige Rolle. Dabei leisten Annotationen einen wichtigen Beitrag.

Neben den im Java-API enthalten Annotationen (z. B. **Deprecated** für veraltete, nicht mehr empfehlenswerte Programmbestandteile) lassen sich auch eigene Exemplare definieren. Dabei ist eine an Schnittstellen erinnernde Syntax zu verwenden (siehe Abschnitt 9.5.1), und der Compiler erzeugt tatsächlich aus jeder Annotationsdefinition, die nicht auf den Quellcode beschränkt bleiben soll (siehe Abschnitt 9.5.4), ein Interface.

---

[1] Wer die Programmiersprache C# kennt, fühlt sich zu Recht an die dortigen Attribute erinnert.

Annotationen mit Sichtbarkeit **public** benötigen wie andere öffentliche Schnittstellen eine eigene Quellcode- und Bytecodedatei.

9.5.1 Definition

Wir starten mit der (im typischen Programmieralltag nur selten erforderlichen) Definition von Annotationen und werden dabei ohne großen Aufwand einen guten Einblick in die Technik gewinnen. Als erstes Beispiel betrachten wir die im Abschnitt 7.5.1 behandelte API-Annotation **Override** (Paket **java.lang**), die dem Compiler signalisiert, dass durch eine Methodendefinition eine Basisklassenvariante oder eine Schnittstellenmethode überschrieben werden soll. Sie enthält keine Annotationselemente (siehe unten) und gehört daher zu den *Marker-Annotationen*:

```
public @interface Override {}
```

Hinter dem optionalen Zugriffsmodifikator **public** steht das Schlüsselwort **interface** mit dem Präfix @ zur Unterscheidung von gewöhnlichen Schnittstellendefinitionen. Dann folgen der Typname und der (bei einer Marker-Annotation leere) Definitionsrumpf.

In der Regel enthält der Definitionsrumpf einer Annotation sogenannte *Annotationselemente*, damit bei der Zuweisung einer Annotation (siehe Abschnitt 9.5.2) Detailinformationen durch (Name-Wert) - Paare übergeben werden können. In der Definition wird ein Annotationselement als Interface-Methode realisiert mit:

- einem Namen
- einem Rückgabetyp für den bei der Zuweisung festzulegenden Wert

Über die folgende, selbst entworfene Annotation `VersionInfos` können bei der Zuweisung Versionsinformationen an Programmbestandteile geheftet werden:

```
public @interface VersionInfos {
   String   version();
   int      build();
   String   date() default "unknown";
   String[] contributors() default {};
}
```

Als Rückgabetypen sind bei Annotationselementen erlaubt:

- Primitive Typen
- Die Klassen **String** und **Class**
- Aufzählungstypen
- Annotationstypen
- Arrays mit einem Elementtyp aus der vorgenannten Liste

Parameter dürfen bei einem Annotationselement *nicht* vereinbart werden.

Nach dem Schlüsselwort **default** kann zu einem Annotationselement ein Voreinstellungswert angegeben werden. Dies spart Aufwand bei der Annotationsvergabe (siehe Abschnitt 9.5.2), wenn der Voreinstellungswert gerade passt, weil man in diesem Fall das Annotationselement weglassen kann. Elemente ohne **default**-Wert müssen bei der Zuweisung einer Annotation mit Werten versorgt werden.

Um bei einem Annotationselement mit Array-Typ einen leeren Array als Voreinstellung zu vereinbaren, setzt man hinter das Schlüsselwort **default** ein Paar geschweifter Klammern (also eine leere Initialisierungsliste), z. B.:

```
String[] contributors() default {};
```

Hat eine Annotation nur ein einziges Element, sollte dieses den Namen **value()** erhalten. Im folgenden Beispiel ist die API-Annotation **Retention** (aus dem Paket **java.lang.annotation**) zu sehen, deren einziges Element den Aufzählungstyp **RetentionPolicy** besitzt:

```
public @interface Retention {
   RetentionPolicy value();
}
```

Dann genügt bei der Zuweisung (siehe Abschnitt 9.5.2) an Stelle der (Name = Wert) - Notation eine Wertangabe, z. B.:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

Wie das Beispiel **Override** zeigt, kann auch eine Annotation (wie jeder andere Typ) Träger von Annotationen werden (im Beispiel: **Target** und **Retention**), wobei man von *Meta-Annotationen* spricht. Die drei im Beispiel auftauchenden API-Annotationen werden im Abschnitt 9.5.4 näher beschrieben.

Die Ableitung von einem Basistyp ist bei Annotationen *nicht* möglich.

9.5.2 Zuweisung

Eine zu vergebende Annotation wird im Quellcode dem Träger vorangestellt. In der Regel setzt man die Annotationen *vor* sonstige Dekorationen (also Modifikatoren), doch ist auch ein Mix erlaubt. Eine Annotation besteht aus einem Namen samt Präfix „@“ und einer durch runde Klammern begrenzte Elementenliste mit (Name = Wert) - Paaren. Im folgenden Beispiel wird einer Methode die Annotation `VersionInfos` zugewiesen, deren Definition im Abschnitt 9.5.1 zu sehen war:

```
@VersionInfos(version="7.1.4", build=3124, contributors={"Häcker", "Kwikki"})
public static void meth() {
   // Not yet implemented
}
```

Für die Elementenliste einer Annotation gelten folgende Regeln:

- Sie wird durch runde Klammern begrenzt.
- Sie kann bei Marker-Annotationen (ohne Elemente) entfallen, z. B.:
  ```
  @Override
  ```
- Ist nur ein Element namens **value** vorhanden, genügt die Wertangabe (ohne „**value** =“), z. B.:
  ```
  @Retention(RetentionPolicy.SOURCE)
  ```
- Elemente mit **default**-Wert dürfen weggelassen werden. Im Beispiel `VersionInfos` ist der Verzicht auf eine Datumsangabe erlaubt, weil das zugehörige Annotationselement einen **default**-Wert besitzt.
- Sind ausschließlich Annotationselemente mit einem **default**-Wert vorhanden, kann die Elementenliste wie bei einer Marker-Annotation entfallen, z. B.:
  ```
  @Deprecated
  ```
- Als Werte sind nur konstante Ausdrücke erlaubt, die der Compiler berechnen kann.
- Bei Elementen mit Referenztyp ist der Wert **null** verboten.

- Sind bei einem Annotationselement mit Array-Typ mehrere Werte zu vergeben, werden diese mit geschweiften Klammern begrenzt, z. B.:
  ```
  contributors = {"Häcker", "Kwikki"}
  ```
  Bei einem einzelnen Wert sind keine geschweiften Klammern erforderlich, z. B.:
  ```
  contributors = "Häcker"
  ```

9.5.3 Runtime-Annotationen per Reflexion auswerten

Soll eine Annotation zwecks Auswertung per Reflexion auch noch zur Laufzeit an einem Träger haften, muss bei ihrer Definition die Meta-Annotation **Retention** (vgl. Abschnitt 9.5.4) entsprechend gesetzt werden:

```
@Retention(RetentionPolicy.RUNTIME)
```

Diese Zuweisung eignet sich auch für die im Abschnitt 9.5.1 vorgestellten Annotation `VersionInfos`, die im folgenden Beispielprogramm einer Methode zum Einsatz kommt:

```
import java.lang.reflect.Method;

class AnnoReflection {
   @VersionInfos(version = "7.1.4", build = 3124, contributors = {"Häcker","Kwikki"})
   public static void meth() {
      // Not yet implemented
   }

   public static void main(String[] args) {
      VersionInfos vi;
      for (Method meth : AnnoReflection.class.getMethods()) {
         System.out.println("\npublic method "+meth.getName()+"()");
         vi = meth.getAnnotation(VersionInfos.class);
         if (vi != null) {
            System.out.println(" "+vi.version()+" ("+vi.build()+") "+vi.date());
            for (String s : vi.contributors())
               System.out.print(" "+s);
            System.out.println();
         }
      }
   }
}
```

Im Beispiel werden die Elementausprägungen der zur Methode `meth()` der Klasse `AnnoReflection` gehörigen `VersionInfos`-Instanz folgendermaßen ermittelt:

- Über das an den Klassennamen `AnnoReflection` per Punktoperator angehängte Schlüsselwort **class** wird ein Objekt der Klasse **Class** angesprochen, das diverse Kenntnisse über die Klasse `AnnoReflection` besitzt:
  ```
  AnnoReflection.class
  ```
  Dasselbe **Class**-Objekt wird übrigens auch von der Instanzmethode **getClass()** geliefert.
- Mit der **Class**-Methode **getMethods()** erhält man einen Array mit Objekten der Klasse **Method** für alle öffentlichen Methoden der Klasse `AnnoReflection` (selbst definiert oder geerbt von der Basisklasse oder von einem implementierten Interface mit **default**-Methoden):
  ```
  AnnoReflection.class.getMethods()
  ```
- Ein **Method**-Objekt kann mit der Methode **getAnnotation()** aufgefordert werden, ggf. eine Referenz zu der per Parameter vom Typ **Class** spezifizierten Annotation zu liefern:
  ```
  VersionInfos vi = meth.getAnnotation(VersionInfos.class);
  ```
- Nun lassen sich die Werte der Annotationselemente ermitteln, z. B.:
  ```
  vi.version()
  ```

Bei der Ausführung des Beispielprogramms erfährt man über die Methode `meth()` der Klasse `AnnoReflection`:

```
public method meth()
 7.1.4 (3124) unknown
 Häcker Kwikki
```

9.5.4 API-Annotationen

Anschließend werden wichtige Annotationen aus dem Java-API beschrieben, die Sie teilweise bereits kennen. Im Paket **java.lang** finden sich u.a. die folgenden, an den Compiler oder an Entwicklungswerkzeuge gerichteten Annotationen:

- **Deprecated**
  Diese Annotation wird an veraltete (überholte, abgewertete) Programmbestandteile (z. B. Methoden oder Klassen) geheftet, um Programmierer von ihrer weiteren Verwendung abzuhalten. Eventuell hat sich die Verwendung des Programmelements als problematisch herausgestellt, oder es ist eine bessere Lösung entwickelt worden. Im Kapitel 15 über Multithreading ist z. B. zu erfahren, dass die Methode **stop()** nicht mehr zum Stoppen von Threads verwendet werden sollte. Wie der Quellcode zur Klasse **Thread** zeigt, hat die Methode **stop()** die Annotation **Deprecated** erhalten:

  ```
  @Deprecated(since="1.2")
  public final void stop() {
     . . .
  }
  ```

  In der **Deprecated**-Definition wird durch die Meta-Annotation **Documented** (siehe unten) empfohlen, die Vergabe der Annotation **Deprecated** durch einen Dokumentationskommentar (vgl. Abschnitt 3.1.5) mit dem Tag **@deprecated** (kleiner Anfangsbuchstabe!) zu erläutern, was bei der **Thread**-Methode **stop()** auch geschieht:

  ```
  /**
   * Forces the thread to stop executing.
    . . .
   * @deprecated This method is inherently unsafe.  Stopping a thread with
   *       Thread.stop causes it to unlock all of the monitors that it
   *       has locked (as a natural consequence of the unchecked
   *       <code>ThreadDeath</code> exception propagating up the stack).  If
   *       any of the objects previously protected by these monitors were in
   *       an inconsistent state, the damaged objects become visible to
   *       other threads, potentially resulting in arbitrary behavior.  Many
   *       uses of <code>stop</code> should be replaced by code that simply
   *       modifies some variable to indicate that the target thread should
   *       stop running. ...
   */
  ```

  Unsere Entwicklungsumgebung IntelliJ warnt vor der Verwendung von abgewerteten Programmbestandteilen, indem die Bezeichnung im Editor durchgestrichen angezeigt wird, z. B.:

  ```
  protected void finalize() throws Throwable {
     super.finalize();
     System.out.println(this + " finalisiert");
  }
  ```

  Die Annotation **Deprecated** kennt folgende Annotationselemente:

  ```
  String since() default "";
  boolean forRemoval() default false;
  ```

- **Override**
  Mit dieser Marker-Annotation kann man seine Absicht bekunden, bei einer Methodendefinition eine Basisklassenvariante oder eine Schnittstellenmethode zu überschreiben (siehe Abschnitt 7.5.1), z. B.:
  ```
  @Override
  public void wo() {
     super.wo();
     System.out.println("Unten rechts: (" + (xpos+2*radius) +
                        ", " + (ypos+2*radius) + ")");
  }
  ```
  Misslingt dieser Plan z. B. wegen eines Tippfehlers, warnt der Compiler.

- **SuppressWarnings**
  Mit dieser Annotation überredet man den Compiler, Warnungen aus bestimmtem Anlass zu unterdrücken. Sie kann auf diverse Programmbestandteile bezogen werden (auf Typen, Felder, Methoden, Konstruktoren, Parameter, lokale Variablen). Es ist anzustreben, den Gültigkeitsbereich der Unterdrückung so klein wie möglich zu halten. Im folgenden Beispiel aus Abschnitt 8.1.3.1 werden für eine Methode die Warnungen vor nicht kontrollierbaren Typumwandlungen abgeschaltet, was stets kommentiert werden sollte:
  ```
  public E get(int index) {
     if (index >= 0 && index < size) {
        // Casting erforderlich, weil kein Array vom Typ E erstellt werden kann.
        // elements kann nur Objekte vom Typ E enthalten.
        @SuppressWarnings("unchecked")
        E result = (E) elements[index];
        return result;
     } else
        return null;
  }
  ```
  Die Annotation **SuppressWarnings** kennt ein Annotationselement:
  ```
  String[] value();
  ```
  Laut Java-Sprachbeschreibung (Gosling et al. 2019, Abschnitt 9.6.4.5) haben drei Warnungen einen in der Programmiersprache Java festgelegten Namen: **unchecked**, **deprecation**, **removal**. Weitere Namen sind von den Compiler-Herstellern abhängig:

  > Vendors are encouraged to cooperate to ensure that the same names work across multiple compilers.

- **SafeVarargs**
  Mit dieser seit Java 7 verfügbaren Marker-Annotation versichert man dem Compiler, dass eine Methode mit ihren Serienparameter von einem konkretisierten generischen Typ *keine* unsicheren Operationen ausführt, sodass der Compiler auf die Warnung verzichtet (siehe Abschnitt 8.1.2.4), z. B.:
  ```
  @SafeVarargs
  private void genVarargs(Method(ArrayList<String>... stringLists) {
      System.out.println(stringLists.length);
  }
  ```

Im Paket **java.lang.annotation** finden sich wichtige Meta-Annotationen, welche z. B. die erlaubte Verwendung oder den Gültigkeitsbereich einer Annotation betreffen:

- **Documented**
  Die Vergabe einer so dekorierten Annotation sollte in einem Dokumentationskommentar erläutert werden.
- **Inherited**
  Eine so dekorierte Annotation wird von einer Klasse an ihre Ableitungen vererbt.

- **Retention**
  Über einen Wert vom Aufzählungstyp **java.lang.annotation.RetentionPolicy** wird festgelegt, wo eine Annotation verfügbar sein soll:
  - **SOURCE**
    Die Annotation ist nur in der Quellcodedatei vorhanden.
  - **CLASS** (= Voreinstellung)
    Die Annotation ist auch in der Bytecodedatei vorhanden, aber zur Laufzeit nicht verfügbar.
  - **RUNTIME**
    Die Annotation ist auch zur Laufzeit verfügbar.

  Um für eine Annotation die im Abschnitt 9.5.3 beschriebene Reflexion zu ermöglichen, muss sie bei der Meta-Annotation **Retention** den Wert **RUNTIME** erhalten.
- **Target**
  Über einen Array mit Werten vom Aufzählungstyp **java.lang.annotation.ElementType** wird festgelegt, für welche Programmelemente eine Annotation verwendbar ist. Z. B. kann eine folgendermaßen dekorierte Annotation
  ```
  @Target({ElementType.CONSTRUCTOR, ElementType.METHOD})
  ```
  einer Methode oder einem Konstruktor zugewiesen werden.

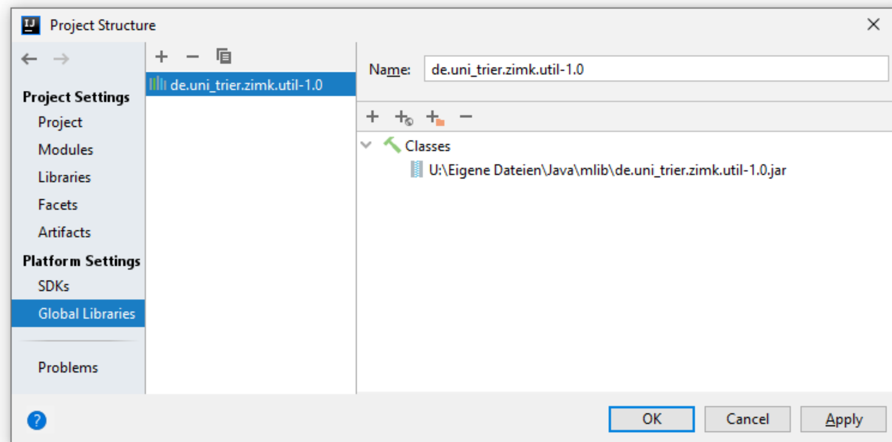
9.6 Übungsaufgaben zum Kapitel 9

1) Welche der folgenden Aussagen sind richtig bzw. falsch?

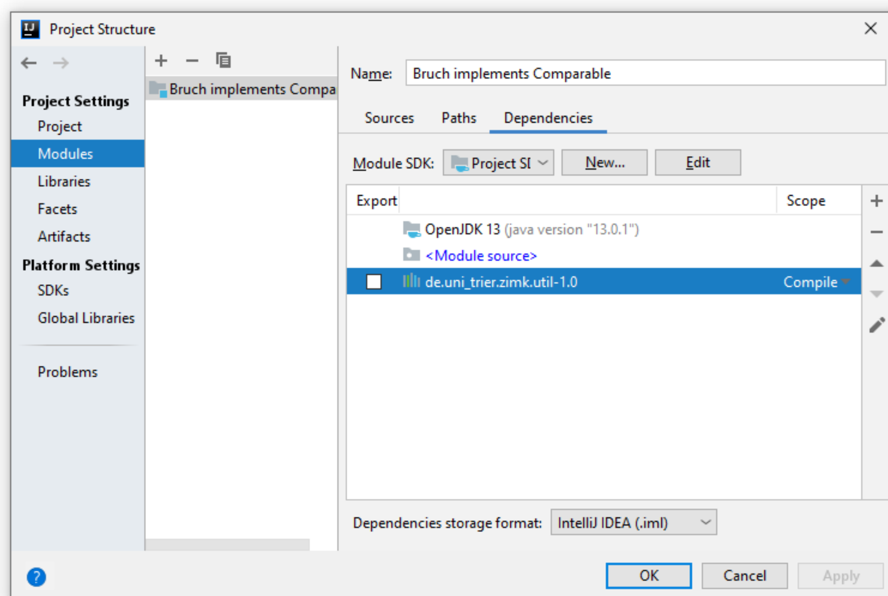
1. Eine Schnittstelle ist per Voreinstellung (ohne Zugriffsmodifikator) als **public** definiert.
2. Die Methoden einer Schnittstelle sind per Voreinstellung (ohne Zugriffsmodifikator) als **public** definiert.
3. Eine Schnittstelle muss mindestens eine Methode enthalten.
4. Die Felder einer Schnittstelle sind implizit als **public**, **static** und **final** deklariert.
5. Annotationen sind spezielle Schnittstellen.

2) Erstellen Sie zur Klasse `Bruch`, die im Kapitel 4 als zentrales Beispiel diente, eine Variante, welche die Schnittstelle **Comparable**`<Bruch>` implementiert, sodass z. B. ein `Bruch`-Array mit der statischen Methode **sort()** aus der Klasse **Arrays** sortiert werden kann.

Nachdem wir uns im Kapitel 6 mit Paketen beschäftigt haben, sollte die Klasse `Bruch` in ein Paket (z. B. `de.uni_trier.zimk.matrain.br`) eingefügt und die in der `Bruch`-Definition verwendete Klasse `Simput` aus dem Paket `de.uni_trier.zimk.util.conio` bezogen werden (vgl. Abschnitt 6.2.7.1). Allerdings können Sie der Einfachheit halber auf die mit Java 9 eingeführte Modultechnik verzichten und die modulare **jar**-Datei **de.uni_trier.zimk.util-1.0.jar** mit dem Modul `de.uni_trier.zimk.util`, das u.a. das Paket `de.uni_trier.zimk.util.conio` enthält, wie eine traditionelle **jar**-Datei behandeln. Diese **jar**-Datei (oder der Ordner mit dieser **jar**-Datei) kann z. B. in IntelliJ (nach dem Menübefehl **File > Project Structure**) als globale Bibliothek vereinbart



und dann dem Projekt zugewiesen werden:



Weil die Klasse `Simput` mit Java 13 übersetzt worden ist (vgl. Abschnitt 6.2.7.1), muss ein passendes Projekt-SDK gewählt werden.

3) Definieren Sie eine generische Methode mit einem Parameter, dessen Typ von einer bestimmten Klasse abstammen und zwei Interfaces implementieren muss.

10 Java Collections Framework

Die in diesem Kapitel vorgestellten Typen zur Verwaltung von Listen, Mengen, (Schlüssel-Wert) - Tabellen (Abbildungen) oder Warteschlangen gehören zu dem mit Java 1.2 eingeführten **Java Collections Framework (JCF)**. Diese Sammlung von Schnittstellen und Klassen aus den Paketen **java.util** und **java.util.concurrent** wird von praktisch jedem Java-Programmierer in vielen Anwendungen intensiv zur Datenverwaltung genutzt.

Das JCF hat enorm von der in Java 5 eingeführten Generizität profitiert und war ein primärer Grund für die Erweiterung der Programmiersprache Java um die Generizität (Naftalin & Wadler 2007). Im Abschnitt 8.1.1 haben Sie einen Eindruck davon erhalten, welchen Fortschritt eine generische Listenverwaltungsklasse wie **ArrayList<E>** gegenüber der auf unsicheren **Object**-Referenzen und expliziter Typumwandlung basierenden Vorgängerlösung bei der häufig benötigten Verwaltung einer Liste mit Elementen *desselben* Typs darstellt.

Wer nach der Lektüre von Kapitel 8 noch Zweifel am Nutzen der generischen Typen und Methoden hatte, lernt nun zahlreiche generische Interfaces und Klassen mit hohem praktischem Nutzwert kennen, was im Hinblick auf die Generizität zu einem Erfahrungs- und Motivationsgewinn führen sollte. Zugleich wird allgemein belegt, dass auch scheinbar abstrakte Java-Sprachmerkmale (wie die Generizität) die (professionelle) Praxis enorm erleichtern.

Für die Objekte der im aktuellen Kapitel vorzustellenden Klassen wird im Manuskript alternativ zur offiziellen Bezeichnung *Kollektionen* aus sprachlichen Gründen gelegentlich auch die Bezeichnung *Container* verwendet.

In diesem Kapitel beschäftigen wir uns nicht damit, eigene generische Kollektionstypen zu definieren (siehe einfache Beispiele im Abschnitt 8.1.3), sondern wir konzentrieren und darauf, die im JCF zahlreich vorhandenen Typen zu nutzen. Diese Typen besitzen ausgefeilte Handlungskompetenzen (Methoden) für typische Aufgabenstellungen (z. B. Vereinigung von zwei Mengen ohne Entstehung von Dubletten), damit Programmierer möglichst selten „das Rad neu erfinden müssen“. Wenn doch eigene Kollektionsklassen erforderlich eignen sich als Basis die abstrakten Klassen **AbstractCollection<E>**, **AbstractList<E>**, **AbstractSet<E>**, **AbstractMap<K,V>** und **AbstractQueue<E>** im JCF.

Weitere Details zum Java Collections Framework liefern z. B. bei Evans & Flanagan (2015) sowie Naftalin & Wadler (2007).

Mit Thread-sicheren Kollektionen werden wir uns im Abschnitt 15.6 beschäftigen.

10.1 Arrays versus Kollektionen

Auch die im Abschnitt 5.1 vorgestellten Arrays taugen als Container für (angeordnete) Elemente mit einem identischen, frei wählbaren Typ und bieten einen schnellen Indexzugriff auf die Elemente. Man kann sich fragen, wozu eigentlich noch weitere Kollektionsklassen benötigt werden.

Beginnen wir bei den häufig auftretenden listenartigen Datenstrukturen. Dabei zeigen Arrays folgende Schwächen:

- Die Größe eines Arrays kann nach der Erzeugung nicht mehr geändert werden.
- Soll ein neues Element an der mittleren Position *k* eingefügt werden, dann müssen alle alten Elemente ab dieser Position aufwändig nach rechts verschoben werden. Ein analoger Aufwand entsteht beim Löschen des Elements an der mittleren Position *k*.

Das JCF bietet hingegen Listenverwaltungsklassen, die eine automatische Vergrößerung sowie performantes Einfügen und Löschen beherrschen.

Sind für Elementsammlungen häufige Existenzprüfungen erforderlich, bietet ein Array wenig Unterstützung. Sind seine Elemente nicht sortiert, muss für jedes Element geprüft werden, ob es mit dem gesuchten Element übereinstimmt. JCF-Kollektionen zur Verwaltung von Mengen bieten hingegen schnelle Detektionsmöglichkeiten und verhindern außerdem identische Elemente (Dubletten).

In der Praxis ist oft eine Menge von (Schlüssel-Wert) - Paaren (Assoziationen) zu verwalten, z. B. eine Tabelle mit den bei einem Web-Dienst aktuell angemeldeten Benutzern, wobei eine eindeutige Kennung als Schlüssel fungiert und auf ein Objekt mit den Eigenschaften des Benutzers zeigt. Eventuell stammen die Eigenschaften aus einer Datenbankzeile, die nach der Anmeldung des Benutzers von einem Datenbankserver bezogen und dann zum schnellen Zugriff im Hauptspeicher aufbewahrt wird. Es melden sich ständig Benutzer an oder ab, und beim Versuch, eine solche Datenstruktur mit einem Array zu verwalten, treten die eben beschriebenen Probleme auf (feste Anzahl von Elementen, umständliches Einfügen und Löschen, aufwändige Existenzprüfungen).

Im zu modellierenden Aufgabenbereich einer Anwendung treten oft Datenstrukturen vom Typ Liste, Menge, (Schlüssel-Wert) - Tabelle oder Warteschlange auf, und im Java Collections Framework finden sich passende Typen, sodass im Vergleich zur Verwendung von Arrays eine bessere Modellierung und ein besser lesbarer Quellcode resultieren.

Manche JCF-Kollektionsklassen verwenden im Hintergrund einen Array zur Datenspeicherung (z. B. **ArrayList<E>**), was in bestimmten Anwendungsfällen zu einer performanten Lösung führt (siehe z. B. Abschnitt 10.3.4 mit Einsatzempfehlungen für die Listenverwaltung.

Im Abschnitt 8.1.2.2 hat sich herausgestellt, dass die sogenannte *Kovarianz* von Arrays regelrecht als Defekt angesehen werden muss. Während der Compiler z. B. **ArrayList<String>** *nicht* als Spezialisierung von **ArrayList<Object>** akzeptiert, übersetzt er leider die folgenden Anweisungen ohne jede Kritik:

```
Object[] arrObject = new String[5];
arrObject[0] = 13;
```

Zur Laufzeit ärgern sich die Benutzer über einen Fehler vom Typ **ArrayStoreException**.

Trotz der guten Argumente für die Kollektionstypen gibt es aber weiterhin berechtigte Einsatzzwecke für Arrays, und das nicht nur als Hintergrundspeicher von Kollektionsklassen wie **ArrayList<E>**. Dass in Kollektionsklassen als Elemente nur *Objekte* erlaubt sind, wird bei der Verwaltung einer großen Anzahl primitiver Werte zum Nachteil:

- Das Auto(un)boxing verursacht einen spürbaren Zeitaufwand (für das Erstellen und Entsorgen von Objekten).
- Der Speicherbedarf für die Objekte ist relativ hoch.

In einem Array kann man demgegenüber auch eine große Anzahl primitiver Werte zeit- und platzsparend abgelegen.

Wenn es erforderlich ist, zwischen einem Array mit Referenz-Elementtyp und einer Liste umzusteigen, bietet das Java-API geeignete Methoden an:

- **public static <E> List<E> asList(E...** *a*)
  Die statische und generische Methode **asList()** der Klasse **Arrays** erstellt aus dem Array, der hinter dem Serienparameter der Methode steckt (vgl. Abschnitt 4.3.1.4.3), ein Objekt vom Typ **List<E>**.[1]
- **public <T> T[] toArray(T[]** *a*)
  Das Interface **Collection<E>** verlangt von implementierenden Klassen die generische Instanzmethode **toArray()**, die einen Array mit den Elementen der Kollektion liefert (siehe Abschnitt 10.2.1).

10.2 Zur Rolle von Schnittstellen beim JCF-Design

Wie bei jedem Framework spielen auch beim JCF Schnittstellen eine wichtige Rolle. Im Kapitel 9 haben Sie erfahren, dass ein *Interface* meist aus einer Liste von Instanzmethoden besteht, die entweder nur abstrakt definiert sind oder eine **default**-Implementierung besitzen.

Das JCF enthält ...

- für wichtige Datenstrukturen (Liste, Menge, Abbildung, Warteschlange) jeweils eine Schnittstelle mit einer Liste von Methoden, die jede Lösung zu erfüllen hat,
- und für jede Schnittstelle verschiedene Implementierungen, welche trotz unterschiedlicher Innenarchitekturen dieselben Methoden beherrschen, sodass problemadäquat ein einfacher Wechsel möglich ist.

Anschließend ist das im Abschnitt 10.2.1 zu beschreibende Interface **Collection<E>** aus dem Paket **java.util** zu sehen, das (quasi als Pflichtenheft) grundlegende Kompetenzen einer Kollektionsklasse vorschreibt:

```
public interface Collection<E> extends Iterable<E> {
   boolean add(E e);
   boolean addAll(Collection<? extends E> c);
   void clear();
   boolean contains(Object o);
   boolean containsAll(Collection<?> c);
   boolean equals(Object o);
   int hashCode();
   boolean isEmpty();
   Iterator<E> iterator();
   default Stream<E> parallelStream() {
      return StreamSupport.stream(spliterator(), true);
   }
   boolean remove(Object o);
   boolean removeAll(Collection<?> c);
   default boolean removeIf(Predicate<? super E> filter) {
      . . .
      return removed;
   }
   boolean retainAll(Collection<?> c);
```

[1] Man erhält ein Objekt der in **Arrays** enthaltenen statischen Mitgliedsklasse **ArrayList<E>** (nicht zu verwechseln mit der Top-Level-Klasse **ArrayList<E>**), die das Interface **List<E>** erfüllt, z. B.:

```
String[] sar = {"a", "b", "c"};
List<String> li = Arrays.asList(sar);
```

Die resultierende Liste speichert ihre Daten intern in einem Array und hat eine feste Länge, was bei Kollektionen ungewöhnlich ist. Auf den Versuch, ein weiteres Element einzufügen, reagiert sie mit einer Ausnahme vom Typ **UnsupportedOperationException**. Ist eine *erweiterbare* Liste das Ziel, kann man einem Vorschlag von Evans & Flanagan (2015, S. 257) folgend die **asList()** - Rückgabe z. B. an einen **ArrayList<E>** - Konstruktor weiterreichen:

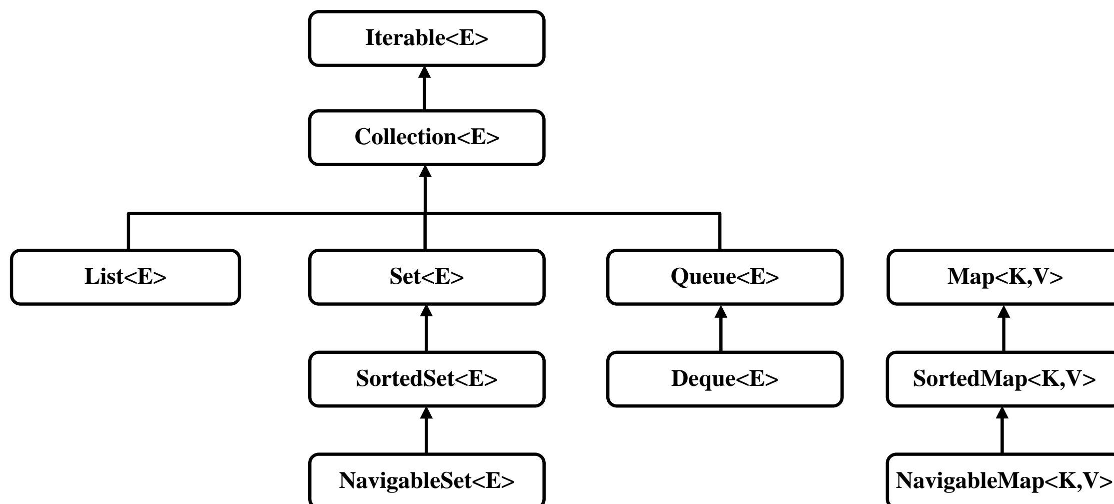
```
List<String> li = new ArrayList<>(Arrays.asList(sar));
```

```
      int size();
      @Override
      default Spliterator<E> spliterator() {
         return Spliterators.spliterator(this, 0);
      }
      default Stream<E> stream() {
         return StreamSupport.stream(spliterator(), false);
      }
      Object[] toArray();
      <T> T[] toArray(T[] a);
   }
```

Will eine Klasse von sich behaupten, das Interface **Collection<E>** zu implementieren, muss sie alle abstrakten Interface-Methoden implementieren. Hinzu kommen noch die Methoden im Interface **Iterable<E>**, das von **Collection<E>** erweitert wird.

Ein Interface ist aber nicht nur ein **Pflichtenheft**, sondern auch ein **Datentyp**. Wird ein Interface z. B. als Datentyp für einen Formalparameter einer Methode vorgeschrieben, ist beim Methodenaufruf als Aktualparameter-Datentyp jede Klasse erlaubt, die das Interface implementiert. So kann die Methode mit diversen, z. B. auch mit später definierten Klassen zusammenarbeiten. Damit leisten Interfaces einen wichtigen Beitrag zur Realisation von Software nach dem Open-Closed - Prinzips (vgl. Abschnitt 4.1.1.3): Neue Anforderungen können durch zusätzliche Klassen realisiert werden, ohne dass die vorhandene Code-Basis geändert werden muss. Um diese Flexibilität zu erzielen, sollten Interface-Datentypen intensiv verwendet werden (z. B. bei Methodendefinitionen).

Analog zur Erweiterung einer Klasse durch abgeleitete Klassen lassen sich zu einem Interface erweiterte (abgeleitete) Varianten definieren, die von implementierenden Klassen zusätzliche Methoden verlangen. Für das Java Collections Framework ist so eine Interface-Hierarchie entstanden, die einen guten Eindruck von den Kompetenzprofilen der im Framework enthaltenen Klassen vermittelt. In der folgenden Abbildung sind die JCF-Schnittstellen zu sehen, die im weiteren Verlauf des Kapitels zusammen mit implementierenden Klassen behandelt werden:



In der Abbildung ist auch das (nicht dem JCF zugerechnete) Interface **Iterable<T>** enthalten, das von **Collection<E>** erweitert wird (siehe Abschnitte 10.2.1 und 10.4).

10.2.1 Das Interface Collection<E> mit Basiskompetenzen

Implementiert eine Klasse direkt oder indirekt das generische Interface **Collection<E>**, dann muss sie u.a. die folgenden Instanzmethoden beherrschen:

- **public boolean add(E** *element***)** (optionale Operation)
  Wenn die Kollektion aufgrund der per **add()** beantragten Neuaufnahme verändert wurde, dann liefert die Methode den Rückgabewert **true**. Manche Kollektionen verweigern die Aufnahme von Dubletten und liefern ggf. den Rückgabewert **false**. Scheitert die Aufnahme aus einem anderen Grund (z. B. Ablehnung von **null**-Elementen), dann wirft die Methode eine Ausnahme (z. B. vom Typ **NullPointerException**).[1] Wie für Arrays gilt auch für Kollektion, dass sie keine vollständigen Objekte aufnehmen, sondern *Referenzen* auf Objekte.
- **public boolean addAll(Collection<? extends E>** *collection***)** (optionale Operation)
  Wenn die angesprochene Kollektion aufgrund der beantragten Neuaufnahme einer kompletten Kollektion verändert wurde, liefert die Methode den Rückgabewert **true**. Auf Fehler reagiert **addAll()** analog zu **add()** (siehe oben). Durch eine gebundene Wildcard-Typdeklaration (siehe Abschnitt 8.3) wird für die Aufnahmekandidaten der Elementtyp der im **addAll()** - Aufruf angesprochenen Kollektion oder eine Spezialisierung vorgeschrieben. Das Verhalten der Methode **addAll()** ist undefiniert, wenn während ihrer Ausführung die Kollektion (z. B. durch einen anderen Thread) verändert wird.
- **public void clear()** (optionale Operation)
  Mit dieser Methode fordert man eine Kollektion auf, alle Elemente zu löschen. Dabei werden die im Kollektionsobjekt vorhandenen Objektreferenzen gelöscht, nicht die Objekte selbst.[2]
- **public boolean contains(Object** *object***)**
  Diese Methode informiert darüber, ob ein Element der Kollektion im Sinne der **equals()** - Methode der Elementklasse mit dem Aktualparameter übereinstimmt.
- **public boolean containsAll(Collection<?>** *collection***)**
  Diese Methode informiert darüber, ob die angesprochene Kollektion im Sinne der **equals()** - Methode der Elementklasse alle Elemente der Parameterkollektion enthält.
- **public boolean isEmpty()**
  Mit dieser Methode findet man heraus, ob die angesprochene Kollektion leer ist.
- **public Iterator<E> iterator()**
  Diese Methode liefert ein Iterator-Objekt, das ein sequentielles Aufsuchen der Kollektionselemente ermöglicht. Sie gehört zum Interface **Iterable<E>**, das vom Interface **Collection<E>** erweitert wird. Details zu Iteratoren folgen im Abschnitt 10.4.
- **public default Stream<E> parallelStream()**
  Diese Methode liefert einen möglicherweise parallelen Strom mit der angesprochenen Kollektion als Quelle (zur Strombearbeitung siehe Abschnitt 12.2).
- **public boolean remove(Object** *obj***)** (optionale Operation)
  Diese Methode entfernt ggf. *ein* Element aus der Kollektion, das sich vom Parameterobjekt gemäß der **equals()** - Methode der Elementklasse nicht unterscheidet. Zeigt der Aktualparameter auf **null**, wird ggf. ein **null**-Element aus der Kollektion entfernt. Mit dem Rückgabewert informiert die Methode darüber, ob die Kollektion tatsächlich geändert wurde.
- **public boolean removeAll(Collection<?>** *collection***)** (optionale Operation) (optionale Operation)
  Diese Methode entfernt *alle* Elemente aus der angesprochenen Kollektion, die mit einem Element der Parameterkollektion gemäß der **equals()** - Methode der Elementklasse gleich

---

[1] Mit der Ausnahmebehandlung werden wir uns bald im Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.** beschäftigen.

[2] Ein Objekt wird in Java bekanntlich vom Garbage Collector gelöscht, wenn im gesamten Programm keine Referenz auf das Objekt mehr vorhanden ist, und die JVM wegen eines Mangels an Heap-Speicher eine Aufräumaktion veranlasst.

sind. Mit dem Rückgabewert informiert die Methode darüber, ob die Kollektion geändert wurde. Durch eine ungebundene Wildcard-Typdeklaration (siehe Abschnitt 8.3) wird für die Parameterkollektion das Implementieren einer Konkretisierung der generischen Schnittstelle **Collection<E>** mit beliebigem Referenzelementtyp vorgeschrieben.

- **public default boolean removeIf(Predicate<? super E>** *bedingung***)**
  Diese Methode entfernt ggf. alle Elemente aus der angesprochenen Kollektion, die eine Bedingung erfüllen, welche durch die Methode **test()** der funktionalen Schnittstelle **Predicate<T>** geprüft wird (zu funktionalen Schnittstellen siehe Abschnitt 12.1.1.1). Mit dem Rückgabewert informiert die Methode darüber, ob die Kollektion geändert worden ist. Durch die **super**-gebundene Wildcard-Typdeklaration (siehe Abschnitt 8.3.1.2) werden auch Tester zugelassen, die nicht Objekte vom Kollektionselementtyp **E** beurteilen, sondern Objekte einer Basisklasse von **E**.
- **public boolean retainAll(Collection<?>** *collection***)** (optionale Operation)
  Diese Methode entfernt ggf. *alle* Elemente aus der angesprochenen Kollektion, die *nicht* mit einem Element der Parameterkollektion gemäß der **equals()** - Methode der Elementklasse gleich sind. Mit dem Rückgabewert informiert die Methode darüber, ob die Kollektion geändert wurde.
- **public int size()**
  Diese Methode liefert die Anzahl der Elemente in der Kollektion.
- **public default Stream<E> stream()**
  Diese Methode liefert einen sequentiellen Strom mit der angesprochenen Kollektion als Quelle (zur Strombearbeitung siehe Abschnitt 12.2).
- **public Object[] toArray()**
  Es wird ein **Object**-Array erstellt, mit den Elementen der Kollektion befüllt und abgeliefert. Ist für den Iterator der Kollektion eine Ordnung definiert, dann sind die Array-Elemente entsprechend angeordnet. Die Methode liefert keine Referenz auf das eventuell von einer Kollektion intern zur Datenablage verwendete Array-Objekt, sondern erstellt auf jeden Fall ein neues Array-Objekt, das also gefahrlos vom Aufrufer geändert werden darf.
- **public <T> T[] toArray(T[]** *a***)**
  Es wird ein Array mit den Elementen der Kollektion geliefert, wobei als Datentyp für diesen Array der Laufzeittyp des Parameter-Arrays übernommen wird. Es ist zu beachten, dass **T** *nicht* der Typformalparameter von **Collection<E>** ist, sondern für einen beliebigen kompatiblen Typ steht. Bei **toArray(T[]** *a***)** handelt es sich um eine generische Methode. Wenn die erforderliche Typumwandlung scheitert, kommt es zu einem Laufzeitfehler vom Typ **ArrayStoreException**. Ist die Kapazität des Parameter-Arrays ausreichend, werden die Kollektionselemente dort eingefüllt. Anderenfalls wird ein neuer Array passender Größe erstellt.

Die beiden **toArray()** - Überladungen unterstützen die Kooperation von Kollektionen mit Software, die Array-Datentypen verwendet (z. B. als Methodenparameter).

Für alle zu einer Änderung der Kollektion führenden Methoden (z. B. **add()**, **addAll()**, **clear()**, **remove()** usw.) ist in der API-Dokumentation der Zusatz *optional operation* angegeben (vgl. Abschnitt 10.2.2). Es ist einer Klasse erlaubt, sich in der Implementation solcher Methoden auf das Werfen einer **UnsupportedOperationException** zu beschränken. Es wird allerdings von jeder implementierenden Klasse erwartet, in der Dokumentation offenzulegen, für welche Methoden nur eine Pseudo-Implementation vorhanden ist.

Es verwundert leicht, dass bei den Methoden **contains()** und **remove()** nicht der Typformalparameter **E** sondern die Klasse **Object** als Parametertyp zum Einsatz kommt. Eine überzeugende Erklä-

rung hat Luke Hutteman auf Stack Overflow präsentiert:[1] Auf diese Weise können die Methoden auch dann eingesetzt werden, wenn der unbeschränkte Wildcard-Datentyp verwendet wird, z. B.:

Quellcode	Ausgabe

<pre>import java.util.*;
class Prog {
    public static void main(String[] args) {
        List<String> mis = new ArrayList<>();
        mis.add("A"); mis.add("B");
        System.out.println(mis);
        List<?> wc = mis;
        wc.remove(1);
        System.out.println(mis);
    }
}</pre>
	<pre>[A, B]
	[A]</pre>

Wo das Verhalten einer Methode von Übereinstimmungsprüfungen abhängt (z. B. **contains()**, **remove()**), ist bei der Interface-Implementierung die **equals()** - Methode des Elementtyps zu verwenden (statt des Identitätsoperators). Dementsprechend wird in der Elementklasse für die von **Object** geerbte **equals()** - Methode eine sinnvolle Überschreibung erwartet, die eine Äquivalenzrelation (im mathematischen Sinn) realisieren muss (vgl. Bloch 2018, S. 38ff), d. h.:

- Reflexivität
  Für jede nicht auf **null** zeigende Referenzvariable `x` muss `x.equals(x)` den Wert **true** liefern.
- Symmetrie
  Für zwei nicht auf **null** zeigende Referenzvariablen `x` und `y` muss gelten: `x.equals(y)` liefert den Wert **true** genau dann, wenn `y.equals(x)` den Wert **true** liefert.
- Transitivität
  Für drei nicht auf **null** zeigende Referenzvariablen `x`, `y` und `z` muss gelten: Aus `x.equals(y)` und `y.equals(z)` folgt `x.equals(z)`.

Außerdem muss eine sinnvolle **equals()** - Überschreibung die folgenden Bedingungen erfüllen:

- Konsistenz
  Für zwei nicht auf **null** zeigende Referenzvariablen `x` und `y` muss `x.equals(y)` bei jedem Aufruf den konstanten Wert **true** oder **false** liefern, solange bei den Objekten keine für den Vergleich relevante Änderung stattgefunden hat.
- Für jede nicht auf **null** zeigende Referenzvariable `x` muss `x.equals(null)` den Wert **false** liefern.

In der API-Klasse **Integer** (Paket **java.lang**) ist z. B. eine geeignete **equals()** - Überschreibung vorhanden:

```
public boolean equals(Object obj) {
    if (obj instanceof Integer) {
        return value == ((Integer)obj).intValue();
    }
    return false;
}
```

Verwendet eine Kollektionsklasse intern einen Array zum Speichern ihrer Elemente, dann ist die maximale Kapazität begrenzt durch den größtmöglichen Array-Index (= **Integer.MAX_VALUE** = 2147483647).

[1] https://stackoverflow.com/questions/4269147/why-does-java-mapk-v-take-an-untyped-parameter-for-the-get-and-remove-methods

10.2.2 Optionale Operationen

Wer die Dokumentation zur Schnittstelle **Collection<E>** studiert, stellt verwundert fest, dass sich bei etlichen Methoden der Zusatz *optional operation* findet, der aber *nicht* als *optional implementation* missverstanden werden darf:

`boolean`	`add(E e)` Ensures that this collection contains the specified element (optional operation).
`boolean`	`addAll(Collection<? extends E> c)` Adds all of the elements in the specified collection to this collection (optional operation).
`void`	`clear()` Removes all of the elements from this collection (optional operation).
`boolean`	`contains(Object o)` Returns `true` if this collection contains the specified element.
...	...

Mit dem Zusatz *optional operation* will der Schnittstellendesigner keinesfalls vorschlagen, eine betroffene Methode beim Implementieren wegzulassen, was zu einem Protest des Compilers führen würde. Es wird vielmehr eine Implementation wie im folgenden Beispiel (entnommen aus dem Quellcode der API-Klasse **AbstractCollection<E>**) verbunden mit einer entsprechenden Dokumentation als *akzeptabel* dargestellt:

```
public boolean add(E e) {
    throw new UnsupportedOperationException();
}
```

Diese Methode führt keine Aufträge aus, sondern meldet nur per Ausnahmeobjekt: „Ich kann das nicht."

Die merkwürdige Lösung mit „optionalen" Schnittstellenmethoden und Pseudoimplementationen dient beim Java Collections Framework dazu, die folgenden Ziele gemeinsam zu realisieren:

- Die Zahl der Schnittstellen soll möglichst klein gehalten werden.
- Spezielle Kollektionsklassen sollen eine Schnittstelle (z. B. **Collection<E>**) erfüllen, aber keine Änderungen (z. B. durch Aufnahme neuer Elemente) erlauben (z. B. die im Abschnitt 10.9 erwähnten unveränderlichen Kollektionen).

Mit dieser Besonderheit befindet sich das JCF zwar nicht syntaktisch, aber doch semantisch in partiellem Widerspruch zu den Erläuterungen über Schnittstellen als Verpflichtungserklärungen (vgl. z. B. Abschnitt 9.1.2), und eine Nachahmung ist *nicht* zu empfehlen.

10.3 Listen

Eine JCF-Liste enthält eine Sequenz von Elementen (Objektreferenzen) desselben Typs mit einer definierten Reihenfolge, auf die man sequentiell sowie wahlfrei über einen nullbasierten Index zugreifen kann. Es ist also wie bei einem Array möglich, das Element an einer bestimmten Position abzurufen oder zu verändern.

Im Unterschied zu einem Array wird eine Liste bei Bedarf automatisch vergrößert. Wir haben also einen größendynamischen Container zur Verfügung, der dank Typgenerizität Elemente von einem wählbaren Referenztyp sortenrein (mit Compiler-Typsicherheit) verwaltet.

Für Listen finden sich sehr viele Einsatzmöglichkeiten bei der Software-Entwicklung. Man verwendet sie z. B. für ...

- die aktuell von einem Steuerelement der Bedienoberfläche angebotenen Optionen,
- die Bestellungen eines Kunden, der aus einer Datenbank geladen wurde,
- die aus einem Text sukzessiv extrahierten Wörter.

Die Elemente einer Liste müssen (im Unterschied zu den Elementen einer Menge, vgl. Abschnitt 10.5) *nicht* verschieden sein, d.h.:

- Mehrere Elemente können dasselbe Objekt referenzieren (also dieselbe Adresse enthalten)
- Mehrere Referenzziele können im Sinn der **equals()** - Methode der Elementklasse inhaltsgleich sein.

Im späteren Kapitel 13 über die Programmierung von grafischen Bedienoberflächen mit JavaFX-Technik werden *beobachtbare* Listen behandelt. Diese können durch ein **ListView<E>** - Steuerelement präsentiert werden. Außerdem lassen sich bei einer solchen Liste Beobachter registrieren, die über bestimmte Veränderungen (z. B. Aufnahme neuer Elemente, Auftreten bestimmter Werte bei einem Element) informiert werden wollen (siehe Abschnitt 13.5.3.3).

10.3.1 Das Interface List<E>

Zur Realisation von Listen enthält das Java Collections Framework mehrere Klassen mit unterschiedlichen Techniken zum Speichern der Elemente, die in verschiedenen Einsatzszenarien stark abweichende Leistungen zeigen. Alle implementieren das von **Collection<E>** abstammende Interface **List<E>** und besitzen folglich über die **Collection<E>** - Methoden (siehe Abschnitt 10.2.1) hinaus u.a. die folgenden Kompetenzen:

- **public void add(int** *index*, **E** *element*) (optionale Operation)
  Es wird ein neues Element an der gewünschten Indexposition eingefügt.
- **public boolean addAll(int** *index*, **Collection<? extends E>** *coll*) (optionale Operation)
  Diese Methode fügt die Elemente der Parameterkollektion an der gewünschten Position in die Liste ein. Wenn die Liste aufgrund des Aufrufs verändert wurde, dann liefert die Methode den Rückgabewert **true**.
- **public E get(int** *index*)
  Das Element mit dem gewünschten Index wird geliefert (wahlfreier Zugriff).
- **public int indexOf(Object** *obj*)
  Sind Elemente vorhanden, die im Sinne der Methode **equals()** der Elementklasse mit dem Parameterobjekt übereinstimmen, wird der kleinste zugehörige Index geliefert, ansonsten der Wert -1. Ist der Parameter gleich **null**, liefert die Methode ggf. den kleinsten Index zu einem Element, das gleich **null** ist.
- **public int lastIndexOf(Object** *obj*)
  Sind Elemente vorhanden, die im Sinne der Methode **equals()** der Elementklasse mit dem Parameterobjekt übereinstimmen, wird der größte zugehörige Index geliefert, ansonsten der Wert -1. Ist der Parameter gleich **null**, liefert die Methode ggf. den größten Index zu einem Element, das gleich **null** ist.
- **public ListIterator<E> listIterator()**
  Diese Methode liefert einen Iterator, der die Schnittstelle **ListIterator<E>** erfüllt und daher mehr Kompetenzen besitzt als eine **Iterator<E>** - Implementation:
    - Es dürfen Elemente eingefügt und verändert werden.
    - Die Rückwärtsbewegung wird unterstützt.
    - Die aktuelle Indexposition kann abgefragt werden.

  Es ist noch eine **listIterator()** - Überladung mit Parameter für die Indexstartposition vorhanden. Details zu Iteratoren folgen im Abschnitt 10.4.

- **public E remove(int** *index***)** (optionale Operation)
  Diese Methode entfernt das Element an der Position *index* aus der Liste und liefert dessen Adresse zurück.
- **public E set(int** *index*, **E** *element***)** (optionale Operation)
  Das Element mit der im ersten Parameter genannten Position wird durch das Objekt im zweiten Parameter ersetzt.
- **public List<E> subList(int** *fromIndex*, **int** *toIndex*,**)**
  Man erhält ein Segment der angesprochenen Liste von der Indexposition *fromIndex* (inklusive) bis zur Indexposition *toIndex* (exklusive). Weil nur die Referenzen, aber nicht die referenzierten Listenelemente kopiert werden, wirken sich alle per Subliste durchgeführten Änderungen auf die Originalliste aus. Über die Subliste sind auch alle optionalen Operationen (z. B. **add**(), **remove**(), **clear**()) zulässig. Werden nach dem Erstellen der Subliste strukturelle Änderungen *über die Originalliste* vorgenommen, führt die anschließende Verwendung der Subliste zu einem Laufzeitfehler vom Typ **ConcurrentModificationException**.

Auf fehlerhafte Aktualparameter (z. B. Indexwerte < 0 oder $\geq$ **size**(), ungeeignete Elemente) reagieren die **List<E>** - Methoden mit einem Ausnahmefehler (z. B. **IndexOutOfBoundsException**, **ClassCastException**).

Die bereits im Interface **Collection<E>** vorhandenen Methoden **add(E** *element***)** und **addAll(Collection<? extends E>** *c***)** müssen von einer Listenverwaltungsklasse so implementiert werden, dass neue Elemente *am Ende* der Liste angehängt werden.

Warum die Methoden **indexOf()** und **lastIndexOf()** den Formalparametertyp **Object** (statt **E**) verwenden, wurde im Abschnitt 10.2.1 anhand analoger Beispiele erläutert.

Seit Java 9 besitzt die Schnittstelle **List<E>** eine statische Fabrikmethode namens **of()**, mit der sich auf einfache Weise *unveränderliche* **List<E>** - Objekte erstellen lassen, z. B.:[1]

Quellcode

```java
import java.util.*;
class Prog {
    public static void main(String[] args) {
        List<String> los = List.of("a", "b", "c", "d");
        System.out.println(los);
        // los.add("f");
    }
}
```

Ausgabe

```
[a, b, c, d]
```

Die auskommentierte Anweisung hätte einen Laufzeitfehler vom Typ **UnsupportedOperationException** zur Folge, weil die **of()** - Produkte unveränderlich sind.

Den Aufruf der statischen, generischen Methode richtet man an den Rohtyp der generischen Schnittstelle **List<E>**. Wegen der Typinferenz-Fähigkeiten des Compilers ist es *nicht* erforderlich, beim Aufruf der generischen Methode **of()** den Typaktualparameter anzugeben:

```java
List<String> los = List.<String>of("a", "b", "c", "d");
```

Wie die Ausgabe des Programms zeigt, sind die Elemente in einem **List<E>** - Container in der Einfügereihenfolge angeordnet. Die auskommentierte Anweisung hätte einen Laufzeitfehler vom Typ **UnsupportedOperationException** zur Folge, weil die **of()** - Produkte unveränderlich sind.

[1] Wenn von ***List<E>*** *- Objekten* die Rede ist, dann sind Objekt aus einer das Interface **List<E>** implementierenden Klasse gemeint. Um mit einem von **of()** gelieferten Objekt erfolgreich arbeiten zu können, müssen wir den Namen seiner Klasse nicht kennen.

Man sollte nach Möglichkeit für Variablen und Parameter, die auf eine Liste zeigen, den Interface-Datentyp **List<E>** verwenden, damit zur Lösung einer konkreten Aufgabe die optimale **List<E>** - Implementierung im OCP-Sinn (Open-Closed - Prinzip, vgl. Abschnitt 4.1.1.3), also praktisch ohne Quellcode-Änderungen, genutzt werden kann.

In Bezug auf die die eingesetzte Technik zum Speichern der Elemente bestehen zwischen den **List<E>** - Implementierungen im Java Collections Framework erhebliche Unterschiede, die gleich behandelt werden.

10.3.2 Beispiel

In einem Beispielprogramm sollen einige **List<E>** - Methoden unter Verwendung der bereits bekannten parametrischen Klasse **ArrayList<String>** erprobt werden:

```
import java.util.*;

class Listen {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    List<String> namen = new ArrayList<>();

    namen.add("Rita"); namen.add("Otto"); namen.add("Leo"); namen.add(0, "Otto");
    System.out.println("Namensliste:");
    for(String s : namen)
      System.out.println(s);
    System.out.println();

    while (true) {
      System.out.println("Zu suchender Name oder leere Eingabe zum Beenden");
      String s = scanner.nextLine();
      if (s.length() == 0) break;
      System.out.println("Position: " + namen.indexOf(s));
    }

    List<String> unikate = new ArrayList<>();
    for(String s : namen)
      if (!unikate.contains(s))
        unikate.add(s);
    System.out.println("Liste der eindeutigen Namen:");
    for(String s : unikate)
      System.out.println(s);
  }
}
```

Im Programm werden zunächst einige (nicht unbedingt eindeutige) Namen per **add()** ohne bzw. mit Angabe der Indexposition in die Liste eingefügt. Dann darf der Benutzer für beliebig viele Namen die Indexposition in der Liste abfragen, wobei die Methode **indexOf()** zum Einsatz kommt. Die zu untersuchenden Zeichenfolgen werden mit der Methode **nextLine()** der Klasse **Scanner** aus dem Paket **java.util** (vgl. Abschnitt 3.4.1) eingelesen. Diese Methode liest eine mit **Enter** quittierte Zeile von der Konsole in ein **String**-Objekt ein. Schließich wird eine zweite Liste mit den *eindeutigen* Namen erstellt, wobei zur Existenzprüfung die **Collection<E>** - Methode **contains()** zum Einsatz kommt. Die **for**-Schleife für Arrays und Kollektionen (vgl. Abschnitt 3.7.3.1) kann verwendet werden, weil eine **List<E>** - Implementation auch die von **Collection<E>** erweiterte Schnittstelle **Iterable<E>** erfüllt und daher ein Iterieren über ihre Elemente ermöglicht (siehe Abschnitt 10.4).

Das Protokoll eines Programmablaufs:

```
Namensliste:
Otto
Rita
Otto
Leo

Zu suchender Name oder leere Eingabe zum Beenden
Leo
Position: 3
Zu suchender Name oder leere Eingabe zum Beenden
Karla
Position: -1
Zu suchender Name oder leere Eingabe zum Beenden

Liste der eindeutigen Namen:
Otto
Rita
Leo
```

10.3.3 Listenarchitekturen

10.3.3.1 Array als Hintergrundspeicher

Die Klasse **ArrayList<E>** arbeitet intern mit einem Array zum Speichern der Elemente und bietet daher einen schnellen wahlfreien Zugriff (engl.: *random access*). Auch das Anhängen neuer Elemente am Ende der Liste verläuft flott, wenn nicht gerade die Kapazität des Arrays erschöpft ist. Dann wird es erforderlich, einen größeren Array zu erzeugen und alle Elemente dorthin zu kopieren. Beim Einfügen bzw. Löschen von *inneren* Elementen müssen die neuen bzw. früheren rechten Nachbarn zeitaufwändig nach rechts bzw. links verschoben werden.

Listenklassen mit schnellem wahlfreiem Zugriff sollten das Marker-Interface **RandomAccess** implementieren. Unter dieser Voraussetzung lässt sich mit Hilfe des **instanceof**-Operators für eine Liste prüfen, ob ein schneller wahlfreier Zugriff möglich ist, z. B.:

```
List<String> ls = new LinkedList<String>();
ls.add("Otto"); ls.add("Luise"); ls.add("Rainer");
if (!(ls instanceof RandomAccess))
    ls = new ArrayList<>(ls);
```

Im Beispiel wird die bei Evans & Flanagan (2015, S. 248) gefundene Idee realisiert, bei Bedarf eine Liste *mit* wahlfreiem Zugang als Kopie zu erstellen. Die Klasse **ArrayList<E>** bietet dazu eine passende Konstruktor-Überladung:

public ArrayList(Collection<? extends E> *c***)**

Während ein **ArrayList<E>** - Objekt den intern zum Speichern der Elemente verwendeten Array bei Bedarf automatisch vergrößert, existiert kein Automatismus zum Verkleinern dieses Arrays nach dem Löschen von Elementen. Dazu ist ein Aufruf der Methode **trimToSize()** erforderlich:

public void trimToSize()

Die ebenfalls Array-basierte Klasse **Vector<E>** war von Beginn an im Java-API enthalten, wurde zwar an das später entwickelte Java Collections Framework angepasst, steht aber trotzdem nicht mehr im besten Ruf. Sie enthält neben den empfohlenen Methoden aus dem Interface **List<E>** auch noch veraltete Methoden, die nicht mehr verwendet werden sollten, weil sie den Wechsel zu einer alternativen Listenverwaltungsklasse verhindern, also die Flexibilität und Wiederverwendung von Software erschweren.

Ein weiterer wesentlicher Unterschied zur Klasse **ArrayList<E>** besteht darin, dass **Vector<E>** Thread-sicher realisiert wurde, sodass ein Container-Objekt ohne Risiko simultan durch mehrere

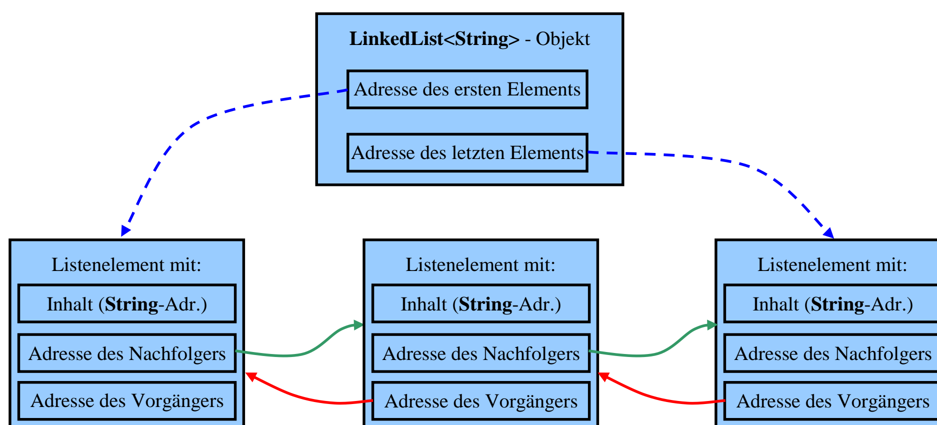
Threads benutzt werden kann.[1] Was das genau bedeutet, werden Sie im Kapitel 15 über Multithreading (nebenläufige Programmierung) erfahren. Allerdings ist die Sicherheit nicht kostenlos zu haben, sodass die Klasse **ArrayList<E>** performanter arbeitet und zu bevorzugen ist, wenn *kein* simultaner Zugriff durch mehrere Threads auftreten kann. Wenn Sie die Klasse **ArrayList<E>** in einer Multithreading - Anwendung einsetzen, müssen Sie allerdings selbst für die Synchronisation der Threads sorgen. Weil eventuell der eine oder andere Leser schon davon profitieren kann, soll hier Optionen für Thread-sicherer Listenverwaltungsklassen erwähnt werden. Die Service-Klasse **Collections** (mit einem *s* am Ende des Namens, siehe Abschnitt 10.9) liefert über die statische Methode **synchronizedList()** zu einer das Interface **List<E>** implementierenden Klasse eine synchronisierte Hüllenklasse, z. B.:

```
List<String> sal = Collections.synchronizedList(new ArrayList<String>());
```

Diese Hüllenklasse kann allerdings bei der Verarbeitungsgeschwindigkeit nicht ganz mit der Klasse **Vector<E>** mithalten.[2] Mit weiteren Thread-sicheren Listenverwaltungsklassen werden wir uns im Abschnitt 15.6 beschäftigen.

10.3.3.2 Verkette Objekte

Die Klasse **LinkedList<E>** arbeitet im Unterschied zur Klasse **ArrayList<E>** intern mit einer **doppelt verketteten Liste** bestehend aus selbständigen Objekten, die jeweils ihren Nachfolger und ihren Vorgänger kennen, z. B.:



Vorteile einer verlinkten Liste im Vergleich zu einem Array:

- Beim Einfügen oder Löschen von inneren Elementen müssen keine anderen Elemente verschoben werden. Es genügt, die Adressketten neu zu verknüpfen.
- Die Länge der Liste ist zu keinem Zeitpunkt festgelegt, sodass keine aufwändigen Maßnahmen zur Kapazitätsanpassung erforderlich werden.

Um ein Listenelement mit bestimmtem Indexwert aufzusuchen, muss die Liste allerdings ausgehend vom ersten oder letzten Element durchlaufen werden. Folglich ist die verkettete Liste beim wahlfreien Zugriff auf vorhandene Elemente einer Liste mit Array-Technik deutlich unterlegen, weil die Elemente eines Arrays im Speicher hintereinander liegen und nach einer einfachen Adressberechnung direkt angesprochen werden können. Außerdem benötigt eine verkettete Liste mehr Speicher. Zum sequentiellen Aufsuchen der Listenelemente muss bei der Klasse **LinkedList<E>** aus Perfor-

[1] Man sollte von einer *bedingten Thread-Sicherheit* sprechen (siehe Naftalin & Wadler 2007, Abschnitt 8.6). Soll z. B. ein Element nur dann in eine Liste eingefügt werden, wenn es noch nicht vorhanden ist, dann genügt es nicht, wenn die beiden Vorgänge (Existenztest, Einfügen) von einer Klasse atomar ausgeführt werden. Es muss trotzdem durch eine explizite Synchronisierung verhindert werden, dass zwischen der Existenzprüfung und dem Einfügen andere Threads auf die Liste zugreifen.

[2] Quelle: http://docs.oracle.com/javase/tutorial/collections/implementations/list.html

manzgründen an Stelle des Indexzugriffs unbedingt ein Iterator-Objekt verwendet werden (siehe Abschnitt 10.4).

Insgesamt sind verlinkte Listen gut geeignet für Algorithmen, die …

- häufig Elemente einfügen oder entfernen und sich dabei nicht auf das Listenende beschränken,
- Elemente überwiegend sequentiell aufsuchen.

Wie die Klasse **ArrayList<E>** bietet auch die Klasse **LinkedList<E>** aus Performanzgründen keine Thread-Sicherheit. Von der **Collections**-Methode **synchronizedList()** erhält man aber zu jedem **List<E>** - Objekt eine Thread-sichere (synchronisierte) Verpackung (siehe Abschnitte 10.3.3.1 und 10.9).

10.3.4 Leistungsunterschiede und Einsatzempfehlungen

Bei der Klasse **LinkedList<E>** machen es die **List<E>** - Methoden mit Index-Parameter (z. B. **get**(), **remove**()) erforderlich, sich vom Startpunkt (Index ≤ halbe Länge) oder Endpunkt (Index > halbe Länge) ausgehend bis zur gesuchten Position vorzuarbeiten, wobei diese aufwändige Prozedur bei jedem Methodenaufruf neu startet. Genügt ein sequentieller Zugriff, sollte bei einer verlinkten Liste unbedingt ein **Iterator**-Objekt verwendet werden, um die Elemente nacheinander zu besuchen (siehe Abschnitt 10.4). Wird ein wahlfreier Zugriff benötigt, ist eine Array-basierte Klasse zu bevorzugen.

Bei den Klassen **ArrayList<E>** und **Vector<E>** entsteht ein großer Aufwand, wenn ein inneres Element eingefügt oder entfernt werden muss.

Man kann je nach Einsatzschwerpunkt und benötigter Thread-Sicherheit zwischen den Listenverwaltungsklassen aus dem Java Collections Framework wählen und unproblematisch wechseln, wenn man ...

- als Datentyp für Variablen und Parameter das Interface **List<E>** verwendet
- und ausschließlich die gemeinsamen, durch das Interface **List<E>** vorgeschriebenen Methoden verwendet.

Im folgenden Programm

```java
import java.util.*;

class Listen {
   static final int ANZ = 40_000;

   static void testList(List<String> plis) {
      StringBuilder sb = new StringBuilder();
      Random ran = new Random();

      // Füllen
      System.out.println("Kollektionsklasse:\t" + plis.getClass());
      long start = System.currentTimeMillis();
      for (int i = 0; i < ANZ; i++) {
         sb.delete(0, 6);
         for (int j = 0; j < 5; j++)
            sb.append((char) (65 + ran.nextInt(26)));
         plis.add(sb.toString());
      }
      System.out.println(" Zeit zum Füllen:\t" +
           (System.currentTimeMillis() - start));
```

```
      // Abrufen per Index-Zugriff
      start = System.currentTimeMillis();
      for (int i = 0; i < ANZ; i++)
         plis.get(ran.nextInt(ANZ));
      System.out.println(" Zeit zum Abrufen:\t" +
            (System.currentTimeMillis() - start));

      // Einfügen am Listenanfang
      start = System.currentTimeMillis();
      for (int i = 0; i < ANZ; i++)
         plis.add(0, "neu");
      System.out.println(" Zeit zum Einfügen:\t" +
            (System.currentTimeMillis() - start));

      // Löschen am Listenanfang
      start = System.currentTimeMillis();
      for (int i = 0; i < 2*ANZ; i++)
         plis.remove(0);
      System.out.println(" Zeit zum Löschen:\t" +
            (System.currentTimeMillis() - start) + "\n");
   }

   public static void main(String[] args) {
      System.out.println("Warmlaufen:\n");
      testList(new ArrayList<String>());
      testList(new Vector<String>());
      testList(new LinkedList<String>());
      System.out.println("Verwertbare Testergebnisse:\n");
      testList(new ArrayList<String>());
      testList(new Vector<String>());
      testList(new LinkedList<String>());
   }
}
```

mit den Aufgaben

- eine Liste mit 40.000 Zeichenketten füllen
- aus der Liste 40.000 Elemente mit zufällig bestimmter Indexposition abrufen
- 40.000 neue Elemente einzeln am Anfang der Liste einfügen
- 80.000 Elemente einzeln am Listenanfang löschen

zeigen die drei Klassen **ArrayList<String>**, **Vector<String>** und **LinkedList<String>** folgende Leistungen:[1]

```
Kollektionsklasse:   class java.util.ArrayList
 Zeit zum Füllen:    7
 Zeit zum Abrufen:   0
 Zeit zum Einfügen:  616
 Zeit zum Löschen:   776

Kollektionsklasse:   class java.util.Vector
 Zeit zum Füllen:    6
 Zeit zum Abrufen:   1
 Zeit zum Einfügen:  559
 Zeit zum Löschen:   741

Kollektionsklasse:   class java.util.LinkedList
 Zeit zum Füllen:    6
 Zeit zum Abrufen:   1594
 Zeit zum Einfügen:  1
 Zeit zum Löschen:   5
```

[1] Die Zeiten stammen von einem PC unter Windows 10 (64 Bit) mit Intel-CPU Core i3 (3,2 GHz) mit vier virtuellen Kernen.

Wir beobachten:

- Das Befüllen verläuft bei allen Klassen recht flott, wobei die Thread-sichere Klasse **Vector<String>** *nicht* mehr Zeit benötigt als die anderen Klassen.
- Beim Abrufen von Werten sind die Array-basierten Klassen erheblich schneller als die verkettete Liste.
- Beim Einfügen und Löschen ist umgekehrt die verkettete Liste überlegen. Allerdings hat sie einen etwas künstlichen Wettbewerbsvorteil erhalten: Weil das Einfügen und Löschen stets am Listenanfang stattfindet, muss das **LinkedList<String>** - Objekt keine Adressen per Listenverfolgung ermitteln und schneidet daher sehr gut ab.

Das Beispielprogramm macht sich zu Nutze, dass eine Schnittstelle als Datentyp zugelassen ist, und dass eine entsprechende Referenzvariable auf ein Objekt aus einer beliebigen implementierenden Klasse zeigen kann (siehe die Definition der Methode `testList()` und deren Aufrufe in der Methode **main**()).[1]

10.4 Iteratoren

Die Schnittstelle **Collection<E>** erweitert die Schnittstelle **Iterable<E>**,[2]

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

sodass eine implementierende Kollektionsklasse eine Methode namens **iterator()** definieren muss, die ein Objekt liefert, das die Schnittstelle **Iterator<E>**[3]

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
}
```

erfüllt und daher u.a. die folgenden Methoden beherrscht, um das Iterieren durch die Elemente der Kollektion zu ermöglichen:

- **public boolean hasNext()**
  Befindet sich hinter der aktuellen Iterator-Position noch ein weiteres Element, wird der Rückgabewert **true** geliefert, sonst **false**.

Position des Iterators (|)	**hasNext**()-Rückgabe
X|YZ	**true**
XYZ|	**false**

[1] Im vorliegenden Fall hätten wir als Datentyp für den `testList()` - Parameter auch die Klasse **AbstractList<E>** verwenden können, weil diese gemeinsame Basisklasse von **ArrayList<E>**, **Vector<E>** und **LinkedList<E>** ebenfalls das Interface **List<E>** implementiert und somit die benötigten Methoden beherrscht (vgl. Abschnitt 7.6).

[2] Der Einfachheit halber wurden die beiden in Java 8 ergänzten **default**-Methoden **forEach**() und **spliterator()** weggelassen.

[3] Der Einfachheit halber wurde die in Java 8 ergänzte **default**-Methode **forEachRemaining**() weggelassen.

- **public E next()**
  Diese Methode liefert das nächste Element hinter dem Iterator und verschiebt den Iterator um eine Position nach rechts:

Position des Iterators (|) vor next()	Position des Iterators (|) nach next()
X|YZ	XY|Z

  Gibt es kein nächstes Element, wirft die Methode eine Ausnahme vom Typ **NoSuchElementException**.

- **public void remove()**
  Ein **remove()** - Aufruf entfernt das zuletzt per **next()** abgerufene Listenelement. Einem **remove()** - Aufruf muss also ein erfolgreicher **next()** - Aufruf vorangehen, der noch nicht durch einen anderen **remove()** - Aufruf verwertet worden ist. Wird nach der Erstellung des Iterators die zugehörige Liste auf andere Weise als durch Aufruf der Iterator-Methode **remove()** geändert, dann darf der Iterator anschließend nicht mehr verwendet werden, weil sein Verhalten unspezifiziert ist. Bei Bedarf muss durch einen erneuten Aufruf der **Iterable<E>** - Methode **iterator()** ein frischer Iterator erstellt werden. Die Methode **remove()** ist in der API-Dokumentation durch den Zusatz *optional operation* markiert (vgl. Abschnitt 9.2). Es ist einer Klasse erlaubt, sich bei der Implementation dieser Methode auf das Werfen einer **UnsupportedOperationException** zu beschränken. Es wird allerdings von jeder implementierenden Klasse erwartet, es in der Dokumentation offenzulegen, wenn nur eine Pseudo-Implementation oder die **default**-Methode der Schnittstelle **Iterator<E>**

  ```
  default void remove() {
      throw new UnsupportedOperationException("remove");
  }
  ```

  vorhanden ist.

Das folgende Programm demonstriert die Verwendung des Iterators zu einem **LinkedList<String>** - Objekt:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
import java.util.*;
class Prog {
    public static void main(String[] args) {
        List<String> ls = new LinkedList<String>();
        ls.add("Otto"); ls.add("Luise"); ls.add("Rainer");
        Iterator<String> ist = ls.iterator();
        while (ist.hasNext())
            System.out.println(ist.next());
        ist.remove();// Letzte next() - Rückgabe entfernen
        System.out.println("\nRest der Liste:");
        for (String s : ls)
            System.out.println(s);
    }
}
```

</td><td>

```
Otto
Luise
Rainer

Rest der Liste:
Otto
Luise
```

</td></tr>
</table>

Die Iteratoren vieler Klassen (z. B. **ArrayList<E>**, **LinkedList<E>**, **Vector<E>**, **HashSet<E>**, **TreeSet<E>**, **HashMap<K,V>**, **TreeMap<K,V>**) sind **fail-fast**: Ist die Kollektion nach der Erstellung des Iterators auf andere Weise als durch die **remove()** - Methode des Iterators strukturell geändert worden, dann werfen die kritischen Methoden des Iterators eine **ConcurrentModificationException**, um erratisches Verhalten zu verhindern. Allerdings gelingt die Detektion nicht zuverlässig, sodass die Korrektheit eines Programms nicht von den **fail-fast** - Bemühungen des Iterators abhängen darf. Multithreading-Programme müssen bei der Verwendung von Iteratoren eine explizite (manuelle) Sperre setzen. Allerdings existieren zu den eben genannten JCF-Klassen für den Einsatz in Multithreading-Programmen überlegene Alternativen aus dem Paket **java.util.concurrent** (siehe Abschnitt 15.6).

Iteratoren haben ihren Einsatzschwerpunkt bei verketteten Listen (siehe Abschnitt 10.3.3), wo sie im Vergleich zum zeitaufwändigen Indexzugriff für einen Performanzschub sorgen; sie sind aber auch bei den Klassen zur Verwaltung von Mengen und Abbildungen verwendbar (vgl. Abschnitt 10.5 und 10.6).

Iteratoren werden meist implizit und bequem über die **for**-Schleife für Arrays und Kollektionen genutzt (auch als *erweiterte for-Schleife* oder *for-each - Schleife* bezeichnet, vgl. Abschnitt 3.7.3.2):

for (*Elementtyp Iterationsvariable* **:** *Serie***)**   *Anweisung*

Die vom Interface **List<E>** geforderte Methode **listIterator()** liefert ein Objekt, welches das vom Interface **Iterator<E>** abstammende Interface **ListIterator<E>** implementiert. Es enthält etliche zusätzliche Methoden, u.a. zur Unterstützung von bidirektionalen Listenpassagen und zur Aufnahme von neuen Elementen:

- **public boolean hasPrevious()**
  Befindet sich vor der aktuellen Iterator-Position noch ein Listenelement, wird der Rückgabewert **true** geliefert, sonst **false**.
- **public E previous()**
  Diese Methode liefert das nächste Listenelement vor dem Iterator und verschiebt den Iterator um eine Position nach links. Gibt es kein vorheriges Element, wirft die Methode eine Ausnahme vom Typ **NoSuchElementException**.
- **public int previousIndex()**
  Von dieser Methode erfährt man den Index des Elements, das ein nachfolgender **previous()** - Aufruf liefern würde, oder den Wert -1, wenn sich der Iterator am Listenanfang befindet.
- **public void add(E** *e***)** (optionale Operation)
  Das Parameterobjekt wird in die Liste aufgenommen und vor dasjenige Element gesetzt, das von **next()** geliefert würde.
- **public void set(E** *element***)** (optionale Operation)
  Das zuletzt von **next()** oder **previous()** gelieferte Element wird durch das Parameterobjekt ersetzt.

Bei den **List<E>** - Implementationen **ArrayList<E>**, **LinkedList<E>** und **Vector<E>** sind auch die **listIterator()** - Rückgaben fail-fast, wobei strukturelle Listenänderungen durch die Methoden **add()** und **remove()** zulässig und damit *kein* Anlass für eine Ausnahme sind.

10.5 Mengen

Zur Verwaltung einer *Menge* von Elementen, die im Unterschied zu einer Liste *keine Dubletten* (im Sinne der **equals()** - Methode der Elementklasse) aufweisen darf, enthält das Java Collections Framework im Paket **java.util** u.a. die generischen Klassen **HashSet<E>**, **LinkedHashSet<E>** und **TreeSet<E>**. Sie implementieren wie ihre gemeinsame (direkte oder indirekte) abstrakte Basisklasse **AbstractSet<E>** das Interface **Set<E>**. Diese Klassen sind nützlich, wenn Mengen im Sinne der Mathematik zu modellieren sind und entsprechende Operationen benötigt werden (z. B. Durchschnitt, Vereinigung oder Differenz von zwei Mengen). Im Vergleich zu anderen Kollektionsklassen können sie Mengenzugehörigkeitsprüfungen sehr schnell ausführen, was sie unverzichtbar macht, wenn derartige Prüfungen in großer Zahl auftreten.

Einige Mengenverwaltungsklassen unterstützen eine Anordnung der Elemente, die entweder auf der Einfügereihenfolge (bei **LinkedHashSet<E>**) oder auf einer Ordnung des Elementtyps (bei **TreeSet<E>**) basiert. Von den Listen (siehe Abschnitt 10.3) unterscheiden sich auch die Klassen mit geordneten Elementen durch die Forderung, dass zur Existenzprüfung ein Algorithmus verfüg-

bar sein muss, dessen Aufwand im ungünstigsten Fall über eine logarithmische Funktion von der Anzahl k der Elemente abhängt, also z. B. von der Ordnung $O(\log_2 k)$ ist (siehe Abschnitt 10.5.4).

10.5.1 Das Interface Set<E>

Das von **Collection<E>** abstammende Interface **Set<E>** verlangt keine zusätzlichen Instanzmethoden, modifiziert aber bei einigen Methoden die Verhaltensanforderungen:

- **public boolean add(E** *element***)** (optionale Operation)
  Das Parameterelement wird in die Menge aufgenommen, falls dort noch kein im Sinne der **equals()** - Methode der Elementklasse inhaltsgleiches Element existiert. Wenn die Menge aufgrund der beantragten Neuaufnahme verändert wurde, liefert die Methode den Rückgabewert **true**, anderenfalls **false**.
- **public boolean addAll(Collection<? extends E>** *collection***)** (optionale Operation)
  Die Elemente der übergebenen Kollektion werden in die Menge aufgenommen, falls sie dort noch nicht (im Sinne der **equals()** - Methode der Elementklasse) vorhanden sind. Ihr Typ muss mit dem Elementtyp der angesprochenen Mengenklasse übereinstimmen oder diesen spezialisieren. Nach einem erfolgreichen Methodenaufruf enthält das angesprochene Objekt die **Vereinigung** der beiden Mengen.
- **public boolean contains(Object** *object***)**
  Diese Methode informiert darüber, ob das fragliche Element bzw. ein im Sinne der **equals()** - Methode der Elementklasse inhaltsgleiches Element in der Kollektion vorhanden ist und arbeitet bei den Mengenverwaltungsklassen erheblich flotter als bei den Listenverwaltungsklassen (siehe Abschnitt 10.3).
- **public boolean remove(Object** *element***)** (optionale Operation)
  Das angegebene Element (bzw. ein im Sinne der **equals()** - Methode der Elementklasse inhaltsgleiches Element) wird aus der Menge entfernt, falls es dort vorhanden ist.
- **public boolean removeAll(Collection<?>** *collection***)** (optionale Operation)
  Die Elemente der Parameterkollektion (bzw. im Sinne der **equals()** - Methode der Elementklasse inhaltsgleiche Elemente) werden ggf. aus der angesprochenen Kollektion entfernt, sodass man nach einem erfolgreichen Methodenaufruf die **Differenz** der beiden Mengen bzw. Kollektionen erhält.
- **public boolean retainAll(Collection<?>** *collection***)** (optionale Operation)
  Aus der angesprochenen Kollektion werden alle Elemente entfernt, die nicht zur Parameterkollektion gehören bzw. im Sinne der **equals()** - Methode der Elementklasse mit einem Element der Parameterkollektion inhaltsgleich sind, sodass man nach einem erfolgreichen Methodenaufruf den **Durchschnitt** der beiden Mengen bzw. Kollektionen erhält.

Die Methoden **add()**, **addAll()**, **remove()**, **removeAll()** und **retainAll()** informieren mit ihrem **boolean**-Rückgabewert darüber, ob die angesprochene Menge durch den Aufruf verändert worden ist.

Wo das Verhalten einer Methode von Übereinstimmungsprüfungen abhängt (z. B. **contains()**, **remove()**), ist bei der Interface-Implementierung die **equals()** - Methode des Elementtyps zu verwenden (statt des Identitätsoperators). Dementsprechend wird in der Elementklasse für die von **Object** geerbte **equals()** - Methode eine sinnvolle Überschreibung erwartet (siehe Abschnitt 10.2.1).

Warum die Methoden **contains()** und **remove()** den Formalparametertyp **Object** (statt **E**) verwenden, wurde im Abschnitt 10.2.1 erläutert.

Seit Java 9 besitzt die Schnittstelle **Set<E>** eine statische Fabrikmethode namens **of()**, mit der sich auf einfache Weise *unveränderliche* **Set<E>** - Objekte erstellen lassen, z. B.:[1]

Quellcode

```
import java.util.*;
class Prog {
    public static void main(String[] args) {
        Set<String> sos = Set.of("a", "b", "c", "d");
        System.out.println(sos);
        // sos.add("f");
    }
}
```

Ausgabe

```
[d, a, c, b]
```

Den Aufruf der statischen, generischen Methode richtet man an den Rohtyp der generischen Schnittstelle **Set<E>**. Wegen der Typinferenz-Fähigkeiten des Compilers ist es *nicht* erforderlich, beim Aufruf der generischen Methode **of()** den Typaktualparameter anzugeben:

```
Set<String> sos = Set.<String>of("a", "b", "c", "d");
```

Wie die Ausgabe des Programms zeigt, besitzen die Elemente in einem **Set<E>** - Container *keine* definierte Anordnung. Die auskommentierte Anweisung hätte einen Laufzeitfehler vom Typ **UnsupportedOperationException** zur Folge, weil die **of()** - Produkte unveränderlich sind.

Man sollte nach Möglichkeit für Variablen und Parameter, die auf eine Menge zeigen, den Interface-Datentyp **Set<E>** (oder eine geordnete Variante, siehe Abschnitt 10.5.5) verwenden, damit zur Lösung einer konkreten Aufgabe die optimale **Set<E>** - Implementierung im OCP-Sinn (Open-Closed - Prinzip, vgl. Abschnitt 4.1.1.3), also praktisch ohne Quellcode-Änderungen, genutzt werden kann.

Einen Indexzugriff auf ihre Elemente bieten die Kollektionsklassen zur Mengenverwaltung *nicht*, ein Iterator (vgl. Abschnitt 10.4) ist jedoch verfügbar.

Eine **Set<E>** - Implementierung kann Dubletten *nicht* verhindern, wenn Objekte *nach* Aufnahme in die Menge geändert werden. Bei manchen API-Klassen ist eine Änderung von Objekten grundsätzlich ausgeschlossen (z. B. **String**, alle Wrapper-Klassen für primitive Datentypen), was sich im augenblicklichen Kontext (und nicht nur dort) als vorteilhaft erweist. In der Regel sind Objekte aber veränderbar, z. B. bei der Klasse `Mint`, die Sie in einer Übungsaufgabe als **int**-Hüllenklasse entworfen haben (vgl. Abschnitt 5.5). Im folgenden Programm entsteht ein **HashSet<Mint>** - Container mit Dublette:

[1] Wenn von ***Set<E>** - Objekten* die Rede ist, sind Objekt aus einer das Interface **Set<E>** implementierenden Klasse gemeint. Um mit einem von **of()** gelieferten Objekt erfolgreich arbeiten zu können, müssen wir den Namen seiner Klasse nicht kennen.

Quellcode	Ausgabe

`import java.util.*;`
 	
`class Dubletten {`
`   public static void main(String[] args) {`
`      Set<Mint> mint = new HashSet<>();`
`      Mint m1 = new Mint(1);`
`      Mint m2 = new Mint(2);`
 	
`      mint.add(m1); mint.add(m1);`
`      mint.add(m2);`
`      System.out.println(mint);`
 	
`      m2.val = 1;`
`      System.out.println(mint);`
`   }`
`}`
	`[1, 2]`
	`[1, 1]`

Aus Performanzgründen sind die Klassen **HashSet<E>**, **LinkedHashSet<E>** und **TreeSet<E>** *nicht* Thread-sicher implementiert. Allerdings liefert die Klasse **Collections** über die statische Methode **synchronizedSet()** zu einer das Interface **Set<E>** implementierenden Klasse eine synchronisierte Hüllenklasse, z. B.:

```
Set<String> shs = Collections.synchronizedSet(new HashSet<String>());
```

Mit weiteren Thread-sicheren Mengenverwaltungsklassen werden wir uns im Abschnitt 15.6 beschäftigen.

10.5.2 Leistungsvorteil bei der Existenzprüfung

Um den Leistungsunterschied zwischen **List<E>** - und **Set<E>** - Implementierungen bei der Existenzprüfung zu untersuchen, wurden in einem Testprogramm folgende Aufgaben gestellt:

- Eine Kollektion mit 20.000 **String**-Objekten füllen
- Für 20.000 neue **String**-Objekte prüfen, ob sie bereits in der Kollektion vorhanden sind

Dabei zeigten die Klassen **ArrayList<String>**, **HashSet<String>** und **TreeSet<String>**folgende Leistungen:[1]

```
Kollektionsklasse:                   class java.util.ArrayList
 Zeit zum Füllen:                    10
 Zeit für die Existenzprüfungen:     2006

Kollektionsklasse:                   class java.util.HashSet
 Zeit zum Füllen:                    13
 Zeit für die Existenzprüfungen:     5

Kollektionsklasse:                   class java.util.TreeSet
 Zeit zum Füllen:                    27
 Zeit für die Existenzprüfungen:     19
```

Bei der Existenzprüfung sind die beiden **Set<E>** - Implementierungen deutlich überlegen.

10.5.3 Hashtabellen

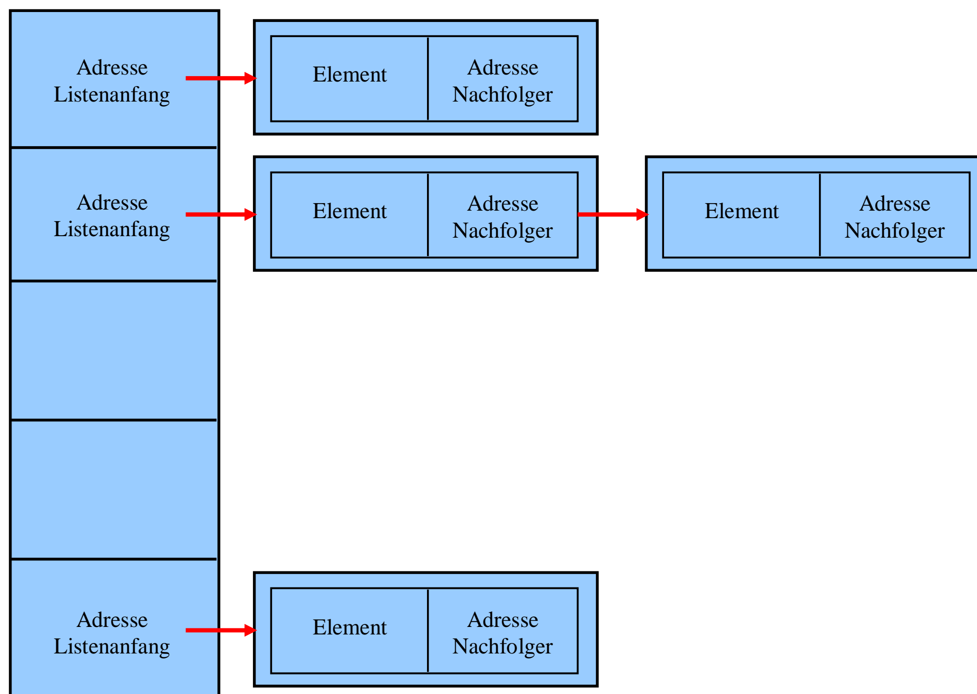
Benötigt ein Algorithmus zahlreiche Mengenzugehörigkeitsprüfungen, sind Kollektionen mit Listenbauform wenig geeignet, weil ein fragliches Element potentiell mit jedem Listenelement über

[1] Die Zeiten stammen von einem PC unter Windows 10 (64 Bit) mit Intel-CPU Core i3 (3,2 GHz).

einen Aufruf der **equals()** - Methode verglichen werden muss. Um diese Aufgabe schneller lösen zu können, kommt bei der Klasse **HashSet<E>** eine sogenannte *Hashtabelle* zum Einsatz. Die zentrale Designidee besteht darin, zur Datenablage einen Array mit sogenannten *Buckets* zu verwenden, wobei es sich um einfach verkettete Listen handelt. Eine effiziente Existenzprüfung ist möglich, weil ...

- für ein aufzunehmendes oder auf Existenz zu prüfendes Element der Array-Index des zugehörigen Buckets direkt zu berechnen ist,
- die Buckets nach Möglichkeit einelementig sind, sodass kaum Listenoperationen anfallen.

Bei leicht vereinfachter Darstellung sieht ein Bucket-Array so aus:[1]



Bei der Aufnahme eines neuen Elements und auch bei der Existenzprüfung wird der Bucket-Index über die typspezifische Implementierung der bereits in der Urahnklasse **Object** definierten Instanzmethode **hashCode()** ermittelt:

public int hashCode()

Beim Einfügen eines neuen Elements ist die Liste zum berechneten Index idealerweise noch leer. Anderenfalls spricht man von einer *Hash-Kollision*, und es entsteht ein kleiner Zusatzaufwand. Wegen der folgenden Anforderungen an eine zum Befüllen einer Hashtabelle einzusetzende **hashCode()** - Methode (bzw. an die in dieser Methode realisierte Hash-Funktion) ist in der Regel in der **E**-Konkretisierungsklasse das **Object**-Erbstück durch eine sinnvolle Implementierung zu überschreiben:

- Während eines Programmlaufs müssen alle Methodenaufrufe für ein Objekt denselben Wert liefern, solange bei diesem Objekt keine Veränderungen mit Relevanz für die **equals()** - Methode auftreten.
- Sind zwei Objekte identisch im Sinne der **equals()** - Methode, dann müssen sie denselben **hashCode()** - Wert erhalten. Daher muss für jede Klasse, welche die **equals()** - Methode überschreibt, auch die **hashCode()** - Methode überschrieben werden (Bloch 2018, S. 50).

---

[1] Es wird mal wieder aus didaktischen Gründen ein wenig gemogelt. Ein Blick in den API-Quellcode zeigt, dass die Klasse **HashSet<E>** intern ein **HashMap<E,V>** - Objekt zur Datenablage verwendet, sodass die realen Buckets etwas anders aussehen.

- Sind zwei Objekte verschieden im Sinne der **equals()** - Methode, dann müssen sie *nicht* unbedingt verschiede **hashCode()** - Werte erhalten. Allerdings leidet die Performanz von Hashtabellen, wenn es oft zu sogenannten Hash-Kollisionen kommt (gleiche **Hash**-Werte für **equals()** - verschiedene Objekte.
- Die **hashCode()** - Werte sollten dazu taugen, Array-Elemente zu indizieren.
- Die **hashCode()** - Werte sollten möglichst gleichmäßig über den möglichen Indexwertebereich verteilt sein.

Aus dem Hashcode eines Objekts und der Hashtabellen-Kapazität wird der Array-Index per Modulo-Operation ermittelt:

$$\text{Array-Index} = \text{Hashcode} \ \% \ \text{Kapazität}$$

Bei der API-Klasse **String** kommt z. B. die folgende Hash-Funktion zum Einsatz:

$$\boldsymbol{u(0)\ 31^{n-1} + u(1)\ 31^{n-2} + ... + u(n\text{-}1)}$$

Dabei steht $\boldsymbol{u(i)}$ für die Unicode-Nummer des Zeichens an der (nullbasierten) Position $\boldsymbol{i}$ und $\boldsymbol{n}$ für die Länge der Zeichenfolge. Für die Zeichenfolge "`Theo`" erhält man z. B.:

$$84 \cdot 31^3 + 104 \cdot 31^2 + 101 \cdot 31^1 + 111 = 2605630$$

Bei einer Hashtabellen-Kapazität von 1024 resultiert der Array-Index

$$2605630 \ \% \ 1024 = 574$$

Um für ein Objekt mit der Methode **contains()** festzustellen, ob es bereits in der Hashtabelle (Menge) enthalten ist, muss es nicht über **equals()** - Aufrufe mit allen Insassen verglichen werden. Stattdessen wird sein Hashcode berechnet und sein Array-Index ermittelt. Befindet sich hier noch keine Listenadresse, ist die Existenzfrage geklärt (**contains()** - Rückmeldung **false**). Anderenfalls ist nur für die Objekte der im Array-Element adressierten Liste eine **equals()** - Untersuchung erforderlich.

Damit es selten zu Hash-Kollisionen kommt, sollte die Array-Größe ungefähr das 1,5 - fache der Anzahl aufzunehmender Elemente betragen (Horstmann & Cornell, 2002, S. 137). Über den **Ladungsfaktor** der Hashtabelle legt man fest, bei welchem Füllungsgrad in einen neuen, ca. doppelt so großen Array umgezogen werden soll (Voreinstellung: 0,75).

Weil die Klasse **HashSet<E>** das Interface **Collection<E>** (siehe Abschnitt 10.2.1) implementiert, kann sie (als Rückgabe der Methode **iterator()**) einen Iterator (siehe Abschnitt 10.4) zur Verfügung stellen, der sukzessive alle Elemente aufsucht und dabei erwartungsgemäß eine zufällig wirkende Reihenfolge verwendet.

Mit der Klasse **LinkedHashSet<E>** steht eine **HashSet<E>** - Ableitung zur Verfügung, deren Objekte sich die Einfügereihenfolge der Elemente merken. Dies wird durch den Zusatzaufwand einer doppelt verlinkten Liste realisiert. Die Elemente merken sich ...

- das nächste Element im selben Bucket
- und das als nächstes eingefügte Element.

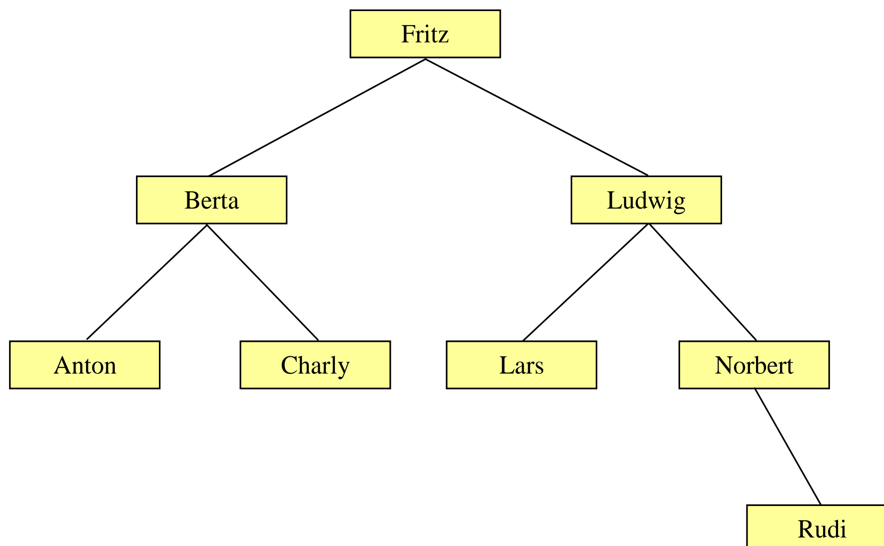
Im Ergebnis erhalten wir einen Iterator, der die Einfügereihenfolge verwendet und außerdem flotter arbeitet als die **HashSet<E>** - Variante, weil die leeren Buckets nicht aufgesucht werden müssen (Naftalin & Wadler 2007, S. 181).

10.5.4 Balancierte Binärbäume

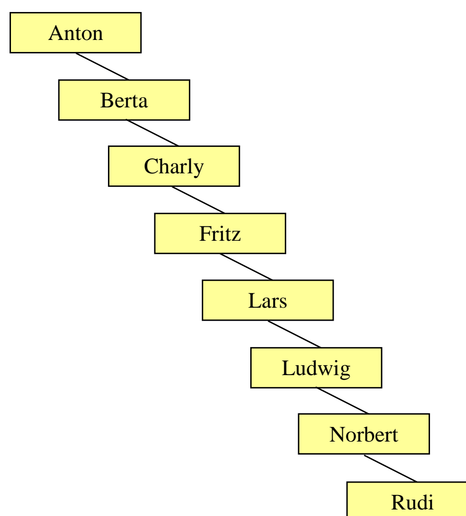
Existiert über den Elementen einer Menge eine **vollständige Ordnung** (z. B. Zeichenketten mit der lexikografischen Ordnung), dann kann man über einen sogenannten *Binärbaum* die Elemente im sortierten Zustand halten, ohne den Aufwand bei den zentralen Mengenverwaltungsmethoden (z. B.

add(), **contains()** und **remove()**) zu groß werden zu lassen (siehe Leistungsvergleich im Abschnitt 10.5.2).

In einem Binärbaum hat jeder Knoten maximal zwei direkte Nachfolger, wobei der linke Nachfolger einen kleineren und der rechte Nachfolger einen höheren Rang hat, was die folgende Abbildung für Zeichenketten illustriert:



Bei einem **balancierten** Binärbaum kommen Forderungen zur maximalen Entfernung zwischen der Wurzel und einem Element (zur Anzahl der Ebenen) hinzu, um den Aufwand beim Suchen und Einfügen von Elementen zu begrenzen. Der bisher betrachtete Namensbaum ist gut balanciert (mit 4 Ebenen), während in der folgenden Abbildung eine extrem unbalancierte Anordnung derselben Elemente zu sehen ist (mit 8 Ebenen), die offenbar aus einer ungünstigen Wahl des Wurzelelements resultiert:



Zur Beurteilung des Aufwands bei der Suche nach einem Element (oder bei der Neuaufnahme eines Elements) gehen wir von einem balancierten und vollständig gefüllten Binärbaum aus. Hier haben alle Knoten, die keine Endknoten sind, genau zwei Nachfolger. In der ersten Variante des Namensbaums lag diese Situation vor der Aufnahme von Rudi vor. Der maximale Aufwand bei einer Existenzprüfung oder Neuaufnahme ist identisch mit der Zahl m der Ebenen, weil pro Ebene eine Identitätsprüfung vorgenommen werden muss. Wir schätzen nun ab, wie viele Ebenen ein balancierter Binärbaum zur Aufnahme von k Elementen benötigt.

Aus der Anzahl m der Ebenen kann nach der folgenden Formel die Anzahl der k der enthaltenen Elemente berechnet werden:

$$k = 2^m - 1$$

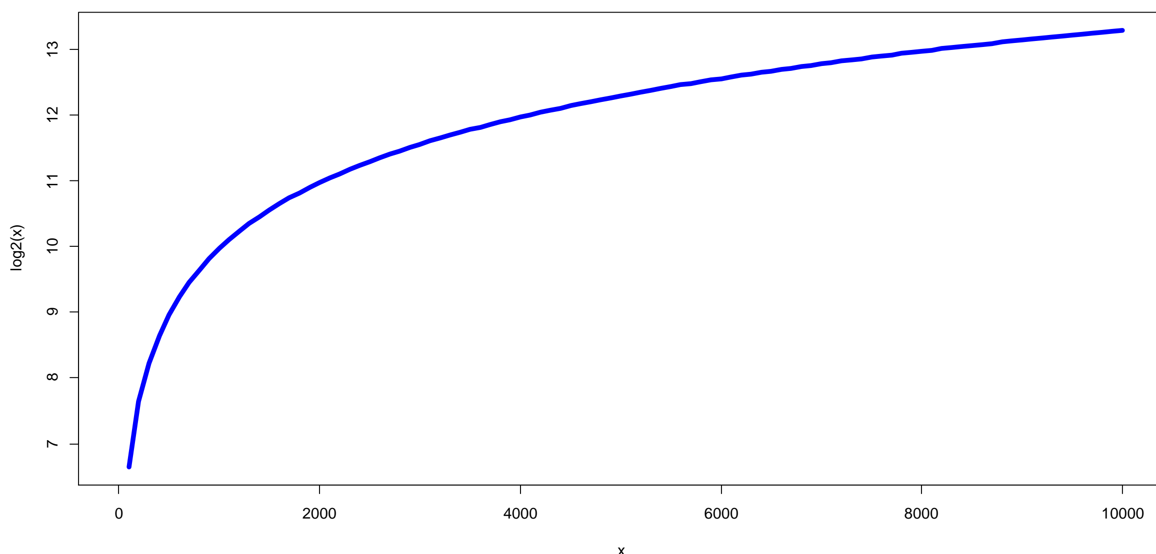
Bei $m = 3$ Ebenen resultieren z. B. 7 Elemente (siehe Beispiel). Man erhält k als Partialsumme der geometrischen Reihe:[1]

$$k = \sum_{i=0}^{m-1} 2^i = \frac{1-2^m}{1-2} = 2^m - 1$$

Für hinreichend großes k kann man die Beziehung zwischen k und m vereinfachen und dann durch Anwendung der Logarithmus-Funktion nach m auflösen, um die zur Verwaltung von k Elementen erforderliche Anzahl von Ebenen zu ermitteln:

$$\begin{aligned} k &= 2^m \\ \Leftrightarrow \quad \log_2(k) &= m \end{aligned}$$

Für hinreichend großes k sind also $\log_2(k)$ Algorithmusschritte erforderlich, um ein Element zu suchen oder die Position für ein neues Element zu bestimmen. Man sagt unter Verwendung einer Notation mit dem griechischen Großbuchstaben O (Omikron), der Algorithmus sei von der Ordnung $O(\log_2 k)$. Weil das monotone Wachstum der Logarithmus-Funktion relativ flach verläuft, steigt der Aufwand nur langsam mit der Anzahl k der Elemente an:[2]



Bei einer Hashtabelle wächst der Aufwand einer Existenzprüfung *nicht* mit der Anzahl der Elemente, und es resultiert die günstigere Ordnung $O(1)$. Bei einer Liste hingegen ist der Aufwand einer Existenzprüfung direkt proportional zur Anzahl der Elemente, und es resultiert die ungünstige Ordnung $O(k)$. Insgesamt verursacht bei einem Binärbaum die Anordnung der Elemente keine allzu großen Kosten für die Mengenoperationen.

---

[1] Der mit elementaren Mitteln zu führende Beweis ist z. B. hier zu finden:
https://de.wikibooks.org/wiki/Mathe_f%C3%BCr_Nicht-Freaks:_Geometrische_Reihe

[2] Der Funktionsplot wurde mit R 3.6 erstellt über den Funktionsaufruf:

```
curve(log2(x), 0, 10000, col="blue", lwd=5)
```

Im Java Collections Framework nutzt u.a. die Klasse **TreeSet<E>** das Prinzip des balancierten Binärbaums, wobei durch die sogenannte **Rot-Schwarz** -Architektur sichergestellt wird, dass der Baum immer balanciert ist.

Der Iterator zu einem **TreeSet<E>** - Objekt, den wir meist implizit im Rahmen der erweiterten **for**-Schleife (vgl. Abschnitt 3.7.3.2) benutzen, durchläuft die Elemente in aufsteigender Ordnung, z. B.:

Quellcode

```
import java.util.*;
class Prog {
  public static void main(String[] args) {
    Set<String> tsi = new TreeSet<>();
    tsi.add("Fritz");
    tsi.add("Lars"); tsi.add("Berta");
    tsi.add("Charly");tsi.add("Ludwig");tsi.add("Anton");tsi.add("Norbert");
    tsi.add("Rudi");
    for(String s : tsi)
      System.out.println(s);
  }
}
```

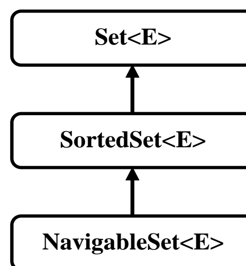
Ausgabe

```
Anton
Berta
Charly
Fritz
Lars
Ludwig
Norbert
Rudi
```

Am Rande sei noch erwähnt, dass seit Java 7 in eine Kollektion vom Typ **TreeSet<E>** kein Element mit dem Referenzziel **null** eingefügt werden kann. Ein Versuch endet mit einer **NullPointerException**. Bei der Klasse **HashSet<E>** und bei den meisten **List<E>** - Implementationen lässt sich eine **null**-Referenz jedoch ungestraft einfügen.

10.5.5 Interfaces für geordnete Mengen

Die Klasse **TreeSet<E>** implementiert über das Interface **Set<E>** hinaus auch das Interface **SortedSet<E>** mit Methoden für geordnete Mengen und das mit Java 6 als **SortedSet<E>** - Erweiterung und designierter Nachfolger hinzugekommene Interface **NavigableSet<E>**. Hier sind die drei Interfaces und Ihre Beziehungen zu sehen:



Es gibt zwei Möglichkeiten, die Ordnung der von einer **SortedSet<E>** - Implementierung verwalteten Elemente zu begründen:

- Der Elementtyp **E** erfüllt das Interface **Comparable<E>**, besitzt also eine Instanzmethode **compareTo()** und somit eine natürliche Ordnung.
- Man übergibt dem Konstruktor ein Objekt, das die Schnittstelle **Comparator<? super E>** erfüllt und folglich für den Typ **E** oder für eine Basisklasse **B** eine Vergleichsmethode mit dem folgenden Definitionskopf bietet:

  public int compare(E *e1*, **E** *e2*) bzw. **public int compare(B** *e1*, **B** *e2*)

Die Methoden **compareTo()** bzw. **compare()** müssen konsistent mit der **equals()** - Methode des Elementtyps sein, denn:

- Das Interface **Set<E>** ist basierend auf der **equals()** - Methode definiert.
- Eine **SortedSet<E>** - Implementation benutzt für alle Elementvergleiche (und damit auch für die Dubletten-Erkennung) die Methoden **compareTo()** bzw. **compare()**.

Konkret heißt das z. B. für die **Comparator<E>** - Methode **compare()**:[1]

> The ordering imposed by a comparator `c` on a set of elements `S` is said to be *consistent with equals* if and only if `c.compare(e1, e2)==0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` in `S`.

Das Interface **SortedSet<E>** fordert von implementierenden Klassen u.a. die folgenden Methoden:

- **public Comparator<? super E> comparator()**
  Es wird das bei der Konstruktion übergebene **Comparator**-Objekt geliefert oder **null**, wenn die natürliche Ordnung der Elementklasse **E** benutzt wird. In diesem Fall muss **E** das Interface **Comaparable<E>** implementieren.

- **public E first()**
  Die Methode liefert das erste (kleinste) Element in der sortierten Menge.

- **public E last()**
  Die Methode liefert das letzte (größte) Element in der sortierten Menge.

- **public SortedSet<E> headSet(E** *obereSchranke***)**
  Man erhält als sogenannte *Sicht* (engl.: *View*) auf die Teilmenge mit allen Elementen der angesprochenen Kollektion, die *kleiner* als die obere Schranke sind, ein Objekt aus einer Klasse, welche das Interface **SortedSet<E>** erfüllt. Alle Methoden des View-Objekts wirken sich auf die Originalkollektion aus, sodass man z. B. mit der Methode **clear()** die komplette **headSet()** - Teilmenge löschen kann:

  Quellcode

  ```
  import java.util.*;
  class Prog {
    public static void main(String[] args) {
      SortedSet<String> ts=new TreeSet<>(Set.of("a","b","c","d"));
      System.out.println(ts);
      SortedSet<String> sos = ts.headSet("c");
      System.out.println(sos);
      sos.clear();
      System.out.println(ts);
    }
  }
  ```

  Ausgabe

  ```
  [a, b, c, d]

  [a, b]

  [c, d]
  ```

  Dem **TreeSet<String>** - Konstruktor wird ein **Set<String>** - Objekt übergeben, das mit Hilfe der seit Java 9 verfügbaren statischen Methode **of()** aus dem Interface **Set<E>** erstellt wird. Aus der von **Set.of()** gelieferten unveränderlichen Menge (vgl. Abschnitt 10.5.1) erstellt der **TreeSet<E>** - Konstruktor eine *veränderliche* Kollektion.

- **public SortedSet<E> tailSet(E** *untereSchranke***)**
  Man erhält ein View-Objekt mit allen Elementen der angesprochenen Kollektion auswirken, die mindestens so groß sind wie die untere Schranke. Alle Methoden des View-Objekts wirken sich auf die Originalkollektion aus, z. B.:

[1] https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Quellcode	Ausgabe

<pre>import java.util.*;
class Prog {
  public static void main(String[] args) {
    SortedSet<String> ts=new TreeSet<>(Set.of("a","b","c","d"));
    SortedSet<String> sos = ts.tailSet("c");
    System.out.println(sos);
    sos.clear();
    System.out.println(ts);
  }
}</pre>
	<pre>[c, d]
	[a, b]</pre>

- **public SortedSet<E> subSet(E** *untereSchranke,* **E** *obereSchranke***)**
  Man erhält eine Sicht auf einen Bereich der angesprochenen Kollektion beginnend mit der unteren Schranke (inklusive) und endend mit der oberen Schranke (exklusive). Alle Methoden des View-Objekts wirken sich auf die Originalkollektion aus.

Im Definitionskopf einer das Interface **SortedSet<E>** implementierenden Klasse (z. B. **TreeSet<E>**) kann und sollte darauf verzichtet werden, den Typformalparamater auf die Schnittstelle **Comparable<E>** zu restringieren, weil die von **SortedSet<E>** geforderte Methode **comparator()** (siehe oben) die Vergleichbarkeit der Elemente sicherstellt.

Im folgenden Beispielprogramm verwendet das **TreeSet<String>** - Objekt `tss` die natürliche Ordnung der Klasse **String**, während im **TreeSet<String>** - Objekt `tssc` ein Objekt der Klasse `CompaS`, welche die Schnittstelle **Comparator<String>** erfüllt, dafür sorgt, dass Otto immer vorne steht:

Quellcode	Ausgabe

<pre>import java.util.*;
 	
class CompaS implements Comparator<String> {
  public int compare(String s1, String s2) {
    if (s1.equals("Otto"))
      return -1;
    if (s2.equals("Otto"))
      return 1;
    return s1.compareTo(s2);
  }
}
 	
class ComparatorTest {
  public static void main(String[] args) {
    SortedSet<String> tss = new TreeSet<>();
    tss.add("Werner"); tss.add("Ludwig"); tss.add("Otto");
    System.out.println(tss);
 	
    TreeSet<String> tssc = new TreeSet<>(new CompaS());
    tssc.add("Werner"); tssc.add("Ludwig"); tssc.add("Otto");
    System.out.println(tssc);
  }
}</pre>
	<pre>[Ludwig, Otto, Werner]
	[Otto, Ludwig, Werner]</pre>

Das seit Java 6 vorhandene Interface **NavigableSet<E>** erweitert das Interface **SortedSet<E>** und ist als dessen Nachfolger vorgesehen. U.a. werden zusätzlich die folgenden Methoden gefordert:

- **public E pollFirst()**
  Das erste (kleinste) Element in der navigierbaren Menge wird als Rückgabe geliefert und gelöscht.
- **public E pollLast()**
  Das letzte (größte) Element in der navigierbaren Menge wird als Rückgabe geliefert und gelöscht.
- **public E ceiling(E** *argument***)**
  Man erhält als Rückgabe das kleinste Element in der navigierbaren Menge, das mindestens ebenso groß ist wie das Argument.
- **public E floor(E** *argument***)**
  Man erhält als Rückgabe das größte Element in der navigierbaren Menge, welches das Argument nicht übertrifft.
- **public E higher(E** *argument***)**
  Man erhält als Rückgabe das kleinste Element in der navigierbaren Menge, welches das Argument übertrifft.
- **public E lower(E** *argument***)**
  Man erhält als Rückgabe das größte Element in der navigierbaren Menge, welches dem Argument unterlegen ist.
- **public Iterator<E> descendingIterator()**
  Diese Methode liefert ein Iterator-Objekt, das ein sequentielles Aufsuchen der Kollektionselemente in umgekehrter Reihenfolge unterstützt (siehe Abschnitt 10.4).

Existiert kein passendes Element, liefern die Methoden **pollFirst()**, **pollLast()**, **ceiling()**, **floor()**, **higher()** und **lower()** die Rückgabe **null**. Im folgenden Programm werden die gerade genannten Methoden vorgeführt:

Quellcode

```
import java.util.*;
class Prog {
  public static void main(String[] args) {
    NavigableSet<String> tss = new TreeSet<>();
    tss.add("a"); tss.add("c"); tss.add("k");
    System.out.println("Ceiling of e:   " + tss.ceiling("e"));
    System.out.println("Floor of b:     " + tss.floor("b"));
    System.out.println("Higher than a:  " + tss.higher("a"));
    System.out.println("Lower than b:   " + tss.lower("b"));
    System.out.println("First: " + tss.pollFirst() +
                       ", Last: " + tss.pollLast());
    System.out.println("Remaining:      " + tss);
  }
}
```

Ausgabe

```
Ceiling of e:   k
Floor of b:     a
Higher than a:  c
Lower than b:   a
First: a, Last: k
Remaining:      [c]
```

Eine geordnete Menge kann wie eine Liste als *Sequenz* bezeichnet werden, doch es bestehen u.a. die folgenden Unterschiede zwischen den beiden geordneten Kollektionen:

- In einer Menge sind keine Dubletten erlaubt, während die Eindeutigkeitsgarantie bzw. -restriktion bei Listen *nicht* besteht.
- Bei einer Liste kann der Programmierer die Position jedes einzelnen Elements uneingeschränkt kontrollieren und z. B. ein neues Element per **add()** - Methodenaufruf an einer frei wählbaren Stelle einfügen. Bei einem geordneten Menge wird hingegen die Position aller Elemente durch eine **compareTo()** - oder eine **compare()** - Methode diktiert.
- Eine Liste bietet im Unterschied zu einer geordneten Menge den Indexzugriff auf die Elemente.

- Eine geordnete Menge kann Existenzprüfungen sehr viel schneller ausführen.
- Während in eine Liste auch ein Element mit **null**-Referenz eingefügt werden kann, führt der Versuch bei einer geordneten Menge zu einer **NullPointerException**.

10.6 Abbildungen

Zur Verwaltung einer Menge von (Schlüssel-Wert) - Paaren (Assoziationen) stellt das Java Collections Framework Klassen zur Verfügung, die das generische Interface **Map<K,V>** erfüllen und als *Abbildungen* bezeichnet werden. Die Schlüssel (mit einer Konkretisierung des Typformalparameters **K** als Datentyp) werden wie eine Menge verwaltet, sodass also Eindeutigkeit herrscht (ohne Dubletten). Über einen Schlüssel ist sein Wert ansprechbar (mit einer Konkretisierung des Typformalparameters **V** als Datentyp). Man könnte z. B. in einem Programm zur Personalverwaltung eine Abbildung verwenden mit ...

- einer eindeutigen Personalnummer (Typ **Integer** als **K**-Konkretisierung)
- und einer geeigneten Klasse `Personal` (mit Instanzvariablen für den Namen, die Mail-Adresse etc.) als **V**- Konkretisierung.

Hinsichtlich der zur Schlüsselverwaltung eingesetzten Technik unterscheiden sich die beiden bekanntesten, das Interface **Map<K,V>** implementierenden Klassen:

- **HashMap<K,V>**
  Die Schlüssel werden in einer Hashtabelle verwaltet (vgl. Abschnitt 10.5.3), sind also sehr schnell auffindbar, aber unsortiert.
- **TreeMap<K,V>**
  Die Schlüssel werden in einen balancierten Binärbaum verwaltet (vgl. Abschnitt 10.5.4), sind also nicht ganz so schnell auffindbar, aber stets sortiert (im Sinne der natürlichen **K**-Ordnung oder per **Comparator<K>**).

Im Unterschied zur traditionsreichen Klasse **Hashtable<K,V>** (kleines *t* im Namen der Klasse!), die mittlerweile ebenfalls das Interface **Map<K,V>** implementiert, sind die Klassen **HashMap<K,V>** und **TreeMap<K,V>** aus Performanzgründen *nicht* Thread-sicher. Allerdings liefert die Klasse **Collections** über die statische Methode **synchronizedMap()** zu einer das Interface **Map<K,V>** implementierenden Klasse eine synchronisierte Hüllenklasse, z. B.:

```
Map<String,Person> shm =
        Collections.synchronizedMap(new HashMap<String,Person>());
```

Daneben bietet das mit Java 5 (alias 1.5) eingeführte Paket **java.util.concurrent** Schnittstellen und Klassen zur Multithreading-Unterstützung bei Abbildungs-Kollektionen, die aus Performanzgründen gegenüber den **synchronizedMap()** - Rückgaben zu bevorzugen sind (siehe Abschnitt 15.6).

Wie die Klasse **Vector<E>** (siehe Abschnitt 10.3.3) steht auch die Klasse **Hashtable<K,V>** trotz der Anpassung an das Java Collections Framework mittlerweile nicht mehr auf der *Best Practice* - Empfehlungsliste für Java-Entwickler. Sie enthält neben den empfohlenen Methoden aus dem Interface **Map<K,V>** auch noch veraltete Methoden, die nicht mehr verwendet werden sollten, weil sie den Wechsel zu einer alternativen Container-Klasse verhindern, also die Flexibilität und Wiederverwendung von Software erschweren.

10.6.1 Das Interface Map<K,V>

Im Interface **Map<K,V>** werden *zwei* Typformalparameter (für ***K**ey* und ***V**alue*) verwendet, und **Map<K,V>** stammt (im Unterschied zu **List<E>** und **Set<E>**) *nicht* von **Collection<E>** ab. Anschließend wird eine Auswahl von den insgesamt 39 Methoden in der Schnittstelle **Map<K,V>** von Java 13 vorgestellt (Überladungen mitgezählt). Insbesondere werden Methoden mit funktionalen

Schnittstellen als Parametern weggelassen, weil wir uns noch nicht mit der funktionalen Programmierung beschäftigt haben (siehe Kapitel 12).

Das Interface **Map<K,V>** verlangt von einer implementierenden Klasse u.a. die folgenden Instanzmethoden zur Aufnahme oder Veränderung von Assoziationen:

- **public V put(K** *key*, **V** *value***)** (optionale Operation)
  Wenn der Schlüssel noch nicht existiert, wird ein neues (Schlüssel-Wert) - Paar angelegt. Anderenfalls wird der alte Wert überschrieben. Um ein neues Paar mit noch unbekanntem Wert anzulegen oder einen vorhandenen Wert zu löschen, kann man das Referenzliteral **null** als Wert angeben. Als Rückgabe liefert die Methode den aktuellen Wert.
- **public void putAll (Map<? extends K,? extends V>** *map***)** (optionale Operation)
  Beim Import der (Schlüssel-Wert) - Paare aus der Parameterkollektion werden ggf. für vorhandene Schlüssel die Werte geändert. Durch die gebundene Wildcard-Typdeklarationen (siehe Abschnitt 8.3) wird für die Kollektion mit den Aufnahmekandidaten gefordert, denselben **K**- bzw. **V**-Typ wie die im **putAll()** - Aufruf angesprochene Abbildung zu verwenden oder eine Spezialisierung (Ableitung).
- **public default V replace(K** *key***, V** *value***)**
  Ist der Schlüssel vorhanden, wird sein Wert neu festgelegt. Anderenfalls passiert nichts (im Unterschied zur Methode **put()**, die eine vorhandene Assoziation ändert oder eine neue einfügt).

Zum Entfernen von Assoziationen schreibt das Interface **Map<K,V>** die folgenden Instanzmethoden vor:

- **public void clear()** (optionale Operation)
  Mit dieser Methode fordert man eine Abbildung auf, alle Assoziationen zu löschen.
- **public V remove(Object** *key***)** (optionale Operation)
  Existiert ein Eintrag mit dem angegebenen Schlüssel, wird dieser Eintrag gelöscht und sein ehemaliger Wert an den Aufrufer geliefert. Anderenfalls erhält der Aufrufer die Rückgabe **null**. Die etwas verblüffende Verwendung des Parametertyps **Object** wird gleich diskutiert.

Für Existenzprüfungen oder Wertabfragen sind die folgenden Methoden zuständig:

- **public boolean containsKey(Object** *key***)**
  Die Methode liefert **true** zurück, wenn der angegebene Schlüssel in der Abbildung vorhanden ist, sonst **false**.
- **public boolean containsValue(Object** *value***)**
  Die Methode liefert **true** zurück, wenn der angegebene Wert in der Abbildung vorhanden ist (eventuell auch mehrfach), sonst **false**. Eine Abbildungsklasse ist für die schnelle Schlüsselsuche konstruiert und muss bei einer Wertsuche zeitaufwändig nacheinander alle Elemente bis zum ersten Treffer inspizieren.
- **public V get(Object** *key***)**
  Man erhält den zum angegebenen Schlüssel gehörigen Wert oder **null**, falls der Schlüssel nicht vorhanden ist.

Warum bei den Methoden **remove()**, **containsKey()**, **containsValue()** und **get()** kein Typformalparameter (also **K** bzw. **V**) sondern die Klasse **Object** als Parametertyp zum Einsatz kommt, erschließt sich nicht sofort. Eine überzeugende Erklärung hat Luke Hutteman auf Stack Overflow präsentiert:[1] Auf diese Weise können die Methoden auch dann verwendet werden, wenn bei einer Abbildungsreferenzvariablen unbeschränkte Wildcard-Datentypen verwendet werden. Die im fol-

---

[1] https://stackoverflow.com/questions/4269147/why-does-java-mapk-v-take-an-untyped-parameter-for-the-get-and-remove-methods

genden Beispiel verwendete **Map<K,V>** - Implementation **HashMap<K,V>** wird im Abschnitt 10.6.2 vorgestellt:

Quellcode	Ausgabe

`import java.util.*;`
`class Prog {`
`    public static void main(String[] args) {`
`        Map<Integer,String> mis = new HashMap<>();`
`        mis.put(1, "A"); mis.put(2, "B");`
`        System.out.println(mis);`
`        Map<?,?> wc = mis;`
`        wc.remove(1);`
`        System.out.println(mis);`
`    }`
`}`
	`{1=A, 2=B}`
	`{2=B}`

Die folgenden Methoden liefern Kollektionen mit den Schlüsseln, Werten oder Assoziationen der angesprochenen Abbildung:

- **public Set<K> keySet()**
  Diese Methode liefert ein Objekt, das die Schnittstelle **Set<K>** erfüllt (vgl. Abschnitt 10.5) und als Sicht (engl.: *View*) auf der Menge aller Schlüssel in der angesprochenen Abbildung operiert. Man kann z. B. mit der **Set<K>** - Methode **clear()** sämtliche Schlüssel und damit sämtliche Elemente der Abbildung, löschen:

Quellcode	Ausgabe

`import java.util.*;`
`class Prog {`
`    public static void main(String[] args) {`
`        Map<Integer,String> mis = new HashMap<>();`
`        mis.put(1, "A"); mis.put(2, "B");`
`        mis.put(3, "C");`
`        System.out.println(mis);`
`        mis.keySet().clear();`
`        System.out.println(mis);`
`    }`
`}`
	`{1=A, 2=B, 3=C}`
	`{}`

  Die im Beispiel verwendete **Map<K,V>** - Implementation **HashMap<K,V>** wird im Abschnitt 10.6.2 vorgestellt. Während das Interface **Map<K,V>** im Unterschied zum Interface **Collection<E>** *keine* Methode **iterator()** besitzt, die ein Hilfsobjekt zur sequentiellen Ansprache aller Kollektionselemente liefert, ist über die per **keySet()** verfügbare Schlüsselmenge genau diese Funktionalität realisierbar.

- **public Collection<V> values()**
  Diese Methode liefert ein Objekt, das die Schnittstelle **Collection<V>** - erfüllt (vgl. Abschnitt 10.2.1) und als Sicht (engl.: *View*) auf der Menge aller Werte in der angesprochenen Abbildung operiert. Man kann z. B. mit der **Collection<V>** - Methode **remove()** die erste Assoziation mit einem bestimmten Wert löschen:

Quellcode	Ausgabe

`import java.util.*;`
`class Prog {`
`    public static void main(String[] args) {`
`        Map<Integer,String> mis = new HashMap<>();`
`        mis.put(1, "A"); mis.put(3, "C"); mis.put(2, "C");`
`        System.out.println(mis);`
`        Collection<String> cs = mis.values();`
`        cs.remove("C");`
	`{1=A, 2=C, 3=C}`
	`{1=A, 3=C}`

```
        System.out.println(mis);
    }
}
```

Die im Beispiel verwendete **Map<K,V>** - Implementation **HashMap<K,V>** wird im Abschnitt 10.6.2 vorgestellt.

- **public Set<Map.Entry<K,V>> entrySet()**
  Diese Methode liefert ein Objekt, das die Schnittstelle **Set<Map.Entry<K,V>>** - erfüllt (vgl. Abschnitt 10.5) und als Sicht auf der Menge aller (Schlüssel-Wert) - Paare aus der angesprochenen Abbildung operiert. Es bietet als Mengenverwaltungsobjekt einen Iterator, mit dem sich die Elemente der Abbildung nacheinander ansprechen lassen.
  Die Kompetenzen eines Elements der Ergebnismenge werden durch das Interface **Map.Entry<K,V>** beschrieben, das als (implizit statisches und öffentliches) Mitglieds-Interface innerhalb von **Map<K,V>** definiert ist:[1]

```
public interface Map<K,V> {
  . . .
   interface Entry<K,V> {
      K getKey();
      V getValue();
      V setValue(V value);
      boolean equals(Object o);
      int hashCode();
   }
  . . .
}
```

Über die von **keySet()**, **values()** oder **entrySet()** gelieferten Kollektionen kann man Elemente aus der Abbildung entfernen, aber keine Elemente aufnehmen.

Mit den folgenden Methoden informiert eine Abbildung über die Anzahl ihrer Elemente:

- **public boolean isEmpty()**
  Mit dieser Methode erfährt man, ob die angesprochene Abbildung leer ist.
- **public int size()**
  Liefert die Anzahl der Elemente in der Abbildung

Alle zu einer Änderung der Kollektion führenden Methoden (z. B. **put()**, **putAll()**, **clear()**, **remove()** usw.) sind in der API-Dokumentation durch den Zusatz *optional operation* markiert. Es ist einer Klasse erlaubt, sich bei der Implementation solcher Methoden auf das Werfen einer **UnsupportedOperationException** zu beschränken. Es wird allerdings von jeder implementierenden Klasse erwartet, in der Dokumentation offenzulegen, für welche Methoden nur eine Pseudo-Implementation vorhanden ist.

Wo das Verhalten einer Methode von Übereinstimmungsprüfungen abhängt (z. B. **containsKey()**, **remove()**, **containsValue()**) ist bei der Interface-Implementierung die **equals()** - Methode des Schlüssel- bzw. Werttyps zu verwenden (statt des Identitätsoperators). Dementsprechend wird in der Schlüssel- bzw. Wertklasse für die von **Object** geerbte **equals()** - Methode eine sinnvolle Überschreibung erwartet.

Man sollte nach Möglichkeit für Variablen und Parameter, die auf eine Abbildung zeigen, den Interface-Datentyp **Map<K,V>** (oder eine Variante mit geordnetem Schlüssel, siehe Abschnitt 10.6.3) verwenden, damit zur Lösung einer konkreten Aufgabe die optimale **Map<K,V>** - Implementie-

[1] Mitglieds-Schnittstellen von Schnittstellen sind implizit öffentlich und statisch, werden also wie Top-Level-Schnittstellen behandelt, müssen aber einen Doppelnamen führen (siehe Abschnitt 9.2.5, vgl. Gosling et al. 2019, Abschnitt 9.5).

rung im OCP-Sinn (Open-Closed - Prinzip, vgl. Abschnitt 4.1.1.3), also praktisch ohne Quellcode-Änderungen, genutzt werden kann.

Seit Java 9 besitzt die Schnittstelle **Map<K,V>** eine statische und generische Fabrikmethode namens **of()**, mit der sich auf einfache Weise *unveränderliche* Abbildungen erstellen lassen, z. B.:

Quellcode:

```
import java.util.*;
class Prog {
    public static void main(String[] args) {
        Map<Integer,String> mis = Map.of(1,"a",  2,"b",  3,"c");
        System.out.println(mis);
//          mis.put(4, "d");
    }
}
```

Ausgabe:

```
{1=a, 3=c, 2=b}
```

Die auskommentierte Anweisung hätte einen Laufzeitfehler vom Typ **UnsupportedOperation-Exception** zur Folge, weil die **of()** - Produkte unveränderlich sind.

Den Aufruf der statischen, generischen Methode richtet man an den Rohtyp der generischen Schnittstelle **Map<K,V>**. Wegen der Typinferenz-Fähigkeiten des Compilers ist es *nicht* erforderlich, beim Aufruf der generischen Methode **of()** die Typaktualparameter anzugeben:

```
Map<Integer,String> mis = Map.<Integer,String>of(1, "a", 2, "b", 3, "c");
```

An die statische **Map<K,V>** - Methode **of()** übergibt man einen **K**- und einen **V**-Aktualparameter, um eine Assoziation einzufügen. An die ebenfalls in Java 9 eingeführte statische **Map<K,V>** - Methode **ofEntries()**, die gleichfalls eine unveränderliche Abbildung erstellt, übergibt man Objekte des innerhalb von **Map<K,V>** definierten Typs **Entry<K,V>**. Im folgenden Beispiel werden diese Objekte von der statischen und generischen Methode **entry()** aus der Schnittstelle **Map<K,V>** erstellt. Die Methode wird statisch in die Quellcodedatei importiert (siehe Abschnitt 6.1.2.2 zum statischen Import):

Quellcode:

```
import java.util.*;
import static java.util.Map.entry;

class Prog {
    public static void main(String[] args) {
        Map<Integer,String> mis = Map.ofEntries(
                entry(1,"a"),
                entry(2,"b"),
                entry(3,"c"));
        System.out.println(mis);
    }
}
```

Ausgabe:

```
{1=a, 3=c, 2=b}
```

10.6.2 Die Klasse HashMap<K,V>

Über einen Mengenverwaltungscontainer (z. B. aus der Klasse **HashSet<E>**) kann man für Objekte eines Typs festhalten, ob sie sich in einer Menge befinden oder nicht. Ein einfaches Beispiel ist etwa die Menge aller Zeichen (**Character**-Objekte), die in einem Text auftreten. Mit den im aktuellen Abschnitt 10.6 behandelten Abbildungsklassen lassen sich zu jedem Objekt im Container noch zusätzliche Informationen aufbewahren. Im gerade erwähnten Beispiel könnte man zu jedem Zeichen noch die Häufigkeit des Auftretens speichern. Für den Text "Otto spielt Lotto" resultiert die folgende Tabelle mit den Zeichen und ihren Auftretenshäufigkeiten:

```
p --> 1
s --> 1
t --> 5
e --> 1
i --> 1
l --> 1
L --> 1
O --> 1
o --> 3
```

Durch die Pfeilnotation wird betont, dass es sich tatsächlich um eine Abbildung im mathematischen Sinn handelt (von einer Teilmenge der Buchstaben in die Menge der natürlichen Zahlen).

Die folgende statische Methode `countLetters()` liefert ein **HashMap<Character,** `Mint`**>** - Objekt mit Paaren aus einem Schlüssel vom Typ **Character** und einem Wert vom Typ `Mint` (eine selbst entworfene **int**-Hüllenklasse, vgl. Übungsaufgabe im Abschnitt 5.5). Als Rückgabetyp wird jedoch nicht die parametrisierte Klasse angegeben, sondern das parametrisierte Interface **Map<Character,** `Mint`**>**, damit der tatsächliche Typ später geändert werden kann:

```
static Map<Character, Mint> countLetters(String text) {
   HashMap<Character, Mint> fred = new HashMap<>();
   Mint temp;
   for (int i = 0; i < text.length(); i++) {
      char pc = text.charAt(i);
      if (Character.isLetter(pc)) {
         Character c = pc;
         if (fred.containsKey(c)) {
            temp = fred.get(c);
            temp.val++;
            fred.replace(c, temp);
         } else
            fred.put(c, new Mint(1));
      }
   }
   return fred;
}
```

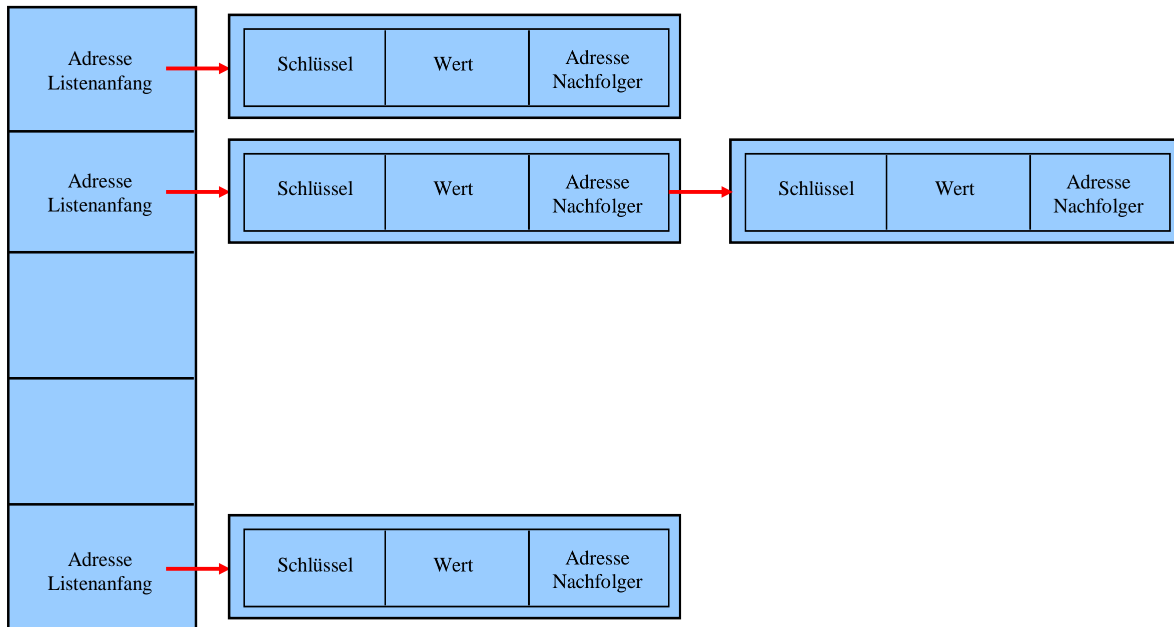
Wie die im Abschnitt 10.5.3 beschriebene Klasse **HashSet<E>** arbeitet auch die Klasse **HashMap<K,V>** mit einer Hashtabelle, verwendet also einen Array mit einfach verketteten Listen (Buckets) als Einträgen.[1] Folglich muss die **K**-Konkretisierungsklasse eine **hashCode()** - Implementierung besitzen, welche die im Abschnitt 10.5.3 angegebenen Bedingungen erfüllt.

---

[1] Ein Blick in den API-Quellcode von Java 13 zeigt übrigens, dass die Klasse **HashSet<E>** intern eine **HashMap<E,Object>** - Kollektion zur Datenablage verwendet und alle Elemente mit einem Dummy-Objekt als **V**-Wert anlegt:

```
public class HashSet<E> extends AbstractSet<E>
      implements Set<E>, Cloneable, java.io.Serializable {
  static final long serialVersionUID = -5024744406713321676L;

  private transient HashMap<E,Object> map;

  // Dummy value to associate with an Object in the backing Map
  private static final Object PRESENT = new Object();

  /**
   * Constructs a new, empty set; the backing {@code HashMap} instance has
   * default initial capacity (16) and load factor (0.75).
   */
  public HashSet() {
     map = new HashMap<>();
  }
   . . .
```

Ein **HashMap<K,V>** - Objekt kann so skizziert werden:



Um die (Schlüssel-Wert) - Paare in einem **HashMap<K,V>** - Container sukzessive anzusprechen, kann man über die Methode **entrySet()** ein Mengenverwaltungsobjekt mit den (Schlüssel-Wert) - Paaren als Elementen anfordern und dessen Iterator (siehe Abschnitt 10.4) benutzen. Dabei zeigt sich erwartungsgemäß eine zufällig wirkende Reihenfolge.

Mit der Klasse **LinkedHashMap<K,V>** steht eine **HashMap<K,V>** - Ableitung zur Verfügung, deren Objekte sich die Einfügereihenfolge der Elemente merken. Dies wird durch den Zusatzaufwand einer doppelt verlinkten Liste realisiert. Die Elemente merken sich ...

- das nächste Element im selben Bucket
- und das als nächstes eingefügte Element.

Im Ergebnis erhalten wir einen Iterator, der die Einfügereihenfolge verwendet und außerdem flotter arbeitet als die **HashMap<E>** - Variante, weil die leeren Buckets nicht aufgesucht werden müssen (Naftalin & Wadler 2007, S. 181).

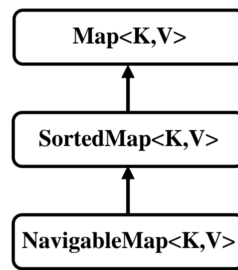
10.6.3 Interfaces für Abbildungen mit geordneten Schlüsseltypen

Analog zu den Verhältnissen bei den Schnittstellen **Set<E>**, **SortedSet<E>** und **NavigableSet<E>** zur Mengenverwaltung (siehe Abschnitt 10.5) existieren für Abbildungen mit geordneten Schlüsseltypen zum Interface **Map<K,V>** die folgenden Erweiterungen:

- das Interface **SortedMap<K,V>**
- das mit Java 6 als designierter Nachfolger hinzu gekommene Interface **NavigableMap<K,V>**

Hier sind die drei Interfaces und Ihre Beziehungen zu sehen:

```
}
```



Es gibt zwei Möglichkeiten, die Ordnung der von einer **SortedMap<E>** - Implementierung verwalteten Elemente zu begründen:

- Der Schlüsseltyp **K** erfüllt das Interface **Comparable<K>**, besitzt also eine Instanzmethode **compareTo()** und somit eine natürliche Ordnung.
- Man übergibt dem Konstruktor ein Objekt, das die Schnittstelle **Comparator<K>** erfüllt und folglich für den Typ **K** oder für eine Basisklasse **B** von **K** eine Vergleichsmethode mit dem folgenden Definitionskopf bietet:
  public int compare(K *k1*, **K** *k2*) bzw. **public int compare(B** *e1*, **B** *e2*)

Die Methoden **compareTo()** bzw. **compare()** müssen konsistent mit der **equals()** - Methode des Elementtyps sein, denn:

- Das Interface **Map<E>** ist basierend auf der **equals()** - Methode definiert.
- Eine **SortedMap<E>** - Implementation benutzt für alle Elementvergleiche (und damit auch für die Dubletten-Erkennung) die Methoden **compareTo()** bzw. **compare()**.

Das Interface **SortedMap<K,V>** fordert von implementierenden Klassen u.a. die folgenden Methoden:

- **public Comparator<? super K> comparator()**
  Es wird das bei der Konstruktion übergebene **Comparator**-Objekt geliefert oder **null**, wenn die natürliche Ordnung der Schlüsselklasse **K** genutzt wird. In diesem Fall muss **K** das Interface **Comparable<K>** implementieren.
- **public K firstKey()**
  Liefert den ersten (kleinsten) Schlüssel in der sortierten Abbildung
- **public K lastKey()**
  Liefert den letzten (größten) Schlüssel in der sortierten Abbildung
- **public SortedMap<K,V> headMap(K** *obereSchranke***)**
  Man erhält ein Objekt aus einer Klasse, welche das Interface **SortedMap<K,V>** erfüllt, und als Sicht (engl.: *View*) auf der Teilmenge der (Schlüssel-Wert) - Paare aus der angesprochenen Abbildung mit einem Schlüssel *unterhalb* der oberen Schranke operiert. Alle Methoden des View-Objekts wirken sich auf die Originalkollektion aus, sodass man z. B. mit der Methode **clear()** die komplette **headMap()** - Teilmenge löschen kann.
- **public SortedMap<K,V> tailMap(K** *untereSchranke***)**
  Die Methoden des resultierenden View-Objekts wirken auf die Teilmenge der (Schlüssel-Wert) - Paare aus der angesprochenen Abbildung mit einem Schlüssel ab der unteren Schranke (inklusive).

- **public SortedMap<K,V> subMap(K** *untereSchranke,* **K** *obereSchranke***)**
  Man erhält eine Sicht auf eine Teilmenge der angesprochenen Abbildung, festgelegt durch einen Schlüsselbereich beginnend mit der unteren Schranke (inklusive) und endend mit der oberen Schranke (exklusive). Alle Methoden des View-Objekts wirken sich auf die Originalkollektion aus.

Im Definitionskopf einer das Interface **SortedMap<K,V>** implementierenden Klasse (z. B. **TreeMap<K,V>**) kann und sollte darauf verzichtet werden, den Typformalparamater **K** auf die Schnittstelle **Comparable<K>** zu restringieren, weil die von **SortedMap<K,V>** geforderte Methode **comparator()** (siehe oben) die Vergleichbarkeit der Elemente sicherstellt.

Das seit Java 6 vorhandene Interface **NavigableMap<K,V>** erweitert das Interface **SortedMap<K,V>** und ist als Nachfolger vorgesehen. U.a. werden zusätzlich die folgenden Methoden gefordert:

- **public Map.Entry<K,V> firstEntry()**
  Aus der navigierbaren Abbildung wird das Element mit dem ersten (kleinsten) Schlüssel als Rückgabe geliefert.
- **public Map.Entry<K,V> lastEntry()**
  Aus der navigierbaren Abbildung wird das Element mit dem letzten (größten) Schlüssel als Rückgabe geliefert.
- **public Map.Entry<K,V> pollFirstEntry()**
  Aus der navigierbaren Abbildung wird das Element mit dem ersten (kleinsten) Schlüssel als Rückgabe geliefert und gelöscht.
- **public Map.Entry<K,V> pollLastEntry()**
  Aus der navigierbaren Abbildung wird das Element mit dem letzten (größten) Schlüssel als Rückgabe geliefert und gelöscht.
- **public K ceilingKey(K** *key***)**
  Man erhält als Rückgabe den kleinsten Schlüssel in der navigierbaren Abbildung, der mindestens ebenso groß ist wie der Aktualparameter.
- **public K floorKey(K** *key***)**
  Man erhält als Rückgabe den größten Schlüssel in der navigierbaren Abbildung, welcher den Aktualparameter *nicht* übertrifft.
- **public K higherKey(K** *key***)**
  Man erhält als Rückgabe den kleinsten Schlüssel in der navigierbaren Abbildung, welcher den Aktualparameter übertrifft.
- **public K lowerKey(K** *key***)**
  Man erhält als Rückgabe den größten Schlüssel in der navigierbaren Abbildung, welcher dem Aktualparameter unterlegen ist.
- **public Map.Entry<K,V> ceilingEntry(K** *key***)**
  Man erhält als Rückgabe den Eintrag in der navigierbaren Abbildung mit dem kleinsten Schlüssel, der mindestens ebenso groß ist wie der Aktualparameter.
- **public Map.Entry<K,V> floorEntry(K** *key***)**
  Man erhält als Rückgabe den Eintrag in der navigierbaren Abbildung mit dem größten Schlüssel, welcher den Aktualparameter *nicht* übertrifft.

- **public Map.Entry<K,V> higherEntry(K** *key***)**
  Man erhält als Rückgabe den Eintrag in der navigierbaren Abbildung mit dem kleinsten Schlüssel, welcher den Aktualparameter übertrifft.
- **public Map.Entry<K,V> lowerEntry(K** *key***)**
  Man erhält als Rückgabe den Eintrag in der navigierbaren Abbildung mit dem größten Schlüssel, welcher dem Aktualparameter unterlegen ist.

Zum Interface-Datentyp **Map.Entry<K,V>** siehe die Beschreibung der **Map<K,V>** - Methode **entrySet()** im Abschnitt 10.6.1.

Existiert kein passendes Element, liefern die Methoden **firstEntry()**, **lastEntry()**, **pollFirstEntry()**, **pollLastEntry()**, **ceilingKey()**, **floorKey()**, **higherKey()**, **lowerKey()**, **ceilingEntry()**, **floorEntry()**, **higherEntry()** und **lowerEntry()** die Rückgabe **null**. Im Abschnitt 10.6.4 über die Klasse **TreeMap<K,V>** findet sich ein Beispielprogramm, das einige **NavigableMap<K,V>** - Methoden demonstriert.

10.6.4 Die Klasse TreeMap<K,V>

Analog zu den Verhältnissen bei den Mengenverwaltungsklassen **HashSet<E>** und **TreeSet<E>** gibt es zur Abbildungsverwaltungsklasse **HashMap<K,V>** für geordnete Schlüsseltypen eine Alternative namens **TreeMap<K,V>** mit einem balancierten Binärbaum zur Verwaltung der Schlüssel. Diese Klasse erfüllt neben dem Interface **Map<K,V>** auch die Schnittstellen **SortedMap<K,V>** und **NavigableMap<K,V>** für Abbildungen mit einem geordneten Schlüsseltyp.

Ersetzt man in der Buchstabenfrequenzen - Methode `countLetters()` aus Abschnitt 10.6.2 das **HashMap<Character,** `Mint`> - Objekt durch ein **TreeMap<Character,** `Mint`> - Objekt,

```java
static NavigableMap<Character, Mint> countLetters(String text) {
   TreeMap<Character, Mint> fred = new TreeMap<>();
   Mint temp;
   for (int i = 0; i < text.length(); i++) {
      char pc = text.charAt(i);
      if (Character.isLetter(pc)) {
         Character c = pc;
         if (fred.containsKey(c)) {
            temp = fred.get(c);
            temp.val++;
            fred.replace(c, temp);
         } else
            fred.put(c, new Mint(1));
      }
   }
   return fred;
}
```

dann sind die Elemente der Rückgabe gemäß der **compareTo()** - Implementierung in der Klasse **Character** sortiert:

```
L --> 1
O --> 1
e --> 1
i --> 1
l --> 1
o --> 3
p --> 1
s --> 1
t --> 5
```

Im folgenden Programm werden einige Methoden aus dem Interface **NavigableMap<Integer, String>** (vgl. Abschnitt 10.6.3) vorgeführt:

Quellcode

```
import java.util.*;
class Prog {
  public static void main(String[] args) {
    NavigableMap<Integer,String> tms = new TreeMap<>();
    tms.put(1,"a"); tms.put(2,"b"); tms.put(4,"d");
    System.out.println(tms);

    Map.Entry<Integer,String> fi = tms.firstEntry();
    System.out.println(fi.getValue());

    tms.pollFirstEntry();
    System.out.println(tms);

    System.out.println("\nceilingKey(3) = "+tms.ceilingKey(3));
    System.out.println("floorKey(3)   = "+tms.floorKey(3));
    System.out.println("heigherKey(4) = "+tms.higherKey(4));
    System.out.println("lowerKey(4)   = "+tms.lowerKey(3));
  }
}
```

Ausgabe

```
{1=a, 2=b, 4=d}
a
{2=b, 4=d}

ceilingKey(3) = 4
floorKey(3)   = 2
heigherKey(4) = null
lowerKey(4)   = 2
```

10.7 Vergleich der Kollektionsarchitekturen

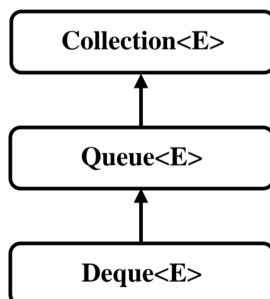
In der folgenden Tabelle sind für die zentralen Kollektionsarchitekturen im JCF einige Merkmale und Operationskomplexitäten zusammengefasst:

Merkmale/Operationen		Kollektionsarchitektur			
		Array-basiert	Verkettete Liste	Hashtabelle	Binärbaum
Merkmale	Definierte Positionen der Elemente[1]	+	+	-	-
	Frei von Dubletten	-	-	+	+
	Automatische Anordnung nach Inhalt	-	-	-	+
Operationen	Existenzprüfung	$O(n)$	$O(n)$	$O(1)$	$O(\log_2 n)$
	Elemente einfügen oder löschen	$O(n)$	$O(1)$	$O(1)$	$O(\log_2 n)$

[1] Per Voreinstellung wird die Einfügereihenfolge konserviert, doch kann die Position eines Elements beliebig festgelegt werden.

10.8 Warteschlangen

Seltener als Listen, Mengen und Abbildungen werden Kollektionen mit einer Warteschlagen- oder Stapel-Architektur benötigt (Evans & Flanagan 2015, 253). Die von einer Warteschlange erwarteten Verhaltenskompetenzen werden im Interface **Queue<E>** beschrieben, das vom Interface **Collection<E>** abstammt:



Das von **Queue<E>** abgeleitete Interface **Deque<E>** (dt.: Deck) beschreibt das Verhalten von Doppelschlagen.

Eine Warteschlange ist wie eine Liste linear organisiert, doch können nur am Kopfende Elemente entnommen und nur am Schwanzende Elemente eingefügt werden, sodass die Elemente nach dem FIFO-Prinzip (First In First Out) bedient werden. Anders als bei Listen ist es *nicht* möglich, das Element an einer bestimmten Position abzurufen. Im Abschnitt 15.2.4 über Verfahren zur automatisierten Thread-Koordination für Produzenten-Konsumenten - Konstellationen wird das von **Queue<E>** abgeleitete Interface **BlockingQueue<E>** vorgestellt.

Bei der ebenfalls linear organisierten, das Interface **Deque<E>** realisierenden **Doppelschlange** können an *beiden* Enden Elemente eingefügt und entnommen werden. Man verwendet diese Kollektion häufig im einseitigen Betrieb als **Stapel**, wobei nach dem LIFO-Prinzip (Last In First Out) das zuletzt eingefügt (bzw. aufgelegte) Element zuerst bedient wird. Eine häufig als Stapel verwendete **Deque<E>** - Implementation ist die Klasse **ArrayDeque<E>**. Von der früher für diesen Zweck verwendeteten Klasse **Stack<E>** wird mittlerweile in der API-Dokumentation abgeraten. Im Kurs können Stapel leider aus Zeitgründen nicht behandelt werden.

10.9 Nützliche Methoden in der Klasse Collections

Die Klasse **Collections** erbringt durch statische und teilweise generische Methoden zahlreiche Dienstleistungen für Kollektionsobjekte, von denen die Fähigkeit, eine Thread-sichere (synchronisierte) Hülle zu einem Kollektionsobjekt zu liefern, bereits mehrfach erwähnt worden ist:

- **public static <T extends Object & Comparable<? super T>>**
T max(Collection<? extends T> *coll***)**
Diese Methode liefert das größte Element einer Kollektion. Durch die erste, scheinbar überflüssige Restriktion (**T extends Object**) für den Typformalparameter **T**, wird aus Kompatibilitätsgründen dafür gesorgt, dass im Bytecode (nach der Typlöschung) der Rückgabetyp **Object** steht (statt **Comparable**).[1] Wie im Abschnitt 8.1.3.2 erläutert wurde, orientiert sich die Typlöschung bei multiplen Bindungen ausschließlich an der *ersten* Bindung. Mit der zweiten Restriktion (**T extends Comparable<? super T>**) wird vom Typ **T** eine Methode **compareTo()** verlangt, wobei **T** selbst oder eine Basisklasse von **T** als Parametertyp erlaubt ist. Damit ist insgesamt als **T**-Konkretisierung auch eine Kollektionsklasse möglich, welche die Methode **compareTo()** nicht selbst implementiert, sondern von einer Basisklasse erbt (vgl. Abschnitt 9.3).

[1] Diese Erklärung stammt von der Webseite:
http://www.angelikalanger.com/GenericsFAQ/FAQSections/ProgrammingIdioms.html#FAQ104

- **public static <T extends Object & Comparable<? super T>>**
  T min(Collection<? extends T> *coll***)**
  Diese Methode liefert das kleinste Element einer Kollektion.
- **public static <T extends Comparable<? super T>> void sort(List<T>** *list***)**
  Eine Liste wird unter Verwendung der (eventuell geerbten) **compareTo()** - Methode ihres Elementtyps sortiert.
- **public static <T> void sort(List<T>** *list***, Comparator<? super T>** *comp***)**
  Eine Liste wird unter Verwendung der vom zweiten Parameter-Objekt beherrschten Methode **compare()** sortiert.
- **public static void reverse(List<?>** *list***)**
  Die Elemente einer Liste erhalten eine umgekehrte Reihenfolge.
- **public static void shuffle(List<?>** *list***)**
  Diese Methode bringt die Elemente einer Liste in eine neue, zufällige Reihenfolge.
- **public static <T> int binarySearch(List<? extends Comparable<? super T>>** *list*, **T** *obj***)**
  Diese Methode durchsucht eine Liste nach einem Objekt unter Verwendung des Halbierungsverfahrens. Sie liefert den Index des Treffers oder den Wert -1.
- **public static <E> Collection<E> synchronizedCollection(Collection<E>** *coll***)**
  public static <E> List<E> synchronizedList(List<E> *list***)**
  public static <E> Set<E> synchronizedSet(Set<E> *set***)**
  public static <K,V> Map<K,V> synchronizedMap(Map<K,V> *map***)**
  Zu einem Container, der die Schnittstelle **Collection<E>**, **List<E>**, **Set<E>** oder **Map<K,V>** erfüllt, erhält man eine Thread-sichere (synchronisierte) Verpackung. Was das genau bedeutet, wird im Kapitel 15 über Multithreading erläutert. Dabei wird sich allerdings im Abschnitt 15.6 zeigen, dass nur eine *bedingte* Thread-Sicherheit vorliegt.
- **public static <E> Collection<E> unmodifiableCollection(Collection<? extends E>** *coll***)**
  public static <E> List<E> unmodifiableList(List<? extends E> *list***)**
  public static <E> Set<E> unmodifiableSet(Set<? extends E> *set***)**
  public static <K,V> Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> *map***)**
  Zu einem Container, der die Schnittstelle **Collection<E>**, **List<E>**, **Set<E>** oder **Map<K,V>** erfüllt, erhält man eine Sicht, die zwar einen lesenden, aber keinen schreibenden Zugriff auf die Elemente erlaubt.
- **public static final <E> List<E> emptyList()**
  public static final <E> Set<E> emptySet()
  public static final <K,V> Map<K,V> emptyMap()
  Wenn eine Methode eine erwartete Rückgabe mit Kollektionstyp nicht erstellen kann, ist die Rückgabe einer leeren Kollektion oft gegenüber alternativen Kommunikationsverfahren (Rückgabe von **null**, Ausnahmefehler) zu bevorzugen.

Im folgenden Programm werden einige **Collections**-Methoden demonstriert:

Quellcode	Ausgabe

<pre>import java.util.*;
class Prog {
  public static void main(String[] args) {
    List<String> ls = new LinkedList<>();
    ls.add("Otto"); ls.add("Luise"); ls.add("Rainer");
    System.out.println("Original: \t"+ls);
    Collections.sort(ls);
    System.out.println("Sortiert: \t"+ls);
    Collections.reverse(ls);
    System.out.println("Invertiert: \t"+ls);
    Collections.shuffle(ls);
    System.out.println("Verwirbelt: \t"+ls);
    System.out.println("Minimum: \t"+Collections.min(ls));
    System.out.println("Maximum: \t"+Collections.max(ls));
  }
}</pre>
	<pre>Original: [Otto, Luise, Rainer]
	Sortiert: [Luise, Otto, Rainer]
	Invertiert: [Rainer, Otto, Luise]
	Verwirbelt: [Otto, Luise, Rainer]
	Minimum: Luise
	Maximum: Rainer</pre>

10.10 Übungsaufgaben zum Kapitel 10

1) Erstellen Sie ein Programm, das 6 Lottozahlen (von 1 bis 49) zieht und sortiert ausgibt. Diese Aufgabe haben Sie schon einmal mit Hilfe von Array-Techniken gelöst (siehe Abschnitt 5.5). Das JCF sorgt dafür, dass sich der Lösungsaufwand deutlich reduziert.

2) Erweitern Sie die als Aufgabe zu Abschnitt 5.2 in Auftrag gegebene Klasse `StringUtil` um statische Methoden mit den folgenden Leistungen:

- `public static List<String> getWordList(CharSequence text)`
  Diese Methode soll für die Parameterzeichenfolge eine Liste mit den enthaltenen Wörtern in der Reihenfolge ihres Auftretens liefern.
- `public static NavigableMap<Character,Integer>`
  `getStartCharFreqs(CharSequence text)`
  Diese Methode soll zur Parameterzeichenfolge als **TreeMap<Character, Integer>** - Objekt eine sortierte Tabelle liefern, die für jeden Buchstaben (die Groß-/Kleinschreibung ignorierend) angibt, wie viele Wörter mit diesem Buchstaben beginnen.

Im folgenden Programm wird die Verwendung der Methoden demonstriert:

```
import java.util.*;
import de.uni_trier.zimk.util.strings.StringUtil;

class StringUtilTest {
   public static void main(String[] args) {
      String s = "In diesem Satz kommt der Anfangsbuchstabe a zweimal vor.";
      System.out.println(StringUtil.getWordList(s));

      NavigableMap<Character,Integer> freqs = StringUtil.getStartCharFreqs(s);
      System.out.println("\nAnfangsbuchstabe  Häufigkeit");
      for (Character c : freqs.keySet())
         System.out.printf("%-17c %d\n", c, freqs.get(c));
   }
}
```

Es sollte die folgende Ausgabe liefern:

```
[In, diesem, Satz, kommt, der, Anfangsbuchstabe, a, zweimal, vor.]

Anfangsbuchstabe  Häufigkeit
a                 2
d                 2
i                 1
k                 1
s                 1
v                 1
z                 1
```

Weil Sie seit Abschnitt 5.2 viel dazugelernt haben, sollte die Klasse `StringUtil` modernisiert werden:

- Die Klasse `StringUtil` sollte in ein explizites Paket aufgenommen werden, damit ihre Verwendbarkeit nicht länger auf das Standardpaket beschränkt ist (vgl. Kapitel 6).
- In vorhandenen Methoden sollte die Klasse **String** als Parameterdatentyp durch das Interface **CharSequence** ersetzt werden (siehe Kapitel 9).

3) Erstellen Sie eine Klasse mit generischen, statischen und öffentlichen Methoden für elementare Operationen aus dem Bereich der Mengenlehre. Realisieren Sie zumindest den Schnitt, die Vereinigung und die Differenz von zwei Mengen (Kollektionsobjekten gem. Abschnitt 10.5) mit identischem (ansonsten beliebigem) Referenztyp. Für zwei Mengen

`A = {'a', 'b', 'c'}, B = {'b', 'c', 'd'}`

sollen ungefähr die folgenden Ausgaben möglich sein:

```
Menge A
a
b
c

Menge B
b
c
d

Durchschnitt von A und B
b
c

Vereinigung von A und B
a
b
c
d

Differenz von A und B
a
```

4) Erstellen Sie ein Programm, das zu den Spalten einer Datenmatrix mit **double**-Elementen jeweils eine Häufigkeitstabelle erstellt und nach den Merkmalsausprägungen aufsteigend sortiert ausgibt, z. B.:

```
Datenmatrix mit 5 Fällen und 3 Merkmalen:
    1,00     2,00     4,00
    1,00     2,00     5,00
    2,00     2,00     6,00
    2,00     1,00     5,00
    3,00     1,00     4,00

Häufigkeiten Merkmal 0:
    Wert       N
    1,00       2
    2,00       2
    3,00       1

Häufigkeiten Merkmal 1:
    Wert       N
    1,00       2
    2,00       3

Häufigkeiten Merkmal 2:
    Wert       N
    4,00       2
    5,00       2
    6,00       1
```

11 Ausnahmebehandlung

Durch Programmierfehler (z. B. versuchter Array-Zugriff mit ungültigem Indexwert) oder durch besondere Umstände (z. B. fehlerhafte Eingabedaten, Speichermangel, unterbrochene Netzwerkverbindungen) kann die reguläre Ausführung einer Methode scheitern. In diesem Kapitel geht es um die Behandlung von Ausnahmesituationen, die während der Laufzeit auftreten. Java bietet ein modernes Verfahren zur Meldung und Behandlung von Laufzeitproblemen: An der Unfallstelle wird ein Ausnahmeobjekt aus der Klasse **java.lang.Exception** oder aus einer problemspezifischen Unterklasse erzeugt und der unmittelbar verantwortlichen Methode „zugeworfen". Diese Methode wird somit über das Problem informiert und mit Daten für die Behandlung versorgt.

Die Initiative beim Auslösen einer Ausnahme kann ausgehen …

- von der JVM
  Sie wirft z. B. ein Ausnahmeobjekt aus der Klasse **ArithmeticException** bei einer versuchten Ganzzahldivision durch 0.
- vom Programm, wozu auch die verwendeten Bibliotheksklassen gehören
  In jeder Methode und in jedem Konstruktor kann mit der **throw**-Anweisung (siehe Abschnitt 11.5) eine Ausnahme geworfen werden (z. B. wegen ungeeigneter Aktualparameterwerte).

Die unmittelbar von einer Ausnahme betroffene Methode steht in der Regel am Ende einer Sequenz verschachtelter Methodenaufrufe, und entlang der Aufrufersequenz haben die beteiligten Methoden jeweils folgende Reaktionsmöglichkeiten:

- das Ausnahmeobjekt abfangen und das Problem behandeln
  Im tatsächlichen Programmablauf fliegen natürlich keine Objekte durch die Gegend, die mit irgendwelchen Gerätschaften eingefangen werden. Stattdessen überprüft die Laufzeitumgebung, ob die betroffene Methode geeigneten Code zur Behandlung des Ausnahmeobjekts (einen sogenannten *Exception-Handler*) enthält. Gegebenenfalls wird dieser Exception-Handler angesprungen und erhält quasi als Aktualparameter das Ausnahmeobjekt mit Informationen über das Problem. Nach der Ausnahmebehandlung kann die Methode ...
  - entweder ihre Tätigkeit mit einem angepassten Handlungsplan fortsetzen
  - oder ihrerseits ein Ausnahmeobjekt werfen (entweder das ursprüngliche oder ein informativeres) und somit die Kontrolle an ihren Aufrufer zurückgeben.
- das Ausnahmeobjekt ignorieren
  In diesem Fall besitzt eine Methode keinen zum Ausnahmeobjekt passenden Exception-Handler. Die Methode wird beendet, und das Ausnahmeobjekt wird dem Vorgänger in der Aufrufersequenz überlassen.

Wir werden uns anhand verschiedener Versionen eines Beispielprogramms damit beschäftigen,

- was bei unbehandelten Ausnahmen geschieht,
- wie man eine Methode auf Ausnahmen vorbereitet, um diese abfangen zu können,
- wie man in einer Methode selbst Ausnahmen wirft,
- wie man eigene Ausnahmeklassen definiert.

Man kann von keinem Programm erwarten, dass es unter allen widrigen Umständen normal funktioniert. Doch müssen Schäden (z. B. Datenverluste) nach Möglichkeit verhindert werden, und der Benutzer sollte eine nützliche Information zum aufgetretenen Problem erhalten. Bei vielen Methodenaufrufen ist es realistisch und erforderlich, auf Störungen des normalen Ablaufs vorbereitet zu sein. Dies folgt schon aus **Murphy's Law** (zitiert nach Wikipedia):[1]

---

[1] https://de.wikipedia.org/wiki/Murphys_Gesetz

Anything that can go wrong will go wrong.

Eine Besonderheit von Java besteht in der Unterscheidung zwischen Ausnahmeklassen, für die eine potentiell betroffene Methode einen Exception-Handler bereithalten muss (z. B. **IOException**) und Ausnahmeklassen, bei denen die Entscheidung über eine Vorbereitung dem Programmierer überlassen wird (z. B. **NumberFormatException**).

11.1 Unbehandelte Ausnahmen

Findet die JVM (als Werfer bzw. Vermittler) zu einer Ausnahme in einem Konsolenprogramm (mit der bislang von uns verwendeten Single-Thread - Architektur) entlang der Aufrufersequenz bis hinauf zur **main()** - Methode keinen Exception-Handler, dann bringt sie den im Ausnahmeobjekt enthaltenen Unfallbericht auf die Konsole und beendet das Programm, z. B.:[1]

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "vier"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
  at java.lang.Integer.parseInt(Integer.java:580)
  at java.lang.Integer.parseInt(Integer.java:615)
  at Fakul.convertInput(Fakul.java:3)
  at Fakul.main(Fakul.java:14)
```

In dieser Ausgabe der automatisch aufgerufenen **Exception**-Methode **printStackTrace()** sind enthalten (vgl. Abschnitt 11.2.3):

- Name der Ausnahmeklasse (im Beispiel: **java.lang.NumberFormatException**)
- Fehlermeldung (im Beispiel: `For input string: "vier"`)
- Aufrufersequenz von der Unfallstelle bis zur Methode **main()**

Trotz der planhaften Vorgehensweise der Laufzeitumgebung und der zahlreichen brauchbaren Informationen erlebt der verärgerte Benutzer einen Programmabsturz und denkt eventuell über alternative Programme nach.

Wird ein Programm im Rahmen unsere Entwicklungsumgebung IntelliJ ausgeführt, dann besitzt der Unfallbericht eine auffällige Färbung und klickbare Verknüpfungen zu den Quellcodezeilen der betroffenen Methoden in der Aufrufersequenz, z. B.:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "vier"
 at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
 at java.lang.Integer.parseInt(Integer.java:580)
 at java.lang.Integer.parseInt(Integer.java:615)
 at Fakul.convertInput(Fakul.java:3)
 at Fakul.main(Fakul.java:14)
```

Der Absturzbericht stammt vom folgenden Programm, das die Fakultät zu einer nichtnegativen ganzen Zahl berechnen soll, die beim Start als Programmargument übergeben wird. Die eigentliche Fakultätsberechnung findet in der **main()** - Methode statt, während die Konvertierung und Validierung der übergebenen Zeichenfolge in der Methode `convertInput()` erfolgt. Diese wiederum stützt sich bei der Konvertierung auf die statische Methode **parseInt()** der API-Klasse **Integer**:

---

[1] Im Beispiel wird die Laufzeitumgebung aus dem OpenJDK 8 verwendet.

```
class Fakul {
   static int convertInput(String instr) {
      int arg = Integer.parseInt(instr);
      if (arg >= 0  &&  arg <= 170)
         return arg;
      else
         return -1;
   }

   public static void main(String[] args) {
      int argument = -1;

      if (args.length > 0)
         argument = convertInput(args[0]);
      else {
         System.out.println("Kein Argument angegeben");
         System.exit(1);
      }

      if (argument != -1) {
         double fakul = 1.0;
         for (int i = 1; i <= argument; i++)
               fakul = fakul * i;
         System.out.printf("%s! = %.0f", args[0], fakul);
      } else
         System.out.printf("Keine ganze Zahl im Intervall [0, 170]: " + args[0]);
   }
}
```

Ein Programm sollte sich generell bemühen, Ausnahmefehler nach Möglichkeit durch Kontrollmaßnahmen zu verhindern. Im Beispiel überprüft die Methode **main**(), ob tatsächlich ein Kommandozeilenargument in `args[0]` vorhanden ist, bevor diese **String**-Referenz beim Aufruf der Methode `convertInput()` als Parameter verwendet wird. Damit wird verhindert, dass es zu einer **ArrayIndexOutOfBoundsException** kommt, wenn der Benutzer das Programm ohne Kommandozeilenargument startet. Weil das Programm in dieser Situation kein Fakultätsargument und auch keine Möglichkeit zum Befragen des Benutzers hat, informiert es über das Problem und beendet sich durch einen Aufruf der Methode **System.exit**(), wobei als Aktualparameter ein **Exitcode** übergeben wird. Dieser landet beim Betriebssystem und steht unter Windows nach dem Programmende in der Umgebungsvariablen ERRORLEVEL zur Verfügung, z. B.:

```
>java Fakul
Kein Argument angegeben

>echo %ERRORLEVEL%
1
```

Diese Reaktion auf ein fehlendes Programmargument kann als akzeptabel gelten. An Stelle der für Benutzer irritierenden und wenig hilfreichen Ausnahmemeldung durch das Laufzeitsystem

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at Fakul.main(Fakul.java:13)
```

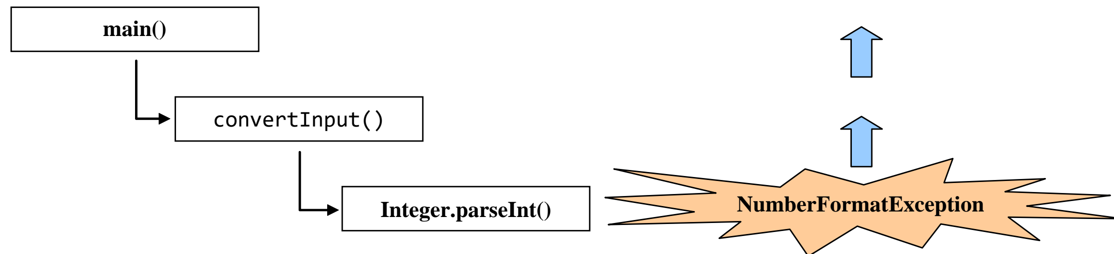
erscheint eine verwertbare Information:

```
Kein Argument angegeben
```

Die Methode `convertInput()` überprüft, ob die aus dem übergebenen **String**-Parameter ermittelte **int**-Zahl außerhalb des zulässigen Wertebereichs für eine Fakultätsberechnung mit **double**-Ergebniswert liegt, und meldet ggf. den Wert -1 als Fehlerindikator zurück. Weil der Ergebnistyp

double verwendet wird, sind nur Argumente bis zum maximalen Wert 170 erlaubt.[1] Die Methode **main()** kennt die spezielle Bedeutung der Rückgabe -1, sodass die unsinnige Fakultätsberechnung für ein negatives Argument und der wenig hilfreiche Ergebniswert Unendlich vermieden werden. Diese traditionelle Fehlerbehandlung per **Rückgabewert** (engl.: *return code*) ist *nicht* grundsätzlich als überholt und ineffizient zu bezeichnen, aber in vielen Situationen doch der gleich vorzustellenden Kommunikation über Ausnahmeobjekte unterlegen (siehe Abschnitt 11.3 zum Vergleich von Fehlerrückmeldung und Ausnahmebehandlung).

Trotz seiner präventiven Bemühungen ist das Programm leicht aus dem Tritt zu bringen, indem man es mit einer nicht konvertierbaren Zeichenfolge füttert (z. B. „vier“). Die zunächst betroffene Methode[2] **Integer.parseInt()** wirft daraufhin eine **NumberFormatException**. Diese wird vom Laufzeitsystem entlang der Aufrufreihenfolge an `convertInput()` und dann an **main()** gemeldet:



Weil beide Methoden keinen Exception-Handler bereithalten, bringt die JVM den im Ausnahmeobjekt enthaltenen Unfallbericht (den *stack trace*) auf die Konsole (siehe oben) und beendet das Programm.[3]

11.2 Ausnahmen abfangen

Die Startversion des Programms zur Fakultätsberechnung beherrscht weder das Behandeln noch das Werfen von Ausnahmen. Wir machen uns nun daran, diese kommunikativen Kompetenzen nachzurüsten.

11.2.1 Die try-catch-finally - Anweisung

In Java wird die Behandlung von Ausnahmen über die **try-catch-finally** - Anweisung unterstützt:

---

[1] Durch Verwendung der Klasse **BigDecimal** ist es möglich, für beliebig große Argumente die Fakultät zu bestimmen, und mit Hilfe der in Java 8 eingeführten Stromoperationen kann ohne nennenswerten Programmieraufwand die für große Argumente erforderliche Rechenzeit durch parallele Ausführung in mehreren Threads begrenzt werden (siehe Abschnitt 12.2.5.4.2). Wir verzichten im aktuellen Kapitel auf diese Verbesserungen, um ein möglichst einfaches Beispiel zur Demonstration der Ausnahmebehandlung zu erhalten. Später werden Sie in einer Übungsaufgabe zum Kapitel 12 ein Programm erstellen, das für beliebige positive Ganzzahlen die Fakultät berechnet und dabei alle verfügbaren CPU-Kerne nutzt.

[2] Aufrufverschachtelungen *innerhalb* der Standardbibliothek ignorieren wir an dieser Stelle. Im Abschnitt 11.2.3 wird die Angelegenheit mit Hilfe des API-Quellcodes genauer untersucht.

[3] Genau genommen, ist das Geschehen in Folge einer nicht abgefangenen Ausnahme komplexer:

- Zunächst wird nur der Thread (Ausführungsfaden) beendet, in dem die Ausnahme aufgetreten ist. Existiert (wie bei unseren bisherigen Konsolenprogrammen) *kein* weiterer Benutzer-Thread, endet das Programm.
- Der zu terminierende Thread wird von der JVM über die statische **Thread**-Methode **getUncaughtExceptionHandler()** nach seinem **UncaughtExceptionHandler** befragt. Dieses Objekt enthält einen Aufruf der Methode **uncaughtException()**, und diese Methode fordert das per Aktualparameter übergebene **Exception**-Objekt auf, die Methode **printStackTrace()** auszuführen.

try {
Überwachter Block mit Anweisungen für den regulären Ablauf
}
catch (*Ausnahmeklassenliste1 parameter1***) {**
Anweisungen für die Behandlung einer Ausnahme aus einer aufgelisteten Klasse
oder aus einer daraus abgeleiteten Klasse
}
// Optional können weitere Ausnahmen abgefangen werden:
catch (*Ausnahmeklassenliste2 parameter2***) {**
Anweisungen für die Behandlung einer Ausnahme aus einer aufgelisteten Klasse
oder aus einer daraus abgeleiteten Klasse
}
. . .
// Optionaler **finally**-Block mit Abschlussarbeiten. Besitzt eine **try**-Anweisung
// einen **finally**-Block, muss kein **catch**-Block vorhanden sein.
finally {
Anweisungen, die unabhängig vom Auftreten einer Ausnahme ausgeführt werden sollen
}

Die Anweisungen für den ungestörten Ablauf setzt man in den **try**-Block. Nachdem eine Anweisung des **try**-Blocks eine Ausnahme verursacht oder aktiv geworfen hat, werden die weiteren Anweisungen des **try**-Blocks *nicht* mehr ausgeführt. In einem **try**-Block sollten Anweisungen zusammengestellt werden, die allesamt erfolgreich ausgeführt werden müssen, damit ein sinnvolles Ergebnis entsteht.

Treten bei der Ausführung dieses überwachten Blocks *keine* Fehler auf, wird das Programm hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**-Block ausgeführt wird (siehe Abschnitt 11.2.1.2).

Java erlaubt folgende Varianten der **try** - Anweisung:

- **try-catch** - Anweisung
- **try-finally** - Anweisung
- **try-catch-finally** - Anweisung

Ein **try**-, **catch**- oder **finally**-Block benötigt auch dann ein einrahmendes Paar geschweifter Klammern, wenn nur *eine* Anweisung enthalten ist.

Weil es der obigen Syntaxbeschreibung im Quellcodedesign trotz Unterstützung durch Kommentare an Präzision fehlt, sollen Sie in einer Übungsaufgabe ein Syntaxdiagramm zur **try-catch-finally** - Anweisung erstellen (siehe Abschnitt 11.9).

11.2.1.1 Ausnahmebehandlung per catch-Block

Tritt im **try**-Block eine Ausnahme auf, wird seine Ausführung *abgebrochen*, und das Laufzeitsystem sucht in der **try**-Anweisung nach einem **catch**-Block, welcher eine Ausnahme der betroffenen Klasse behandeln kann.

Ein **catch**-Block, den man auch als *Exception-Handler* bezeichnet, verfügt in gewisser Analogie zu einer Methode in seinem Kopfbereich über eine Typangabe und einen Formalparameter. Vor Java 7 konnte pro Exception-Handler nur *eine* zu behandelnde Ausnahmeklasse angegeben werden, z. B.:

```
catch (NumberFormatException e) {
  . . .
}
```

Seit Java 7 ist neben diesem **Single-Catch - Block** auch ein **Multiple-Catch - Block** mit einer Liste von Ausnahmeklassen erlaubt, für die eine einheitliche Behandlung vereinbart werden soll, z. B.:

```
catch (NumberFormatException | ArithmeticException e) {
  .  .  .
}
```

Bei einem Multi-Catch - Block sind folgende Regeln zu beachten:

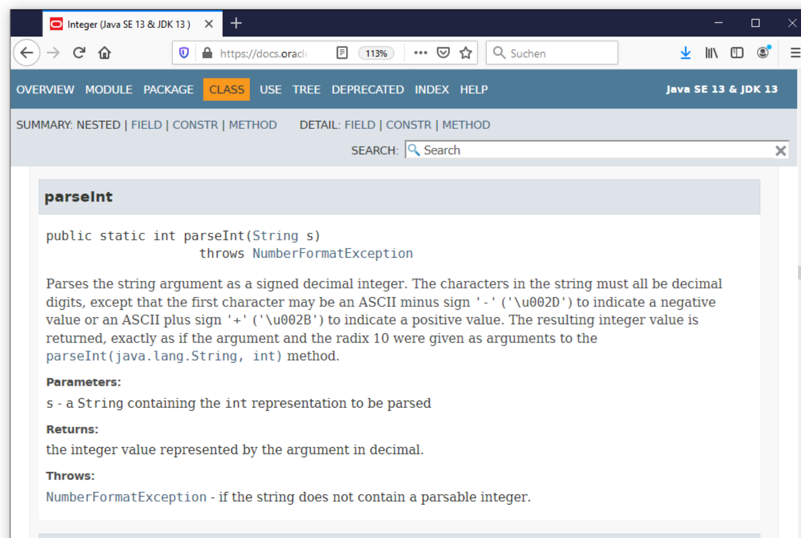
- Die Namen der Ausnahmeklassen werden durch einen senkrechten Strich | getrennt, der bekanntlich (zwischen zwei logischen Ausdrücken) auch für die logische ODER-Operation steht (vgl. Abschnitt 3.5.5). Das ist eine gute Wahl, denn im obigen Beispiel wird der Exception-Handler aktiv, wenn eine **NumberFormatException** *oder* eine **ArithmeticException** aufgetreten ist.
- Es ist es verboten (und auch sinnlos), neben einer Klasse `K` auch von `K` abgeleitete Klassen in die Liste aufzunehmen.
- Ein Multi-Catch - Block wird vom Compiler in entsprechend viele, hintereinander stehende Single-Catch - Blöcke mit identischen Anweisungen umgesetzt.

Das Laufzeitsystem sucht für ein zu behandelndes Ausnahmeobjekt nach einem **catch**-Block mit einer passenden Ausnahmeklasse und führt ggf. den zugehörigen Anweisungsblock aus. Für jedes Ausnahmeobjekt wird maximal *ein* **catch**-Block ausgeführt. Weil die Liste der **catch**-Blöcke von oben nach unten durchsucht wird, müssen *breitere* Ausnahmeklassen stets *unter* spezielleren stehen. Freundlicherweise stellt der Compiler die Einhaltung dieser Regel sicher.

In der folgenden Variante der Methode `convertInput()` aus unserem Beispielprogramm zur Fakultätsberechnung wird eine von **Integer.parseInt()** ausgelöste **NumberFormatException** abgefangen. Der **catch**-Block beendet die Methodenausführung mit dem Rückgabewert -2, der als Fehlerindikator zu verstehen ist:

```
static int convertInput(String instr) {
   int arg;
   try {
      arg = Integer.parseInt(instr);
   } catch (NumberFormatException e) {
      return -2;
   }
   if (arg < 0  || arg > 170) {
      return -1;
   } else
      return arg;
}
```

Wie die API-Dokumentation zu Java 13 zeigt, sind von **parseInt()** keine Ausnahmen aus anderen Klassen zu erwarten:



In der Methode **main()** muss der neue Fehlerindikator berücksichtigt werden:

```
public static void main(String args[]) {
  int argument = -1;

  if (args.length > 0)
    argument = convertInput(args[0]);
  else {
    System.out.println("Kein Argument angegeben");
    System.exit(1);
  }

  switch (argument) {
    case -1:  System.out.print("Keine ganze Zahl im Intervall [0, 170]: " + args[0]);
              break;
    case -2:  System.out.printf("Fehler beim Konvertieren von: \"%s\"", args[0]);
              break;
    default:  double fakul = 1.0;
              for (int i = 1; i <= argument; i++)
                fakul = fakul * i;
              System.out.printf("%s! = %.0f", args[0], fakul);
  }
}
```

Beim Programmstart mit einem nicht-konvertierbaren Kommandozeilenargument erscheint nun eine informative Fehlermeldung an Stelle eines „Absturzprotokolls“ der JVM, z. B.:

```
Fehler beim Konvertieren von: "vier"
```

Je nach Algorithmus kommen als Aufgaben für einen **catch**-Block in Frage (selbstverständlich auch im Kombination):

- Reparatur
  Manchmal ist es möglich, den aufgetretenen Fehler zu beheben oder zu kompensieren (z. B. zu umgehen).
- Rückabwicklung
  Man kann versuchen, bereits realisierte und aufgrund der Ausnahme nunmehr unerwünschte Effekte des unterbrochenen **try**-Blocks wieder rückgängig zu machen.
- Ersetzung der Ausnahme durch eine informativere Alternative
  Viele **catch**-Blöcke betätigen sich als Informationsvermittler und werfen selbst eine Ausnahme, um dem Aufrufer einen leichter verständlichen Unfallbericht zu liefern (siehe Abschnitt 11.5).

- Fehlermeldung und/oder Fehlerprotokollierung
  Wenn eine gescheiterte Operation abgebrochen werden muss, sollte der Benutzer eine gut verständliche Fehlermeldung erhalten. Ein Eintrag in eine Logdatei kann den Software-Entwickler oder einen Administrator dabei unterstützen, die Ursache des Fehlers zu finden (siehe Kapitel 14 zur Dateiausgabe). Nach einer Fehlermeldung oder -protokollierung ist es in der Regel sinnvoll, die abgefangene Ausnahme erneut zu werfen.

Es sollte verhindert werden, dass ein Objekt durch einen gescheiterten Methodenaufruf in einen defekten Zustand gerät. Wenn das nicht möglich ist, müssen andere Programmierer durch eine klare Dokumentation davon abhalten werden, ein havariertes Objekt weiter zu verwenden.

Eine Ausnahme nur scheinbar zu behandelt und letztlich zu ignorieren, ist eine riskante Praxis, die auf jeden Fall in einem Kommentar begründet werden muss. Die folgende **try**-Anweisung stammt aus einem Beispielprogramm, das im Abschnitt 11.2.1.2 vorgestellt wird:

```
try {
   Thread.sleep(3000);
} catch (InterruptedException ignored) {
   // Die Ausnahme darf ignoriert werden, weil der Thread nicht abgebrochen wird.
}
```

Durch die statische **Thread**-Methode **sleep()** wird der aktuelle Thread in einen Schlaf von 3000 Millisekunden Dauer versetzt. Beantragt in dieser Zeit ein anderer Thread, dass der schlafende Thread abgebrochen werden soll, wird die Methode **sleep()** aktiv, um eine Verzögerung bei der Antragsbearbeitung zu vermeiden. Die Methode **sleep()** wirft in dieser Situation eine **InterruptedException**, die den Schläfer reaktiviert und veranlasst, eine Terminierung seiner Tätigkeit in Erwägung zu ziehen. Im konkreten Beispielprogramm findet aber kein Unterbrechungsversuch statt, sodass keine **InterruptedException** auftreten kann und keine ernsthafte Behandlung erforderlich ist. Eine **InterruptedException** darf keinesfalls generell ignoriert werden, und jedes Ignorieren irgendeiner Ausnahme muss sorgfältig erwogen und begründet werden. Im konkreten Fall der **InterruptedException** handelt sich um übrigens um eine sogenannte *geprüfte* Ausnahme, auf die man sich vorbereiten muss (siehe Abschnitt 11.4.2).

Wenn ein **catch**-Block erfolgreich (d.h. ohne weiteres Ausnahmeereignis) ausgeführt worden ist, dann wird die Methode hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**-Block ausgeführt wird. Der **finally**-Block wird auch dann ausgeführt, wenn die Methode im **catch**-Block per **return**-Anweisung verlassen wird. Weitere Details zum Programmablauf bei der Ausnahmebehandlung werden im Abschnitt 11.2.2 behandelt.

Die eventuell im überwachten **try**-Block auf die Anweisung, die zur Ausnahme geführt hat, noch folgenden Anweisungen werden *nicht* ausgeführt. Java verwendet also in Bezug auf den betroffenen **try**-Block eine *terminierende Form* der Ausnahmebehandlung, wobei aber nicht die gesamte Methode und erst recht nicht die ganze Anwendung enden müssen. Nach einer Ausnahmebehandlung kann der Benutzer z. B. die Gelegenheit erhalten, den gescheiterten Vorgang (z. B. einen Netzwerkzugriff) zu wiederholen (z. B. nach erfolgter WLAN-Aktivierung). Viele Ausnahmebehandlungen bestehen darin, den Benutzer über das aufgetretene Problem zu informieren und bei einem erfolgreichen Neuversuch zu unterstützen.

11.2.1.2 Aufräumarbeiten im finally-Block

In einen **finally**-Block gehören Anweisungen, die auf jeden Fall ausgeführt werden sollen:

- Nach der ungestörten Ausführung des **try**-Blocks
- Nach einer Ausnahmebehandlung in einem **catch**-Block (auch beim Verlassen des **catch**-Blocks durch eine neue Ausnahme)
- Nach dem Auftreten einer unbehandelten Ausnahme im **try**-Block
- Beim Beenden der Methode durch eine **return**-Anweisung im **try**-Block oder in einem **catch**-Block

Vor Java 7 wurde der **finally**-Block meist dazu verwendet, Ressourcen wie Datei - und Netzverbindungen freizugeben. Für diesen Zweck stellt Java seit der Version 7 mit der **try-with-resources** - Anweisung jedoch eine weitaus bessere Lösung zur Verfügung (siehe Abschnitt 11.8). Daher fällt es etwas schwer, ein plausibles und einfaches Anwendungsbeispiel für den **finally**-Block zu finden.

Für das folgende Beispiel ist ein Vorgriff auf das Kapitel über Multithreading erforderlich. Es wird die Verwaltung eines Bankkontos durch zwei Threads (nebenläufige Ausführungsfäden des Programms) simuliert:

- Ein freundlicher Thread zahlt ständig Geldbeträge auf das Konto ein.
- In einem gleichzeitig aktiven Thread darf der Benutzer Geld abheben.

Über ein Sperrobjekt aus der Klasse **ReentrantLock** (Paket **java.util.concurrent.locks**) wird verhindert, dass beide Threads gleichzeitig auf das Konto zugreifen, weil dabei ein fehlerhaftes Verhalten des Programms resultieren könnte (vgl. Abschnitt 15.2.3). Die Methode zum Abheben aktiviert die Sperre, erfragt dann beim Benutzer den gewünschten Betrag, erleichtert das Konto und gibt die Sperre schließlich frei, damit der Einzahlungs-Thread wieder Zugang zum Konto erhält. Um die Wünsche des Benutzers entgegenzunehmen, wird mit der Methode **nextLine()** der Klasse **Scanner** aus dem Paket **java.util** (vgl. Abschnitt 3.4.1) eine mit **Enter** quittierte Zeile von der Konsole gelesen und anschließend mit der Methode **parseInt()** der Klasse **Integer** eine Interpretation der Eingabe versucht. Die Methode **parseInt()** reagiert auf eine nicht interpretierbare Eingabe mit dem Werfen eines Ausnahmeobjekts vom Typ **NumberFormatException** (siehe Abschnitt 11.1). Daher wird sie im Rahmen einer **try**-Anweisung mit **catch**-Block für die **NumberFormatException** aufgerufen:

```
import java.util.*;
import java.util.concurrent.locks.ReentrantLock;

class FinallyDemo implements Runnable {
   int konto;
   ReentrantLock lock = new ReentrantLock();
   Random ran = new Random();

   public void run() {
      while (true) {
         lock.lock();
         konto += ran.nextInt(30);
         System.out.print("\nKontostand erhöht auf: "+konto);
         lock.unlock();
         try {
            Thread.sleep(1000);
         } catch (InterruptedException ignored) {
            // Die Ausnahme darf ignoriert werden, weil der Thread nicht abgebrochen wird.
         }
      }
   }
```

```
    void abheben() {
       Scanner input = new Scanner(System.in);
       int amount;
       while (true) {
          try {
             Thread.sleep(3000);
          } catch (InterruptedException ignored) {
             // Die Ausnahme darf ignoriert werden, weil der Thread nicht abgebrochen wird.
          }
          lock.lock();
          try {
             System.out.print("\n\nWelcher Betrag soll abgehoben werden (Beenden mit Betrag < 0): ");
             amount = Integer.parseInt(input.nextLine());
             if (amount < 0) {
                System.exit(0);
             }
             konto -= amount;
             System.out.println("Neuer Kontostand:  " + konto);
          } catch(NumberFormatException e) {
             System.out.println("Kein gültiger Betrag!");
          } finally {
             lock.unlock();
          }
       }
    }

    public static void main(String args[]) {
       FinallyDemo fd = new FinallyDemo();
       (new Thread(fd)).start();
       fd.abheben();
    }
 }
```

Es ist sicherzustellen, dass die ggf. von einem Ausnahmeobjekt betroffene Methode `abheben()` unter allen Umständen (also auch bei gestörter Ausführung) das Sperrobjekt wieder freigibt, damit weitere Einzahlungen durch den zweiten Thread möglich sind. Daher wird der erforderliche **unlock**() - Aufruf in einen **finally**-Block platziert.

Ansonsten demonstriert die Methode `abheben()`, dass nach der Behandlung einer Ausnahme durchaus der Normalbetrieb wieder aufgenommen werden kann:

```
Kontostand erhöht auf: 24
Kontostand erhöht auf: 36
Kontostand erhöht auf: 63
Kontostand erhöht auf: 64

Welcher Betrag soll abgehoben werden (Beenden mit Betrag < 0): vier
Kein gültiger Betrag!

Kontostand erhöht auf: 84
Kontostand erhöht auf: 91
Kontostand erhöht auf: 112

Welcher Betrag soll abgehoben werden (Beenden mit Betrag < 0): 4
Neuer Kontostand:  108

Kontostand erhöht auf: 112
Kontostand erhöht auf: 113
Kontostand erhöht auf: 126
```

Im Programm wird an zwei Stellen die **InterruptedException** ignoriert, die von der statischen **Thread**-Methode **sleep()** zu erwarten ist:

```
try {
   Thread.sleep(3000);
} catch (InterruptedException ignored) {
   // Die Ausnahme darf ignoriert werden, weil der Thread nicht abgebrochen wird.
}
```

Im Abschnitt 11.2.1.1 wurde begründet, warum in diesem Fall das Ignorieren bzw. Verschlucken einer Ausnahme gerechtfertigt ist.

11.2.2 Programmablauf bei der Ausnahmebehandlung

Findet die JVM für eine Ausnahme in der aktuellen Methode keinen zuständigen **catch**-Block, dann sucht sie entlang der Aufrufersequenz weiter. Dies macht es leicht, die Behandlung einer Ausnahme der bestgerüsteten Methode zu überlassen. Im folgenden Beispiel dürfen Sie allerdings keine optimierte Einsatzplanung erwarten. Es demonstriert einige Programmabläufe infolge von Ausnahmen, die auf verschiedenen Stufen einer Aufrufhierarchie geworfen bzw. behandelt werden. Um das Beispiel einfach zu halten, wird auf Praxisnähe verzichtet. Das Programm nimmt via Kommandozeile ein Argument entgegen, interpretiert es numerisch und ermittelt den Rest aus der Division der Zahl 10 durch das Argument:

```
class Sequenzen {
   static int calc(String instr) {
      int arg = 0;
      try {
         System.out.println("try-Block von calc()");
         arg = Integer.parseInt(instr);
         arg = 10 % arg;
      } catch (NumberFormatException e) {
         System.out.println("NumberFormatException-Handler in calc()");
      } finally {
         System.out.println("finally-Block von calc()");
      }
      System.out.println("Nach try-Anweisung in calc()");
      return arg;
   }

   public static void main(String[] args) {
      try {
         System.out.println("try-Block von main()");
         System.out.println("10 % "+args[0]+" = "+calc(args[0]));
      } catch (ArithmeticException e) {
         System.out.println("ArithmeticException-Handler in main()");
      } finally {
         System.out.println("finally-Block von main()");
      }
      System.out.println("Nach try-Anweisung in main()");
   }
}
```

Die Methode **main()** lässt die eigentliche Arbeit von der Methode `calc()` erledigen und bettet deren Aufruf in eine **try**-Anweisung mit **catch**-Block für die **ArithmeticException** ein, die das Laufzeitsystem z. B. bei einer versuchten Ganzzahldivision durch null auslöst. `calc()` benutzt die Klassenmethode **Integer.parseInt()** sowie den Modulo-Operator in einem **try**-Block, wobei nur die potentiell von **Integer.parseInt()** zu erwartende **NumberFormatException** abgefangen wird.

Wir betrachten einige Konstellationen mit ihren Konsequenzen für den Programmablauf:

a) Normaler Ablauf
b) Exception in `calc()`, die dort auch behandelt wird
c) Exception in `calc()`, die in **main()** behandelt wird
d) Exception in **main()**, die nirgends behandelt wird

a) Normaler Ablauf

Beim Programmablauf *ohne* Ausnahmen (hier mit Kommandozeilen-Argument „8“) werden die **try**- und die **finally**-Blöcke von **main()** und `calc()`ausgeführt. Es kommt zu folgenden Ausgaben:

```
try-Block von main()
try-Block von calc()
finally-Block von calc()
Nach try-Anweisung in calc()
10 % 8 = 2
finally-Block von main()
Nach try-Anweisung in main()
```

b) Exception in `calc()`, die dort auch behandelt wird

Wird beim Ausführen der Anweisung

```
arg = Integer.parseInt(instr);
```

eine **NumberFormatException** an `calc()` gemeldet (z. B. wegen Kommandozeilen-Argument „acht“ von **parseInt()** geworfen), kommt der zugehörige **catch**-Block zum Einsatz. Dann folgen:

- **finally**-Block in `calc()`
- restliche Anweisungen in `calc()` (hinter der **try**-Anweisung)
  Im **try**-Block von `calc()` hinter dem Unfallort stehende Anweisungen werden *nicht* ausgeführt. So wird verhindert, dass ein Algorithmus mit fehlerhaften Zwischenergebnissen weiterläuft. Wenn eine Methode auf traditionelle Weise per Rückgabewert einen Fehler signalisiert, kann es hingegen passieren, dass die warnende Rückgabe ignoriert und der laufende Algorithmus fortgesetzt wird (vgl. Abschnitt 11.3).

An **main()** wird keine Ausnahme gemeldet, also werden hier nacheinander ausgeführt:

- **try**-Block
- **finally**-Block
- restliche Anweisungen

Insgesamt erhält man die folgenden Ausgaben:

```
try-Block von main()
try-Block von calc()
NumberFormatException-Handler in calc()
finally-Block von calc()
Nach try-Anweisung in calc()
10 % acht = 0
finally-Block von main()
Nach try-Anweisung in main()
```

Zu der unsinnigen Ausgabe

```
10 % acht = 0
```

kommt es, weil die **NumberFormatException** in `calc()` *nicht sinnvoll* behandelt wird. Wenn ein **catch**-Block lediglich eine Fehlermeldung ausgibt und/oder einen Logdateieintrag schreibt, sollte er in der Regel die gefangene Ausnahme erneut werfen oder stattdessen eine informativere Ausnahme werfen. Das aktuelle Beispiel soll nur dazu dienen, Programmabläufe bei der Ausnahmebehandlung zu demonstrieren.

c) Exception in `calc()`, die in main() behandelt wird

Wird vom Laufzeitsystem eine **ArithmeticException** an `calc()` gemeldet (z. B. wegen Kommandozeilen-Argument „0“), dann findet sich in dieser Methode kein passender Handler. Weil die Ausnahme im **try**-Block einer **try-catch-finally** - Anweisung auftrat, wird noch der zugehörige **finally**-Block ausgeführt, bevor die Methode verlassen wird, um entlang der Aufrufsequenz nach einem geeigneten Handler zu suchen.

In **main()** findet sich ein **ArithmeticException**-Handler, der nun zum Einsatz kommt. Dann geht es weiter mit dem zugehörigen **finally**-Block. Schließlich wird das Programm hinter der **try-**Anweisung der Methode **main()** fortgesetzt:

```
try-Block von main()
try-Block von calc()
finally-Block von calc()
ArithmeticException-Handler in main()
finally-Block von main()
Nach try-Anweisung in main()
```

d) Exception in main(), die nirgends behandelt wird

Übergibt der Benutzer kein Kommandozeilen-Argument, tritt in **main()** bei Zugriff auf `args[0]` eine **ArrayIndexOutOfBoundsException** auf (vom Laufzeitsystem geworfen). Weil sich kein zuständiger Handler findet, wird das Programm vom Laufzeitsystem beendet. Zuvor wird der **finally**-Block von **main()** noch ausgeführt, die Anweisungen hinter der **try**-Anweisung aber nicht mehr:

```
try-Block von main()
finally-Block von main()
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at Sequenzen.main(Sequenzen.java:23)
```

In der Praxis ist es oft sinnvoll, **try**-Anweisungen zu schachteln, wobei sowohl innerhalb eines **try**- als auch innerhalb eines **catch**- oder **finally**-Blocks wiederum eine komplette **try**-Anweisung stehen darf. Daraus ergeben sich weitere Ablaufvarianten für eine flexible Ausnahmebehandlung.

11.2.3 Diagnostische Ausgaben

Ein Ausnahmeobjekt enthält viele Informationen, die sich für diagnostische Ausgaben eignen. Statt im **catch**-Block eine eigene Fehlermeldung zu formulieren, kann man die **toString()** - Methode des übergebenen Ausnahmeobjekts aufrufen, was hier implizit im Rahmen eines **println()** - Aufrufs geschieht:

```
catch (NumberFormatException e) {
     System.out.println(e);
}
```

Das Ergebnis enthält den Namen der Ausnahmeklasse und eventuell eine situationsspezifische Information, falls eine solche beim Erstellen des Ausnahmeobjekts an den Konstruktor übergeben wurde, z. B.:

```
java.lang.NumberFormatException: For input string: "vier"
```

Wer nur die situationsspezifische Fehlerinformation, aber nicht den Namen der Ausnahmeklasse sehen möchte, verwendet die Methode **getMessage()**, z. B.:

```
System.out.println(e.getMessage());
```

In Beispiel erscheint nur noch:

```
For input string: "vier"
```

Eine weitere nützliche Information, die ein Ausnahmeobjekt parat hat, ist die Aufrufersequenz (engl.: *stack trace*) von der **main()** - Methode bis zur Unfallstelle. Mit der **Throwable**-Methode **printStackTrace()**befördert man den Namen der Ausnahmeklasse, die Fehlermeldung und die Aufrufersequenz zur Standardfehlerausgabe (**System.err**), die per Voreinstellung (ohne Umleitung) mit der Standardausgabe (**System.out**) identisch ist:

```
catch (NumberFormatException e) {
   e.printStackTrace();
    . . .
}
```

Im Beispiel erscheint:

```
java.lang.NumberFormatException: For input string: "vier"
        at java.base/java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.base/java.lang.Integer.parseInt(Unknown Source)
        at java.base/java.lang.Integer.parseInt(Unknown Source)
        at Sequenzen.calc(Sequenzen.java:6)
        at Sequenzen.main(Sequenzen.java:23)
```

Bleibt ein Ausnahmeobjekt unbehandelt, erhält es von der JVM die Aufforderung **printStackTrace()**, bevor das Programm endet.[1] Daher haben wir schon mehrfach das Ergebnis eines **printStackTrace()** - Aufrufs gesehen.

Vielleicht wundern Sie sich darüber, dass in der zuletzt präsentierten Aufrufersequenz gleich *zwei* **Integer**-Methoden namens **parseInt()** auftauchen. Ein Blick in den API-Quellcode zeigt, dass die von unserer Methode `convertInput()` aufgerufene **parseInt()** - Überladung mit einem Parameter vom Typ **String**

```
public static int parseInt(String s) throws NumberFormatException {
    return parseInt(s, 10);
}
```

die eigentliche Arbeit einer Überladung mit einem zusätzlichen Parameter für die Basis des Zahlensystems überlässt, die schließlich auf das Problem stößt und die **NumberFormatException** wirft:

```
public static int parseInt(String s, int radix)
        throws NumberFormatException {
    . . .
}
```

11.3 Ausnahmeobjekte im Vergleich zur Fehlerkommunikation per Rückgabewert

Die konventionelle Fehlerbehandlung verwendet meist den **Rückgabewert** einer Methode, um über Störungen bei der Ausführung der Methode zu informieren. Ein Rückgabewert kann …

- ausschließlich zur Fehlermeldung dienen. Meist wird dann ein ganzzahliger **Returncode** mit Datentyp **int** verwendet, wobei die 0 einen erfolgreichen Ablauf signalisiert, während andere Zahlen für bestimmte Fehlertypen stehen. Soll nur zwischen Erfolg und Misserfolg unterschieden werden, bietet sich der Rückgabewert **boolean** an.
- neben den Ergebnissen einer ungestörten Ausführung durch spezielle Werte auch Störungen signalisieren. Diese Technik wird im Beispielprogramm von Abschnitt 11.2.1.1 und auch in der Methode **read()** der API-Klasse **FileInputStream** verwendet:

[1] Genau genommen, verläuft die Kommunikation etwas komplizierter: Der zu terminierende Thread wird von der JVM über die statische **Thread**-Methode **getUncaughtExceptionHandler()** nach seinem **UncaughtExceptionHandler** befragt. Dieses Objekt enthält einen Aufruf der Methode **uncaughtException()**, und diese Methode fordert das per Aktualparameter übergebene **Exception**-Objekt auf, die Methode **printStackTrace()** auszuführen.

```
public int read()
         throws IOException
```

Reads a byte of data from this input stream. This method blocks if no input is yet available.

Specified by:

`read` in class `InputStream`

Returns:

the next byte of data, or `-1` if the end of the file is reached.

Throws:

`IOException` - if an I/O error occurs.

Eine für die aktuelle Diskussion interessante, durchaus nicht ungewöhnliche Eigenschaft der Methode **read()** besteht darin, dass einerseits per Rückgabewert (mit kombinierter Bedeutung) über ein zu erwartendes Problem beim Lesen informiert wird, und andererseits per Ausnahmeobjekt (aus der Klasse **IOException**) über weniger erwartete Fehler berichtet wird.

Sollen z. B. drei Methoden, deren Rückgabewerte ausschließlich zur Fehlermeldung dienen, nacheinander aufgerufen werden, dann wird die vom Algorithmus diktierte simple Sequenz:

```
public static void main(String[] args) {
    m1();
    m2();
    m3();
}
```

nach Ergänzen der Fehlerbehandlungen zu einer unübersichtlichen Konstruktion:

```
public static void main(String[] args) {
    int returncode;

    returncode = m1();
    // Behandlung für diverse m1() - Fehler
    if (returncode == 1) {
        // ...
        System.exit(1);
    }
    // ...

    returncode = m2();
    // Behandlung für diverse m2() - Fehler
    if (returncode == 1) {
        // ...
        System.exit(2);
    }
    // ...

    returncode = m3();
    // Behandlung für diverse m3() - Fehler
    if (returncode == 1) {
        // ...
        System.exit(3);
    }
    // ...
}
```

Mit Hilfe der Ausnahmetechnik bleibt hingegen beim Kernalgorithmus die Übersichtlichkeit erhalten. Wir nehmen nun an, dass die drei Methoden `m1()`, `m2()` und `m3()` durch Ausnahmeobjekte über Fehler informieren:

```
public static void main(String[] args) {
    try {
        m1();
        m2();
        m3();
    } catch (ExA a) {
        // Behandlung von Ausnahmen aus der Klasse ExA
    } catch (ExB b) {
        // Behandlung von Ausnahmen aus der Klasse ExB
    } catch (ExC c) {
        // Behandlung von Ausnahmen aus der Klasse ExC
    }
}
```

Es ist zu beachten, dass z. B. nach der Behandlung einer durch die Methode `m1()` verursachten Ausnahme die weiteren Anweisungen des überwachten **try**-Blocks *nicht* mehr ausgeführt werden.

Das traditionelle Verfahren der Fehlerrückmeldung hat neben dem unübersichtlichen Quellcode noch weitere Nachteile:

- Ungesicherte Beachtung von Rückgabewerten
  Gute gesetzte Rückgabewerte nutzten nichts, wenn sich die Aufrufer nicht darum kümmern.
- Umständliche Weiterleitung von Fehlern
  Wenn ein Fehler nicht an Ort und Stelle behandelt werden soll, muss die Fehlerinformation aufwändig entlang der Aufrufersequenz nach oben gemeldet werden.

Wenn eine Methode per Rückgabewert eine Nutzinformation (z. B. ein Berechnungsergebnis) übermitteln soll, und bei einer ungestörten Methodenausführung *jeder* Wert des Rückgabetyps auftreten kann, dann sind keine Werte als Fehlerindikatoren verfügbar. In diesem Fall verwendet die klassische Fehlersignalisierung einen per Methodenaufruf oder Variable zugänglichen **Fehlerstatus** als Kommunikationsmittel, wobei die Beachtung ebenso wenig garantiert ist wie bei einem Returncode. Auch die Klasse `Simput`, die wir zur Vereinfachung der Werteingabe in zahlreichen Konsolenprogrammen verwendet haben (vgl. Abschnitt 3.4), informiert per Fehlerstatus bei solchen Methoden, die keine Ausnahmen werfen (z. B. `gint()` zum Erfassen eines **int**-Werts). Die Methode `frage()` unserer Demonstrationsklasse `Bruch` (siehe z. B. Abschnitt 1.1.2) verwendet die Methode `Simput.gint()` und überprüft den Erfolg eines Aufrufs über die statische Methode `Simput.checkError()`:

```
do {
   System.out.print("Zähler: ");
   setzeZaehler(Simput.gint());
} while (Simput.checkError());
```

Auch die Methoden der zur Ausgabe in Textdateien geeigneten API-Klasse **PrintWriter** (siehe Abschnitt 14.4.1.5) werfen *keine* **IOException**, sondern setzen ein Fehlersignal, das mit einer Methode namens **checkError**() abgefragt werden kann.

Gegenüber der konventionellen Fehlerbehandlung hat die Kommunikation über Ausnahmeobjekte u.a. folgende Vorteile:

- Garantierte Beachtung von Ausnahmen
  Im Unterschied zu einem Returncode oder einem Fehlerstatus können Ausnahmen nicht ignoriert werden. Ist ein Ausnahmeobjekt (gleich aus welcher Ausnahmeklasse) erst einmal geworfen, muss es behandelt werden. Anderenfalls wird das Programm (genauer: der betroffenen Thread) vom Laufzeitsystem beendet. Allerdings gibt es leider doch eine (zu oft benutzte) Möglichkeit, die Absichten der Java-Designer zu durchkreuzen und Ausnahmen zu ignorieren. Sollte der „Trick“ ausnahmsweise akzeptabel sein, muss der höchst verdächtige Quellcode unbedingt kommentiert werden, z. B. (vgl. Abschnitt 11.2.1.2):

```
try {
   Thread.sleep(3000);
} catch (InterruptedException ignored) {
   // Die Ausnahme darf ignoriert werden, weil der Thread nicht abgebr. wird.
}
```

- Obligatorische Vorbereitung auf Ausnahmen
  In Java wird zwischen der obligatorischen und der freiwilligen Ausnahmebehandlung unterschieden (siehe Abschnitt 11.4.2). Beim Einsatz von Methoden, die obligatorisch zu behandelnde Ausnahmen werfen können, *muss* sich der Aufrufer vorbereiten (z. B. durch eine **try**-Anweisung mit geeignetem **catch**-Block). Unabhängig von der Pflicht zur Vorbereitung, muss jede *geworfene* Ausnahme behandelt werden, um die Beendigung des Programms (genauer: des betroffenen Threads) zu verhindern.[1]
- Automatische Weitermeldung bis zur bestgerüsteten Methode
  Manchmal ist der unmittelbare Verursacher nicht gut gerüstet zur Behandlung einer Ausnahme, z. B. nach dem vergeblichen Öffnen einer Datei. Dann sollte eine „höhere“ Methode über das weitere Vorgehen entscheiden und z. B. beim Benutzer eine alternative Datei erfragen. Oft ist es aber sinnvoll, dass die unmittelbar betroffene Methode ihrem Aufrufer einen besser verständlichen Fehlerbericht liefert.
- Bessere Lesbarkeit des Quellcodes
  Mit Hilfe einer **try-catch-finally** - Anweisung erreicht man eine bessere Trennung zwischen den Anweisungen für den normalen Programmablauf und den diversen Ausnahmebehandlungen, sodass der Quellcode übersichtlich bleibt.
- Umfangreiche Fehlerinformationen für den Aufrufer
  Über ein **Exception**-Objekt kann der Aufrufer beliebig genau über einen aufgetretenen Fehler informiert werden, was bei einem traditionellen Rückgabewert nicht der Fall ist.

Allerdings ist die Fehlermeldung per Rückgabewert oder Fehlerstatus nicht in jedem Fall der moderneren Kommunikation per Ausnahmeobjekt unterlegen. Die Verwendung der traditionellen Technik im Beispielprogramm von Abschnitt 11.2 kann z. B. als akzeptabel gelten. Im weiteren Verlauf von Kapitel 11 wird eine alternative Variante der Methode `convertInput()` zu sehen sein, die ihren Aufrufer durch das Werfen von Ausnahmeobjekten über Probleme informiert. Bei der Entscheidung für eine Technik zur Fehlerkommunikation ist u.a. die Wahrscheinlichkeit für das Auftreten des Fehlers relevant:

- Wenn ein **Problem mit erheblicher Wahrscheinlichkeit** auftritt, sollte eine routinemäßige, aktive Kontrolle stattfinden. Daher sollte eine Methode, die ein solches Problem zu melden hat, davon ausgehen, dass der Aufrufer mit dem Problem rechnet und per Rückgabewert oder Fehlerstatus kommunizieren. Über ein mit erheblicher Wahrscheinlichkeit auftretendes Problem per Ausnahmeobjekt zu informieren, wäre eine unangemessen aufwändige Kommunikationstechnik. Die Ausnahmebehandlung sollte *nicht* zum Bestandteil der Programmablaufsteuerung werden.

[1] Durch eine unbehandelte Ausnahme wird zunächst nur der betroffene Thread beendet. Wenn ein Programm keine Benutzer-Threads mehr besitzt, sondern nur noch sogenannte *Daemon-Threads*, die mit niedriger Priorität im Hintergrund arbeiten und ein Programm *nicht* am Leben erhalten können, dann wird das Programm beendet (siehe Abschnitt 15.8.1). In unseren Konsolenprogrammen ist nur ein Benutzer-Thread vorhanden. Wenn dort ein unbehandelter Ausnahmefehler auftritt, endet das Programm.

- Bei **außergewöhnlichen Problemen (mit einer geringen Auftretenswahrscheinlichkeit**) haben jedoch häufige, meist überflüssige Kontrollen eine Leistungseinbuße zur Folge. Hier sollte man es besser auf eine Ausnahme ankommen lassen. Eine Überwachung über die Ausnahmetechnik verursacht praktisch nur dann Kosten, wenn tatsächlich eine Ausnahme geworfen wird. Diese Kosten sind allerdings deutlich größer als bei einer Fehleridentifikation auf traditionelle Art.

Eine gut entworfene Klasse bietet für kritische (zustandsabhängige) Aktionen eine Methode zur Prüfung der Realisierbarkeit an, sodass scheiternde Aufrufe vermieden werden können. In der Klasse **Scanner**, die sich auch dazu eignet, aus einer Textdatei Werte primitiver Datentypen zu lesen (vgl. Abschnitt 14.5), finden sich z. B. die beiden folgenden Methoden:

- **public double nextDouble()**
  Es wird versucht, aus der Eingabedatei eine abgegrenzte Zeichenfolge zu ermitteln und als **double**-Zahl zu interpretieren. Wenn dies misslingt, wirft die Methode eine Ausnahme.
- **public boolean hasNextDouble()**
  Es wird überprüft, ob das eben beschriebene Unterfangen realisierbar ist.

Weil es bei einem **nextDouble()** - Aufruf leicht zu Problemen kommen kann (Ende der Eingabedatei erreicht, Fehler bei der Interpretation), empfiehlt sich eine vorherige Kontrolle, z. B.:

```
while (input.hasNextDouble()) {
    sum += input.nextDouble();
    n++;
}
```

Zur Unterstützung der funktionalen, strombasierten Programmierung (siehe Kapitel 12) wurde in Java 8 die Klasse **Optional<T>** eingeführt. Ihre Objekte enthalten als Container entweder eine von **null** verschiedene Referenz oder sind leer, wobei die Methode **isPresent()** die Rückgabe **false** liefert. Der Rückgabetyp **Optional<T>** eignet sich für Methoden, die unter Umständen eine angeforderte Rückgabe nicht liefern können, weil ein leerer Container als Rückgabe sinnvoller sein kann als ...

- das Werfen einer Ausnahme
  Das Werfen einer Ausnahme sollte nur unter außergewöhnlichen Umständen passieren.
- die Rückgabe von **null**
  Wenn der Aufrufer nicht mit der **null**-Rückgabe rechnet, kommt es eventuell im weiteren Programmverlauf zu einer **NullPointerException**. Der Rückgabetyp **Optional<T>** signalisiert dem Aufrufer explizit, dass er auch ein leeres Rückgabeobjekt erhalten kann.

Durch eine **Optional<T>** - Rückgabe kann in manchen Situationen das Werfern einer Ausnahme und die Ausnahmebehandlung per **try**-Anweisung vermieden werden. Bei außergewöhnlichen Problemen ist aber weiterhin die Kommunikation per Ausnahmeobjekt überlegen, weil der Aufrufer detaillierter informiert werden kann, als es mit einem **Optional<T>** - Objekt möglich ist.

Im folgenden Beispielprogramm nach Bloch (2018, S. 250) liefert die statische und generische Methode `max()` eine Rückgabe vom Typ **Optional<E>**. Der Aufrufer erhält das maximale Element einer nicht-leeren Kollektion mit komparablem Elementtyp oder ein leeres Objekt, wenn eine die Parameter-Kollektion leer ist:

Quellcode	Ausgabe

<pre>import java.util.*;
class Prog {
  static <E extends Comparable<E>> Optional<E> max(Collection<E> c) {
    if(c == null || c.isEmpty())
      return Optional.empty();
    E result = null;
    for(E e : c)
      if (result == null || e.compareTo(result) > 0)
        result = Objects.requireNonNull(e);
    return Optional.of(result);
  }
  public static void main(String[] args) {
    List<String> los = Arrays.asList("b", "d", "c", "a");
    System.out.println(max(los));
    List<String> elos = Arrays.asList();
    System.out.println(max(elos));
  }
}</pre>
	<pre>Optional[d]
	Optional.empty</pre>

Die statische **Objects**-Methode **requireNonNull()** prüft für eine Referenzvariable, ob das Verweisziel existiert, und wirft bei negativem Prüfungsergebnis eine **NullPointerException**.

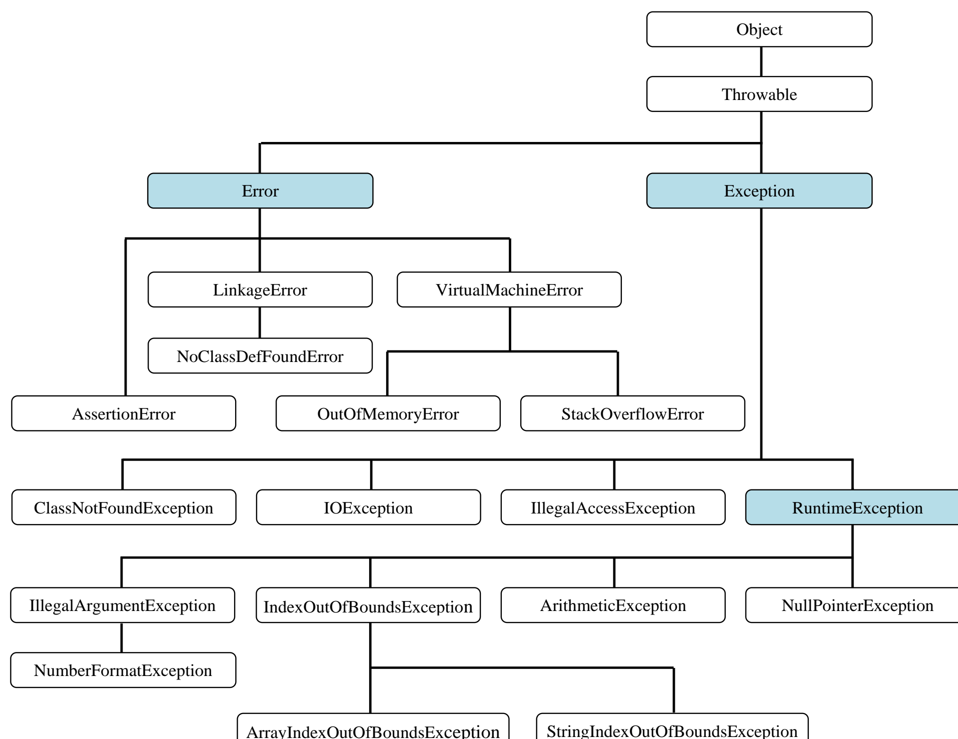
Die **Optional<T>** - Rückgabe ist ...

- für API-Nutzer bequemer (aber auch weniger informativ) als ein Ausnahmeobjekt
- weniger fehleranfällig als die **null** - Rückgabe, weil dem Aufrufer signalisiert wird, dass er mit einem leeren Rückgabeobjekt rechnen muss.

Für primitive Datentypen sind seit Java 8 die analog zu **Optional<T>** arbeitenden Klassen **OptionalInt**, **OptionalDouble** etc. vorhanden.

11.4 Ausnahmen und Fehler

In der folgenden Abbildung sind wichtige, teilweise im weiteren Textverlauf noch anzusprechende Klassen für Ausnahmen und noch üblere Fehler mit ihren Vererbungsbeziehungen zu sehen:



Wo im bisherigen Kursverlauf von *Ausnahmeobjekten* die Rede war, hätte also eigentlich von **Throwable**-Objekten gesprochen werden müssen, um neben den **Exception**-Objekten auch die **Error**-Objekte einzubeziehen. Die Fehler sind allerdings so gravierend, dass sie in der Regel nicht behandelt werden.

In einem **catch**-Block einer **try**-Anweisung können auch *mehrere* Ausnahmesorten durch Wahl einer entsprechend breiten Ausnahmeklasse abgefangen werden.

Sind mehrere **catch**-Blöcke vorhanden, dann werden diese beim Auftreten einer Ausnahme sequentiell von oben nach unten auf Zuständigkeit untersucht, wobei pro Ausnahmeobjekt nur *eine* Behandlung stattfindet. Folglich müssen speziellere Ausnahmeklassen *vor* allgemeineren stehen, was der Compiler sicherstellt.

11.4.1 Error

Wie die obige Klassenhierarchie zeigt, gilt neben der **Exception** auch der **Error** als **Throwable**. Hier geht es gravierende Probleme, vor denen nach laut offizieller API-Dokumentation ein Programm kapitulieren sollte:[1]

> An `Error` is a subclass of `Throwable` that indicates serious problems that a reasonable application should not try to catch.

Typische Beispiele sind:

- **NoClassDefFoundError**
  Kann die JVM eine für den Programmablauf benötigte Klasse nicht finden, meldet sie einen **NoClassDefFoundError**, z. B.:[2]
  ```
  Exception in thread "main" java.lang.NoClassDefFoundError: demopack/A
      at PackDemo.main(packdemo.java:7)
  ```
- **OutOfMemoryError**
  Fordert ein Programm zu viel Heap-Speicher an (z. B. für einen sehr großen Array), dann meldet die JVM einen **OutOfMemoryError**, z. B.:
  ```
  Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
      at Prog.main(Prog.java:8)
  ```
- **StackOverflowError**
  Wird z. B. bei einem fehlerhaften rekursiven Algorithmus die Anzahl der verschachtelten Methodenaufrufe zu groß, meldet die JVM einen **StackOverflowError**, z. B.:
  ```
  Exception in thread "main" java.lang.StackOverflowError
  ```

Es ist durchaus möglich, einen **Error** per **try-catch** - Anweisung abzufangen:

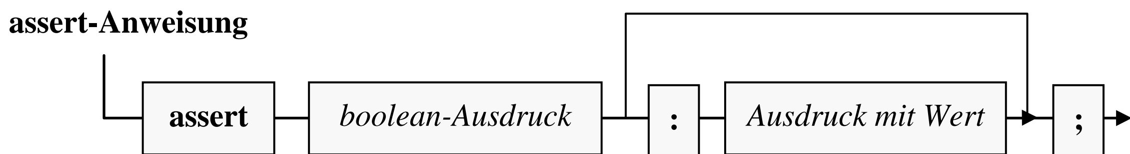
---

[1] https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Error.html

[2] Die von **LinkageError** abstammenden Ausnahmeklasse **NoClassDefFoundError** wird verwendet, wenn eine im Quellcode über Ihren *Namen* angesprochene
```
Katze cat = new Katze();
```
und beim Übersetzen auch vorhandene Klasse zur Laufzeit fehlt. Daneben kennt Java die von **Exception** abstammende Ausnahmeklasse **ClassNotFoundException**. Diese Ausnahme wird von der (abgewerteten) **Class**-Methode **newInstance**() geworfen, wenn eine im Quellcode per *Zeichenfolge* identifizierte Klasse nicht zu finden ist, z. B.:
```
Object obj = Class.forName("Katze").newInstance();
```
Beim Übersetzen wird nicht geprüft, ob zur angegebenen Zeichenfolge eine Klasse existiert.

- Man kann eine Meldung ausgeben oder einen Logeintrag schreiben und die Anwendung anschließend beenden.
- Eventuell ist nur eine irrelevante Funktion des Programms betroffen, und es kann fortgesetzt werden. Eine Anwendung nach einem **Error** fortzusetzen, ist aber riskant und nur akzeptabel, wenn man die Ursache des Fehlers mit sehr hoher Wahrscheinlichkeit kennt. Eventuell kann man die Entscheidung über die Fortsetzung dem Benutzer überlassen.

Einen **Error** zu werfen, sollte der JVM vorbehalten bleiben. Eine Ausnahme stellt die Verwendung der zur Unterstützung der Fehlersuche in Java 1.4 eingeführten **assert**-Anweisung dar, die eine notwendige Bedingung für die reguläre Programmausführung überprüft und bei negativem Ergebnis einen **AssertionError** wirft. Nach dem Schlüsselwort **assert** gibt man einen booleschen Ausdruck mit der Bedingung an. In der Regel lässt man einen Doppelpunkt und eine an den **AssertionError**-Konstruktor zu übergebende Fehlermeldung folgen, wobei ein beliebiger Ausdruck *mit* Wert erlaubt, also ein Methodenaufruf mit Rückgabe **void** verboten ist:



Es folgt ein mäßig sinnvolles Beispiel:

```
class Prog {
    public static void main(String[] args) {
        int arg = Integer.parseInt(args[0]);
        assert arg != 0 : "int-Division durch 0";
        System.out.println(3/arg);
    }
}
```

Zur Laufzeit werden **assert**-Anweisungen per Voreinstellung ignoriert, sodass keine Überprüfungen mit negativem Einfluss auf das Zeitverhalten des Programms stattfinden. Im Beispiel wird also nach dem regulären Programmstart mit

```
>java Prog 0
```

eine **ArithmeticEception** geworfen:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Prog.main(Prog.java:6)
```

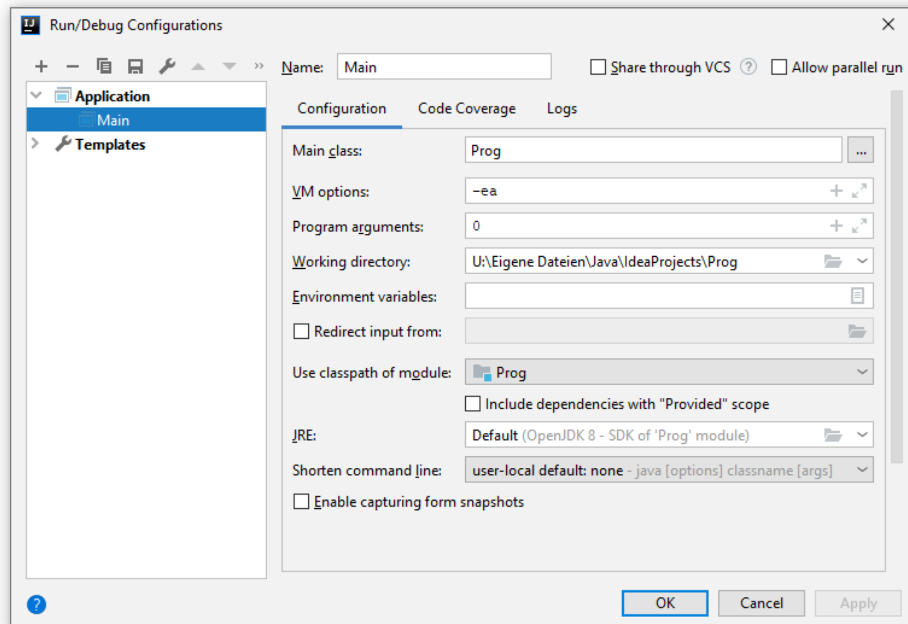
Um die **assert**-Anweisungen zu aktivieren, ist beim Aufruf des Java-Starters der Schalter **-ea** (*enable assertions*) zu setzen. Zur Aktivierung für das unbenannte Paket lässt man auf den Schalter einen Doppelpunkt und dann drei Punkte folgen, z. B.:

```
>java -ea:... Prog 0
```

Im Beispiel resultiert:

```
Exception in thread "main" java.lang.AssertionError: int-Division durch 0
        at Prog.main(Prog.java:5)
```

In IntelliJ ist per Ausführungskonfiguration eine **VM option** zu setzen, um die **assert** - Anweisungen zu aktivieren, z. B.:



Eine eigene Klasse von **Error** abzuleiten, ist möglich, aber nicht empfehlenswert (siehe Bloch 2018, S. 297).

11.4.2 Geprüfte und ungeprüfte Ausnahmen

11.4.2.1 Unterschiedliche Behandlung durch den Compiler

Bei Ausnahmeobjekten aus der Klasse **RuntimeException** und aus daraus abgeleiteten Klassen (siehe die Klassenhierarchie zu Beginn von Abschnitt 11.4) ist es dem Programmierer *freigestellt*, ob er sich auf eine Behandlung vorbereiten möchte. Weil der Compiler *nicht* prüft, ob eine Behandlung erfolgt, spricht man von *ungeprüften Ausnahmen* (engl.: *unchecked exceptions*). Alle übrigen Ausnahmeobjekte (z. B. aus der Klasse **IOException**) *müssen* hingegen behandelt werden. Weil der Compiler dies kontrolliert, spricht man von *geprüften Ausnahmen* (engl.: *checked exceptions*.

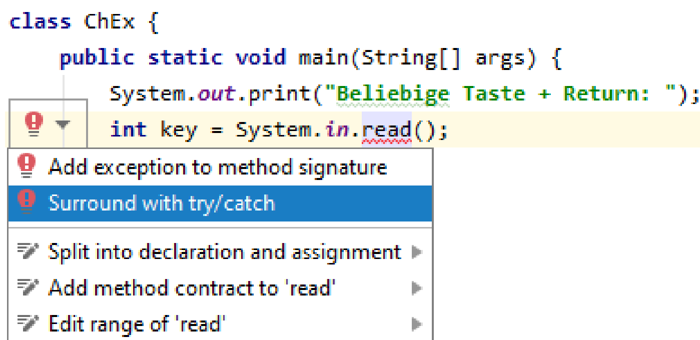
Bei Verwendung einer Methode, die Laufzeitprobleme über geprüfte Ausnahmen meldet, muss der Aufrufer …

- entweder den Aufruf in einer **try**-Anweisung mit geeignetem **catch**-Block vornehmen (vgl. Abschnitt 11.2)
- oder im eigenen Definitionskopf das Weiterreichen der Ausnahmen an den Aufrufer ankündigen (vgl. Abschnitt 11.5.2).

Bei einer Ausnahmeklasse *ohne* Behandlungszwang ist eine solche Vorbereitung nicht erforderlich, aber erlaubt und oft sinnvoll. Eine *geworfene* Ausnahme muss unabhängig von der Klassenzugehörigkeit auf jeden Fall behandelt werden, um die Beendigung des Programms (genauer: des betroffenen Threads) durch das Laufzeitsystem zu verhindern.

Ausnahmeobjekte werden auch in vielen anderen Programmiersprachen unterstützt, wobei aber nur Java zwischen geprüften und ungeprüften Ausnahmen unterscheidet. In anderen Sprachen (z. B. C#, C++, Kotlin, Scala) sind *alle* Ausnahmen vom ungeprüften Typ.

Im folgenden Programm soll mit der Methode **read()** aus der Klasse **InputStream**, zu der auch das Standardeingabe-Objekt **System.in** gehört, ein Zeichen (bzw. ein Byte) von der Tastatur gelesen werden. Weil die von **read()** potentiell zu erwartende **java.io.IOException** behandlungspflichtig ist, muss sie entweder im Kopf der Methode **main()** angekündigt oder in einem **catch**-Block abgefangen werden:



Da wir mittlerweile die **try**-Anweisung beherrschen, ist das Problem leicht zu lösen (mit oder ohne Hilfe der Entwicklungsumgebung):

```
class ChEx {
   public static void main(String[] args) {
      int key = 0;
      System.out.print("Beliebige Taste + Return: ");
      try {
         key = System.in.read();
      } catch(java.io.IOException e) {
         System.out.println("Fehler beim Lesen von der Konsole: " + e);
         System.exit(1);
      }
      System.out.println(key);
   }
}
```

Allerdings ist der Compiler nicht in der Lage, eine *wirksame* Ausnahmebehandlung einzufordern und akzeptiert z. B. auch Exception-Handler mit einem leeren Anweisungsblock, z. B.:

```
try {
   Thread.sleep(3000);
} catch (InterruptedException ignored) {
   // Die Ausnahme darf ignoriert werden, weil der Thread nicht abgebrochen wird.
}
```

Wie im Abschnitt 11.2.1.1 erläutert, ist diese Praxis nur mit einer triftigen Begründung erlaubt, die am besten im Quellcode dokumentiert wird.

11.4.2.2 Eine schwierige Unterscheidung

Die API-Designer haben sich bei der Klassifikation einer Ausnahme als geprüft oder ungeprüft offenbar von der Ursache und vom Ausmaß des Problems leiten lassen. Die Ausnahmen zur Meldung eines **Programmierfehlers** (z. B. **NullPointerException**, **ArrayIndexOutOfBoundsException**) sind *ungeprüft* (mit der Basisklasse **RuntimeException**). Beim Aufruf einer Methode muss man sich nicht darauf vorbereiten, auf einen solchen Fehler zu reagieren. Stattdessen muss der Fehler schleunigst beseitigt werden.

Haben vom Programmierer nicht zu kontrollierende **externe Bedingungen** das Problem verursacht (z. B. Netzwerkstörungen), dann orientiert sich die Entscheidung für oder gegen den Behandlungszwang daran, ob das laufende Programm das Problem lösen kann (siehe auch Ullenboom 2016, Abschnitt 7.5.5):

- **Lösbares Problem**
  Auf ein lösbares Problem wird mit einer geprüften Ausnahme hingewiesen. Wird z. B. mit einer **FileNotFoundException** ein fehlgeschlagener Leseversuch gemeldet, kann das Programm den Benutzer nach einer alternativen Datei fragen.

- **Unlösbares Problem**
  Kann das Problem vom laufenden Programm kaum behoben bzw. kompensiert werden, wird auf den Behandlungszwang verzichtet. Wenn die JVM z. B. eine zur Fortsetzung des Programms benötigte **class**-Datei nicht findet, wirft sie einen **NoClassDefFoundError**, auf den man sich nicht vorbereiten muss.

Wer als Programmierer für die Kommunikation eines Problems die Wahl zwischen vorhandenen Ausnahmeklassen hat oder für eine selbst definierte Ausnahmeklasse (siehe Abschnitt 11.7) eine Basisklasse wählt und damit über die Compiler-Kontrolle entscheidet, sollte sich ebenfalls an Ursache und Schwere des Problems orientieren. Bei Bloch (2018, S. 297) findet sich die folgende Empfehlung:

> To summarize, throw checked exceptions for recoverable conditions and unchecked exceptions for programming errors. When in doubt, throw unchecked exceptions.

Aufgrund der eben referierten Empfehlungen zur Entscheidung zwischen geprüften und ungeprüften Ausnahmen sollte z. B. die von der API-Konvertierungsmethode **Integer.parseInt()** zu erwartende **NumberFormatException** (im Paket **java.lang**) eine *geprüfte* Ausnahme sein, weil z. B. oft eine aus externen Quellen stammende Zeichenfolge konvertiert werden muss (externe Ursache, lösbares Problem). Tatsächlich stammt die Klasse **NumberFormatException** aber von der Klasse **RuntimeException** ab, und der Compiler verlangt daher vom Aufrufer der Methode **Integer.parseInt()** *keine* Ausnahmebehandlung. Hier wird offenbar der Standpunkt vertreten, dass bei einem **parseInt()** - Aufruf mit ungeeignetem Argument ein Fehler des Programmierers vorliegt, der das Argument hätte prüfen müssen.

Die permanenten Schwierigkeiten bei der Differenzierung zwischen geprüften und ungeprüften Ausnahmen mögen der Grund dafür gewesen sein, warum praktisch alle anderen Programmiersprachen auf diese Differenzierung verzichten.

11.5 Ausnahmen in einer eigenen Methode auslösen und ankündigen

11.5.1 Ausnahmen auslösen (throw), ankündigen (throws) und dokumentieren

Unsere eigenen Methoden und Konstruktoren müssen sich nicht auf das Abfangen von Ausnahmen beschränken, die vom Laufzeitsystem oder von Bibliotheksmethoden stammen, sondern sie können sich auch als „Werfer“ betätigen, um bei misslungenen Aufrufen den Absender mit Hilfe der flexiblen **Exception**-Technologie zu informieren.

Insbesondere sollten Methoden und Konstruktoren die übergebenen Parameterwerte routinemäßig prüfen und ggf. die Ausführung durch das Werfen einer Ausnahme abbrechen (siehe Abschnitt 11.6). In der folgenden Variante unseres Beispielprogramms zur Fakultätsberechnung wird in der Methode `convertInput()` ein Ausnahmeobjekt aus der Klasse **IllegalArgumentException** (im Paket **java.lang**) erzeugt, wenn der Aktualparameter entweder nicht interpretierbar ist, oder aber die erfolgreiche Interpretation ein unzulässiges Fakultätsargument ergibt:

```
/**
 * Converts the input string into an int-Value.
 *
 * @throws IllegalArgumentException if argument parsing fails
 *         or value is not in [0 ... 170]
 */
```

```
static int convertInput(String instr) {
   int arg;
   try {
      arg = Integer.parseInt(instr);
      if (arg < 0  || arg > 170)
         throw new IllegalArgumentException(
               "Unzulässiges Argument (erlaubt: 0 bis 170): " + arg);
      else
         return arg;
   } catch (NumberFormatException e) {
      throw new IllegalArgumentException("Fehler beim Konvertieren: " + instr, e);
   }
}
```

Zum Auslösen einer Ausnahme dient die **throw**-Anweisung. Hier ist nach dem Schlüsselwort **throw** eine Referenz auf ein Ausnahmeobjekt anzugeben. Dieses Objekt wird oft per **new**-Operator mit nachfolgendem Konstruktor vor Ort erzeugt (siehe Beispiel).

Die meisten Ausnahmeklassen besitzen u.a. folgende Konstruktoren:

- einen parameterfreien Konstruktor
- einen Konstruktor mit einem **String**-Parameter für eine Fehlermeldung zur näheren Beschreibung der Ausnahme, die im Exception-Handler über die Methode **getMessage()** abgerufen werden kann (vgl. Abschnitt 11.2.3)
- einen Konstruktor mit einem **String**-Parameter für eine Fehlermeldung und einem Verweis auf ein ursprüngliches (inneres) Ausnahmeobjekt, dessen Behandlung zum Erstellen der aktuellen Ausnahme geführt hat (siehe den **NumberFormatException - catch**-Block im Beispiel).

Mit der Fehlerbehebung beauftragte Personen erfahren eventuell nur den Namen der Ausnahmeklasse, die Fehlermeldung und eventuell noch die Aufrufsequenz (siehe Abschnitt 11.2.3). Eine möglichst informative Fehlermeldung mit Details zur Unfallursache (z. B. erlaubte Werte, fehlerhafter Wert) kann daher eine große Hilfe sein (siehe den obigen Quellcode der Methode `convertInput()`).

Viele **catch**-Blöcke betätigen sich als Informationsvermittler und werfen selbst eine Ausnahme, um dem Aufrufer einen leichter verständlichen Unfallbericht zu liefern. Wird wie im folgenden Beispiel

```
throw new IllegalArgumentException("Fehler beim Konvertieren: " + instr, e);
```

in die neue Ausnahme die Adresse der ursprünglichen aufgenommen, kann der Aufrufer über die Methode **getCause()** Ursachenforschung betreiben (siehe unten).

Hat eine Ausnahmehandlung weder zur Lösung geführt, noch zusätzliche Informationen erbracht, kann ein **catch**-Block das ursprüngliche Ausnahmeobjekt erneut werfen, um die Vorgänger in der Aufrufersequenz davon in Kenntnis zu setzen.

Für die Benutzung einer Methode (durch andere Programmierer) ist es außerordentlich wichtig, dass alle von dieser Methode zu erwartenden Ausnahmen dokumentiert werden, was in einem Dokumentationskommentar geschehen sollte (siehe obigen Quellcode der Methode `convertInput()`).

Bei *geprüften* Ausnahmen (z. B. **IOException**) besteht der Compiler darauf, dass diese entweder behandelt oder deklariert werden müssen. Zur Deklaration ist im Definitionskopf eine **throws**-Klausel mit dem Namen der Ausnahmeklasse anzugeben, z. B.:[1]

```
public static BufferedReader newBufferedReader(Path path, Charset cs)
        throws IOException {
    CharsetDecoder decoder = cs.newDecoder();
    Reader reader = new InputStreamReader(newInputStream(path), decoder);
    return new BufferedReader(reader);
}
```

Durch Kommata getrennt können nach dem Schlüsselwort **throws** auch *mehrere* Ausnahmeklassen deklariert werden.

Bei *ungeprüften* Ausnahmen (**RuntimeException** und Unterklassen, siehe Abschnitt 11.4) ist es dem Programmierer freigestellt, ob er die in seiner Methode (direkt oder indirekt) ausgelösten, aber nicht behandelten Ausnahmen deklarieren möchte, z. B.:

```
static int convertInput(String instr) throws IllegalArgumentException { ... }
```

Die freiwillige Deklaration einer Ausnahme hat keinen Behandlungszwang durch den Aufrufer zur Folge.

Es hat sich offenbar die Auffassung etabliert, dass die freiwillige Deklaration von ungeprüften Ausnahmen *keine* gute Praxis sei.[2] Joshua Bloch (2018, S. 304) argumentiert, die freiwillige Deklaration würde den Unterschied zwischen ungeprüften und geprüften Ausnahmen verwischen, und empfiehlt unmissverständlich:

> Use the Javadoc @throws tag to document each unchecked exception that a method can throw, but do *not* use the throws keyword to include unchecked exceptions in the method declaration.

Die Java-Sprachspezifikation (Gosling et al. 2019, Abschnitt 8.4.6) sagt dazu:

> It is permitted but not required to mention unchecked exception classes (§11.1.1) in a `throws` clause.

Der schon im Abschnitt 11.2.1.1 präsentierte Definitionskopf der **Integer**-Methode **parseInt()** enthält eine **throws**-Klausel mit der ungeprüften **NumberFormatException**:

```
public static int parseInt(String s) throws NumberFormatException { ... }
```

Aus dem Auftritt eine Ausnahmeklasse in der **throws**-Klausel einer API-Methode kann man also keinesfalls schließen, dass es sich um eine geprüfte Ausnahme handelt.

In der Dokumentation zu einer Methode sollten auf jeden Fall *alle* von ihr zu erwartenden Ausnahmen erscheinen (geprüft der ungeprüft). So erfahren andere Programmierer, welche Fehler zu einer Ausnahme führen und zu vermeiden sind.

Um auf das nunmehr von `convertInput()` zu erwartende Ausnahmeobjekt aus der Klasse **IllegalArgumentException** reagieren zu können, muss die Methode im Rahmen einer **try**-Anweisung aufgerufen werden, z. B.:

---

[1] Zu sehen ist die statische Methode **newBufferedReader()** der Klasse **Files** (im Paket **java.nio.file**), die wir im Abschnitt 14.4.2.4 verwenden werden.

[2] https://stackoverflow.com/questions/25743574/java-best-practice-for-declaring-unchecked-exception

```
try {
   argument = convertInput(args[0]);
} catch (IllegalArgumentException iae) {
   System.out.println(iae.getMessage());
   if (iae.getCause() != null)
      System.out.println(" Ursache: "+iae.getCause().getMessage());
   System.exit(1);
}
```

Dass eine Methode selbst geworfene Ausnahmen auch wieder auffängt, ist nicht unbedingt der Standardfall, aber in manchen Situationen eine praktische Möglichkeit, von verschiedenen potentiellen Schadstellen aus zur selben Ausnahmebehandlung zu verzweigen. Wir könnten z. B. in der **main()** - Methode unseres Fakultätsprogramms beliebige Argumentprobleme (nicht vorhanden, nicht konvertierbar, außerhalb des legitimes Wertebereichs) zentral behandeln:

```
try {
   if (args.length == 0)
      throw new IllegalArgumentException ("Kein Argument angegeben");
   argument = convertInput(args[0]);
} catch (IllegalArgumentException iae) {
   System.out.println(iae.getMessage());
   if (iae.getCause() != null)
      System.out.println(" Ursache: " + iae.getCause().getMessage());
   System.exit(1);
}
```

Im Zusammenhang mit dem Überschreiben von Instanzmethoden (siehe Abschnitt 7.5.1) ist noch zu beachten, dass eine überschreibende Methode keine geprüfte Ausnahme per **throws**-Klausel ankündigen darf, die breiter ist also eine von das überschriebenen Methode angekündigte geprüfte Ausnahme.

11.5.2 Pflicht zur Ausnahmebehandlung abschieben

Im Abschnitt 11.4.2 haben Sie erfahren, dass man beim Aufruf einer Methode, die potentiell geprüfte Ausnahmen (checked exceptions) wirft, „präventive Maßnahmen“ ergreifen *muss*. In der Regel ist es empfehlenswert, die kritischen Aufrufe in einem **try**-Block vorzunehmen und Ausnahmen in einem **catch**-Block zu behandeln. Es ist aber auch erlaubt, über das Schlüsselwort **throws** im Definitionskopf der aufrufenden Methode die Verantwortung auf den Vorgänger in der Aufrufhierarchie abzuschieben. Im Beispielprogramm aus Abschnitt 11.4.2 kann sich die Methode **main()**, welche den potentiellen **IOException**-Werfer **read()** aufruft, der Pflicht zur Ausnahmebehandlung auf folgende Weise entziehen:

```
class ChEx {
   public static void main(String[] args) throws java.io.IOException {
      System.out.print("Beliebige Taste + Return: ");
      int key = System.in.read();
      System.out.println(key);
   }
}
```

Man kann also mit **throws** nicht nur selbst geworfene Ausnahmen anmelden (siehe Abschnitt 11.5.1), sondern auch von aufgerufenen Methoden stammende Ausnahmen weiterleiten. Im Falle von geprüften Ausnahmen kann man sich so der Behandlungspflicht entledigen.

Unbehandelte Ausnahmen sollten nicht unverändert an den Aufrufer weitergeleitet werden, wenn sie dort nur schlecht zu verstehen sind. Stattdessen sollte man sich in einer eigenen Ausnahmebehandlung als Informationsvermittler bemühen, dem Aufrufer einen leichter verständlichen Unfallbericht zu liefern. Wird in die neue Ausnahme die Adresse der ursprünglichen aufgenommen, kann der Aufrufer über die Methode **getCause()** Ursachenforschung betreiben.

11.5.3 Compiler-Intelligenz beim erneuten Werfen von abgefangenen Ausnahmen

Seit Java 7 bietet der Compiler beim erneuten Werfen einer geprüften Ausnahme durch einen **catch**-Block eine kleine Erleichterung, wenn mehrere Ausnahmetypen im Spiel sind. Eventuell müssen Sie aber zum Lesen der folgenden Erklärung mehr Zeit aufwenden, als Sie jemals durch die beschriebene Technik einsparen können. Im folgenden Beispiel[1] sind von einem **try**-Block zwei Checked Exceptions zu erwarten. Diese werden der Einfachheit halber (zur Vermeidung von Code-Wiederholung) in *einem* **catch**-Block behandelt, der als Ausnahmetyp die Basisklasse **Exception** angibt. Im **catch**-Block wird die abgefangene Ausnahme erneut geworfen:

```
class FirstException extends Exception { }
class SecondException extends Exception { }
  .  .  .
static void rethrowException(String name) throws FirstException, SecondException {
   try {
      if (name.equals("First"))
         throw new FirstException();
      else
         throw new SecondException();
   } catch (Exception e) {
      e.printStackTrace(System.out);
      throw e;
   }
}
```

Eigentlich müsste man daher in der **throws** - Klausel des Methodenkopfes die Ausnahmeklasse **Exception** angeben. Seit Java 7 ist es erlaubt, stattdessen die beiden tatsächlich möglichen Ausnahmetypen anzugeben, sodass der Aufrufer präziser informiert wird. Der Compiler kann durch eine Analyse des **try**-Blocks die Korrektheit der Angaben in der **throws**-Klausel verifizieren.

Ein älterer Java-Compiler würde hingegen den unbehandelten Ausnahmetyp **Exception** reklamieren, und man müsste ...

- entweder im Methodenkopf den Ausnahmetyp **Exception** anmelden, was eine unerwünschte Informationsreduktion zur Folge hätte,
- oder für die beiden Ausnahmetypen jeweils einen separaten **catch**-Block erstellen, was zu einer ebenfalls unerwünschten Code-Wiederholung führen würde.

Seit Java 7 kann man allerdings auch mit dem Multi-Catch - Block (vgl. Abschnitt 11.2.1.1) beide Nachteile vermeiden, wobei der Schreibaufwand im Vergleich zur obigen Lösung nur unwesentlich ansteigt:

```
catch (FirstException | SecondException e) {
   e.printStackTrace(System.out);
   throw e;
}
```

11.6 Ausnahmen bei der Parameter-Validierung

Eine Methode sollte ihre Aktualparameterwerte validieren und auf Fehler mit einer Ausnahme reagieren; dasselbe gilt für einen Konstruktor. Als Ausnahmeklasse eignet sich oft **IllegalArgument-Exception** aus dem API-Paket **java.lang**. Nach Möglichkeit sollte das Problem präziser dargestellt werden, z. B. durch Verwendung der folgenden Ausnahmeklassen:

[1] Übernommen von:
https://docs.oracle.com/javase/8/docs/technotes/guides/language/catch-multiple.html

- **NullPointerException**
  Ein Aktualparameter mit dem Wert **null** ist das Problem.
- **IndexOutOfBoundsException**
  Ein Indexwert liegt außerhalb des zulässigen Bereichs, der z. B. bei Arrays von 0 (inklusive) bis **length** (exklusive) reicht.

Als Beispiel betrachten wir die Methode **get()** in der Klasse **ArrayList<E>** aus dem Java Collections Framework. Sie verwendet zur Indexvalidierung die (seit Java 9 vorhandene) statische Methode **checkIndex()** der Klasse **Objects** (im Paket **java.util**), die bei unpassendem Parameterwert eine **IndexOutOfBoundsException** - Ausnahme wirft:

```
public E get(int index) {
   Objects.checkIndex(index, size);
   return elementData(index);
}
```

In früheren Kursbeispielen haben wir aus didaktischen Gründen die noch unbekannte Ausnahmebehandlung vermieden. So liefert z. B. die Methode `get()` aus der im Abschnitt 8.1.3.1 entwickelten Klasse `SimpleList<E>` bei unpassendem Parameterwert (Elementindex) die Rückgabe **null**:

```
public E get(int index) {
   if (index >= 0 && index < size) {
      // Casting erforderlich, weil kein Array vom Typ E erstellt werden kann.
      // elements kann nur Objekte vom Typ E enthalten.
      @SuppressWarnings("unchecked")
      E result = (E) elements[index];
      return result;
   } else
      return null;
}
```

Im Abschnitt 11.3 wurde der Rückgabetyp **Optional<E>** mit der Möglichkeit, ein leeres Objekt zu liefern, als bessere Alternative im Vergleich zur Rückgabe von **null** oder zum Werfen einer Ausnahme vorgestellt. Begründung:

- Das Werfen einer Ausnahme sollte nur unter außergewöhnlichen Umständen passieren.
- Die Rückgabe von **null** ist fehleranfällig, weil der Aufrufer eventuell nicht damit rechnet, sodass es im weiteren Programmverlauf zu einer **NullPointerException** kommen kann.

11.7 Ausnahmen definieren

Mit Hilfe von Ausnahmeobjekten kann eine Methode beim Auftreten von Laufzeitfehlern die aufrufende Methode präzise über Ursachen und Begleitumstände informieren. Dabei müssen sich Software-Entwickler nicht auf die im Java-API vorhandenen Ausnahmeklassen beschränken, sondern können auch eigene Ausnahmentypen definieren. Das sollte allerdings zurückhaltend geschehen, weil ungewohnte Ausnahmeklassen die Lesbarkeit des Quellcodes erschweren, und im Java-API eigentlich Ausnahmeklassen für fast alle Fälle vorhanden sind. Wir haben im Manuskript schon von etlichen ASPI-Ausnahmeklassen erfahren, z. B.:

- **NullPointerException**
- **IllegalArgumentException**
- **NumberFormatException**
- **IndexOutOfBoundsException**
- **ConcurrentModificationException** (siehe Abschnitt 10.4)
- **UnsupportedOperationException** (siehe Abschnitt 10.2)

Trotzdem wird in diesem Abschnitt eine Eigenentwicklung namens `BadFactorialArg-Exception` vorgeführt, die im Fakultäts-Beispielprogramm zur Fehlerkommunikation verwendet werden kann:

```
public class BadFactorialArgException extends RuntimeException {
   protected int error = -1, value = -1;
   protected String instr;
   public BadFactorialArgException(String message, String instr_,
                                    int error_, int value_) {
      super(message);
      instr = instr_;
      if (error_ == 1 || error_ == 2)
         error = error_;
      if (error_ == 3 && (value_ < 0 || value_ > 170)) {
         error = error_;
         value = value_;
      }
   }

   public String getInstr() {return instr;}
   public int getError() {return error;}
   public int getValue() {return value;}
}
```

Der `BadFactorialArgException()` - Konstruktor verwendet seinen ersten Parameter in einem Aufruf eines Basisklassenkonstruktors, sodass die **String**-Adresse in der von **Throwable** geerbten Instanzvariablen **detailMessage** landet, die als Rückgabewert der ebenfalls von **Throwable** geerbten Methode **getMessage()** dient.

Durch Verwendung der handgestrickten, aus **RuntimeException** abgeleiteten Ausnahmeklasse `BadFactorialArgException` kann die Methode `convertInput()` beim Auftreten von irregulären Argumenten neben einer Fehlermeldung noch weitere Informationen an aufrufende Methoden übergeben:

- in `instr` die zu konvertierende Zeichenfolge (abrufbar mit `getInstr()`)
- in `error` einen numerischen Indikator für die Fehlerart (abrufbar mit `getError()`):
  - -1: unbekannter Fehler
  - 1: **String**-Parameter ist **null**.
  - 2: Fehler beim Konvertieren
  - 3: Unzulässiger Wert (erlaubt: 0 bis 170)
- in `value` das Konvertierungsergebnis (falls vorhanden, sonst -1; abrufbar mit `getValue()`)

Eine eigene Ausnahmeklasse passt gut zur der Strategie, Probleme aus diversen Teilschritten einer Aufgabenbearbeitung an einer Stelle zusammenzuführen, wo gut über das weitere Vorgehen nach einem Scheitern entschieden werden kann. An dieser Entscheidungsstelle muss dann nur *ein* Ausnahmeobjekt behandelt werden, das genügend Informationen über die Art des Problems enthält.

Durch die Wahl der Basisklasse **RuntimeException** ist eine *unchecked exception* entstanden. Im Abschnitt 11.4.2 wurden einige Kriterien für die Entscheidung zwischen einer geprüften und einer ungeprüften Ausnahme genannt. Oft fällt diese Entscheidung schwer, und viele Entwickler entziehen sich der Mühe, indem sie generell ungeprüfte Ausnahmen verwenden. Das kann so falsch nicht sein, weil andere Programmiersprachen (z. B. C#, C++, Kotlin, Scala) ausschließlich ungeprüfte Ausnahmen kennen.

In der finalen Variante der Methode `convertInput()` läuft die Fehlerbehandlung komplett über die `BadFactorialArgException`. Wir halten uns an die Empfehlung von Joshua Bloch (2018,

S. 304), ungeprüfte Ausnahme sorgfältig zu dokumentieren, aber nicht im Methodenkopf zu deklarieren (siehe Abschnitt 11.5.1):

```
/**
 * Converts the input string into an int-Value.
 *
 * @throws BadFactorialArgException if argument is missing, parsing fails
 *         or value is not in [0 ... 170]
 */
static int convertInput(String instr) {
   int arg;
   try {
      if (instr == null)
         throw new BadFactorialArgException("String-Parameter ist null.", "", 1, -1);
      arg = Integer.parseInt(instr);
      if (arg < 0  || arg > 170)
         throw new BadFactorialArgException(
            "Unzulässiger Wert (erlaubt: 0 bis 170)", instr, 3, arg);
      else
         return arg;
   } catch (NumberFormatException e) {
      throw new BadFactorialArgException("Fehler beim Konvertieren", instr, 2, -1);
   }
}
```

Ein `convertInput()` - Aufrufer *muss* sich nicht um die `BadFactorialArgException` kümmern, sollte es aber tun, z. B.:

```
public static void main(String[] args) {
   int argument = -1;
   try {
      argument = convertInput(args[0]);
      double fakul = 1.0;
      for (int i = 1; i <= argument; i++)
         fakul = fakul * i;
      System.out.println("Fakultät: " + fakul);
   } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Kein Argument angegeben");
   } catch (BadFactorialArgException e) {
      System.out.println("Fehlerhaftes Argument: " + e.getMessage());
      switch (e.getError()) {
         case 2 :   System.out.println("Zeichenfolge: \"" + e.getInstr() + "\"");
                 break;
         case 3 :   System.out.println("Wert: " + e.getValue());
                 break;
      }
   }
}
```

Im Beispiel kann der Kernalgorithmus ungestört durch Fehlerbehandlungen formuliert werden

```
argument = convertInput(args[0]);
double fakul = 1.0;
for (int i = 1; i <= argument; i++)
  fakul = fakul * i;
System.out.println("Fakultät: " + fakul);
```

Die meisten potentiellen Ausnahmefehler sind berücksichtigt, und der Benutzer wird jeweils gut informiert.

11.8 Freigabe von Ressourcen

Von einem Programm belegte externe Ressourcen wie Datei-, Netzwerk- oder Datenbankverbindungen müssen möglichst früh wieder freigegeben werden, um den Benutzer und andere Programme möglichst wenig zu behindern. Außerdem muss sichergestellt werden, dass die Freigabe unter allen Umständen erfolgt, insbesondere auch nach einem Ausnahmefehler.

11.8.1 Traditionelle Lösung per finally-Block

Vor Java 7 war der **finally**-Block einer **try-catch-finally** - Anweisung der ideale Ort zur Freigabe von Ressourcen wie Datei-, Netzwerk- oder Datenbankverbindungen. Seit Java 7 bietet eine spezielle **try**-Variante eine bequemere und zuverlässigere Lösung. Um den Fortschritt deutlich zu machen, betrachten wir zuerst die traditionelle, in vorhandenem Code noch oft anzutreffende Dateifreigabe per **finally**-Block mit **close()** - Aufruf. Im folgenden Beispiel wird (teilweise dem Kapitel 14 vorgreifend) zur Demonstration der traditionellen Dateifreigabe eine statische Methode namens `mean()` definiert, die mit Hilfe eines **DataInputStream**-Objekts aus einer Binärdatei dort erwartete **double**-Zahlen liest und den Mittelwert daraus berechnet:

```
import java.io.*;

class FinallyClose {
   static void mean(String eingabe) {
      DataInputStream dis = null;
      try {
         dis = new DataInputStream(new FileInputStream(eingabe));
         double sum = 0.0;
         int n = 0;
         while (dis.available() > 0) {
            n++;
            sum += dis.readDouble();
         }
         System.out.println("Mittelwert zur Datei " + eingabe + ":  " + sum/n);
      } catch (IOException ioe) {
         ioe.printStackTrace();
      } finally {
         if (dis != null)
            try {dis.close();} catch (IOException ioc) {
               ioc.printStackTrace();
            };
      }
   }

   public static void main(String args[]) {
      mean("eingabe.dat");
   }
}
```

Bei Beendigung einer Anwendung werden alle von ihr geöffneten Dateien automatisch geschlossen, sodass im obigen Beispiel das Bemühen um das frühe Schließen (kurz vor dem Programmende) eigentlich irrelevant ist. Oft bleiben Programme aber deutlich länger aktiv. Anwender sind irritiert und verärgert, wenn sich z. B. eine Datei mit den Mitteln des Betriebssystems nicht umbenennen oder löschen lässt, weil sie vor geraumer Zeit mit einem Programm bearbeitet wurde, das noch aktiv ist und die Datei ohne Grund weiterhin blockiert.

Methoden zur Dateibearbeitung müssen in der Regel in einer **try**-Anweisung mit passendem **catch**-Block aufgerufen werden, weil sie über Ausnahmeobjekte aus der Klasse **IOException** (im Paket **java.io**) oder aus einer daraus abgeleiteten Klasse (z. B. **FileNotFoundException**, **EOFException**) kommunizieren, auf die sich ein Aufrufer obligatorisch vorbereiten muss (vgl. Abschnitt 11.4.2). Im Beispiel sind der **FileInputStream**-Konstruktor und die **DataInputStream**-Methode **readDouble()**

betroffen. Es könnte z. B. passieren, dass sich die Eingabedatei öffnen lässt, aber später beim Lesen eine **IOException** auftritt.

Im Beispiel wird der gesamte Algorithmus in einem **try**-Block ausgeführt. Damit das möglichst frühe Schließen der Datei auch im Ausnahmefall (z. B. **EOFException** beim Lesen) sichergestellt ist, findet der erforderliche **close()** - Aufruf im **finally**-Block der **try**-Anweisung statt. Stünde er z. B. am Ende des **try**-Blocks, bliebe die Datei im Ausnahmefall bis zu einem Garbage Collector - Einsatz oder bis zum Programmende geöffnet.[1]

Ist bereits das Öffnen der Datei im **FileInputStream**-Konstruktor misslungen, existieren keine zu schließende Datei und kein Adressat für den **close()** - Aufruf. Das Programm unterlässt den Fehlversuch, der eine **NullPointerException** zur Folge hätte.

Weil auch die **close()** - Methode eine **IOException** werfen kann, und Ausnahmeobjekte aus dieser Klasse entweder behandelt oder angemeldet werden müssen, findet der **close()** - Aufruf in einer **try-catch** - Anweisung stattfinden, und es resultiert eine **try**-Verschachtelung.

Die in einem **finally**-Block (im Beispiel: beim **close()** - Aufruf) möglichen Ausnahmen müssen vor Ort abgefangen werden, weil ansonsten eine zuvor im Hauptalgorithmus der Methode aufgetretene und an den Aufrufer zu übermittelnde unbehandelte Ausnahme verdeckt würde. Weil an den Aufrufer nur *eine* Ausnahme gemeldet werden kann, würde er nur von der Sekundär-Ausnahme aus dem **finally**-Block erfahren, aber nicht von der primären Ursache des Problems.

Neben dem nicht ganz unerheblichen Aufwand besteht ein weiterer Nachteil der traditionellen Lösung darin, dass die **DataInputStream**-Variable nicht im **try**-Block deklariert werden kann, weil sie sonst im **finally**-Block unbekannt wäre. In dem allgemeineren, umgebenden Block ist sie aber einem leicht erhöhten Fehlerrisiko ausgesetzt.

11.8.2 Try With Resources

Seit Java 7 lässt sich das Schließen der in einem **try**-Block benötigten Ressourcen automatisieren, sofern die Klassen, welche die Ressourcen repräsentieren, das Interface **AutoCloseable** im Paket **java.lang** implementieren. Um diese sehr empfehlenswerte Option zu nutzen, erzeugt man ein automatisch zu schließendes Objekt in einem Ausdruck, der durch runde Klammern begrenzt zwischen das Schlüsselwort **try** und den überwachten Block gesetzt wird. Das Beispiel aus dem letzten Abschnitt kann so erheblich vereinfacht werden:

```
import java.io.*;

class TryWithResources {
   static void mean(String eingabe) {
      try (DataInputStream dis = new DataInputStream(new FileInputStream(eingabe))) {
         double sum = 0.0;
         int n = 0;
         while (dis.available() > 0) {
            n++;
            sum += dis.readDouble();
         }
         System.out.println("Mittelwert zur Datei " + eingabe + ":  " + sum/n);
      } catch (IOException ioe) {
         ioe.printStackTrace();
      }
   }
```

[1] In der Klasse **FileInputStream** ist eine **finalize()** - Methode definiert, die ggf. vom Garbage Collector aufgerufen wird und für das Schließen der Datei sorgt.

```
    public static void main(String args[]) {
      mean("eingabe.dat");
    }
  }
```

Die **finally**-Klausel mit der **close()** - Anweisung ist überflüssig geworden, und die **DataInput-Stream**-Variable ist nur im **try**-Block sichtbar.

Für einen **try**-Block lässt sich auch eine *mehrelementige* Ressourcenliste definieren, wobei zwischen zwei Elemente ein Semikolon zu setzen ist.

```
try (DataInputStream dis = new DataInputStream(new FileInputStream(eingabe));
    DataOutputStream dos = new DataOutputStream(new FileOutputStream(ausgabe))) {
  . . .
}
```

Bis Java 8 ist in einem **try-with-resources** - Block eine außerhalb definierte Ressource nur über eine redundante lokale Variable zu verwenden, z. B.:

```
static void mean(String eingabe) throws FileNotFoundException {
    DataInputStream diso = new DataInputStream(new FileInputStream(eingabe));
    try (DataInputStream dis = diso) {
        double sum = 0.0;
        int n = 0;
        while (dis.available() > 0) {
            n++;
            sum += dis.readDouble();
        }
        System.out.println("Mittelwert: " + sum/n);
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
```

Seit Java 9 kann die **try**-lokale Variable entfallen, sofern die Referenzvariable zur außerhalb definierten Ressource effektiv final ist:

```
static void mean(String eingabe) throws FileNotFoundException {
    DataInputStream dis = new DataInputStream(new FileInputStream(eingabe));
    try (dis) {
        double sum = 0.0;
        int n = 0;
        while (dis.available() > 0) {
            n++;
            sum += dis.readDouble();
        }
        System.out.println("Mittelwert: " + sum/n);
    } catch (IOException ioe) {ioe.printStackTrace();}
}
```

Die effektiv finale Ressourcen-Referenzvariable kann auch als Aktualparameter angeliefert werden:

```
static void mean(DataInputStream dis) {
    try (dis) {
        double sum = 0.0;
        int n = 0;
        while (dis.available() > 0) {
            n++;
            sum += dis.readDouble();
        }
        System.out.println("Mittelwert: " + sum/n);
    } catch (IOException ioe) {ioe.printStackTrace();}
}
```

11.9 Übungsaufgaben zum Kapitel 11

1) Welche der folgenden Aussagen sind richtig bzw. falsch?

1. Eine Ausnahme aus der Klasse **RuntimeException** muss nicht behandelt werden.
2. In einem **catch**-Block kann das abgefangene Ausnahmeobjekt erneut geworfen werden.
3. Nach der ausnahmslos erfolgreichen Ausführung eines **try**-Blocks, wird die Methode hinter der **try-catch-finally** - Anweisung fortgesetzt.
4. In einem **catch**- oder **finally**-Block sind Methoden, die Ausnahmen werfen können, verboten.
5. Es ist auch eine **try-finally** - Anweisung (ohne **catch**-Block) erlaubt.

2) Erstellen Sie ein Syntaxdiagramm zur **try-catch-finally** - Anweisung (vgl. Abschnitt 11.2.1). Die im Abschnitt 11.8.2 vorgestellte **try**-Variante mit automatisierter Ressourcen-Freigabe muss dabei *nicht* berücksichtigt werden.

3) Erstellen Sie ausnahmsweise ein Programm, das eine **NullPointerException** auslöst, indem es auf ein nicht existentes Objekt zugreift.

4) Beim Rechnen mit Gleitkommazahlen produziert Java in kritischen Situationen üblicherweise *keine* Ausnahmen, sondern operiert mit speziellen Werten wie **Double.POSITIVE_INFINITY** oder **Double.NaN**. Dieses Verhalten ist sicher oft nützlich, kann aber eventuell die Fehlersuche erschweren, wenn mit den speziellen Funktionswerten weitergerechnet wird, und am Ende eines längeren Rechenwegs das Ergebnis **Double.NaN** steht. Im folgenden Beispiel wird eine Methode namens `duaLog()` zur Berechnung des dualen Logarithmus (Logarithmus zur Basis 2) verwendet, welche auf die statische Methode **log()** der Klasse **Math** im Paket **java.lang** zurückgreift und bei ungeeigneten Argumenten (≤ 0) als Rückgabewert **Double.NaN** liefert.[1]

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
public class DuaLog {
   final static double LOG2 = Math.log(2);

   public static double duaLog(double arg) {
      return Math.log(arg) / LOG2;
   }

   public static void main(String[] args) {
      double a = duaLog(8);
      double b = duaLog(-1);
      System.out.println(a*b);
   }
}
```

</td><td>

```
NaN
```

</td></tr>
</table>

Erstellen Sie eine Variante, die bei ungeeigneten Argumenten eine **IllegalArgumentException** wirft.

[1] Für positive Zahlen *a* und *b* ist der Logarithmus von *a* zur Basis *b* definiert durch:

$$\log_b(a) := \frac{\log(a)}{\log(b)}$$

Dabei steht log() für den natürlichen Logarithmus zur Basis e (Eulersche Zahl).

12 Funktionales Programmieren

Java 8 hat als wesentlichen Fortschritt die Unterstützung der *funktionalen Programmierung* gebracht, was Horstmann (2014b) so formuliert:

> The principal enhancement in Java 8 is the addition of functional programming constructs to its object-oriented roots.

Als wesentliche Konstrukte zur Unterstützung der funktionalen Programmierung wurden in Java 8 eingeführt:

- **Lambda-Ausdrücke**
  Ein Lambda-Ausdruck ist ein Stück Code (bestehend aus einem einzelnen Ausdruck oder aus einem Anweisungsblock) zusammen mit den vom Code erwarteten Parametern, also letztlich eine Methode. Es wird sich noch zeigen, dass tatsächlich ein *Objekt* im Spiel ist, das die Methode ausführt (siehe Abschnitt 12.1.1.3). Man kann den Lambda-Ausdruck aber auch als *Funktion* bezeichnen. Er wird z. B. an eine andere Methode zur Ausführung übergeben, um deren Verhalten zu komplettieren oder zu konfigurieren. Im folgenden Beispiel erhält eine Methode namens **filter()** einen Aktualparameter vom Interface-Datentyp **Predicate<String>** (siehe Abschnitt 12.1.1.1):

  ```
  filter(s -> s.length() == 4)
  ```

  Der Lambda-Ausdruck empfängt einen Parameter vom Typ **String** und liefert eine Rückgabe vom Typ **boolean**, die genau dann **true** ist, wenn die Parameterzeichenfolge die Länge vier besitzt. Was die Methode **filter()** unter Verwendung des Lambda-Funktionsobjekts tut, ist gleich anschließend zu sehen.

  Bei den Lambda-Ausdrücken handelt es sich um **Erweiterung der Programmiersprache**, die den Umgang mit Funktionsobjekten erleichtert.

- **Ströme**
  Ein Strom ist eine Sequenz von Elementen aus einer Quelle (z. B. Kollektion, Array, Datei) und unterstützt Operationen zur sequentiellen oder parallelen Massenabfertigung der Elemente (engl.: *bulk operations* oder *aggregate operations*). Ein wesentliches, aber nicht das einzige Ziel beim Design der Stromverarbeitung in Java 8 war die bequeme (und damit *tatsächlich genutzte*) Parallelisierung von Operationen bei Sequenzen mit dem Ergebnis guter Leistungswerte auf Multi-Core - Systemen.

  Bei den Java 8 - Strömen handelt es sich um eine **Erweiterung der Standardbibliothek**, die Datenbank-artige Operationen (z. B. Filtern, Gruppieren, Auswerten) mit Kollektionsobjekten erleichtert. Im folgenden Beispiel entsteht aus einem **List<String>** - Objekt durch einen Aufruf seiner **stream()** - Methode ein **Stream<String>** - Objekt mit einer Sequenz von Namen. Um die Anzahl der Namen mit vier Zeichen zu ermitteln, werden per **filter()** die Namen mit einer abweichenden Länge ausgefiltert. Dann wird per **count()** die Anzahl der verbliebenen Stromelemente ermittelt:

  ```
  List<String> als = Arrays.asList("Rudolf", "Emma", "Otto", "Kurt","Walter");
  long n4 = als.stream()
              .filter(s -> s.length() == 4)
              .count();
  ```

12.1 Lambda-Ausdrücke

Wer zu einem Buch mit Unterstützung für Java 8 greift, wird mit einer bis Java 7 völlig ungewohnten Syntax wie im folgenden Aufruf der Methode **setOnAction()** konfrontiert:

```
final Button button = new Button("Klick");
  . . .
button.setOnAction(event -> label.setText("Hallo JavaFX"));
```

Hier wird die Klickereignisbehandlung für einen Befehlsschalter mit Hilfe eines Lambda-Ausdrucks realisiert. Wie das Beispiel zeigt, werden Lambda-Ausdrücke nicht nur im Rahmen von Java 8 - Strömen verwendet. Sie sind für diverse Aufgaben so attraktiv und mittlerweile so verbreitet, dass wir uns mit diesem Ausdrucksmittel beschäftigen müssen.

12.1.1 Traditionelle und moderne Realisation von Funktionsobjekten

Ein weiteres Zitat von Horstmann (2014b) ist dazu geeignet, den Nutzen und die Bedeutung von Lambda-Ausdrücken zu umreißen:

> The single most important change in Java 8 enables faster, clearer coding and opens the door to functional programming.

Wir beschreiben anschließend einige typische Aufgabenstellungen, deren traditionelle Lösung und die seit Java 8 mögliche Lösungsalternative mit Lambda-Ausdrücken.

12.1.1.1 Funktionale Schnittstellen

In Java ist es oft erforderlich, eine Methode zu erstellen, die zu bestimmten Gelegenheiten aufgerufen werden soll und daher bestimmte Voraussetzungen erfüllen muss, z. B.:

- Eine Ereignisbehandlungsmethode soll ausgeführt werden, wenn der Benutzer eines Programms mit JavaFX-Bedienoberfläche auf eine Schaltfläche klickt. Dazu definiert man eine Klasse, die das Interface **EventHandler<ActionEvent>** erfüllt, also eine entsprechende Methode

  public void handle(ActionEvent *event***)**

  besitzt, erzeugt ein Objekt aus dieser Klasse und registriert es bei der Schaltfläche.[1]
- Eine Methode soll in einem separaten Thread ausgeführt werden. Dazu definiert man eine Klasse, die das Interface **Runnable** erfüllt, also eine entsprechende Methode

  public void run()

  besitzt, erzeugt ein Objekt aus dieser Klasse und übergibt es z. B. an den Konstruktor der Klasse **Thread**.
- Eine Methode soll zum Vergleich von Objekten eines bestimmten Typs herangezogen werden. Dazu definiert man eine Klasse, die das passend parametrisierte Interface **Comparator<T>** erfüllt, also eine entsprechende Methode

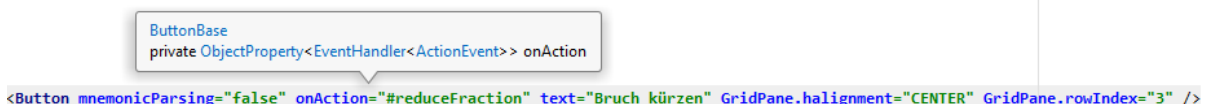
  public int compare(T *o1***, T** *o2***)**

  besitzt, und übergibt ein Objekt aus dieser Klasse z. B. an eine Methode zum Sortieren eines Arrays mit Elementen vom Typ **T**.

In allen Beispielen ist ein Interface beteiligt, das genau *eine* abstrakte Methode enthält, z. B.:

```
public interface EventHandler<ActionEvent> extends EventListener {
   void handle(ActionEvent event);
}
```

[1] Wird die JavaFX-Bedienoberfläche deklarativ per FXML-Datei gestaltet, passiert einiges im Verborgenen. Wie die IntelliJ-Erläuterung zum Attribut **onAction** im Element **Button** aus der FXML-Datei zu unserem Beispielprogramm im Abschnitt 4.9 zeigt, ändert sich nichts an der Grundlogik der Ereignisbehandlung:



Seit Java 8 spricht man hier von einem *funktionalen Interface*, weil es bei der funktionalen Programmierung eine zentrale Rolle spielt. Neben der einen abstrakten Methode können beliebig viele Instanzmethoden mit **default**-Implementierung sowie statische Methoden vorhanden sein (vgl. Abschnitt 9.2.3). Wenn ein Interface eine abstrakte und öffentliche Methode deklariert, die eine öffentliche Methode von **java.lang.Object** überschreibt, ist diese Methode irrelevant für das Kriterium für funktionale Schnittstellen, weil jede Klasse eine Implementation für diese Methode besitzt (von **Object** oder einer anderen Klasse geerbt). Ein Beispiel für die beschriebene Konstellation ist die Schnittstelle **Comparator<T>**, welche die **Object**-Methode **equals()** explizit fordert:[1]

```
@FunctionalInterface
public interface Comparator<T> {
   int compare(T o1, T o2);
   boolean equals(Object obj);
    . . .
}
```

In Java 8 wurde die Standardbibliothek um das Paket **java.util.function** erweitert, das über 40 funktionale Schnittstellen enthält.

Um dem Compiler für ein als funktional konzipiertes Interface die Kontrolle der eben beschriebenen Regel (genau eine abstrakte Methode) zu ermöglichen, kann man der Definition die Marker-Annotation **@FunctionalInterface** voranstellen, was in der Java SE - Standardbibliothek regelmäßig geschieht, z. B. beim generischen Interface **Predicate<T>** im Paket **java.util.function**, das eine abstrakte Methode namens **test()** mit einem Parameter vom Typ **T** und einer Rückgabe vom Typ **boolean** verlangt:

```
package java.util.function;

import java.util.Objects;

@FunctionalInterface
public interface Predicate<T> {

   boolean test(T t);

   default Predicate<T> and(Predicate<? super T> other) {
      Objects.requireNonNull(other);
      return (t) -> test(t) && other.test(t);
   }

   default Predicate<T> negate() {
      return (t) -> !test(t);
   }

   default Predicate<T> or(Predicate<? super T> other) {
      Objects.requireNonNull(other);
      return (t) -> test(t) || other.test(t);
   }
```

[1] Man hat die implizit ohnehin in jeder Schnittstellen-Implementation vorhandene Methode **equals()** deshalb explizit deklariert, um das erwartete Verhalten in der API-Dokumentation beschreiben zu können. Dort ist lesen, dass die **equals()** - Methode so überschrieben werden kann, dass sie den Wert **true** liefert für eine per Parameter bestimmte **Comparator<T>** - Implementation, die im folgenden Sinne mit der angesprochenen Implementation kompatibel ist: Die **compare(T** *o1*, **T** *o2*) - Methoden der beiden Implementationen liefern für alle Paare von Objekten des Typs **T** eine übereinstimmende Beurteilung (dasselbe Vorzeichen oder den übereinstimmenden Wert 0). Die **equals()** - Methode der Klasse **Object** liefert nur dann den Wert **true**, wenn das Parameterobjekt mit dem angesprochenen Objekt identisch ist. Die Dokumentation einer Klasse oder Schnittstellen ist als Bestandteil des Vertrags zu diesem Typ aufzufassen.

```
    static <T> Predicate<T> isEqual(Object targetRef) {
       return (null == targetRef)
             ? Objects::isNull
             : object -> targetRef.equals(object);
    }
 }
```

Bei älteren Standardbibliothekschnittstellen nach dem funktionalen Muster wurde darauf *verzichtet*, die Annotation **@FunctionalInterface** nachträglich einzufügen. Trotzdem sind Lambda-Ausdrücke auch mit diesen Schnittstellen kompatibel.

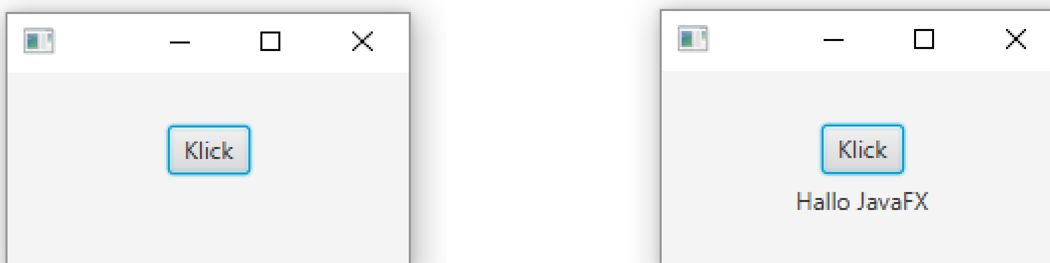
Ein Objekt, das eine funktionale Schnittstelle implementiert, wird auch als *Funktionsobjekt* bezeichnet. Durch die Übergabe eines solchen Objekts als Methoden- oder Konstruktorparameter wird eine Funktionalität in eine Methode oder Klasse injiziert. Dabei wird das sogenannte *Strategie-Entwurfsmuster* realisiert.[1]

Im zeitgemäßen Java-Programmierstil wird diese Option zur Verhaltenskonfiguration oft als Alternative zur Definition von abgeleiteten Klassen verwendet (siehe z. B. Bloch 2018, S. 199). Die Funktionalität einer Klasse kann modifiziert werden, indem einem Feld mit dem Datentyp einer funktionalen Schnittstelle ein Funktionsobjekt zugewiesen wird. Allerdings wird die Vererbung, die ja eine von drei Kernmerkmalen der objektorientierten Programmierung ist (vgl. Abschnitt 4.1.1), von den Funktionsobjekten nicht überflüssig gemacht. Die Definition einer abgeleiteten Klasse ist z. B. empfehlenswert, um eine Basisklasse um zusätzliche Member zu erweitern.

In den folgenden Abschnitten werden Optionen zur Definition von Funktionsobjekten beschrieben.

12.1.1.2 Anonyme Klassen

Die den Beispielen von Abschnitt 12.1.1.1 gemeinsame Aufgabenstellung, Funktionalität an Methoden oder Klassen zu übergeben, wird in Java bis zur Version 7 häufig mit Hilfe von sogenannten *anonymen Klassen* realisiert. Dabei wird die benötigte Funktionalität an Ort und Stelle realisiert, ohne eine anderenorts (eventuell sogar in einer eigenen Datei) definierte Klasse zu benötigen. Anonyme Klassen werden u.a. bei der Ereignisbehandlung in GUI-Programmen verwendet. Das gilt auch für JavaFX, was beim Verzicht auf die deklarativen Oberflächengestaltung per FXML besonders deutlich wird. Im folgenden Programm wird per Mausklick ein Text zur Anzeige gebracht:



Wie ein Blick in den Quellcode des Programms zeigt, wird seine Bedienoberfläche komplett durch Anweisungen erzeugt, was in JavaFX nach wie möglich und bei anderen GUI-Techniken (z. B. Swing) alternativlos ist:

[1] https://de.wikipedia.org/wiki/Strategie_(Entwurfsmuster)

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
 . . .
import javafx.scene.layout.VBox;

public class EventHandlerAnon extends Application {
   @Override
   public void start(Stage primaryStage) {
     final Label label = new Label();
     final Button button = new Button("Klick");

     button.setOnAction(new EventHandler< ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
           label.setText("Hallo JavaFX");
        }
     });

     final VBox root = new VBox(5);
     root.setAlignment(Pos.CENTER);
     root.getChildren().addAll(button, label);
     final Scene scene = new Scene(root, 200, 100);
     primaryStage.setScene(scene);
     primaryStage.show();
   }

   public static void main(String[] args) {
      launch(args);
   }
}
```

Mit dem Aufbau dieses Quellcodes werden wir uns im Kapitel 13 näher beschäftigen. Momentan konzentrieren wir uns darauf, wie für den Befehlsschalter (ein Objekt aus der Klasse **Button**) die Klickbehandlung realisiert wird:

```
button.setOnAction(new EventHandler<ActionEvent>() {
   @Override
   public void handle(ActionEvent event) {
      label.setText("Hallo JavaFX");
   }
});
```

Durch einen Aufruf der **Button**-Methode **setOnAction()** wird zur Behandlung von Klickereignissen ein Objekt vereinbart, dessen Klasse die Schnittstelle **EventHandler<ActionEvent>** erfüllt. Dabei wird nicht nur das Objekt dynamisch erzeugt, sondern eine komplette Klasse an Ort und Stelle definiert. Einen Namen erhält die nur lokal benötigte Klasse nicht, und es resultiert eine sogenannte **anonyme Klasse**.[1]

Seit Java 9 kann der sogenannte Diamond Operator (siehe Abschnitt 8.1.1.2) auch bei anonymen Klassen verwendet werden, sodass sich das letzte Quellcodesegment leicht vereinfachen lässt:

[1] Man kann die anonyme Klasse als spezielle *lokale Klasse* betrachten (vgl. Abschnitt 4.8.2). Dieser Standpunkt wird auch im Java-Tutorial der Firma Oracle vertreten, siehe
https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

```
button.setOnAction(new EventHandler<>() {
   @Override
   public void handle(ActionEvent event) {
      label.setText("Hallo JavaFX");
   }
});
```

Eine anonyme Klasse kann in verschiedenen Umgebungen definiert werden, z. B.:

- In einer Instanzmethode oder in einem Konstruktor
- In einer statischen Methode oder in einem statischen Initialisierer
- Bei der Initialisierung eines (statischen) Felds.

Die anonyme Klasse kann als Aktualparameter verwendet oder einer Variablen zugewiesen werden. Sie hat Zugriff auf Variablen und Methoden aus dem jeweiligen Kontext. Im folgenden Programm, das wegen der Nutzung des Diamond-Operators mindestens Java 9 voraussetzt, werden drei Umgebungsvarianten und die jeweils möglichen Zugriffe auf Kontextvariablen demonstriert:

```
import java.util.function.ToIntFunction;

class Umgebungen {
    static java.util.List<String> ls = java.util.Arrays.asList("1", "22", "333");
    static String statEnv = "Stat";
    String instEnv = "Inst";

    static ToIntFunction<String> stoiAnonS = new ToIntFunction<>() {
        @Override
        public int applyAsInt(String s) {
            return (statEnv+s).length();
        }
    };

    void instMeth() {
        String methEnv = "Meth";

        ToIntFunction<String> stoiAnonIM = new ToIntFunction<>() {
            @Override
            public int applyAsInt(String s) {
                return (statEnv+instEnv+methEnv+s).length();
            }
        };

        System.out.println("\nAnon. Kl. in umgebender Instanzmethode: "+
                ls.stream().mapToInt(stoiAnonIM).sum());
    }

    public static void main(String[] args) {
        String methEnv = "Meth";

        ToIntFunction<String> stoiAnonSM = new ToIntFunction<>() {
            @Override
            public int applyAsInt(String s) {
                return (statEnv+methEnv+s).length();
            }
        };

        System.out.println("Anon Kl. mit statischem Kontext:        "+
                ls.stream().mapToInt(stoiAnonS).sum());
        System.out.println("\nAnon. Kl. in umgebender stat. Methode:  "+
                ls.stream().mapToInt(stoiAnonSM).sum());
        Umgebungen u = new Umgebungen();
        u.instMeth();
    }
}
```

Die Funktionsobjekte aus den anonymen Klassen werden jeweils einer Variablen vom Typ **ToIntFunction<String>** zugewiesen, die als Parameter für die **Stream<String>** - Methode

mapToInt() dient, die wir im Abschnitt 12.2 als intermediäre Operation für Ströme kennenlernen werden. Um praxisrelevante Anwendungsfälle für anonyme Klassen und Lambda-Ausdrücke zu erhalten, verwenden wir im aktuellen Abschnitt oft in möglichst einfacher Form die im Abschnitt 12.2 vorzustellenden **Stream**-Typen. Im Augenblick interessiert am Beispielprogramm ausschließlich, dass die anonymen Klassen auf Variablen in ihrer jeweiligen Umgebung zugreifen dürfen.

Wichtige Eigenschaften von anonymen Klassen:

- Definition und Instanzierung finden in einem **new**-Operanden statt, wobei im Konstruktoraufruf der fehlende Klassenname durch den Namen der implementierten Schnittstelle oder der beerbten Basisklasse vertreten wird. Es folgt ein Klassendefinitionsblock, der wie üblich durch geschweifte Klammern zu begrenzen ist. Im folgenden Beispiel
  ```
  button.setOnAction(new EventHandler<ActionEvent>() {
     @Override
     public void handle(ActionEvent event) {
        label.setText("Hallo JavaFX");
     }
  });
  ```
  wird die Schnittstelle **EventHandler<ActionEvent>** angegeben und deren einzige Methode **handle()** implementiert. Es kann nur eine einzige Instanz der anonymen Klasse erzeugt werden. Werden mehrere Instanzen benötigt, ist eine alternative Lösung zu verwenden (lokale Klasse, Mitgliedsklasse oder Top-Level - Klasse).
- Weil der Klassenname fehlt, sind keine Konstruktoren möglich. Über die Instanzinitialisierer (vgl. Abschnitt 4.4.4) ist jedoch eine Ersatzlösung verfügbar (siehe Beispiel im Abschnitt 13.5.3.2.2).
- Statische Methoden sind verboten, und statische Variablen müssen finalisiert sein.
- Eine anonyme Klasse kann auf die *finalisierten* lokalen Variablen der umgebenden Methode zugreifen. Seit Java 8 wird nur noch die *effektive Finalität* verlangt. Diese besteht, wenn nach der Initialisierung keine Wertveränderung stattfindet. Der Modifikator **final** ist also nicht mehr erforderlich. Seit Java 8 kann eine anonyme Klasse auch auf effektiv finale *Parameter* der umgebenden Methode zugreifen. Weil der Zugriff auf effektiv finale Variablen bzw. Parameter beschränkt ist, sind natürlich nur *lesende* Zugriffe erlaubt. Über eine *Referenz*variable in der lokalen Umgebung sind aber durchaus Schreibzugriffe auf das ansprechbare Objekt möglich, weil sich die Referenzvariable dabei ja nicht ändert.
- Außerdem kann eine anonyme Klasse auf die statischen Variablen und Methoden der umgebenden Klasse zugreifen. Wird eine anonyme Klasse in einer Instanzmethode definiert, kann sie auch auf die Instanzvariablen und -methoden des handelnden Objekts zugreifen. Alle genannten Zugriffe sind auch bei Methoden und Feldern mit **private**-Deklaration möglich. Auf Felder kann lesend und schreibend zugegriffen werden. Es ist Vorsicht geboten, wenn der Code einer anonymen Klasse in verschiedenen Threads ausgeführt wird.
- Felder und lokale Variablen der anonymen Klasse überdecken gleichnamige Variablen der Umgebung. Überdeckte statische Variablen der umgebenden Klasse können in der anonymen Klasse über den Klassennamen als Präfix angesprochen werden, z. B. bei einer umgebenden Klasse namens `Aussen` mit der statischen Variablen `statEnv`:
  ```
  Aussen.statEnv
  ```
  Überdeckte Instanzvariablen eines umgebenden Objekts können in der anonymen Klasse über einen Präfix aus dem Klassennamen und dem Schlüsselwort **this** angesprochen werden, z. B. bei einer umgebenden Klasse namens `Aussen` mit der Instanzvariablen `instEnv`:
  ```
  Aussen.this.instEnv
  ```
- Der Compiler erzeugt auch für eine anonyme Klasse eine eigene **class**-Datei, in deren Namen der Bezeichner für die umgebende Klasse eingeht, z. B.: **Main$1.class**.

Die gleich vorzustellenden Lambda-Ausdrücke können oft statt einer anonymen Klasse verwendet werden und dabei für einen besser lesbaren Quelltext sorgen. In anderen Fällen sind anonyme Klassen weiterhin zu bevorzugen, weil sie aufgrund der reichhaltigeren syntaktischen Optionen u.a. die folgenden Vorteile gegenüber Lambda-Ausdrücken haben:

- Aus einem Lambda-Ausdruck resultiert stets ein Objekt, das genau *eine* Methode beherrscht. Im Unterschied dazu kann eine anonyme Klasse beliebig viele Instanzmethoden besitzen.
- Ein Lambda-Objekt muss auf Instanzvariablen verzichten, während diese bei einem anonymen Objekt verfügbar sind.
- Eine anonyme Klasse kann eine Schnittstelle implementieren oder eine Basisklasse erweitern. Letzteres ist bei Lambda-Ausdrücken nicht möglich.
- Ein Lambda-Objekt kann sich nicht selbst (mit dem Schlüsselwort **this**) ansprechen, was bei anonymen Klassen genauso klappt wie bei Top-Level - Klassen.

12.1.1.3 Compiler-Kompetenz statt Boilerplate-Code

Mit *Boilerplate-Code* sind Syntaxfragmente gemeint, die sich häufig in weitgehend identischer Form wiederholen und daher lästig werden. Die seit Java 8 mögliche Realisierung einer Ereignisbehandlungsmethode durch einen Lambda-Ausdruck enthält im Vergleich zur traditionellen Lösung durch eine anonyme Klasse deutlich weniger monoton und umständlich zu wiederholenden Boilerplate-Code z. B.:

```
button.setOnAction(event -> label.setText("Hallo JavaFX"));
```

Hinter den Kulissen bleibt im Wesentlichen alles beim Alten, wobei der Compiler viele Routinearbeiten übernimmt:

- Er kennt den Interface-Parametertyp **EventHandler<ActionEvent>** von **setOnAction()** und akzeptiert einen Lambda-Ausdruck, der die erforderliche Methode **handle()** realisiert, sodass sich eine passende anonyme Klasse erstellen lässt.
- Der im Lambda-Ausdruck vor dem Pfeil (`->`) angegebene Parameter muss vom passenden Typ sein. Man kann jedoch auf eine Typangabe verzichten, wobei der Compiler den Parameterdatentyp aus der Interface-Definition entnimmt.
- Die vom Code-Block produzierte Rückgabe muss vom passenden Typ sein. Im Beispiel hat die Interface-Methode **handle()** den Rückgabetyp **void**, und eine **return**-Anweisung mit Rückgabe als Bestandteil des Lambda-Ausdrucks würde zum Übersetzungsfehler führen, z. B.:

  ```
  button.setOnAction(event -> {label.setText("Hallo JavaFX"); return 13;});
  ```
  Unexpected return value

- Weil die Typen **EventHandler** und **ActionEvent** im Unterschied zur Verwendung einer anonymen Klasse (siehe Abschnitt 12.1.1.2) im Quellcode nicht mehr auftauchen, sind keine Import-Deklarationen erforderlich.
- Wenn das GUI-Framework später (nach einem Mausklick auf den Schalter) die **EventHandler<ActionEvent>** - Methode **handle()** aufruft, wird der Code im Lambda-Ausdruck ausgeführt.

Ab Java 8 gilt generell: Wo der Compiler ein Objekt vom Datentyp einer funktionalen Schnittstelle erwartet, ist ein Lambda-Ausdruck passender Bauart erlaubt. Das zu Beginn von Kapitel 12 vorgestellte Beispiel

```
s -> s.length() >= 5
```

eignet sich z. B. als Parameter für die Methode **filter()** im Interface **Stream<String>**:

```
List<String> als = Arrays.asList("Rudolf","Emma","Otto","Agnes","Frank");
Stream<String> str = als.stream();
str = str.filter(s -> s.length() >= 5);
```

In diesem Beispiel wird die im Abschnitt 10.1 vorgestellte statische und generische Methode **asList()** der Klasse **Arrays** dazu verwendet, ein **List<String>** - Objekt zu erstellen:

public static <T> List<T> asList(T... *a***)**

Aus dem **List<String>** - Objekt entsteht über die Methode **stream()** aus dem Interface **Collection<E>** (siehe Abschnitt 10.2.1) ein Objekt der Klasse **Stream<String>**. Der **Stream<String>** - Methode **filter()** wird ein Lambda-Ausdruck übergeben, der auf jedes **String**-Element im Strom angewandt wird. Es entsteht ein neuer Strom aus den Elementen des alten Stroms mit mindestens 5 Zeichen.

Die Methode **filter()** erwartet einen Parameter vom Interface-Typ **Predicate<String>** aus dem Paket **java.util.function** (siehe Quellcode im Abschnitt 12.1.1.1). Man kann den Lambda-Ausdruck einer Referenzvariablen von diesem Typ zuweisen

```
Predicate<String> ps = s -> s.length() >= 5;
```

und die Variable anschließend als **filter()** - Parameter verwenden, z. B.:

```
str = str.filter(ps);
```

Bei einem mehrfach benötigten Lambda-Ausdruck ist die Verwendung einer Variablen zu empfehlen, weil eine Code-Wiederholung gegen das DRY-Prinzip (*Don't Repeat Yourself*) verstoßen würde.

Man darf sich vorstellen, dass der Compiler aus dem Lambda-Ausdruck in der folgenden Zuweisung

```
str = str.filter(s -> s.length() >= 5);
```

eine anonyme Klasse konstruiert:

```
str = str.filter(new Predicate<>() {
    public boolean test(String s) {
        return s.length() >= 5;
    }
});
```

Mit einem Lambda-Ausdruck gelingt es in der Regel, die benötigte Funktionalität auf sehr kompakte Weise zu implementieren. Gelegentlich ist aber doch eine direkt definierte anonyme Klasse wegen der folgenden (schon am Ende von Abschnitt 12.1.1.2 aufgelisteten) Vorteile zu bevorzugen:

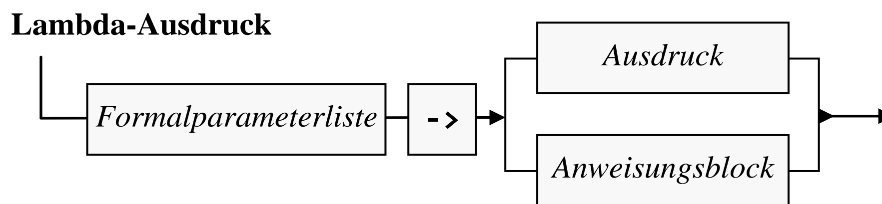
- Aus einem Lambda-Ausdruck resultiert stets ein Objekt, das genau *eine* Methode beherrscht. Im Unterschied dazu kann eine anonyme Klasse beliebig viele Instanzmethoden besitzen.
- Ein Lambda-Ausdruck besteht aus einer Methoden-Implementation, sodass keine Instanzvariablen definiert werden können. Ein Lambda-Objekt muss also auf Instanzvariablen verzichten, während diese bei einem anonymen Objekt verfügbar sind. Wenn z. B. ein **ActionEvent**-Handler zu einem GUI-Bedienelement über die einzelnen Aufrufe hinweg Daten speichern muss, ist ein „vollwertiges" Objekt erforderlich. Um ein Objekt einer anonymen Klasse mehrfach verwenden zu können, muss man seine Adresse natürlich in einer Referenzvariablen aufbewahren.
- Eine anonyme Klasse kann eine Schnittstelle implementieren oder eine Basisklasse erweitern. Letzteres ist bei Lambda-Ausdrücken nicht möglich.
- Ein Lambda-Objekt kann sich nicht selbst (mit dem Schlüsselwort **this**) ansprechen, was bei anonymen Klassen genauso klappt wie bei Top-Level - Klassen.

Unser großes Vorbild Joshua Bloch (2018, Item 42) rät, Lambda-Ausdrücke gegenüber anonymen Klassen zu bevorzugen, solange der Lambda-Code kurz (max. 3 Zeilen lang) und selbsterklärend bleibt. Gegen einen unübersichtlichen Lambda-Code lässt sich leicht etwas unternehmen, weil ein Lambda-Ausdruck (wie eine anonyme Klasse) auf (statische) Methoden der umgebenden Klasse zugreifen kann.

Während zu einer anonymen Klasse beim Übersetzen eine eigene Bytecode-Datei entsteht (vgl. Abschnitt 12.1.1.2), wird die zu einem Lambda-Ausdruck gehörige Klasse zur Laufzeit bei Bedarf dynamisch erstellt (Sosnoski 2014).

12.1.2 Definition von Lambda-Ausdrücken

Um mit der Syntax des Lambda-Ausdrucks vertraut zu werden, verschaffen wir uns zunächst einen Überblick mit Hilfe eines Syntaxdiagramms:



Der Rückgabetyp, den der Compiler aus der vom Lambda-Ausdruck implementierten Schnittstellenmethode kennt, ist *nicht* anzugeben.

12.1.2.1 Formalparameterliste

Grundsätzlich gilt für die Formalparameterliste eines Lambda-Ausdrucks wie für die Formalparameterliste einer Methode (vgl. Abschnitt 4.3.1.4):

- Die Formalparameterliste wird durch ein Paar runder Klammern begrenzt.
- Für jeden Formalparameter sind ein Datentyp und ein Name anzugeben.
- Die Formalparameter sind durch ein Komma voneinander zu trennen.
- Am Ende kann ein Serienparameter stehen.
- Die Parameter können als **final** deklariert werden.

Der Compiler erlaubt allerdings bei der Formalparameterliste eines Lambda-Ausdrucks einige signifikante Vereinfachungen:

- Man kann auf die Angabe der Parametertypen verzichten, weil sich diese aus dem zu erfüllenden Interface zwingend ergeben. Es ist zu beachten, dass der Datentyp für alle Parameter *einheitlich* entweder anzugeben oder wegzulassen ist. Im folgenden Beispiel
  ```
  IntBinaryOperator absMax = (a, b) -> Math.abs(a) >= Math.abs(b) ? a : b;
  ```
  wird per Lambda-Ausdruck ein Objekt namens `absMax` erstellt, dessen Klasse das Interface **IntBinaryOperator** (im Paket **java.util.function**) erfüllt:
  ```
  public interface IntBinaryOperator {
     int applyAsInt(int left, int right);
  }
  ```
  Der Lambda-Ausdruck liefert zu zwei **int**-Werten die Zahl mit dem größten Betrag. Er kann z. B. als Argument der **Stream**-Methode **reduce()** verwendet werden, um aus einem **IntStream**-Objekt das Element mit dem größten Betrag zu fischen:
  ```
  IntStream is = IntStream.of(-3, 7, -12, 5);
  OptionalInt amax = is.reduce(absMax);
  ```
  Die Methode **reduce()** liefert ein Objekt der Klasse **OptionalInt**, das die Betrags-maximale Zahl aus dem Strom enthält, oder (bei einem leeren Strom) *keinen* Wert besitzt. Ausführliche Erläuterungen zu **reduce()** und anderen Stromoperationen folgen im Abschnitt 12.2.5.

- Bei einem *einzelnen*, implizit typisierten Parameter kann man die runden Klammern weglassen. Das Beispiel

  ```
  Predicate<String> ps = (String s) -> s.length() >= 5;
  ```

  kann also kompakter notiert werden:

  ```
  Predicate<String> ps = s -> s.length() >= 5;
  ```

Wie bei einer Methodendefinition muss im Falle einer leeren Parameterliste ein paar runder Klammern angegeben werden, z. B.:

```
() -> 1
```

Dieser scheinbar sinnlose Lambda-Ausdruck eignet sich übrigens als Parameter der **IntStream**-Methode **generate()** dazu, einen unendlich langen Strom mit Einsen zu erzeugen, der per **limit()** - Aufruf die tatsächlich benötigte Länge erhält, z. B.:

```
IntStream one = IntStream.generate(() -> 1).limit(10);
```

Weil die Ausführung der Strommethoden im Java generell ökonomisch bzw. faul (engl.: *lazy*) erfolgt, wird *keinesfalls* ein „unendlich" langer Strom erzeugt und anschließend gekappt. Stattdessen entstehen genau die benötigten 10 Elemente.

12.1.2.2 Rumpf

Der Lambda-Rumpf kann aus einem geschweift eingeklammerten Anweisungsblock bestehen

```
IntBinaryOperator absMax = (a, b) -> {
    if (Math.abs(a) >= Math.abs(b))
        return a;
    else
        return b;
};
```

oder aus einem einzelnen Ausdruck (im Sinn von Abschnitt 3.5):

```
IntBinaryOperator absMax = (a, b) -> Math.abs(a) >= Math.abs(b) ? a : b;
```

Ist der Lambda-Rumpf ein Anweisungsblock und der Rückgabetyp der zu erfüllenden Interface-Methode von **void** verschieden, dann muss für jeden möglichen Ausführungspfad per **return**-Anweisung ein Rückgabewert vom passenden Typ geliefert werden (siehe erstes Beispiel).

Wenn im Anweisungsblock eines Lambda-Ausdrucks eine Methode aufgerufen wird, die eine geprüfte Ausnahme (vgl. Abschnitt 11.4.2) werfen kann, und diese Ausnahme in der implementierten abstrakten Interface-Methode *nicht* deklariert wird, dann muss der Lambda-Block die Ausnahme in einer **try**-Anweisung mit geeignetem **catch**-Block abfangen (vgl. Abschnitt 11.2).
Im folgenden Beispielprogramm

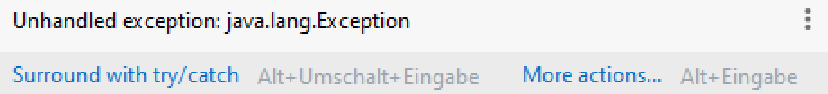
```
interface Tester<T> {
  boolean test(T t) throws Exception;
}
class Prog {
  public static void main(String[] args) throws Exception {
    java.util.function.Predicate<String> pstr = s ->
      {if (s.length() == 13) throw new RuntimeException(); return s.length() >= 5;};

    Tester<String> tester = s ->
      {if (s.length() == 13) throw new Exception(); return s.length() >= 5;};
  }
}
```

werden die beiden Ausnahmen werfenden Lambda-Ausdrücke vom Compiler akzeptiert:

- Im Lambda-Ausdruck vom Typ **Predicate<String>** (zur Definition siehe Abschnitt 12.1.1.1) wird eine ungeprüfte Ausnahme (aus der Klasse **RuntimeException**, vgl. Abschnitt 11.4.2) geworfen, was generell erlaubt ist.
- Im Lambda-Ausdruck vom Typ `Tester<String>` wird eine geprüfte Ausnahme geworfen, die im implementierten Interface angemeldet wird (siehe Definition der Methode `test()`).

Einem das Interface **Predicate<String>** implementierenden Lambda-Ausdruck ist es hingegen nicht erlaubt, eine *ungeprüfte* Ausnahme zu werfen:

```
java.util.function.Predicate<String> pstr = s ->
    {if (s.length()==13) throw new Exception(); return s.length() >= 5;};
```



12.1.2.3 Definitionsumgebungen

Ein Lambda-Ausdruck kann in verschiedenen Umgebungen definiert werden, z. B.:

- In einer Instanzmethode oder in einem Konstruktor
- In einer statischen Methode oder in einem statischen Initialisierer
- Bei der Initialisierung eines (statischen) Felds.

Er hat dementsprechend Zugriff auf unterschiedliche Variablen und Methoden der Umgebung:

- Wird ein Lambda-Ausdruck in einer Methode definiert, hat er Zugriff auf effektiv finale Variablen und Parameter der umgebenden Methode. Eine lokale Variable ist effektiv final, wenn ihr Wert nach der ersten Zuweisung unverändert bleibt. Ein Parameter ist effektiv final, wenn sein Wert in der Methode nicht verändert wird. Weil der Zugriff auf (effektiv) *finale* Variablen bzw. Parameter beschränkt ist, sind natürlich nur *lesende* Zugriffe erlaubt. Über eine *Referenz*variable in der lokalen Umgebung sind aber durchaus Schreibzugriffe auf das ansprechbare Objekt möglich, weil sich die Referenzvariable dabei ja nicht ändert (siehe Beispiel unten).
- Wird ein Lambda-Ausdruck in einer Instanzmethode oder in einem Konstruktor definiert, dann hat er Zugriff auf ...
    - die Instanzvariablen und -methoden des umgebenden Objekts,
    - die statischen Variablen und Methoden der umgebenden Klasse.
- Wird ein Lambda-Ausdruck in einer statischen Methode oder in einem statischen Initialisierer definiert, dann hat er Zugriff auf die statischen Variablen und Methoden der umgebenden Klasse.

Lokale Variablen eines Lambda-Ausdrucks überdecken (statische) Felder der Umgebung. Überdeckte statische Variablen der umgebenden Klasse können im Lambda-Ausdruck über den Klassennamen als Präfix angesprochen werden, z. B. bei einer umgebenden Klasse namens `Aussen` mit der statischen Variablen `statEnv`:

```
Aussen.statEnv
```

Überdeckte Instanzvariablen eines umgebenden Objekts können im Lambda-Ausdruck über das Schlüsselwort **this** angesprochen werden, z. B. bei einer umgebenden Klasse namens `Aussen` mit der Instanzvariablen `instEnv`:

```
this.instEnv
```

Auf Instanz- und Klassenvariablen der Umgebung kann in einem Lambda-Ausdruck auch schreibend zugegriffen werden. Es ist Vorsicht geboten, wenn der Code eines Lambda-Ausdrucks in verschiedenen Threads ausgeführt wird.

Das folgende Beispielprogramm demonstriert für einen im Konstruktor definierten Lambda-Ausdruck die erlaubten Zugriffe auf Umgebungsvariablen:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
import java.util.function.Supplier;

class LambdaScoping {
   static int statEnv = 0;
   int instEnv = 10;
   Supplier<String> sups;

   LambdaScoping() {
      int locEnv = 13;
      int[] locEnvArr = {100};
      sups = () -> {
         // int locEnv = 14; // Verboten
         return String.valueOf(++statEnv)+" "+
                String.valueOf(++instEnv)+" "+
                String.valueOf(locEnv)+" "+
                String.valueOf(++locEnvArr[0]);
      };
   }

   void prot() {
      System.out.println(sups.get());
   }

   public static void main(String[] args) {
      LambdaScoping ls = new LambdaScoping();
      ls.prot(); ls.prot(); ls.prot();
   }
}
```

</td><td>

```
1 11 13 101
2 12 13 102
3 13 13 103
```

</td></tr>
</table>

Die Zusammenfassung eines Lambda-Ausdrucks mit den „eingefangenen“ Variablen aus der Umgebung wird als *Abschluss* (engl. *closure*) bezeichnet.

Beim Zugriff auf Umgebungsvariablen gelten für anonyme Klassen und Lambda-Ausdruck weitgehend identische Regeln mit folgenden Ausnahmen:

- Eine anonyme Klasse begründet einen eigenen Gültigkeitsbereich, und in ihren Methoden dürfen lokale Variablen mit einem Namen angelegt werden, den auch lokale Variablen einer umgebenden Methode verwenden. Dabei werden die Umgebungsvariablen überdeckt. Ein Lambda-Ausdruck gehört hingegen wie ein gewöhnlicher eingeschachtelter Block zum Gültigkeitsbereich einer umgebenden Methode, sodass die dortigen lokalen Variablennamen im Lambda-Ausdruck *nicht* verwendet werden dürfen. In der englischsprachigen Literatur wird dafür die Bezeichnung *lexical scoping* verwendet.
- Sowohl in einer anonymen Klasse als auch in einem Lambda-Ausdruck werden Instanzvariablen eines umgebenden Objekts durch lokale Variablen überdeckt. Um die Instanzvariablen des umgebenden Objekts weiterhin ansprechen zu können, genügt im Lambda-Ausdruck das Schlüsselwort **this**, das sich hier auf das umgebende Objekt bezieht, z. B.:

  ```
  String.valueOf(this.instEnv)
  ```

  Daraus ergibt sich allerdings die in seltenen Fällen relevante Einschränkung, dass sich das Lambda-Objekt nicht selbst ansprechen kann. In einer anonymen Klasse bezieht sich **this** hingegen auf das anonyme Objekt, und zum Zugriff auf eine überdeckte Instanzvariable des umgebenden Objekts ist dem Schlüsselwort der Klassenname voranzustellen, z. B.:

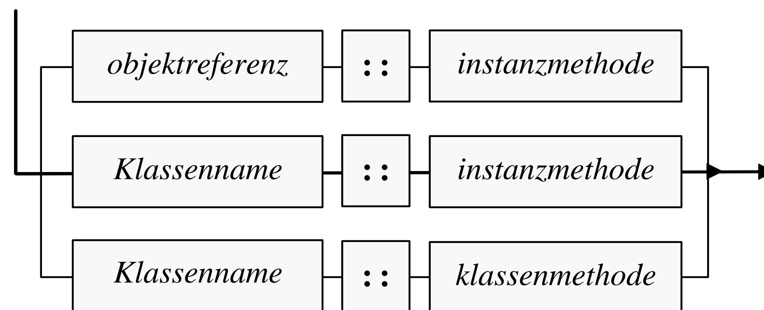
  ```
  String.valueOf(LambdaScoping.this.instEnv)
  ```

12.1.3 Methoden- und Konstruktorreferenzen

Man glaubt es kaum, doch in vielen Situationen bietet Java seit der Version 8 zur Erstellung von Funktionsobjekten für die Lambda-Syntax noch prägnantere Alternativen. Dadurch gibt es allerdings für Einsteiger noch mehr Syntaxvarianten kennenzulernen.

12.1.3.1 Methodenreferenzen

Wenn zu einem geplanten Lambda-Ausdruck eine Methode existiert, bei der die Formalparameterliste und der Rückgabetyp exakt passen, dann kann der Lambda-Ausdruck durch eine sogenannte *Methodenreferenz* ersetzt werden:



Bei einer Instanzmethode wird der Auftragnehmer entweder durch eine konkrete Objektreferenz (z. B. **System.out**) oder durch eine Klasse angegeben. Bei einer statischen Methode ist der Klassenname anzugeben. Hinter den Auftragnehmer ist der **::** - Operator zu setzen. Schließlich folgt der Methodenname *ohne* Parameterliste, womit prinzipiell überflüssige Code-Bestandteile eingespart werden, was Anhänger eines kompakten Programmierstils erfreut (siehe z. B. Bloch 2018, S. 197f).

Gibt man eine Klasse zusammen mit einer Instanzmethode an (Fall 2 im Syntaxdiagramm), dann wird der erste Parameter der zu implementierenden Schnittstellenmethode zum Ansprechpartner für den Aufruf der Instanzmethode, und die restlichen Parameter der zu implementierenden Methode müssen zu den Parametern der Instanzmethode passen. Ebenso muss der Rückgabetyp kompatibel sein. Eine Methodenreferenz von der mittleren Bauart aus dem obigen Syntaxdiagramm ist also genau dann zulässig, wenn der Typ des ersten Parameters in der zu implementierenden Schnittstellenmethode eine Instanzmethode beherrscht, welche genau die restlichen Parameter aus der zu implementierenden Schnittstellenmethode verarbeitet und einen passenden Rückgabewert liefert.

Im folgenden Beispiel wird für die **String**-Objekte in einer Liste die mittlere Länge berechnet, wobei ein Stromobjekt vom Typ **Stream<String>** zum Einsatz kommt (vgl. Abschnitt 12.2). Der **Stream<String>** - Methode **mapToInt()** wird als Parameter vom Interface-Typ **ToIntFunction<? super String>** ein Lambda-Ausdruck übergeben:

```
OptionalDouble ml =
    Arrays.asList("Viktor","Otto","Emma","Kurt","Isolde","Frank")
                  .stream()
                  .mapToInt(s -> s.length())
                  .average();
```

Von **mapToInt()** wird die Schnittstellenmethode

public int applyAsInt(String *value***)**

mit jedem Stromelement als Aktualparameter aufgerufen. Ein Lambda-Ausdruck verwendet dieselbe Formalparameterliste wie die zu implementierende Schnittstellenmethode. Im Beispiel wird der erste (und einzige) Parameter (Typ **String**) zum Ansprechpartner für den Aufruf der **String**-Instanzmethode **length()**. Hier passt eine Instanzmethodenreferenz gemäß Fall 2 aus dem obigen Syntaxdiagramm:

- Aus dem ersten (und einzigen) **applyAsInt()** - Parameter wird die Klassenangabe **String**.
- Bei der **String**-Instanzmethode **length()** passen die restliche Parameterliste und der Rückgabetyp **int**.

Folglich kann der Lambda-Ausdruck durch eine Instanzmethodenreferenz mit der Methode **length()** ersetzt werden:

```
OptionalDouble ml =
    Arrays.asList("Viktor", "Otto", "Emma", "Kurt", "Isolde", "Frank")
                  .stream()
                  .mapToInt(String::length)
                  .average();
```

Man darf sich vorstellen, dass der Compiler aus der Methodenreferenz eine anonyme Klasse synthetisiert:

```
OptionalDouble ml =
    Arrays.asList("Viktor", "Otto", "Emma", "Kurt", "Isolde", "Frank")
            .stream()
            .mapToInt(new ToIntFunction<String>() {
                public int applyAsInt(String s) {
                    return s.length();
                }
             })
            .average();
```

Ist der Auftragnehmer ein *konkretes* Objekt (z. B. **System.out**) oder eine Klasse, dann werden alle Parameter der zu implementierenden Schnittstellenmethode auf die Parameter der Instanz- oder Klassenmethode abgebildet. Im folgenden Beispiel wird an die **Stream<String>** - Methode **forEach()** ein Objekt übergeben, das die Schnittstelle **Consumer<? super String>** im Paket **java.util.function** implementiert:

```
Arrays.asList("Viktor", "Otto", "Emma", "Kurt", "Isolde", "Frank")
        .stream()
        .forEach(s -> System.out.println(s));
```

Der Lambda-Ausdruck

```
s -> System.out.println(s)
```

sorgt für die Ausgabe der Stromelemente. Er hat einen Parameter vom Typ **String** sowie den Rückgabetyp **void** und kann daher durch die folgende Methodenreferenz

```
System.out::println
```

ersetzt werden:

```
Arrays.asList("Viktor", "Otto", "Emma", "Kurt", "Isolde", "Frank")
        .stream()
        .forEach(System.out::println);
```

Ein Beispielprogramm aus Abschnitt 10.5.5, das ein **Comparator<String>** - Objekt als Parameter für den **TreeSet<String>** - Konstruktor verwendet, um eine geordnete Namenssammlung mit bevorzugter Einordnung von „Otto“ zu erstellen, lässt sich leicht zur Demonstration einer Methodenreferenz vom statischen Typ umbauen, wobei ausnahmsweise *kein* Vorgriff auf die im Abschnitt 12.2 vorzustellenden **Stream**-Klassen stattfindet:

Quellcode	Ausgabe

`import java.util.*;`
 	
`class StatMethRef {`
`  public static int compare(String s1, String s2) {`
`    if (s1.equals("Otto"))`
`      return -1;`
`    if (s2.equals("Otto"))`
`      return 1;`
`    return s1.compareTo(s2);`
`  }`
 	
`  public static void main(String[] args) {`
`    TreeSet<String> tssc = new TreeSet<>(StatMethRef::compare);`
`    tssc.add("Werner"); tssc.add("Ludwig"); tssc.add("Otto");`
`    System.out.println(tssc);`
`  }`
`}`
	`[Otto, Ludwig, Werner]`

Der **TreeSet<String>** - Konstruktor erwartet ein Funktionsobjekt, das die Schnittstelle

Comparator<? super String>

erfüllt. Dazu ist eine Methode mit dem folgenden Definitionskopf erforderlich:

int compare (? super String *s1*, **? super String** *s2*)

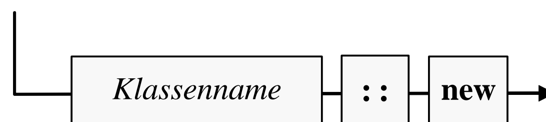
Statt wie im Abschnitt 10.5.5 eine Klasse zu definieren, die das Interface erfüllt, wird eine statische Methode von der geforderten Bauart definiert und per Methodenreferenz an den Konstruktor übergeben.

Weitere Details zu Methodenreferenzen sind z. B. bei Horstmann (2014b) zu finden.

12.1.3.2 Konstruktorreferenzen

Wenn ein Lambda-Ausdruck nichts anderes tut, als ein Objekt per Konstruktoraufruf zu instanzieren, dann kann der Lambda-Ausdruck durch eine sogenannte *Konstruktorreferenz* ersetzt werden:

Konstruktorreferenz



Eine Konstruktorreferenz unterscheidet sich von einer Methodenreferenz (siehe Abschnitt 12.1.3.1) dadurch, dass ein Konstruktor statt einer Methode aufgerufen wird, was syntaktisch folgende Konsequenzen hat:

- Vor dem **::** - Operator befindet sich stets ein Klassenname.
- An der Stelle des Methodennamens befindet sich das Schlüsselwort **new**.

Im folgenden Beispiel sollen **String**-Objekte in Objekte der Klasse **BigDecimal** gewandelt werden. Wir erstellen aus einem Kollektionsobjekt der Klasse **List<String>** ein Stromobjekt vom Typ **Stream<String>** und verwenden dessen Methode **map()**, um daraus ein Stromobjekt vom Typ **Stream<BigDecimal>** zu erzeugen. Die Methode **map()** erwartet als Parameter ein Funktionsobjekt vom Interface-Typ **Function<String, BigDecimal>**, der die Methode **apply()** vorschreibt:

public BigDecimal apply(String *s*)

Die folgende Überladung des **BigDecimal** - Konstruktors

public BigDecimal(String *val*)

erfüllt den Job und kann daher per Konstruktorreferenz an **map()** übergeben werden:

Quellcode	Ausgabe

<pre>import java.math.BigDecimal;
import java.util.Arrays;
 	
class KonstruktorReferenzen {
   public static void main(String[] args) {
      Arrays.asList("3.14", "9.99", "47.11")
         .stream()
         .map(BigDecimal::new)
         .forEach(System.out::println);
}</pre>
	<pre>3.14
	9.99
	47.11</pre>

Die Konstruktorreferenz

```
BigDecimal::new
```

ist äquivalent zum Lambda-Ausdruck:

```
s -> new BigDecimal(s)
```

12.2 Ströme

12.2.1 Elementare Begriffe und Beispiel

Neben den Lambda-Ausdrücken waren die **Stream<T>** - Typen im Paket **java.util.stream** die bedeutsamste Neuerung in Java 8. Stromobjekte erlauben Abfrage- und/oder Verarbeitungsoperationen in Massenabfertigung für alle von einer Datenquelle (z. B. von einer Kollektion oder einer Datei) gelieferten Elemente.

Dabei ist eine *deklarative Programmierung* möglich, und das explizite Iterieren über die Stromelemente bei ständiger Aktualisierung von Variablen mit Zwischenergebnissen wird in die Tiefen der Standardbibliothek verlagert. Man kann z. B. (analog zu einer Datenbankabfrage per SQL-Abfrage mit SELECT-Spaltenwahl, AVG-Funktion und WHERE-Bedingung) für eine Serie von Kontoobjekten den mittleren Stand für die Konten eines bestimmten Typs ermitteln, ohne sich um Details bei der Iteration über die Elemente und bei der Fallauswahl kümmern zu müssen. Anwendungsprogrammierer können sich auf das *Was* konzentrieren und viele Implementierungsdetails (also Aspekte des *Wie*) der Standardbibliothek überlassen.

In günstigen Fällen gelingt es, von der sequenziellen Verarbeitung mit geringem Aufwand auf die *parallele, mehrere Prozessorkerne nutzende Verarbeitung* umzustellen. Bei parallelen Stromoperationen werden ...

- die Daten in Teilmengen zerlegt,
- die Teilmengen in eigenständigen Threads parallel verarbeitet,
- und die Teilergebnisse am Ende zusammengeführt.

In vielen Situationen kann man sich eine eigene Multithreading-Lösung, die typischerweise mit Aufwand und Fehlerrisiko verbunden ist, ersparen und die parallelisierte Strombearbeitung den ausgefeilten Methoden der Systembibliothek überlassen. Dabei kommt im Hintergrund das *Fork-Join* - Framework zum Einsatz, das im Abschnitt 15.5.1 vorgestellt wird.

Bevor es zu abstrakt wird, betrachten wir ein Beispiel. Die etwas künstliche Aufgabe besteht darin, für eine Sequenz von Namen die mittlere Länge aller Namen mit mindestens fünf Zeichen zu ermitteln.

```
List<String> als = Arrays.asList("Viktor","Otto","Emma","Kurt","Isolde","Frank");
OptionalDouble mlge5 = als.stream()
      .filter(s -> s.length() >= 5)
      .mapToInt(s -> s.length())
      .average();
System.out.println(mlge5);
```

Ausgehend von einer Liste mit **String**-Objekten, erstellt von der statischen **Arrays**-Methode **asList()**, wird über die (im Abschnitt 10.2.1 erwähnte) **Collection<T>** - Methode **stream()** ein Objekt vom Typ **Stream<String>** erstellt.

Daraus entsteht durch Anwendung der Operation **filter()** ein neues Stromobjekt vom selben Typ, das nur noch die **String**-Objekte mit mindestens 5 Zeichen enthält. Zur Bewertung der Zeichenfolgen im ursprünglichen Strom dient ein Funktionsobjekt einer anonymen, das Interface **Predicate<String>** implementierenden Klasse, die per Lambda-Ausdruck definiert wird und die Instanzmethode

public boolean test(String *s*)

besitzt.

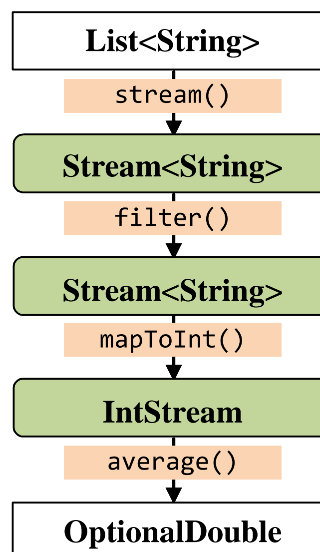
Wie das Beispiel zeigt, bietet die **Stream**-Bibliothek ein *Fluent API*, weil ihre Methoden ein flüssiges Programmieren durch Verketten von Aufrufen erlauben.

Über die Operation **mapToInt()** erhält man durch elementweise Abbildung ein Stromobjekt vom Typ **IntStream**. Für die Produktion des **int**-Werts zu einer Zeichenfolge ist ein Funktionsobjekt einer anonymen, das Interface **ToIntFunction<String>** implementierenden Klasse zuständig, die per Lambda-Ausdruck definiert wird und die Instanzmethode

public int applyAsInt(T *value***)**

beherrscht.

Auf das **IntStream**-Objekt wird die Stromoperation **average()** angewendet, um ein Ergebnisobjekt vom Typ **OptionalDouble** zu produzieren, das bei einem nicht-leeren Strom die gesuchte Durchschnittslänge als **double**-Wert enthält und nach Aufforderung durch **getAsDouble()** ausliefert. In der folgenden Abbildung ist die gesamte Stromverarbeitung dargestellt:



Insgesamt wird das sogenannte **Filter-Map-Reduce** - Muster realisiert.

Ein Stromobjekt ist *kein* Container, sondern eine Station in einer Verarbeitungskette für die Daten in einer Sequenz. Außerdem dürfen die in Java 8 eingeführten **Stream**-Typen zur Massenabfertigung von Elementen in Sequenzen nicht mit den viel älteren I/O - Stream - Klassen verwechselt

(z. B. **InputStream** and **OutputStream**, siehe Kapitel 14), die schon in Java 1.0 vorhanden waren und im Kapitel 14 behandelt werden.

Als Datenquellen für ein Stromobjekt kommen z. B. in Frage:

- eine Kollektion
- ein Array
- eine Methode zum Generieren von (potentiell unendlich vielen) Werten
- ein anderer Strom

Auf diese Daten kann ein Stromobjekt serielle und parallele **Aggregat-Operationen** anwenden, wobei ein neues Stromobjekt oder ein Endergebnis entsteht. Die Besonderheit von Aggregat-Operationen besteht darin, dass sie auf den Strom *im Ganzen* wirken, also auf *alle* Elemente der Quelle. Es ist nicht möglich, eine Aggregat-Operation auf einzelne Elemente eines Stroms zu beschränken.

In der Regel werden mehrere Stromoperationen hintereinander gesetzt, sodass eine **Pipeline** entsteht (siehe Beispiel). Diese ....

- startet mit einer Datenquelle,
- durchläuft beliebig viele intermediäre Operationen (eventuell aber auch keine)
- und endet mit einer terminalen Operation, die ein Endergebnis produziert (z. B. eine Zahl oder eine Kollektion)

Liegt das Ergebnis vor, ist die Pipeline mit ihren Stromobjekten verbraucht und kann *nicht* erneut durchlaufen werden. Ein Versuch führt zum Laufzeitfehler:

```
java.lang.IllegalStateException: stream has already been operated upon or closed
```

12.2.2 Externe versus interne Iteration

Bei der traditionellen *externen* Iteration über die Elemente einer Kollektion (z. B. per **for**-Schleife) ...

- muss der Programmierer Schritt für Schritt festlegen, *wie* vorzugehen ist, sodass Zeitaufwand und Fehlergefahr gegeben sind,
- ist eine eventuell mögliche Parallelisierung nur schwer zu realisieren, sodass sie meist unterbleibt.

Ein wesentliches Kennzeichen der in Java 8 eingeführten Stromoperationen besteht darin, dass Iterationen in der Standardbibliothek gekapselt, also aus dem Anwendungs-Code ferngehalten werden. Bei den mit *interner* Iteration arbeitenden Stromoperationen ...

- legt der Programmierer fest, *was* geschehen soll (z. B. eine Summenbildung) und überlässt die Implementierungsdetails der Standardbibliothek,
- erfordert der Wechsel vom Single- in den Multithread-Betrieb nur eine simple Änderung bei der Erstellung des Stromobjekts (z. B. die Änderung eines Methodennamens). Dieser Wechsel muss allerdings mit Bedacht geschehen, weil er auch zu einer verschlechterten Performanz und sogar zu einem fehlerhaften Programm führen kann (Bloch 2018, S. 225).

Im folgenden Programm werden die Elemente eines Arrays summiert:

- zunächst mit der vertrauten externen Iteration per **for**-Schleife
- anschließend mit der Stromoperation **sum()**

Quellcode	Ausgabe

<pre>import java.util.Arrays;
 	
public class Prog {
 	
    public static void main(String[] args) {
        int[] daten = {2, 4, 5, 7, 8, 11};
 	
        // Externe Iteration
        int sumex = 0;
        for(int wert : daten)
            sumex += wert;
 	
        // Interne Iteration
        int sumint = Arrays.stream(daten).sum();
 	
        System.out.println(sumex+"\n"+sumint);
    }
}</pre>
	<pre>37
	37</pre>

In der strombasierten Lösung ist vom Initialisieren und vom wiederholten Verändern einer Summenvariablen nichts zu sehen, sodass Aufwand und Fehlergefahr entfallen.

Spätestens bei der Parallelisierung ist die traditionelle Technik hoffnungslos unterlegen, weil die moderne Konkurrenz im Beispiel dazu lediglich einen zusätzlichen Methodenaufruf benötigt, der aus dem seriell arbeitenden Strom einen parallel arbeitenden erstellt:

```
int sumint = Arrays.stream(daten)
                   .parallel()
                   .sum();
```

Eine das Interface **Collection<E>** implementierende Kollektion bietet bei der Erstellung eines Stromobjekts die Wahl zwischen der Methode **stream()**, die einen seriell arbeitenden Strom liefert, und der Methode **parallelStream()**, die einen parallel arbeitenden Strom erstellt. Wegen der unvermeidlichen Fixkosten einer Multithreading-Lösung wird im konkreten Beispiel (mit der Summe 37) allerdings der parallele Strom deutlich *mehr* Zeit benötigen als der serielle.

12.2.3 Eigenschaften von Strömen

12.2.3.1 Datentyp der Elemente

Java 8 besitzt im Paket **java.util.stream** folgende Schnittstellen, die das Verhalten von Strom-Objekten definieren:

- Die generische Schnittstelle **Stream<T>** für Elemente mit einem Referenztyp
- Die Schnittstellen **IntStream**, **LongStream** und **DoubleStream** für Elemente vom primitiven Typ **int**, **long** bzw. **double**

Die vier Schnittstellen verfügen zwar über analoge Methoden, doch bestehen etliche Unterschiede.

Die Implementationen der Schnittstellen mit einem primitiven Elementtyp arbeiten performanter als vergleichbare Ströme mit einer Verpackungsklasse als Elementtyp (**Stream<Integer>**, **Stream<Long>**, **Stream<Double>**). Außerdem enthalten die drei Schnittstellen für Ströme mit primitiven Elementen bequeme Methoden, die für einen Strom statistische Kennwerte wie die Summe oder den Mittelwert durch einen einfachen Aufruf liefern (z. B. **average()** oder **sum()**, siehe Abschnitt 12.2.5.4.3).

12.2.3.2 Sequentiell oder parallel

Die Ströme ...

- beschränken sich entweder auf die sequentielle Ausführung von Operationen in einem einzigen Thread
- oder versuchen, eine Operation nach Möglichkeit in Teilaufgaben zu zerlegen, die parallel in mehreren Threads ausgeführt werden können, um später die Ergebnisse zusammenzuführen.

Weil die CPUs in moderner Computer-Hardware (ob Desktop-System, Smartphone oder Server) mehrere (virtuelle) Prozessorkerne besitzen (ca. 2 bis 16), sind *parallele* Ströme von hoher Relevanz für die Entwicklung von leistungsfähigen Anwendungen. Die für Multithreading typische Komplexität mit dem Risiko von Programmierfehlern (siehe Kapitel 15) lässt sich mit Hilfe der Java 8 - Ströme manchmal vermeiden, weil die kritischen Aufgaben von der Standardbibliothek übernommen werden.

Bei parallelen Stromoperationen kommt im Hintergrund das *Fork-Join* - Framework zum Einsatz, das Sie im Kapitel 15 über Multithreading kennenlernen werden.

Für den Programmierer verbleibt auf jeden Fall die Ermessensentscheidung für oder gegen den Einsatz der Parallelisierung. Mehrere Threads zu starten, zu koordinieren und deren Ergebnisse zusammen zu führen, verursacht einen unvermeidlichen Aufwand, der sich bei kleinen Problemen nicht lohnt.

Zudem kann die vollautomatische Parallelisierung mit Hilfe von Java 8 - Strömen bei allzu optimistischer (unbedachter) Anwendung scheitern und zu Programmen führen, die nicht nur eine miserablen Performanz besitzen, sondern eventuell sogar fehlerhafte Ergebnisse liefern (Bloch 2018, 222ff). Im Abschnitt 12.2.5.4.2 wird allerdings ein Beispiel für die erfolgreiche Anwendung von paralleler Stromverarbeitung präsentiert.

12.2.4 Erstellung von Stromobjekten

In diesem Abschnitt werden verschiedene Optionen zur Erstellung von Stromobjekten vorgestellt.

12.2.4.1 Stromobjekt aus einer Kollektion (stream, parallelStream)

Im Interface **Collection<E>** (siehe Abschnitt 10.2.1) sind zum Erstellen eines (parallelen) Stroms die Instanzmethoden **stream()** und **parallelStream()** vorhanden (mit **default**-Implementationen), z. B.:

```
List<String> als = Arrays.asList("Rudolf","Emma","Otto","Agnes","Kurt");
Stream<String> sos = als.stream();
Stream<String> psos = als.parallelStream();
```

In diesem Beispiel wird zunächst die im Abschnitt 10.1 vorgestellte statische und generische Fabrikmethode **asList()** der Klasse **Arrays** dazu verwendet, eine **List<String>** - Kollektion zu erstellen:

public static <T> List<T> asList(T... *a*)

12.2.4.2 Stromobjekt aus einem Array oder aus einer Serie von Werten (stream, of)

Um einen sequentiellen Strom aus einem Array zu erstellen, kann man die in diversen Überladungen vorhandene statische Methode **stream()** aus der Klasse **Arrays** verwenden, z. B.:

```
Stream<String> sos = Arrays.stream(new String[] {"Emma", "Otto", "Kurt"});
IntStream is = Arrays.stream(new int[] {1, 4, 14, 39});
```

Bei einer kleinen Serie von Werten ist die in allen **Stream**-Interfaces vorhandene statische Fabrikmethode **of**(), die einen Serienparameter besitzt, besonders bequem einzusetzen, z. B.:

```
IntStream is = IntStream.of(1, 4, 14, 39);
```

Über die in allen **Stream**-Interfaces vorhandene Methode **parallel()** lässt sich indirekt auch ein paralleler Strom aus einem Array erstellen, z. B.:

```
IntStream paris = Arrays.stream(new int[] {1, 4, 14, 39}).parallel();
```

12.2.4.3 Stromobjekte mit einer Sequenz ganzer Zahlen (range, rangeClosed)

In den Schnittstellen **IntStream** und **LongStream** ermöglichen die statischen Methoden **range()** und **rangeClosed()** das bequeme Erstellen von Strömen bestehend aus einer Sequenz mit ganzen Zahlen von einem Start- bis zu einem Endwert. Der einzige Unterschied zwischen den beiden Methoden besteht darin, dass der Endwert bei **range()** ausgeschlossen und bei **rangeClosed()** eingeschlossen ist, was im folgenden Programm demonstriert wird:

Quellcode

```
import java.util.stream.IntStream;

class Prog {
    public static void main(String[] args) {
        long summe = IntStream.range(1,3).sum();
        System.out.println("Summe: "+summe);

        summe = IntStream.rangeClosed(1,3).sum();
        System.out.println("Summe: "+summe);
    }
}
```

Ausgabe

```
Summe: 3
Summe: 6
```

Die Methode **sum()** liefert bei Strömen vom Typ **IntStream**, **LongStream** oder **DoubleStream** die Summe der Elemente (siehe Abschnitt 12.2.5.4.3 zu weiteren Methoden für Ströme mit primitiven Elementen).

12.2.4.4 Unendliche Ströme (iterate, generate)

Die statischen Methoden **iterate()** und **generate()** in den Strom-Schnittstellen können einen endlosen Strom produzieren.

An die folgenden **iterate()** - Überladungen

static <T> Stream<T> iterate(T *seed*, **UnaryOperator<T>** *f*)
static IntStream iterate (int *seed*, **IntUnaryOperator** *f*)
static LongStream iterate(long *seed*, **LongUnaryOperator** *f*)
static DoubleStream iterate (double *seed*, **DoubleUnaryOperator** *f*)

übergibt man einen Startwert mit dem Parameternamen *seed* sowie eine Funktion mit dem Parameternamen *f*, die durch iterative Anwendung die Stromelemente produziert:

$$seed, f(seed), f(f(seed)), \ldots$$

Im folgenden Beispiel resultiert ein Objekt vom Typ **IntStream** mit den Zweierpotenzen als Elementen:

```
IntStream ins = IntStream.iterate(1, i -> 2*i).limit(11);
```

Per **limit**() wird der Strom auf die ersten 11 Elemente (2^0 bis 2^{10}) beschränkt (vgl. Abschnitt 12.2.5.3.1).

In der Schnittstelle **IntStream** erwartet die **iterate()** - Methode als zweiten Parameter ein Objekt vom Typ **IntUnaryOperator**. Es beherrscht die Methode **applyAsInt()**, die für einen **int**-wertigen Parameter eine Rückgabe vom selben Typ liefert. Im Beispiel wird der **IntUnaryOperator** per Lambda-Ausdruck implementiert.

Seit Java 9 ist eine **iterate()** - Überladung mit *drei* Parametern verfügbar, z. B. im Interface **IntStream**:

static IntStream iterate(int *seed*, **IntPredicate** *hasNext*, **IntUnaryOperator** *next***)**

Im Vergleich zur oben beschriebenen Überladung wird über das Parameterobjekt *hasNext* eine Fortsetzungsbedingung geprüft und bei negativem Ergebnis die Produktion von Stromelementen eingestellt.

Eine einfache Anwendung von **generate()** besteht darin, einen konstanten Strom mit Einsen zu produzieren, z. B.:

```
IntStream ins = IntStream.generate(() -> 1).limit(100);
```

12.2.4.5 Sonstige Erstellungsmethoden (lines)

In der Standardbibliothek sind noch weitere Methoden in der Lage, ein **Stream**-Objekt abzuliefern. Ein Beispiel ist die statische Methode **lines()** der Klasse **Files** im Paket **java.nio.file**, die ihrem Aufrufer ein Objekt der Klasse **Stream<String>** liefert, das die Verarbeitung der Zeilen in einer Textdatei erleichtert. Im folgenden Programm werden mit der Stromoperation **count()** (siehe Abschnitt 12.2.5.4.3) die Zeilen in der Datei gezählt:

Quellcode	Ausgabe

<pre>import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;
 	
class Prog {
  public static void main(String[] args) {
    try (Stream<String> sol = Files.lines(Paths.get("test.txt"))) {
      System.out.println("Anzahl der Zeilen: " + sol.count());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}</pre>
	`Anzahl der Zeilen: 5`

Die im Beispiel verwendete **try-with-resources** - Anweisung wurde im Abschnitt 11.8.2 vorgestellt.

12.2.5 Stromoperationen

Java bietet seit der Version 8 viele aus funktionalen Programmiersprachen (z. B. Haskell, Clojure, Scala) bekannte Operationen zur Listenbearbeitung. Die beteiligten Schnittstellen **Stream<T>**, **IntStream**, **LongStream** und **DoubleStream** im Paket **java.util.stream** enthalten ähnliche, aber in Details doch abweichende Methoden bzw. Operationen (siehe API-Dokumentation). Die wesentliche Funktionserweiterung für die Software-Entwicklung mit Java besteht darin, dass Datenbank-artige Operationen (z. B. Filtern, Gruppieren, Auswerten) mit Kollektionsobjekten möglich werden, wobei die Listenbearbeitung in Vordergrund steht (Urma 2014).

12.2.5.1 *Intermediäre und terminale Stromoperationen*

Die in den Strom-Schnittstellen (**Stream<T>**, **IntStream**, **LongStream**, **DoubleStream**) definierten Methoden (Stromoperationen) lassen sich in zwei Kategorien einteilen:

- **Intermediäre Operationen**
  Intermediäre Operationen liefern ein neues Stromobjekt als Rückgabe, sodass sie hintereinander gekoppelt werden können. Wichtige Beispiele sind:
  - **filter()**
    Im resultierenden Strom sind nur noch die Elemente enthalten, die eine Bedingung erfüllen (siehe Abschnitt 12.2.5.3.1).
  - **map()**
    Die Elemente des neuen Stroms entstehen durch elementweise Abbildung der Elemente des alten Stroms, wobei sich auch der Elementtyp ändern kann (siehe Abschnitt 12.2.5.3.3).
  - **sorted()**
    Der neue Strom entsteht aus dem alten durch das Sortieren der Elemente (siehe Abschnitt 12.2.5.3.4).
  - **distinct()**
    Der neue Strom entsteht aus dem alten durch das Entfernen von Dubletten (siehe Abschnitt 12.2.5.3.1).

  Die in einer Pipeline hintereinander gekoppelten intermediären Operationen verbleiben in Wartestellung, bis eine *terminale* Operation ausgeführt wird. Dann laufen alle Operationen in der Pipeline ab. Dank dieser als *lazy* (dt.: *faul*) bezeichneten Arbeitsweise sind Optimierungen möglich (siehe Abschnitt 12.2.5.2).
- **Terminale Operationen**
  Terminale Operationen liefern ein Ergebnis, das kein Strom ist (z. B. eine Zahl oder eine Liste). Wichtige Beispiele sind:
  - **reduce()**
    Die Elemente im Strom werden durch iterative Anwendung einer binären Operation auf einen Wert reduziert (z. B. auf eine Zahl). So kann man z. B. aus einem Strom mit den natürlichen Zahlen von 1 bis *K* durch iterative Multiplikation die Fakultät von *K* berechnen (siehe Abschnitt 12.2.5.4.2).
  - **average()**
    Für einen Strom mit Elementen vom Typ **int**, **long** oder **double** erhält man den Durchschnittswert (siehe Abschnitt 12.2.5.4.3).
  - **collect()**
    Aus dem Strom kann man z. B. eine Liste oder eine Abbildung erstellen (siehe Abschnitt 12.2.5.4.4).

  Nach der Ausführung einer terminalen Operation sind die Stromobjekte in der Pipeline verbraucht und können keine weiteren Operationen mehr ausführen. Um aus der Quelle ein weiteres Ergebnis zu ermitteln, muss eine neue Pipeline aufgebaut werden.

Bei den intermediären Operationen unterscheidet man:

- **Zustandslose Operationen**
  Jedes Element kann unabhängig von allen anderen verarbeitet werden (Beispiele: **filter()**, **map()**). Sind in einer Pipeline alle intermediären Operationen zustandslos, ist (bei serieller oder paralleler) Verarbeitung nur *ein* Durchlauf erforderlich.

- **Zustandsbehaftete Operationen**
  Bei der Verarbeitung eines Elementes muss eventuell der Zustand von früher verarbeiteten Elementen berücksichtigt werden (Beispiele: **distinct**(), **sorted**()). Enthält eine Pipeline zustandsbehaftete intermediäre Operationen, sind bei paralleler Verarbeitung eventuell mehrere Durchläufe oder eine Speicherung von Zwischenergebnissen erforderlich.

12.2.5.2 Faulheit ist nicht immer dumm

Intermediäre Operationen werden erst dann ausgeführt, wenn es sich nicht weiter aufschieben lässt, weil für die zugehörige Pipeline eine terminale Operation angefordert worden ist. Dank dieser als *lazy* (dt.: *faul*) bezeichneten Arbeitsweise sind folgende Optimierungen möglich:

- Ausführung von mehreren Operationen bei *einer* Datenpassage
  Nach Möglichkeit werden mehrere Operationen bei *einer einzigen* Datenpassage erledigt. Das spart Zeit im Vergleich zu mehreren, nacheinander ausgeführten Iterationen.
- Einschränkung von Operationen auf tatsächlich betroffene Elemente
  Wenn z. B. eine spätere Operation den Strom auf die ersten 10 Elemente begrenzt, werden auch die früheren Operationen (z. B. Abbildungen) nur für die ersten 10 Elemente ausgeführt. Bei den Stromoperationen findet also eine Kurzschlussauswertung statt (engl.: *short-circuiting*), vergleichbar zum Verhalten der logischen Operatoren && und || (vgl. Abschnitt 3.5.5).

Im folgenden Beispielprogramm (nach einer Idee von Urma 2014) soll ausgehend von einer Liste mit Namen eine neue Liste erstellt werden, welche die beiden ersten Namen mit Mindestlänge 5 in Großbuchstaben enthält.

Quellcode

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

class LacyOp {
  public static void main(String[] args) {
    List<String> als = Arrays.asList("Rudolf","Emma","Otto",
                                      "Agnes","Kurt","Walter");
    List<String> f2ge5 = als.stream()
         .filter(s -> {
            System.out.println("Filtern von " + s);
            return s.length() >= 5;})
         .map(s -> {
            System.out.println(" Abbilden von " + s);
            return s.toUpperCase();})
         .limit(2)
         .collect(Collectors.toList());
    System.out.println("\n" + f2ge5);
  }
}
```

Ausgabe

```
Filtern von Rudolf
 Abbilden von Rudolf
Filtern von Emma
Filtern von Otto
Filtern von Agnes
 Abbilden von Agnes

[RUDOLF, AGNES]
```

Die Ausgabe zeigt,

- dass für die Fälle mit positivem Filterergebnis (Rudolf, Agnes) die Operationen **filter**() und **map**() gemeinsam (bei *einer* Datenpassage) ausgeführt worden sind,
- dass für die Fälle mit negativem Filterergebnis (Emma, Otto) keine Abbildung vorgenommen wurde,
- dass nach dem Vorliegen von zwei positiven Fällen keine weiteren (überflüssigen) Operationen mehr ausgeführt wurden (für Kurt, Walter).

Zum Erstellen eines neuen Objekts vom Typ **List<String>** dient die terminale Operation **collect()** (siehe Abschnitt 12.2.5.4.4).

12.2.5.3 Intermediäre Operationen

12.2.5.3.1 Filtern der Elemente (filter)

Die Stromoperation **filter()** liefert einen neuen Strom bestehend aus allen Elementen des alten Stroms, die einen Test bestanden haben. Sie benötigt als Parameter ein Funktionsobjekt vom Interface-Typ **Predicate<T>**, das eine Methode namens **test()** mit einem booleschen Rückgabewert zur Beurteilung eines einzelnen Stromelements beherrscht:

public boolean test(T *value***)**

Der Konkretheit halber betrachten wir anschließend das Interface **Stream<String>**. Hier verlangt die **filter()** - Methode ein Parameterobjekt vom Typ **Predicate<? super String>**:

public Stream<String> filter(Predicate<? super String> *predicate***)**

Die Verwendung des gebundenen Wildcard-Datentyps (vgl. Abschnitt 8.3.1.2) für den Parameter stellt eine Liberalisierung im Vergleich zum Datentyp **Predicate<String>** dar. Neben einer **test()** - Methode mit dem Parametertyp **String** sind daher auch Methoden mit einem generelleren Parametertyp erlaubt.

Im folgenden Programm werden aus einem Strom vom Typ **Stream<String>** alle Elemente mit genau vier Zeichen in einen neuen Strom vom selben Typ geleitet. Anschließend wird mit der terminalen Operation **count()** (siehe Abschnitt 12.2.5.4.3) die Anzahl der Elemente im neuen Strom ermittelt.

Quellcode:

```
import java.util.Arrays;
import java.util.List;

class Filter {
  public static void main(String[] args) {
    List<String> als = Arrays.asList("Rudolf","Emma","Otto",
                                     "Agnes","Kurt","Walter");
    long n4 = als.stream()
        .filter(s -> s.length() == 4)
        .count();
    System.out.println(n4);
  }
}
```

Ausgabe:

```
3
```

Das benötigte Objekt vom Typ **Predicate<? super String>** wird per Lambda-Ausdruck realisiert:

```
s -> s.length() == 4
```

Neben **filter()** liefern noch weitere intermediäre Stromoperationen einen neuen Strom mit den Elementen des alten Stroms, die einen Test überstanden haben:

- **distinct()**
  Man erhält einen neuen Strom ohne Dubletten. Im folgenden Code-Segment wird der resultierende Strom mit der terminalen Operation **collect()** (siehe Abschnitt 12.2.5.4.4) in eine Liste geleitet:

```
List<Integer> ali = Arrays.asList(1, 1, 2, 3, 3, 4, 5, 5);
List<Integer> alin = ali.stream()
      .distinct()
      .collect(Collectors.toList());
```

- **limit(long *n*)**
  Man erhält einen neuen Strom mit den ersten *n* Elementen des alten Stroms.
- **skip(long *n*)**
  Im neuen Strom fehlen die ersten *n* Elemente des alten Stroms.

12.2.5.3.2 Eine Startsequenz zulassen oder ausschließen (takeWhile, dropWhile)

Beim Filtern wird *jedes* Element des Eingangsstroms daraufhin überprüft, ob es in den Ergebnisstrom gelangen soll. Wenn stattdessen bei einem Eingangsstrom mit definierter Anordnung der Elemente (z. B. Sequenz von Qualitätsmessungen aus einer Produktionsanlage) die Zulässigkeitsprüfung beendet werden soll, sobald erstmals über ein Element negativ entschieden wird, dann bietet sich die in Java 9 eingeführte zustandsbehaftete intermediäre Operation **takeWhile()** an. Der Ergebnisstrom enthält die Startsequenz aus dem Eingangsstrom bis zum letzten positiv beurteilten Element, wobei auch ein leerer Ergebnisstrom entstehen kann.

Die **default**-Methode **takeWhile()** ist sowohl in der generischen Schnittstelle **Stream<T>** als auch in den Schnittstellen **IntStream**, **LongStream** und **DoubleStream** für Elemente vom primitiven Typ **int**, **long** bzw. **double** vorhanden, z. B.:

default Stream<T> takeWhile (Predicate<? super T> *predicate***)**

Im folgenden Beispiel enthält der Eingabestrom eine Serie von ganzen Zahlen. Die Weiterleitung der Eingabewerte in den Ergebnisstrom stoppt, sobald erstmals ein Wert über 50 auftritt:

Quellcode

```
import java.util.stream.IntStream;

class TakeWhile {
  public static void main(String[] args) {
    IntStream instr = IntStream.of(48, 8, 35, 52, 82, 24);
    IntStream whileLE50 = instr.takeWhile(i -> i <= 50);
    System.out.println("\nBis zur ersten"+"\nBeobachtung > 50");
    whileLE50.forEach(System.out::println);
  }
}
```

Ausgabe

```
Bis zur ersten
Beobachtung > 50
48
8
35
```

Es ist zu beachten, dass der Wert 24 *nicht* in den Ergebnisstrom gelangt, obwohl er das Take-Kriterium erfüllt, weil nach dem ersten negativen Prüfungsergebnis (52 > 50) keine weitere Beurteilung mehr stattfindet.

Sollen aus einem Eingangsstrom alle Elemente von der Weiterleitung in den Ergebnisstrom ausgeschlossen werden, bis erstmals ein Wert mit einem positiven Prüfungsergebnis auftritt, dann bietet sich (als Gegenstück zu **takeWhile()**) die ebenfalls in Java 9 eingeführte zustandsbehaftete intermediäre Operation **dropWhile()** an. Der Ergebnisstrom enthält die Startsequenz aus dem Eingangsstrom ab dem ersten positiv beurteilten Element, wobei auch ein leerer Ergebnisstrom entstehen kann.

Die **default**-Methode **dropWhile()** ist sowohl in der generischen Schnittstelle **Stream<T>** als auch in den Schnittstellen **IntStream**, **LongStream** und **DoubleStream** für Elemente vom primitiven Typ **int**, **long** bzw. **double** vorhanden, z. B.:

default Stream<T> dropWhile (Predicate<? super T> *predicate***)**

Im folgenden Beispiel enthält der Eingabestrom die schon im letzten Beispiel verwendete Serie von ganzen Zahlen. Diesmal startet die Weiterleitung der Eingabewerte in den Ergebnisstrom, sobald erstmals ein Wert über 50 auftritt:

Quellcode

```java
import java.util.stream.IntStream;

class DropWhile {
  public static void main(String[] args) {
    IntStream instr = IntStream.of(48, 8, 35, 52, 82, 24);
    IntStream droppedLE50 = instr.dropWhile(i -> i <= 50);
    System.out.println("\nAb der ersten"+"\nBeobachtung > 50");
    droppedLE50.forEach(System.out::println);
  }
}
```

Ausgabe

```
Ab der ersten
Beobachtung > 50
52
82
24
```

Es ist zu beachten, dass der Wert 24 in den Ergebnisstrom gelangt, obwohl er das Drop-Kriterium erfüllt, weil nach dem ersten positiven Prüfungsergebnis (52 > 50) keine weitere Kontrolle mehr stattfindet.

Wenn für die Elemente im Eingangsstrom keine Anordnung definiert ist (z. B. nach der Erstellung eines Stroms aus einer Menge), dann liefern **takeWhile()** und **dropWhile()** kein sinnvolles Ergebnis.

12.2.5.3.3 Elementweise Abbildung (map)

Mit der generischen Methode **map()** aus dem Interface **Stream<T>** gewinnt man einen neuen Strom mit Abbildungs-Elementen, die aus den Urbild-Elementen im alten Strom durch eine elementweise Abbildung entstehen:

public <R> Stream<R> map(Function<? super T, ? extends R> *mapper*)

Das für die Abbildung zuständige Parameterobjekt muss das generische funktionale Interface **Function<T,R>** aus dem Paket **java.util.function** implementieren, das die folgende Methode mit einem Parameter vom Typ **T** und einer Rückgabe vom Typ **R** verlangt:

public R apply(T *t*)

In der Definition der generischen Methode **map()** steht **R** für den Elementtyp des Rückgabestroms, und **T** steht für den Elementtyp des angesprochenen Stroms (Typ **Stream<T>**). Für das **map()** - Parameterobjekt ist es in Ordnung, wenn die von ihm beherrschte **apply()** - Methode ...

- den Parameter **T** oder einen generelleren Typ akzeptiert,
- eine Rückgabe vom Typ **R** oder einem spezielleren Typ liefert.

Im folgenden Beispiel entsteht aus einem Strom mit Elementen vom Typ **String** ein neuer Strom mit Elementen vom Typ **Integer**, der für jedes Element des alten Stroms die Anzahl der Zeichen enthält, wobei das **map()** - Parameterobjekt auf einer Methodenreferenz basiert (siehe Abschnitt 12.1.3.1). Für die Ausgabe der **String**-Längen sorgt die terminale Operation **forEach()** (siehe Abschnitt 12.2.5.4.1).

Quellcode	Ausgabe

<pre>import java.util.Arrays;
class Mapping {
   public static void main(String[] args) {
      Arrays.asList("Rudolf","Emma","Otto","Agnes","Kurt")
         .stream()
         .map(String::length)
         .forEach(i -> System.out.print(i+" "));
   }
}</pre>
	6 4 4 5 4

Neben der generischen Methode **map()** für Ergebnisströme mit *Objekten* als Elementen existieren im Interface **Stream<T>** noch Methoden für Ergebnisströme mit primitivem Elementtyp. Wenn für die Namensliste im letzten Beispiel die Gesamtzahl der Buchstaben interessiert, bietet es sich an, mit der Operation **mapToInt()** ein **IntStream**-Objekt zu erstellen. Mit den Elementen eines solchen Stroms sind arithmetische Operationen wie die Addition ohne (Un-)boxing möglich, was der Performanz zu Gute kommt. Außerdem existieren in den Schnittstellen für Ströme mit einem primitiven Elementtyp einige Operationen, die Stromstatistiken mit einem einfachen Aufruf liefern (vgl. Abschnitt 12.2.5.4.3). So kann man z. B. die Summe der **int**-Elemente von der Methode **sum()** ermitteln lassen, was im folgenden Beispiel geschieht:

Quellcode	Ausgabe

<pre>import java.util.Arrays;
 	
class Mapping {
  public static void main(String[] args) {
    int n = Arrays.asList("Rudolf","Emma","Otto","Agnes","Kurt")
      .stream()
      .mapToInt(String::length)
      .sum();
    System.out.println("\nSumme: "+n);
  }
}</pre>
	Summe: 23

In den Schnittstellen für Ströme mit primitivem Elementtyp (z. B. **IntStream**) befinden sich ...

- die Methode **map()** für einen Ergebnisstrom mit demselben Elementtyp
- Methoden für Ergebnisströme mit einem anderen primitiven Elementtyp (z. B. **mapToDouble()**)
- die Methode **mapToObj()** für einen Ergebnisstrom mit Objekten als Elementen

12.2.5.3.4 Sortieren (sorted)

Mit der Methode **sorted()** aus dem Interface **Stream<T>** gewinnt man einen neuen Strom mit den gemäß ihrer natürlichen Ordnung sortierten Elementen des angesprochenen Stroms, wobei der Datentyp der Elemente das Interface **Comparable<T>** erfüllen muss.

Im folgenden Beispiel entsteht aus einem Strom mit Elementen vom Typ **String** ein aufsteigend sortierter Strom mit denselben Elementen:

Quellcode	Ausgabe

<pre>import java.util.Arrays;
 	
class Sorted {
   public static void main(String[] args) {
      Arrays.asList("Rudolf","Emma","Otto","Agnes","Kurt")
         .stream()
         .sorted()
         .forEach(System.out::println);
   }
}</pre>
	<pre>Agnes
	Emma
	Kurt
	Otto
	Rudolf</pre>

Die Schnittstellen **IntStream**, **LongStream** und **DoubleStream** für primitive Elementtypen enthalten jeweils eine analoge arbeitende **sorted()** - Methode.

Das Interface **Stream<T>** enthält zusätzlich eine **sorted()** - Überladung mit einem Parameter vom Typ **Comparator<? super T>**, sodass sich ein alternatives Sortierkriterium realisieren lässt:

public Stream<T> sorted(Comparator<? super T>)

Im folgenden Beispiel entsteht aus einem Strom mit Elementen vom Typ **String** ein *ab*steigend sortierter Strom mit denselben Elementen, wobei der **Comparator** per Lambda-Ausdruck realisiert wird:

Quellcode	Ausgabe

<pre>import java.util.Arrays;
 	
class Sorted {
   public static void main(String[] args) {
      Arrays.asList("Rudolf","Emma","Otto","Agnes","Kurt")
      .stream()
      .sorted((s1, s2) -> s2.compareTo(s1))
      .forEach(System.out::println);
   }
}</pre>
	<pre>Rudolf
	Otto
	Kurt
	Emma
	Agnes</pre>

12.2.5.3.5 Zwischenberichte (peek)

Veranlasst man eine Ausgabe der Stromelemente über die im Abschnitt 12.2.5.4.1 zu beschreibende terminale Stromoperation **forEach()**, ist die gesamte Pipeline anschließend verbraucht und inoperabel, was die Fehlersuche erschwert. Daher stellen die Strom-Schnittstellen für Diagnosezwecke die Operation **peek()** zur Verfügung:

public Stream<T> peek(Consumer<? super T> *action***)**

Man erhält als Rückgabe einen Strom mit den Elementen des angesprochenen Stroms und kann außerdem über das Parameterobjekt eine elementweise durchzuführende Aktion vereinbaren (z. B. eine Protokollausgabe). Die **peek()** - Aktion wird wie jede andere intermediäre Operation ausgeführt, wenn die terminale Operation der Pipeline ansteht. Im folgenden Beispiel erfolgt für einen Strom mit den Zahlen von 1 bis 4 eine erste Kontrollausgabe unmittelbar hinter der Quelle. Nachdem die ungeraden Zahlen ausgefiltert worden sind, erfolgt eine erneute Kontrollausgabe:

Quellcode	Ausgabe

`import java.util.stream.IntStream;`
 	
`class Peek {`
`   public static void main(String[] args) {`
`      int sum = IntStream.rangeClosed(1, 4)`
`         .peek(System.out::println)`
`         .filter(i-> i%2 == 0)`
`         .peek(i-> System.out.println(" filtered: "+i))`
`         .sum();`
`      System.out.println("\nSumme: "+sum);`
`   }`
`}`
	`1`
	`2`
	` filtered: 2`
	`3`
	`4`
	` filtered: 4`
	
	`Summe: 6`

Gemäß der ökonomischen Stromverarbeitungslogik nach dem „Lazy"-Prinzip werden die ungeraden Zahlen nur einmal, die geraden Zahlen hingegen zweimal nacheinander protokolliert (siehe Abschnitt 12.2.5.2).

Die Strommethode **peek()** ist wie **forEach()** (vgl. Abschnitt 12.2.5.3.3) ein sogenannter *Nebeneffekt-Produzent*, ohne (wie **forEach()**) die Pipeline zu terminieren.

12.2.5.4 Terminale Operationen

Terminale Operationen liefern ein Ergebnis, das kein Strom ist, oder führen eine Verarbeitung für jedes einzelne Element aus. Auf jeden Fall sind die Stromobjekte in der Pipeline anschließend verbraucht und können keine weiteren Operationen mehr ausführen.

12.2.5.4.1 Elementweise Verarbeitung (forEach)

Mit der in allen Stromschnittstellen vorhandenen Methode **forEach()** sorgt man für die elementweise Verarbeitung eines Stroms. Im folgenden Beispiel werden die Elemente eines durch die **IntStream**-Methode **iterate()** erstellten Stroms bestehend aus den ersten 8 Zweierpotenzen ausgegeben (beginnend mit $1 = 2^0$), wobei das **forEach()** - Parameterobjekt auf einer Methodenreferenz basiert (siehe Abschnitt 12.1.3.1):

Quellcode	Ausgabe

`import java.util.stream.IntStream;`
 	
`class ForEach {`
`   public static void main(String[] args) {`
`      IntStream is = IntStream.iterate(1, i -> 2*i).limit(8);`
`      is.forEach(System.out::println);`
`   }`
`}`
	`1`
	`2`
	`4`
	`8`
	`16`
	`32`
	`64`
	`128`

Eine terminale Operation produziert entweder ein Ergebnis oder einen Nebeneffekt, wobei **forEach()** ein Nebeneffekt-Produzent ist. Man sollte Stromoperationen mit Nebeneffekten möglichst vermeiden und insbesondere die Operation **forEach()** ausschließlich zu Reportzwecken einsetzen (Bloch 2018, S. 210ff).

Mit der zur Fehlersuche konzipierten Stromoperation **peek()** lassen sich ebenfalls elementweise Nebeneffekte produzieren, wobei jedoch die Pipeline *nicht* terminiert wird (siehe Abschnitt 12.2.5.3.5).

12.2.5.4.2 Reduktion eines Stroms auf einen Wert durch eine assoziative Funktion (reduce)

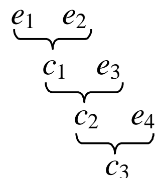
Über die Strommethode **reduce()** lässt sich eine beliebige assoziative Funktion von zwei Variablen zum Reduzieren eines Stroms verwenden. Die Funktion wird so lange iterativ auf jeweils zwei benachbarte Elemente angewendet, bis schließlich ein einzelner Wert resultiert.

Eine binäre Funktion *f* ist genau dann assoziativ, wenn für beliebige Argumente *a*, *b* und *c* gilt:

$$f(f(a, b), c) = f(a, f(b, c))$$

Es spielt also keine Rolle, ob die Funktion zuerst auf *a* und *b* oder zuerst auf *b* und *c* angewendet wird. Folglich kann die Anwendung der Methode **reduce()** auf den gesamten Strom parallelisiert werden, d.h. es ist eine parallele Ausführung durch mehrere Threads möglich. Um die Zusammenfassung der Teilergebnisse kümmert sich die Standardbibliothek.

Von **reduce()** wird die Funktion *f* zunächst auf die beiden ersten Stromelemente angewendet und dann iterativ auf das aktuelle Zwischenergebnis c_i und das aktuelle Stromelement e_j. Bei einem Strom mit den vier Elementen e_1 bis e_4 resultiert die folgende Verarbeitungskette:



Wegen der iterativen Arbeitsweise wird eine Reduktion auch als *Faltung* (engl.: *folding*) bezeichnet. Man kann sich vorstellen, dass bei einem langen Papierstreifen mit vielen Segmenten so lange das jeweils erste Segment Richtung Ende gefaltet wird, bis nur noch *ein* (ziemlich dickes) Segment übrig ist (siehe Urma 2014).

Von der anzuwendenden Funktion erfährt die Methode **reduce()** über einen Parameter vom Typ einer funktionalen Schnittstelle. Die Methode **reduce()** der generischen Schnittstelle **Stream<T>** erwartet einen Parameter vom Typ **BinaryOperator<T>**:

public Optional<T> reduce(BinaryOperator<T> *op***)**

Man erhält von **reduce()** als Rückgabe ein Objekt vom Typ **Optional<T>**. Dieses Objekt enthält nach einer erfolgreichen Stromreduktion das Ergebnis und liefert es nach Aufforderung per **get()** ab. Ob ein Wert vorhanden ist, erfährt man über die boolesche Rückgabe der Methode **isPresent()**.

Das als **reduce()** - Parameter erwartete Funktionsobjekt vom Typ **BinaryOperator<T>** muss die folgende Methode beherrschen:

public T apply(T *first*, **T** *second***)**

In der Schnittstelle **IntStream** ist eine analoge **reduce()** - Methode

public OptionalInt reduce(IntBinaryOperator *op***)**

vorgeschrieben, die als Parameter ein Funktionsobjekt vom Typ **IntBinaryOperator** mit der folgenden Methode

public int applyAsInt(int *left*, **int** *right***)**

erwartet und als Rückgabe ein Objekt vom Typ **OptionalInt** liefert. Dieses Objekt enthält nach einer erfolgreichen Stromreduktion das Ergebnis und liefert es nach Aufforderung per **getAsInt()** ab. Ob ein Wert vorhanden ist, erfährt man über die boolesche Rückgabe der Methode **isPresent()**.

Im folgenden Beispiel wird ein Strom vom Typ **IntStream** mit Elementen vom primitiven Typ **int** durch die statische **IntStream**-Methode **range()** erzeugt, die einen inklusiven Startwert und einen exklusiven Endwert vom Typ **int** erwartet und die zugehörige Sequenz von ganzen Zahlen produziert:

Quellcode	Ausgabe

<pre>import java.util.OptionalInt;
import java.util.stream.IntStream;
 	
class Reduce {
  public static void main(String[] args) {
    OptionalInt sq = IntStream.range(1,4)
                              .map(i -> i*i)
                              .reduce((c,i) -> c+i);
    System.out.println((sq.isPresent() ? sq.getAsInt() : "Fehler"));
  }
}</pre>
	14

Per **map()** - Operation mit dem Lambda-Ausdruck (`i -> i*i`) als **IntUnaryOperator** resultiert ein neuer Strom mit den quadrierten ganzen Zahlen.

Weil wir mit einem **IntStream** arbeiten, erwartet **reduce()** in der Rolle der assoziativen Funktion einen **IntBinaryOperator**, und wir liefern den Lambda-Ausdruck `((c,x) -> c+x)`.[1]

Wir betrachten noch eine zweite **reduce()** - Überladung, die im ersten Parameter das neutrale Element *z* der assoziativen Funktion im folgenden Sinn

$$f(z, a) = a$$

erwartet, sodass für den Typ **Stream<T>** der folgende Methodendefinitionskopf resultiert:

public T reduce(T *identity*, **BinaryOperator<T>** *op***)**

Diesmal erhalten wir eine Rückgabe vom Typ **T**, die bei einem leeren Strom mit dem ersten **reduce()** - Parameter identisch ist, sodass auf jeden Fall eine Rückgabe vom Typ **T** resultiert, und der Aufwand mit einer **Optional<T>** - Rückgabe entfällt.

Für eine **IntStream**-Implementation sieht der Definitionskopf der **reduce()** - Überladung mit einem Parameter für das neutrale Element so aus:

public int reduce(int *identity*, **IntBinaryOperator** *op***)**

Im nächsten Beispiel wird die Fakultät einer natürlichen Zahl mit Stromoperationen berechnet. Um dies für große Argumente zu ermöglichen, kommt der Datentyp **BigDecimal** zum Einsatz. Zunächst entsteht ein Strom vom Typ **IntStream** mit Elementen vom primitiven Typ **int** mit Hilfe der statischen **IntStream**-Methode **rangeClosed()**, die eine Sequenz ganzer Zahlen von einem Start- bis zu einem Endwert (beide inklusive) produziert. Mit der generischen **IntStream**-Methode **mapToObj()** wird der **IntStream** in einen **Stream<BigDecimal>** gewandelt. Ein Aufruf der entsprechend parametrisierten Methode kann folgendermaßen aussehen (mit expliziter Konkretisierung des Typformalparameters, vgl. Abschnitt 8.2):

```
Stream<BigDecimal> sbd = IntStream.rangeClosed(1,500)
                           .<BigDecimal>mapToObj(new IntFunction<BigDecimal>() {
                              public BigDecimal apply(int i) {
                                return new BigDecimal(i);
                              }
                            });
```

Als Aktualparameter dient ein Objekt einer anonymen Klasse, welche das funktionale Interface **IntFunction<BigDecimal>** erfüllt. Dazu implementiert die Klasse eine Methode namens **apply()**

---

[1] Weil wir gerade mit einem **IntStream** arbeiten und lediglich eine Summenbildung benötigen, steht als deutlich bequemere Alternative zur **reduce()** - Operation mit **IntBinaryOperator**-Parameter die spezielle Stromoperation **sum()** zur Verfügung (siehe Abschnitt 12.2.5.4.3). Wir verwenden trotzdem die umständlichere Lösung, um mit einem besonders einfachen **IntBinaryOperator** arbeiten zu können (Addition).

mit einem **int**-Parameter und einer Rückgabe vom Typ **BigDecimal**, die von einem Konstruktor der Klasse **BigDecimal** produziert wird. Per Konstruktorreferenz (vgl. Abschnitt 12.1.3.2) lässt sich der Aktualparameterausdruck drastisch vereinfachen, wobei dank Typinferenz auch auf die explizite Typkonkretisierung im Methodennamen verzichtet werden kann:

```
Stream<BigDecimal> sbi = IntStream.rangeClosed(1,500).mapToObj(BigDecimal::new);
```

Nach der Stromkonstruktion und -transformation kommt es zum Reduktionsschritt unter Verwendung der **reduce()** - Überladung mit einem neutralen Element im ersten und einer assoziativen Funktion im zweiten Parameter. Die Eins als neutrales Element der Multiplikation kann in der Klasse **BigDecimal** so notiert werden:

BigDecimal.ONE

Zur Multiplikation von zwei **BigDecimal**-Objekten beauftragt man den ersten Faktor mit der Methode **multiply()** und übergibt per Parameter den zweiten Faktor:

public BigDecimal multiply(BigDecimal *multiplicand***)**

Auf die recht lange Beschreibung folgt ein angenehm kurzes Programm, das den **Binary-Operator<BigDecimal>** (die assoziative Funktion) per Methodenreferenz vereinbart (vgl. Abschnitt 12.1.3.1):

Quellcode

```
import java.math.BigDecimal;
import java.util.stream.IntStream;

class Reduce {
    public static void main(String[] args) {
        final int arg = 500;
        BigDecimal fak = IntStream
                .rangeClosed(1, arg)
                .mapToObj(BigDecimal::new)
                .reduce(BigDecimal.ONE, BigDecimal::multiply);
        System.out.printf("%e", fak);
    }
}
```

Ausgabe

```
1,220137e+1134
```

Mit dem aktuellen Beispiel lässt sich demonstrieren, wie leicht die serielle Strombearbeitung auf eine parallele, mehrere Prozessorkerne nutzende Strombearbeitung umgestellt werden kann. Dazu ist lediglich mit der **IntStream**-Methode **parallel()** aus dem seriellen Strom ein paralleler Strom zu erstellen. Wir erweitern das letzte Beispielprogramm außerdem um eine Zeitmessung und erhalten:

Quellcode

```
import java.math.BigDecimal;
import java.util.stream.IntStream;

class Prog {
  public static void main(String[] args) {
  long start = System.currentTimeMillis();
  final int arg = 50_000;
  BigDecimal fak = IntStream.rangeClosed(1, arg)
    .parallel()
    .mapToObj(BigDecimal::new)
    .reduce(BigDecimal.ONE, BigDecimal::multiply);
  System.out.printf("\nFakultät von %d:\n %e\nZeit in Millisek.: %d",
                    arg, fak, System.currentTimeMillis() - start);
 }
}
```

Ausgabe

```
Fakultät von 50000:
 3,347321e+213236
Zeit in Millisek.: 309
```

Die Berechnung der Fakultät von 500 dauert nach der Parallelisierung *länger* (25 statt 10 Millisekunden). Offenbar wiegt bei dieser Problemgröße der durch Multithreading bedingte organisatori-

sche Zusatzaufwand den Gewinn durch Verwendung mehrerer Kerne mehr als auf. Zur Berechnung der Fakultät von 50.000 benötigt der parallele Strom mit 309 Millisekunden allerdings deutlich weniger Zeit als der serielle, der erst nach 1812 Millisekunden zum Ergebnis kommt. Wir stellen fest:

- Multithreading ist bei der funktionalen Strombearbeitung in Java sehr leicht zu realisieren.
- Die mit einem **int**-Range startende Beispielaufgabe ist gut parallelisierbar. Man darf die Erfahrung keinesfalls leichtfertig auf beliebige Stromverarbeitungsaufgaben übertragen.
- Weil die Erstellung und Koordination von mehreren Threads Zeitaufwand verursacht, entscheidet die Problemgröße darüber, ob ein Nutzen zu erzielen ist. Im Beispiel fällt die Geschwindigkeitssteigerung sogar stärker aus, als es bei vier logischen Prozessoren zu erwarten war. Ohne Kenntnis der jeweils realisierten Algorithmen ist allerdings keine exakte Gewinnerwartung zu berechnen.

12.2.5.4.3 Statistische Berechnungen ausführen (count, min, max, sum, average)

Für einige Reduktionsaufgaben (z. B. Ermittlung des maximalen Elements) sind spezielle Stromoperationen verfügbar, die im Vergleich zu der parametrisierbaren Methode **reduce()** einfacher handhabbar sind.

Von jedem beliebigen Strom kann man über die Methode **count()** die Anzahl seiner Elemente erfahren, z. B.:

Quellcode-Segment	Ausgabe
`IntStream isp = IntStream.of(1, 4, 10, 20);` `System.out.println(isp.count());`	4

Ströme, die einen primitiven Elementtyp besitzen, also das Interface **IntStream**, **LongStream** oder **DoubleStream** implementieren, beherrschen Operationen zur Berechnung statistischer Kennwerte:

- **sum(), average()**
  Man erhält die Summe bzw. den Mittelwert der Stromelemente, z. B.:

Quellcode-Segment	Ausgabe
`IntStream isp = IntStream.of(1, 4, 10, 20);` `System.out.println(isp.sum());`	35

- **min(), max()**
  Diese Methoden liefern das kleinste bzw. größte Element, z. B.:

Quellcode-Segment	Ausgabe
`IntStream isp = IntStream.of(1, 4, 10, 20);` `System.out.println(isp.max());`	OptionalInt[20]

Auch im Interface **Stream<T>** für Ströme mit Referenzelementtyp sind Methoden **max()** und **min()** zur Ermittlung des größten bzw. kleinsten Elements vorhanden, wobei ein Parameterobjekt vom Typ **Comparator<? super T>** zu übergeben ist.

12.2.5.4.4 Stromelemente in einer Kollektion sammeln (collect)

Die in allen Stromschnittstellen vorhandene Methode **collect()** erstellt aus einem Strom eine Kollektion und benötigt dazu folgende Informationen:

- Wie wird ein Objekt vom gewünschten Kollektionstyp **R** erzeugt?
  Der erste Parameter von **collect()** muss das Interface **Supplier<R>** implementieren, das eine parameterfreie Methode namens **get()** vorschreibt. Es wird erwartet, dass **get()** als Rückgabe eine Kollektion vom Typ **R** liefert.
- Wie wird ein Element in die Kollektion eingefügt?

Der zweite Parameter von **collect()** muss das Interface **BiConsumer<R, ? super T>** implementieren, das eine Methode namens **accept()** mit dem Rückgabetyp **void** vorschreibt. Diese muss als ersten Parameter den Typ der Kollektion und als zweiten Parameter den Elementtyp des Stroms oder eine Verallgemeinerung akzeptieren. Es wird erwartet, dass mit **accept()** ein Stromelement in die Kollektion eingefügt werden kann.

- Wie werden *alle* Elemente einer Kollektion in ein typgleiches Kollektionsobjekt eingefügt? Diese Operation ist relevant, wenn bei paralleler Stromverarbeitung mehrere Zwischenergebnisse entstehen, die vereinigt werden müssen. Der dritte Parameter von **collect()** muss das Interface **BiConsumer<R, R>** implementieren, und diesmal wird erwartet, dass per **accept()** alle Elemente einer Kollektion vom Typ **R** in ein Kollektionsobjekt desselben Typs aufgenommen werden können.

Im folgenden Beispiel wird der aus einer **String**-Liste entstandene Strom per **distinct()** von Dubletten befreit, per **sorted()** aufsteigend sortiert und schließlich per **collect()** in eine neue Liste überführt:

Quellcode

```java
import java.util.*;

public class Collect {
  public static void main(String[] args) {
    List<String> als = Arrays.asList("Charly","Anton","Berta","Ben",
                                     "Clara","Anton","Anette","Charly");
    List<String> als2 = als.stream()
        .distinct()
        .sorted()
        .collect(ArrayList<String>::new,
                 ArrayList<String>::add,
                 ArrayList<String>::addAll);
    for(String s: als2)
      System.out.println(s);
  }
}
```

Ausgabe

```
Anette
Anton
Ben
Berta
Charly
Clara
```

Als **collect()** - Parameter fungieren Konstruktor- bzw. Methodenreferenzen (siehe Abschnitt 12.1.3). Der Konstruktor im ersten Parameter taugt zur Produktion eines **ArrayList<String>** - Kollektionsobjekts. Als zweiter Parameter wird die Instanzmethode **add()** der Klasse **ArrayList<String>** übergeben, die einen Parameter vom Typ **String** erwartet und die Rolle der zu implementierenden Schnittstellenmethode **accept()** spielen kann:

- Der erste **accept()** - Parameter (also das Kollektionsobjekt) wird zum Ansprechpartner für den Aufruf der Instanzmethode **add()**.
- Der zweite **accept()** - Parameter (also das Stromelement) wird an **add()** übergeben.

Als dritter **collect()** - Parameter wird die Instanzmethode **addAll()** der Klasse **ArrayList<String>** übergeben, die einen Parameter vom Typ **ArrayList<String>** erwartet. Auch hier klappt offenbar die Zuordnung der **accept()** - Parameter, die beide vom selben Kollektionstyp sein müssen.

Zu der etwas umständlichen **collect()** - Methode mit drei Parametern existiert eine Überladung mit einem *einzigen* Parameter vom Interface-Typ **Collector<? super T,A,R>**, der die oben beschriebenen Aufgaben (Kollektion erstellen, Element einfügen, andere Kollektion einfügen) zusammenfasst. Besonders vorteilhaft ist, dass man ein Objekt des benötigten, nicht ganz trivialen Typs von einer statischen Methode der Klasse **Collectors** aus dem Paket **java.util.stream** erstellen lassen kann. Um ein passendes Kollektorobjekt für den **collect()** - Aufruf im Beispielprogramm zu erstellen, verwendet man die Methode **toList()**. Schlussendlich erhält man bei deutlich reduziertem Aufwand als Rückgabe von **collect()** einen Container vom Typ **ArrayList<String>**:

```
List<String> als2b = als.stream()
       .distinct()
       .sorted()
       .collect(Collectors.toList());
```

Im nächsten Beispiel entsteht mit Hilfe der **Collectors**-Methode **groupingBy()** aus einem Strom mit Vornamen nach dem Entfernen von Dubletten und dem Sortieren ein Kollektionsobjekt vom Abbildungs-Typ **Map<Character, List<String>>**, das eine Gruppierung der Vornamen nach den Anfangsbuchstaben leistet:

Quellcode	Ausgabe

<pre>import java.util.*;
import java.util.stream.Collectors;
 	
public class GroupingBy {
   public static void main(String[] args) {
      List<String> als = Arrays.asList("Charly","Anton",
        "Berta","Ben","Clara","Anton","Anette","Charly");
      Map<Character, List<String>> map = als.stream()
       .distinct()
       .sorted()
       .collect(Collectors.groupingBy(s -> s.charAt(0)));
      for(Character c : map.keySet())
         System.out.println(c + " " + map.get(c));
   }
}</pre>
	<pre>A [Anette, Anton]
	B [Ben, Berta]
	C [Charly, Clara]</pre>

Man kann hier von einer Abbildung mit (Single Key - Multiple Values) - Struktur sprechen.

Mit dem Gespann aus **collect()** und **Collectors** lassen sich Stromelemente nicht nur in Kollektionen sammeln, sondern auch zu anderen Resultaten verarbeiten (siehe API-Dokumentation). Im folgenden Beispiel werden **String**-Objekte mit der **Collectors**-Methode **joining()** unter Verwendung einer Separatorzeichenfolge zu einer Ergebniszeichenfolge verkettet:

Quellcode	Ausgabe

<pre>import java.util.*;
import java.util.stream.Collectors;
 	
public class Joining {
  public static void main(String[] args) {
    List<String> ls=Arrays.asList("Charly","Anton","Berta");
    String s = ls.stream()
                 .collect(Collectors.joining(", "));
    System.out.println(s);
  }
}</pre>
	<pre>Charly, Anton, Berta</pre>

12.2.5.4.5 Stromelemente in einem Array sammeln (toArray)

Im folgenden Beispiel wird ein **IntStream**-Objekt von Dubletten befreit und anschließend mit der **IntStream**-Methode **toArray()** in einen **int**-Array gewandelt:

Quellcode	Ausgabe

<pre>import java.util.stream.IntStream;
 	
public class ToArray {
   public static void main(String[] args) {
      int[] istar = IntStream.of(1, 1, 2, 3, 3, 4, 5, 5)</pre>
	<pre>1
	2
	3
	4
	5</pre>

```
            .distinct()
            .toArray();
      for(int i : istar)
         System.out.println(i);
   }
}
```

12.2.5.4.6 Strombezogene Bedingungen prüfen (anyMatch, allMatch, noneMatch)

Mit den Methoden **anyMatch()**, **allMatch()** und **noneMatch()**, die allesamt einen booleschen Rückgabewert liefern, lässt sich für einen Strom feststellen, ob eine Bedingung bei mindestens einem Element, bei allen Elementen oder bei keinem Element erfüllt ist. Alle Methoden benötigen als Parameter ein Objekt, das eine Methode namens **test()** mit einem booleschen Rückgabewert zur Beurteilung eines einzelnen Stromelements beherrscht. Bei den **Stream<T>** - Methoden zur strombezogenen Bedingungsprüfung muss besagtes Objekt zu einer Klasse gehören, welche das Interface **Predicate<? super T>** erfüllt, sodass sich z. B. für die Methode **anyMatch()** der folgende Definitionskopf ergibt:

public boolean anyMatch(Predicate<? super T> *predicate***)**

Im folgenden Programm wird für ein Objekt vom Typ **Stream<String>** mit der Methode **anyMatch()** geprüft, ob mindestens ein Element mit genau 5 Zeichen vorhanden ist:

Quellcode:

```
import java.util.Arrays;
import java.util.List;

class Matching {
  public static void main(String[] args) {
    List<String> als=Arrays.asList("Rudolf","Emma","Otto","Agnes","Kurt");
    boolean test = als.stream()
                        .anyMatch(s -> s.length() == 5);
    System.out.println(test);
  }
}
```

Ausgabe:

```
true
```

Das benötigte Objekt zur strombezogenen Bedingungsprüfung hat im Beispiel den Typ **Predicate<String>** und wird per Lambda-Ausdruck realisiert:

```
s -> s.length() == 5
```

12.2.5.4.7 Extrahieren eines Elements (findFirst, findAny)

Mit den Methoden **findFirst()** bzw. **findAny()** erhält man in **Optional**-Verpackung das erste bzw. irgendein Element des angesprochenen Stroms oder ein leeres **Optional**-Objekt (siehe Abschnitt 12.1.2.1), falls der Strom leer ist. In der Regel wird man den Strom vorher filtern, um an ein interessantes Objekt heranzukommen.

Weil **findAny()** mit dem Ziel maximaler Performanz bei paralleler Stromverarbeitung explizit die Freiheit hat, *irgendein* Element zu liefern, ist bei mehreren Aufrufen mit unterschiedlichen Rückgaben zu rechnen. Durch Verwendung der Methode **findFirst()** lässt sich dieser Indeterminismus beseitigen.

Im folgenden Code-Segment wird aus einer Liste mit Vornamen das erste Exemplar mit 4 Zeichen abgerufen:

```
List<String> als = Arrays.asList("Rudolf","Emma","Otto","Agnes","Kurt");
Optional<String> os = als.stream()
```

```
                              .filter(s -> s.length() == 4)
                              .findFirst();
```

Das folgende Programm wendet die beiden Methoden **findAny()** und **findFirst()** auf denselben gefilterten Strom an und vermeidet dabei die Wiederholung der Stromdefinition, um nicht gegen das DRY-Prinzip zu verstoßen (*Don't Repeat Yourself*). Dazu wird die Methode `crFStream()` eingesetzt, die ein frisches gefiltertes Stromobjekt basierend auf einer festen Namensliste liefert:

Quellcode	Ausgabe

<pre>import java.util.*;
import java.util.stream.Stream;
class FindAnyFirst {
  static List<String> als =
    Arrays.asList("Rudolf","Emma","Otto","Agnes","Kurt");
 	
  static Stream<String> crFStream() {
    return als.stream().filter(s -> s.length() == 4);
  }
 	
  public static void main(String[] args) {
    Optional<String> einTreffer = crFStream().findAny();
    System.out.println("Ein Treffer: " + einTreffer);
    Optional<String> ersterTreffer = crFStream().findFirst();
    System.out.println("Erster Treffer: " + ersterTreffer);
  }
}</pre>
	<pre>Ein Treffer: Optional[Emma]
	Erster Treffer: Optional[Emma]</pre>

12.3 Empfehlungen für erfolgreiches funktionales Programmieren

Subramaniam (2014, S. 12ff) empfiehlt den Java-Entwicklern einige Änderungen ihres Programmierstils, um einen hohen Nutzen aus der funktionalen Option in Java zu ziehen.

12.3.1.1 Deklarieren statt Kommandieren

Was sich hinter dieser Empfehlung verbirgt, soll durch ein Beispiel geklärt werden. Wir gehen aus von einer Liste mit Vornamen:

```
List<String> als = Arrays.asList("Viktor", "Otto", "Emma", "Kurt");
```

Dieses Listenobjekt ist vom parametrisierten Interface-Datentyp **List<String>** und wird durch die Methode **asList()** der Klasse **Arrays** erstellt.

Um die mittlere Länge der Vornamen mit mindestens 5 Zeichen zu ermitteln, ist im traditionellen Stil durch eine längliche Serie von Anweisungen zu *kommandieren*, *wie* vorzugehen ist:

```
double summe = 0.0;
int n = 0;
for (String s : als)
    if (s.length() >= 5) {
        summe += s.length();
        n++;
    }
System.out.println("Mittlere Länge der Namen mit >= 5 Zeichen: " +
        (n > 0 ? summe/n : "nicht vorhanden"));
```

Im funktionalen Stil von Java *deklariert* man, *was* zu tun ist:

- Aus der **String**-Liste einen Strom mit **String**-Elementen erstellen
- Elemente mit weniger als 5 Zeichen ausschließen
- **Stream<String>** in einen **IntStream** wandeln, der zu jeder Zeichenfolge die Länge enthält
- Mittelwert der Elemente im **IntStream** bestimmen

In der folgenden Lösung wird eine Verarbeitungs-Pipeline mit drei Operationen nach dem **Filter-Map-Reduce** - Schema deklariert:

```
OptionalDouble mlge5 = als.stream()
                          .filter(s -> s.length() >= 5)
                          .mapToInt(s -> s.length())
                          .average();
System.out.println("Mittlere Länge der Namen mit >= 5 Zeichen: " +
        (mlge5.isPresent() ? mlge5.getAsDouble() : "nicht vorhanden"));
```

Die Details der Ausführung bleiben den beteiligten Bibliotheksklassen überlassen:

- Es werden keine Hilfsvariablen (wie `summe` und `n` in der traditionellen Lösung) deklariert, initialisiert und aktualisiert.
- Die Iterationen laufen intern (gekapselt in Bibliotheksklassen) ab.

Man erspart sich viel Aufwand und viele Fehlermöglichkeiten.

Im Beispiel werden die beiden zu Beginn von Kapitel 12 erwähnten Kerntechniken der funktionalen Programmierung mit Java eingesetzt:

- Aus der **String**-Liste macht die Methode **stream()** aus dem parametrisierten Interface **Collection<String>** eine Sequenz von Elementen mit der Potenz zur bequemen Massenbearbeitung, *Stream* genannt. Es folgen zwei intermediäre Stromoperationen, die zu neuen Strömen führen (**filter()**, **mapTpInt()**) sowie eine terminale Stromoperation (**average()**).
- Die beiden ersten Stromoperationen benötigen eine Funktion, die auf jedes Element des Stroms angewendet werden soll. Während man bis Java 7 in einer solchen Situation meist ein Objekt einer ad hoc definierten anonymen Klasse als Parameter übergeben hat, ist es seit Java 8 möglich, die benötigte Funktionalität syntaktisch einfacher und eleganter per Lambda-Ausdruck zu definieren.

12.3.1.2 Veränderliche Variablen vermeiden

Code mit vielen veränderlichen Variablen ist fehleranfällig, relativ schwer zu verstehen und schlecht zu parallelisieren, d.h. auf mehrere Prozessorkerne zu verteilen. Im eben vorgestellten Beispiel (siehe Abschnitt 12.3.1.1) enthält die traditionelle Lösung sehr viele Wertzuweisungen, die funktionale Lösung (abgesehen von der Ergebnisübergabe) hingegen keine. Dabei führt die funktionale Lösung keinesfalls zu statischen Verhältnissen im Speicher. Es werden neue Objekte erzeugt (z. B. vom Typ **Stream<String>**, **IntStream**, **OptionalDouble**), allerdings keine vorhandenen modifiziert.

12.3.1.3 Seiteneffekte vermeiden

Ein grundlegendes Designmerkmal funktionaler Programmiersprachen, das in Java seit der Version 8 ermöglicht, aber nicht erzwungen wird, ist der Verzicht auf Seiteneffekte in Methoden. Wenn sich Methoden strikt darauf beschränken, aus den Parametern ein Ergebnis zu produzieren und als Rückgabe abzuliefern, steigt die Chance auf eine quasi - automatische Parallelisierung.

12.3.1.4 Ausdrücke bevorzugen gegenüber Anweisungen

Subramaniam (2014, S. 13f) empfiehlt, Ausdrücke gegenüber Anweisungen zu bevorzugen. Während Anweisungen zu vielen Wertveränderungen führen, lassen sich Ausdrücke gut zu Verarbeitungsketten zusammensetzen. Die traditionelle Lösung im Abschnitt 12.3.1.1 arbeitet mit 6 Anweisungen:

```
double summe = 0.0;                          // 1
int n = 0;                                   // 2
for (String s : als)                         // 3
    if (s.length() >= 5) {                   // 4
        summe += s.length();                 // 5
        n++;                                 // 6
    }
```

Demgegenüber beschränkt sich die funktionale Lösung auf eine *einzige* Anweisung, wobei einer Ergebnisvariablen ein Ausdruck zugewiesen wird, der aus einer Sequenz von Methodenaufrufen besteht:

```
OptionalDouble mlge5 =
     als.stream().filter(s->s.length()>=5).mapToInt(s->s.length()).average();
```

12.3.1.5 Verwendung von Funktionen höherer Ordnung

Beim funktionalen Programmierstil ist oft erforderlich, Funktionen als Parameter an andere Funktionen zu übergeben. In unserem Beispiel aus Abschnitt 12.3.1.1 wird an die **Stream<String>** - Methode **filter()** als Aktualparameter ein Lambda-Ausdruck übergeben:

```
s -> s.length() >= 5
```

Dabei handelt sich um eine Funktion, die auf jedes Element des Stroms angewendet werden soll.

Weil an die Funktion **filter()** eine andere Funktion per Parameter übergeben wird, bezeichnet man **filter()** als *Funktion höherer Ordnung*.

Um die Übergabe einer Funktion an eine Funktion höherer Ordnung darzustellen, musste das Typsystem von Java nicht geändert werden. Hinter den Kulissen entsteht aus dem Lambda-Ausdruck eine anonyme Klasse, und ein Objekt dieser Klasse wird an **filter()** als Aktualparameter übergeben. Als Datentyp verlangt **filter()**

Stream<T> filter(Predicate<? super T> *predicate*)

bei seinem Parameter ein Objekt einer Klasse welche das Interface **Predicate<? super T>** erfüllt und daher die folgende Methode implementiert:

public boolean test(? super T *t***)**

Der obige Lambda-Ausdruck passt zu dieser Methode:

- Aus der Parameterliste vor dem Pfeil ergibt sich, dass *ein* Parameter vorhanden ist.
- Als Datentyp wird für diesen Parameter der Elementtyp des Stroms (**String**) angenommen.
- Die Methode **length()** befindet sich im Handlungsrepertoire der Klasse **String** befindet.
- Der Lambda-Ausdruck liefert den korrekten Rückgabewert **boolean**.
- Damit lässt sich das benötigte Objekt aus einer passenden anonymen Klasse erstellen und an **filter()** übergeben:

```
new Predicate<String>() {
    public boolean test(String s) {
        return s.length() >= 5;
    }
}
```

Im Vergleich zur expliziten Verwendung eines Objekts aus einer (anonymen) Klasse besteht die Neuerung eigentlich nur aus syntaktischer Bequemlichkeit. Trotzdem verwendet man eine neue Begrifflichkeit, indem man von der Übergabe *von Funktionen an Funktionen* spricht.

Auch eine Funktion, die andere Funktionen erstellt und als Rückgabe abliefert, bezeichnet man als *Funktion höherer Ordnung*. Wenn im Beispiel aus Abschnitt 12.3.1.1 die mittlere Länge nicht nur

für Vornamen mit der Mindestlänge 5, sondern für mehrere Mindestlängen interessiert, dann ist es wenig attraktiv, entsprechend viele **Predicate<String>** - Objekte bzw. Lambda-Ausdrücke zu erstellen. Stattdessen definiert man eine Methode, die zu einer gewünschten Mindestlänge das passende **Predicate<String>** - Objekt liefert. In der folgenden Lösung

```
import java.util.*;
import java.util.OptionalDouble;
import java.util.function.Predicate;

public class Func2ndOrder {

   static Predicate<String> lenTest(int k) {
      return s -> s.length() >= k;
   }

   public static void main(String[] args) {
      List<String> als = Arrays.asList("Viktor", "Otto", "Emma", "Kurt");
      int k = 4;

      OptionalDouble mlgtk = als.stream()
            .filter(lenTest(k))
            .mapToInt(s -> s.length())
            .average();
      System.out.println("Mittlere Länge der Namen mit >= " + k + " Zeichen: " +
            (mlgtk.isPresent() ? mlgtk.getAsDouble() : "nicht vorhanden"));
   }
}
```

arbeitet `lenTest()` als Funktion höherer Ordnung und produziert Parameterobjekte vom Typ **Predicate<String>** für die Stromoperation **filter()**.

12.4 Übungsaufgaben zum Kapitel 12

1) Erstellen Sie eine Anweisung zur Berechnung der Summe aus den ersten 100 quadrierten natürlichen Zahlen. Vermutlich werden Sie eine elementweise abbildende Stromoperation verwenden (vgl. Abschnitt 12.2.5.3.3). Definieren Sie den Mapper übungshalber über eine anonyme Klasse und über einen Lambda-Ausdruck.

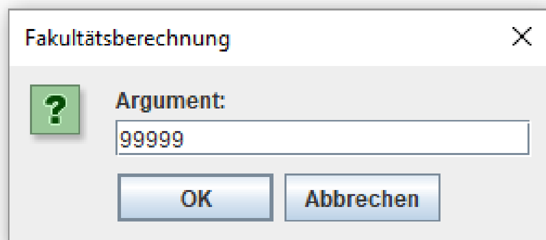
2) Ein Lambda-Ausdruck wird vom Compiler überall dort akzeptiert, wo eine Referenz vom Typ einer funktionalen Schnittstelle erwartet wird. Er steht also für ein Objekt einer speziellen Klasse, die im Java-Typsystem direkt oder indirekt von der Urahnklasse **Object** abstammt. Trotzdem kann einer **Object**-Referenzvariablen kein Lambda-Ausdruck zugewiesen werden, wie z. B. die folgende Fehlermeldung von IntelliJ IDEA zeigt:

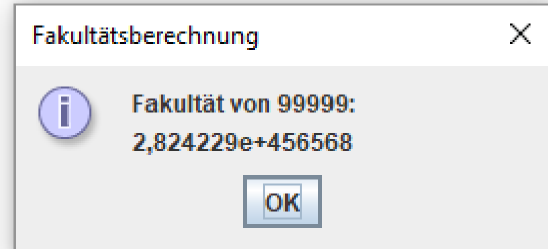
```
Object obj = (String s) -> s.length() >= 5;
```

Target type of a lambda conversion must be an interface

Wir sind daran gewöhnt, dass in einer **Object**-Referenzvariablen die Adresse eines beliebigen Objekts abgelegt werden kann. Welche Gründe erzwingen eine Ausnahme bei der Zuweisungskompatibilität?

3) Erstellen Sie ein Programm zur Fakultätsberechnung, das vom Benutzer per **JOptionPane**-Standarddialog (vgl. Abschnitt 3.8) ein Argument entgegennimmt. Verwenden Sie den Datentyp **BigDecimal**, um praktisch beliebig große Argumente erlauben zu können. Nutzen Sie je nach Problemgröße (Argument) einen seriell oder parallel arbeitenden Strom vom Typ **LongStream**. Die Bedienoberfläche Ihres Programms könnte ungefähr so aussehen:





Geben Sie dem Benutzer unter Verwendung der im Kapitel 11 (über die Ausnahmebehandlung) erlernten Techniken eine Möglichkeit, eine falsche Eingabe zu korrigieren.

13 GUI-Programmierung mit JavaFX

Mit den Eigenschaften und Vorteilen einer grafischen Benutzeroberfläche (engl.: ***G**raphical **U**ser **I**nterface*) sind Sie sicher sehr gut vertraut. Eine GUI-Anwendung präsentiert dem Anwender standardisierte Bedienelemente zur Datenpräsentation und Benutzerinteraktion, z. B.:

- Texteingabefelder
- Befehlsschalter
- Kontrollkästchen und Optionsfelder
- Schieberegler und Auswahllisten
- Baumansichten (z. B. für Dokumentenstrukturen)
- Tabellen zur Datenpräsentation
- Menüs
- Fortschrittsbalken
- Komponenten zur Präsentation von Bildern und audio-visuellen Medien

Die von einer GUI-Bibliothek (in unserem Fall von JavaFX) zur Verfügung gestellten Bedienelemente bezeichnet man oft als *Komponenten*, *controls*, *Steuerelemente* oder *widgets*.[1]

Von standardisierten Bedienelementen profitieren Entwickler *und* Anwender:

- Entwickler können dank fertiger und dabei auch noch flexibel konfigurierbarer Komponenten die Bedienoberfläche einer Anwendung zügig aufbauen. Für eine weitere RAD-Beschleunigung (***R**apid **A**pplication **D**evelopment*) sorgen grafische GUI-Designer (z. B. der Scene Builder zu JavaFX, den wir schon im Abschnitt 4.9 kennengelernt haben).
- Weil die Steuerelemente intuitiv (z. B. per Maus oder Finger) und in verschiedenen Programmen weitgehend konsistent zu bedienen sind, erleichtern sie dem Anwender den Umgang mit Software.

Die in einer leistungsfähigen GUI-Bibliothek wie JavaFX (alias OpenJFX) enthaltenen Standardkomponenten erlauben durch ihre Vielfalt, ihre Konfigurierbarkeit und durch die Möglichkeiten zur flexiblen (z. B. hierarchisch strukturierten) Anordnung die Erstellung von individuellen und ergonomischen Bedienoberflächen für sehr viele Anwendungen. Bei manchen Programmen genügen die Standardkomponenten aber *nicht* für die spezielle Präsentation und/oder Bearbeitung von zwei- oder dreidimensionalen Daten (z. B. bei einem Editor für statistische Diagramme), und es wird eine individuelle Grafikprogammierung erforderlich. Wir beschränken uns in diesem Kapitel auf einen Einstieg in die Erstellung von Bedienoberflächen mit Hilfe von Standardkomponenten aus der JavaFX-Bibliothek.

13.1 Einordnung

13.1.1 Vergleich von Konsolen- und GUI-Programmen

Im Vergleich zu Konsolenprogrammen geht es bei GUI-Anwendungen nicht nur anschaulicher und intuitiver zu, sondern vor allem auch ereignisreicher und mit mehr Mitspracherechten für den Anwender. Ein Konsolenprogramm entscheidet selbst darüber, welche Anweisung als nächstes ausgeführt wird, und wann der Benutzer eine Eingabe machen darf. Um seine Aufgaben zu erledigen, verwendet ein Konsolenprogramm diverse Dienste des Laufzeitsystems, z. B. bei der Aus- oder Eingabe von Zeichen.

Für den Ablauf eines Programms mit grafischer Bedienoberfläche ist hingegen ein **ereignisorientiertes und benutzergesteuertes Paradigma** wesentlich, wobei das Laufzeitsystem als Vermittler

[1] Diese Wortkombination aus *window* und *gadgets* steht für ein *praktisches Fenstergerät*.

oder (seltener) als Quelle von Ereignissen in erheblichem Maße den Ablauf mitbestimmt, indem es Methoden der GUI-Applikation aufruft, z. B. zum Zeichnen von Fensterinhalten. Ausgelöst werden die Ereignisse in der Regel vom Benutzer, der mit der Hilfe von Eingabegeräten wie Maus, Tastatur, Touch Screen etc. praktisch permanent in der Lage ist, unterschiedliche Wünsche zu artikulieren. Ein GUI-Programm präsentiert mehr oder weniger viele Bedienelemente, die dem Anwender das Auslösen von Ereignissen ermöglichen. Das Programm wartet die meiste Zeit darauf, auf ein vom **Benutzer** ausgelöstes **Ereignis** mit einer vorbereiteten Ereignisbehandlungsmethode zu reagieren.

Im Vergleich zu einem Konsolenprogramm ist bei einem GUI-Programm die dominante Richtung im Kontrollfluss zwischen Programm und Laufzeitsystem invertiert. Die Ereignisbehandlungsmethoden einer GUI-Anwendung sind Beispiele für sogenannte *Call Back - Routinen*. Man spricht auch vom *Hollywood-Prinzip*, weil in dieser Gegend oft nach der Divise kommuniziert wird: „*Don't call us. We call you*".

Während sich ein Konsolenprogramm gegenüber dem Anwender autoritär und gegenüber dem Laufzeitsystem fordernd verhält, präsentiert ein GUI-Programm dem Anwender Service-Angebote und befolgt die Anweisungen des Laufzeitsystems:

- Eine Konsolenanwendung diktiert den Ablauf und erlaubt dem Benutzer gelegentlich eine Eingabe. Um seinen Job erledigen zu können, verlangt das Programm Dienstleistungen vom Laufzeitsystem, z. B.: „Bitte den nächsten Tastendruck übermitteln." Das Laufzeitsystem erledigt solche Anforderungen und gibt die Kontrolle dann wieder an die Konsolenanwendung zurück. Eine Konsolenanwendung benimmt sich so, als wäre sie das einzige Anwendungsprogramm und hätte das Laufzeitsystem zu ihrer Verfügung.
- Eine GUI-Anwendung besteht hingegen aus einer Sammlung von Ereignisbehandlungsmethoden, wobei die zugehörigen Ereignisse vom Benutzer ausgelöst werden, indem er eines der zahlreichen Bedienelemente benutzt. Die Ereignisse werden zunächst vom Laufzeitsystem registriert, das daraufhin Methoden des GUI-Programms aufruft.

Betrachten wir zur Illustration eine Konsolen- und eine GUI-Anwendung zum Addieren von Brüchen. Bei der Konsolenanwendung (vgl. Abschnitt 1.1.4)

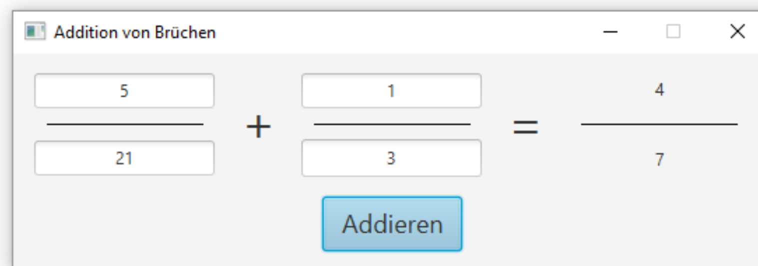
```
Eingabeaufforderung
Zaehler: 2
Nenner : 3
   2
 -----
   3

2. Bruch
Zaehler: 1
Nenner : 7
   1
 -----
   7

Summe
   17
 -----
   21
```

wird der gesamte Ablauf vom Programm diktiert:

- Es fragt nach dem Zähler und dem Nenner des ersten Bruchs.
- Es fragt nach dem Zähler und dem Nenner des zweiten Bruchs.
- Es schreibt das Ergebnis auf die Konsole.

Wenn der Benutzer z. B. nach der Eingabe des Nenners zum ersten Bruch den Zähler dieses Bruchs noch einmal ändern möchte, muss er das Programm beenden und neu starten.

Im Unterschied zu diesem **programmgesteuerten Ablauf** wird bei der GUI-Variante



das Geschehen vom Benutzer diktiert, der die 5 Bedienelemente (vier Eingabefelder und eine Schaltfläche) in beliebiger Reihenfolge verwenden kann, wobei das Programm mit seinen Ereignisbehandlungsmethoden reagiert (**benutzergesteuerter Ablauf**).

Grundsätzlich ist das Erstellen einer GUI-Anwendung mit erheblichem Aufwand verbunden. Allerdings enthält das Java-API leistungsfähige Klassen zur GUI-Programmierung, und deren Verwendung wird durch Hilfsmittel der Entwicklungsumgebungen (z. B. Fensterdesigner) sehr gut unterstützt.

Wie man mit statischen Methoden der Klasse **JOptionPane** einfache Standarddialoge erzeugt, um Nachrichten auszugeben oder Informationen abzufragen, wissen Sie schon seit Abschnitt 3.8. Allerdings kommen nur wenige GUI-Anwendungen mit diesen elementaren Gestaltungs- bzw. Interaktionsmöglichkeiten aus. Wir beschäftigen uns in diesem Kapitel damit, individuelle Bedienoberflächen mit Hilfe der JavaFX-Bibliothek zu erstellen.

13.1.2 Desktop-GUI-Lösungen in Java

Java hat von Beginn an die plattformunabhängige GUI-Programmierung ermöglicht, und die dazu verwendete Technologie wurde sukzessive weiterentwickelt. Die ursprüngliche, als *Abstract Windowing Toolkit* (**AWT**) bezeichnete GUI-Technologie wurde schon in Java 1.2 durch das **Swing Toolkit** erweitert und teilweise ersetzt. Nachdem Swing über viele Jahre der unangefochtene Standard für GUI-Desktop-Anwendungen in Java war, wurde diese Rolle 2014 mit der Java-Version 8 von **JavaFX** übernommen. Zu diesem Zeitpunkt hatte JavaFX schon einige Entwicklungsschritte hinter sich:

- **JavaFX 1.x**
  Von der Firma Sun wurde 2008 mit eher mäßigem Erfolg die Programmiersprache **JavaFX Script** auf den Markt gebracht, um die Entwicklung von grafischen Bedienoberflächen zu vereinfachen.
- **JavaFX 2.x**
  Im Jahr 2011 hat die Firma Oracle (nach Übernahme der Firma Sun) unter der Bezeichnung **JavaFX 2.0** ein deutlich attraktiveres Angebot vorgestellt:
  - Man hat die separate Programmiersprache JavaFX Script aufgegeben und die attraktiven Teile von JavaFX als Klassenbibliotheken in Java integriert (z. B. eine neue Steuerelementfamilie, Grafikausgabe mit plattformspezifischer Hardware-Beschleunigung).
  - Es wurde eine XML-basierte GUI-Deklaration ermöglicht.
- **JavaFX 8**
  Als JavaFX im Jahr 2014 zum Standard für grafische Desktop-Bedienoberflächen in Java 8 befördert wurde, machte die JavaFX-Version einen Sprung von 2.2 auf 8. Seitdem stimmen die Versionsstände von JavaFX und Java SE überein. Somit arbeiten wir aktuell (im März 2020) mit JavaFX 13.

- **JavaFX (alias OpenJFX) ab Version 11**
  Beginnend mit Java 11 wurde JavaFX von der Firma Oracle aus dem JDK entfernt, wovon auch die OpenJDK-Distributionen betroffen sind. JavaFX bleibt unter dem Namen OpenJFX aber als Open Source - Software frei verfügbar (wie das OpenJDK unter der GPL) und wird aktiv weiterentwickelt,[1] wobei die Firma Gluon die Federführung übernommen hat.[2] Man kann davon ausgehen, dass für JavaFX wie für das JDK in absehbarer Zukunft halbjährliche Updates erscheinen werden. Für Entwickler hat die neue Situation folgende Konsequenzen:
  - Weil JavaFX in keiner OpenJDK-Distribution mit einer Version > 8 enthalten ist, muss es explizit in eigene Anwendungen integriert werden.
  - Long-Term-Support ist nur kommerziell von der Firma Gluon zu haben. Wer JavaFX kostenfrei und sicher nutzen will, muss also halbjährlich auf die aktuelle Version umsteigen. Beim OpenJDK ist die Lage durch etliche kostenlos verfügbare Distributionen mit langfristig zugesagter Update-Versorgung wesentlich günstiger (siehe Abschnitt 2.1.2).

Neben Steuerelementen kann ein JavaFX-Fenster auch 2- oder 3-dimensionale Grafikelemente zeigen, was in diesem Kapitel ebenso wenig thematisiert wird wie die JavaFX-Kompetenzen zur Darstellung von Effekten (z. B. Schatten, Weichzeichnen) und Animationen (z. B. wandernde, größenvariable Elemente). Leider fehlen in diesem unfertigen, aber hoffentlich trotzdem nützlichen Kapitel noch weitere JavaFX-Techniken, z. B.:

- Erscheinungsbild einer Anwendung mit CSS (*Cascading **S**tyle **S**heets*) individualisieren
- Multimedia-Inhalte darstellen (Audio, Video)

Die meisten GUI-Programme kommen allerdings ohne die in diesem Kapitel noch fehlenden JavaFX-Techniken aus.

Den Multithreading-Bezügen von JavaFX gehen wir im aktuellen Kapitel aus dem Weg, holen deren Behandlung aber später nach (siehe Abschnitt 15.7).

JavaFX (alias OpenJFX) sowie das als grafischer Fensterdesigner sehr empfehlenswerte Programm Scene Builder zu beziehen und zu installieren sind, wurde im Abschnitt 2.5 beschrieben.

An Literatur zu JavaFX herrscht kein Mangel. Eine kompakte Beschreibung bietet z. B. Inden (2015), eine sehr ausführliche ist z. B. bei Sharan (2015) zu finden.

Während bei neuen Projekten mit grafischer Desktop-Bedienoberfläche auf JavaFX gesetzt werden sollte, müssen viele Swing-basierte Lösungen sicher noch für geraume Zeit gepflegt werden. Wer eine Einführung in die GUI-Programmierung mit Swing sucht, wird z. B. bei Baltes-Götz & Götz (2016) fündig.

Neben den GUI-Toolkits der (bzgl. JavaFX segmentierten) Java-Standardbibliothek sind noch andere Lösungen verfügbar, wobei besonders das im Eclipse-Projekt entwickelte *Standard Widget Toolkit* (SWT) zu erwähnen ist.

13.2 Einstieg in JavaFX

Während mit der älteren GUI-Technik Swing die Bedienoberfläche einer Anwendung grundsätzlich programmgesteuert erstellt wird, bietet JavaFX neben dieser traditionellen Vorgehensweise auch die Möglichkeit zur deklarativen GUI-Gestaltung über einen XML-Dialekt namens **FXML** (*Java**FX** **M**arkup **L**anguage*). Diese nicht nur in JavaFX, sondern z. B. auch im Windows Presenta-

[1] https://openjfx.io/

[2] https://gluonhq.com/products/javafx/

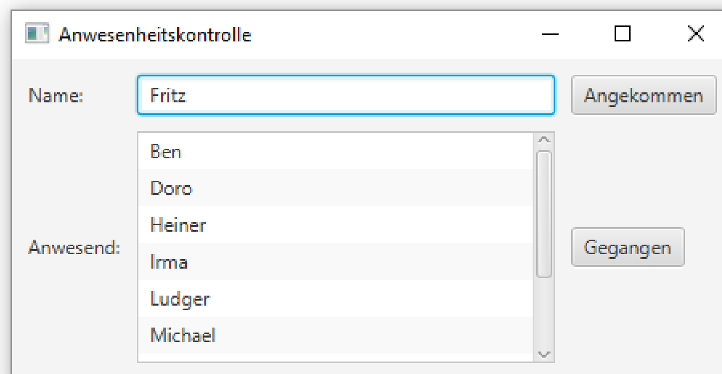
tion Framework (WPF) des .NET - Frameworks zur Windows-Programmierung sowie im Android-SDK bevorzugte deklarative Technik bietet u.a. die folgenden Vorteile:

- Bessere Trennung von Design und Funktion
  Diese Trennung erleichtert die Pflege der beiden Bestandteile und ermöglicht eine arbeitsteilige Kooperation von Programmierern und GUI-Designern.
- Bessere Übersicht bei komplexen Bedienoberflächen

Die deklarative GUI-Gestaltung auf FXML-Basis wird durch den grafischen Fensterdesigner Scene Builder wesentlich erleichtert. Wir haben beim ersten JavaFX-Einsatz im Abschnitt 4.9 die deklarative GUI-Gestaltung mit Hilfe des Scene Builders verwendet, und diese Arbeitsweise ist auch für den Alltag der GUI-Software-Entwicklung in Java zu empfehlen. Im aktuellen Kapitel kommt allerdings vielfach die programmatische Vorgehensweise zum Einsatz, weil sie einen unverstellten Blick auf die JavaFX-Anwendungsstruktur erlaubt.

13.2.1 Beispiel Anwesenheitsliste

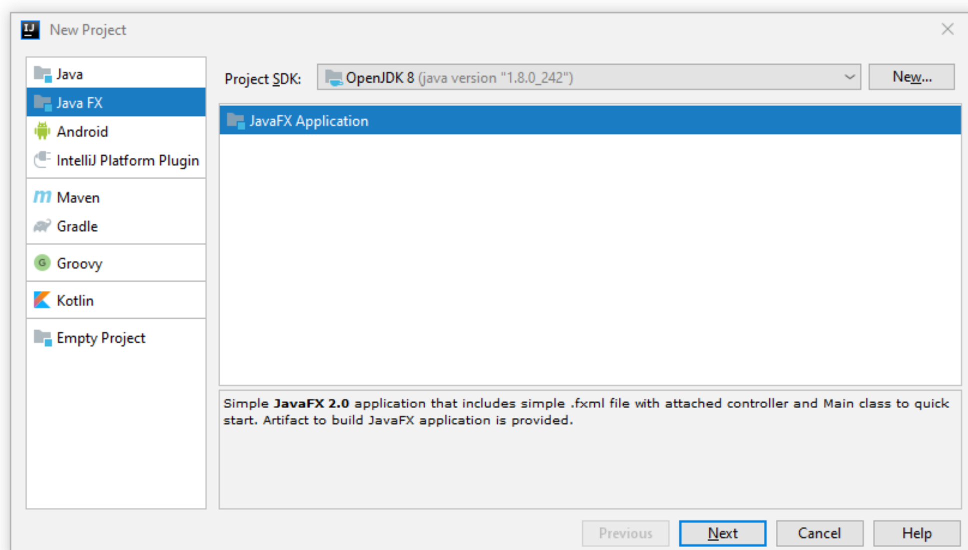
Als erstes Beispiel erstellen wir ein Programm, das z. B. zur Anwesenheitsüberwachung durch einen Blockwart mit freiem Blick auf den einzigen Hauseingang dienen kann:



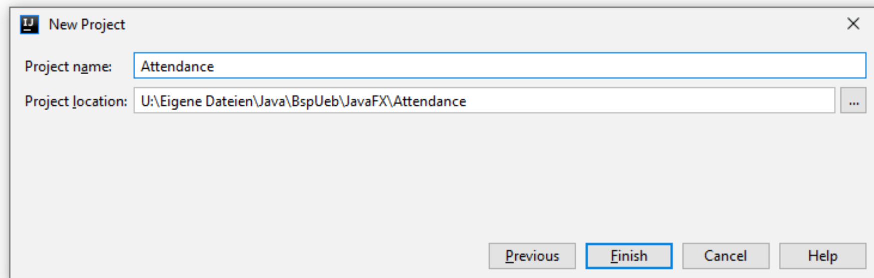
Um in IntelliJ ein Projekt für dieses Programm anzulegen, starten wir mit dem Menübefehl

File > New > Project

und wählen im folgenden Fenster



eine neue **JavaFX Application** basierend auf dem **OpenJDK 8**. Nach einem Klick auf Next vereinbaren wir im einzigen Assistentendialog den Projektnamen `Attendance` (dt.: *Anwesenheit*):

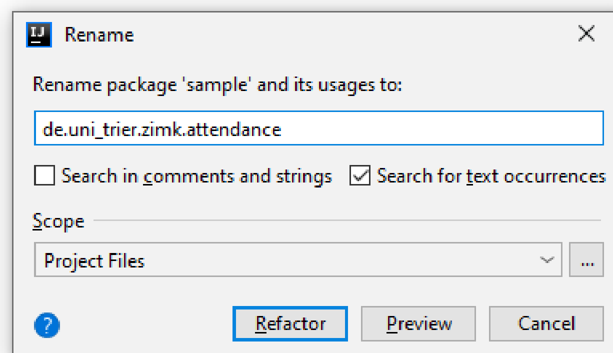


IntelliJ legt für das entstehende Programm eine Hauptklasse mit dem Namen `Main` an, die von der API-Klasse **Application** abstammt:

```
package sample;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));
        primaryStage.setTitle("Hello World");
        primaryStage.setScene(new Scene(root, 300, 275));
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Selbstverständlich hat IntelliJ für das neue Programm ein (wenn auch nicht vorbildlich benanntes) Paket angelegt (siehe erste Zeile im Quellcode). Als Paketname eignet sich z. B. `de.uni_trier.zimk.attendance`. Weil der Paketname vermutlich an mehreren Stellen im Projekt auftaucht und außerdem für die Projektdateiorganisation relevant ist, nutzen wir wieder einmal die Refaktorisierungs-Kompetenz von IntelliJ:

- Markieren Sie den Paketnamen in der Quellcodedatei **Main.java**.
- Wählen Sie den Menübefehl **Refactor > Rename** oder die Tastenkombination **Umschalt + F6**.
- Tragen Sie im folgenden Fenster den gewünschten Paketnamen ein, und quittieren Sie mit **Refactor**:

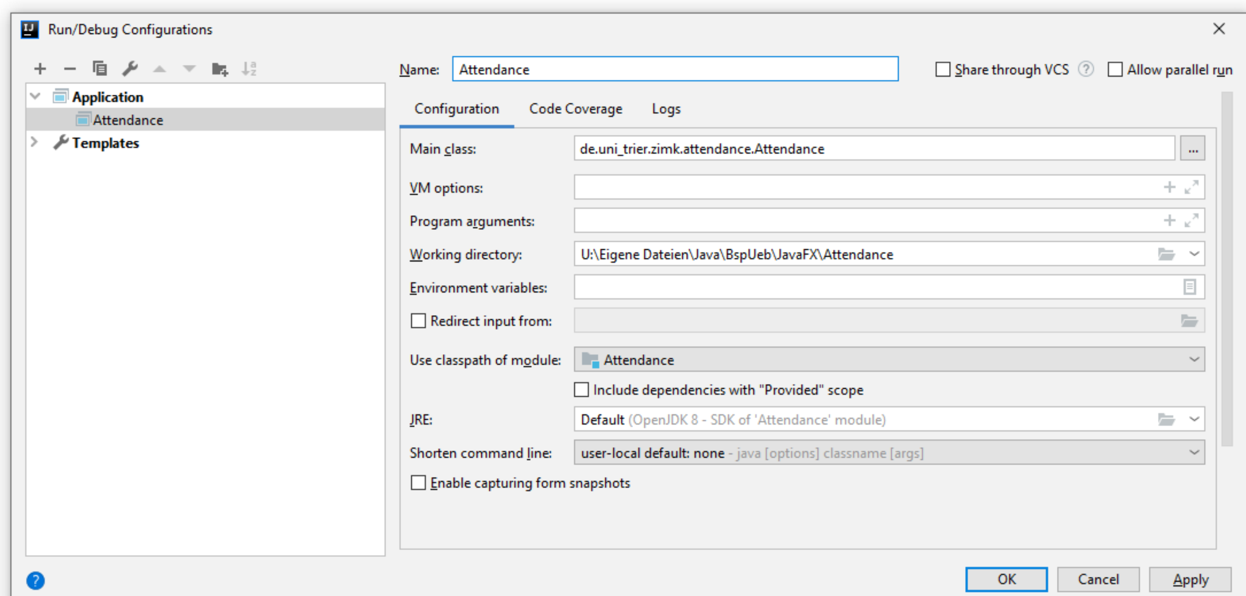


Ersetzen Sie außerdem per Refaktorierung den Klassennamen `Main` durch `Attendance`, indem Sie den Klassennamen im Quellcodeeditor markieren und dann die Tastenkombination **Umschalt + F6** betätigen:

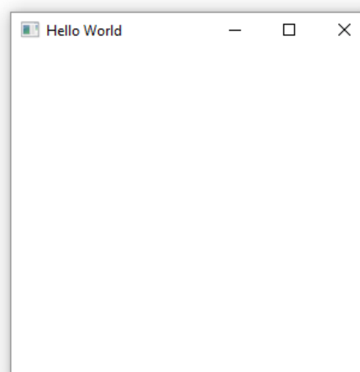
```
public class Attendance extends Application {
```

IntelliJ ändert dabei auch den Namen der zugehörigen Quellcodedatei.

In der **Run Configuration** ist der Name der **Main class** anzupassen. Es bietet sich an, auch den Namen der Konfiguration zu ändern:



Das von IntelliJ erstellte Programm ist schon startfähig, zeigt aber bisher nur eine leere Szenerie:



13.2.2 Starten und Beenden einer JavaFX-Anwendung

Die Haupt- bzw. Anwendungsklasse einer JavaFX-Anwendung muss von der Klasse **Application** im Paket **javafx.application** abgeleitet werden und den Zugriffsmodifikator **public** erhalten. z. B.:

```
public class Attendance extends Application {
  . . .
}
```

Damit ein Objekt der Anwendungsklasse erzeugt werden kann, muss ein parameterfreier Konstruktor öffentlich verfügbar sein. Im Beispiel ist diese Bedingung erfüllt, weil der Standardkonstruktor vorhanden ist, der die Zugriffsstufe **public** von der Klasse übernimmt.

In der Anlaufphase einer JavaFX-Anwendung spielt die in der Basisklasse **Application** abstrakt definierte, also auf jeden Fall zu implementierende Methode **start()** eine wichtige Rolle. Sie wird (indirekt) von der Methode **launch()** aufgerufen, die bis zum Ende der Anwendung läuft. Für den Aufruf von **launch()** sorgt die Methode **main()**, und mehr sollte dort bei einer JavaFX-Anwendung nicht passieren:

```
public static void main(String[] args) {
    launch(args);
}
```

Die Methode **main()** darf sogar fehlen, wobei dann die Laufzeitumgebung für den Aufruf der Methode **launch()** sorgt.

Demgegenüber ist die Methode **start()** unverzichtbar, weil hier die Bühne vorbereitet und schließlich der Vorhang gezogen wird:

```
@Override
public void start(Stage primaryStage) throws Exception{
    Parent root = FXMLLoader.load(getClass().getResource("attendance.fxml"));
    primaryStage.setTitle("Hello World");
    primaryStage.setScene(new Scene(root, 300, 275));
    primaryStage.show();
}
```

Die von IntelliJ erstellte **start()** - Implementation lädt die GUI-Deklaration mit Hilfe der statischen Methode **load()** der Klasse **FXMLLoader** aus der Datei **attendance.fxml**:

```
<?import javafx.geometry.Insets?>
<?import javafx.scene.layout.GridPane?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<GridPane fx:controller="sample.Controller"
          xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10">
</GridPane>
```

Dort wird ein **GridPane**-Objekt als Wurzel-Container für das Anwendungsfenster verwendet, und die Referenz auf dieses Objekt landet in der lokalen **start()** - Variablen **root** vom Typ **Parent**, von dem (indirekt) die JavaFX - Layout-Container abstammen (siehe Abschnitt 13.2.3):

```
Parent root = FXMLLoader.load(getClass().getResource("attendance.fxml"));
```

Unter Verwendung des Wurzel-Containers wird im Beispielprogramm ein **Scene**-Objekt mit bestimmten Ausmaßen erstellt. Das der Methode **start()** per Parameter bekannte Objekt vom Typ **Stage** (dt.: *Bühne*) erhält den Auftrag, diese Szene auf die Bühne zu bringen

```
primaryStage.setScene(new Scene(root, 300, 275));
```

und sichtbar zu machen:

```
primaryStage.show();
```

Außerdem ist das **Stage**-Objekt für den Fenstertitel verantwortlich, den wir im Beispielprogramm anpassen:

```
primaryStage.setTitle("Anwesenheitskontrolle");
```

Für Initialisierungsarbeiten, die vor dem eigentlichen Anwendungsstart stattfinden sollen, eignet sich die Methode **init()**, die ggf. nach dem Laden und Instanzieren der Anwendungsklasse ausgeführt wird:

public void init() throws Exception

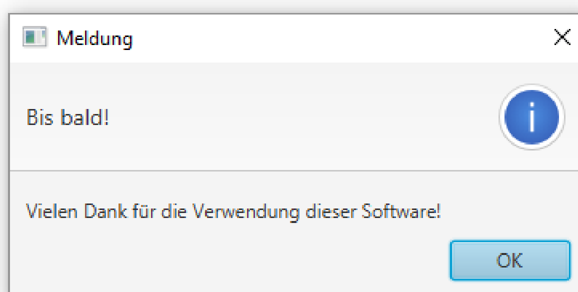
Weil die Methode noch nicht im JavaFX-Anwendungs-Thread läuft, kann hier noch kein **Stage**- oder **Scene**-Objekt erstellt werden. Nach Rückkehr von **init()** wird **start()** aufgerufen.

Im Vergleich zur älteren Swing - GUI-Technik ist anzumerken, dass beim Starten keine Vorsichtsmaßnahmen auf Seiten des Anwendungsprogrammierers erforderlich sind, um Multithreading-Probleme zu verhindern.

Werden alle Fenster (Bühnen) einer Anwendung geschlossen, endet sie per Voreinstellung. Soll ein Programm in dieser Situation noch tätig werden, bietet die automatisch aufgerufene **Application**-Methode **stop()** dazu Gelegenheit. Im folgenden Beispiel

```
@Override
public void stop() {
    Alert alert = new Alert(Alert.AlertType.INFORMATION,
            "Vielen Dank für die Verwendung dieser Software!");
    alert.setHeaderText("Bis bald!");
    alert.showAndWait();
}
```

wird noch ein Standarddialog angezeigt:



Mit der statischen Methode **exit()** der Klasse **Platform** erreicht man die sofortige Beendigung des Programms, wobei ggf. noch ein **stop()** - Aufruf erfolgt:

```
Platform.exit();
```

Während in der älteren GUI-Technik Swing durch spezielle Vorkehrungen (z. B. durch einen **WindowListener**) verhindert werden muss, dass nach dem Schließen aller Fenster im Hintergrund ein unsichtbares Programm weiterläuft, endet eine JavaFX-Anwendung per Voreinstellung nach dem Schließen ihrer Fenster.[1]

[1] Allerdings kann in JavaFX mit der folgenden Anweisung

```
Platform.setImplicitExit(false);
```

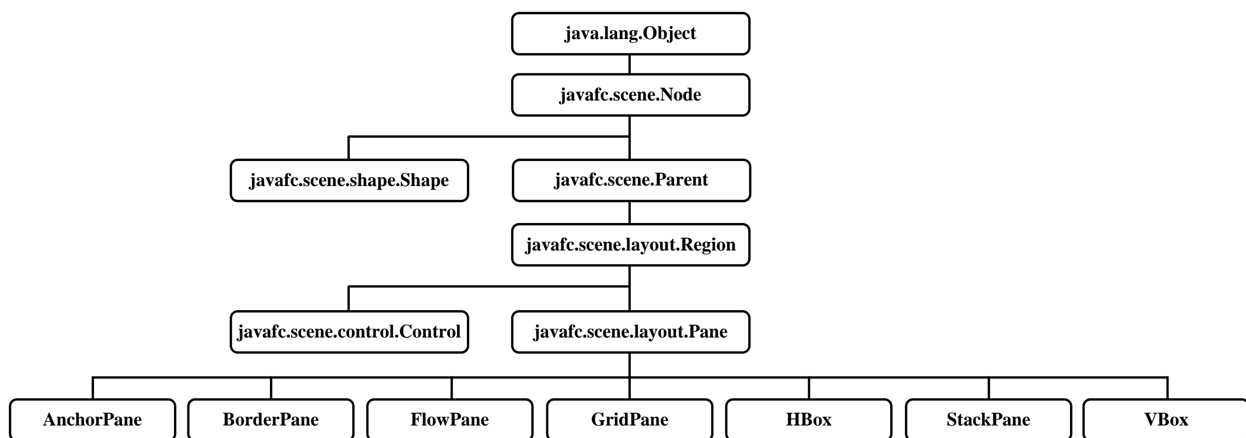
in der **start()** - Methode das automatische Programmende nach dem Schließen der Fenster verhindert werden. Dann muss man analog zu Swing für das Programmende sorgen.

13.2.3 Bühne, Szene und Szenengraph

In einer JavaFX-Anwendung fungiert ein Fenster des Wirtsbetriebssystems als Bühne, wobei auch mehrere Fenster bzw. Bühnen simultan bespielt werden können. Zur primären Bühne erhält die Methode **start()** beim Aufruf eine Referenz vom Typ **Stage** (aus dem Paket **javafx.stage**) per Parameter.

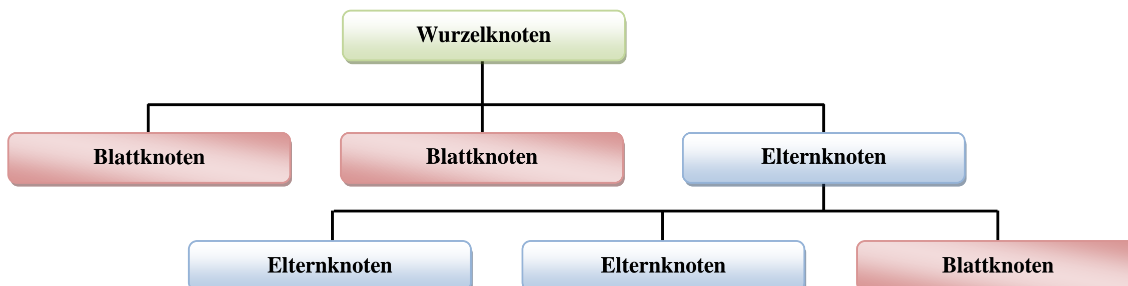
Um Leben auf eine Bühne zu bringen, muss ein Objekt der Klasse **Scene** (aus dem Paket **javafx.scene**) eingesetzt werden. Ein **Stage**-Objekt enthält (zu einem Zeitpunkt) genau ein **Scene**-Objekt, das seinerseits als Container für beliebig viele visuelle Komponenten (Bedienelemente und/oder Grafikelemente) dient.

Die Komponenten in der Szene stammen von der Klasse **Node** im Paket **javafc.scene** ab und sind hierarchisch in einem **Szenengraphen** (engl. *scene graph*) angeordnet mit einem Wurzelknoten an der Spitze, der in den **Scene**-Konstruktoren vereinbart wird. Alle Knoten im Szenengraphen, die Kinder (eingeschachtelte Knoten) haben können, stammen von der Klasse **Parent** im Paket **javafc.scene** ab:



Die Spezialisierungen der Klasse **javafx.scene.layout.Pane** sind nicht nur aufnahmefähig für Kindelemente, sondern auch mit Kompetenzen zum Layout-Management ausgestattet. Sie sind daher als Wurzelknoten des Szenengraphen besonders geeignet. Auch die von **javafx.scene.control.Control** abstammenden normalen Steuerelemente (z. B. **TextField**, **Button**) können untergeordnete Knoten aufnehmen. Die von **javafc.scene.shape.Shape** abstammenden Grafikelemente (z. B. **Circle**, **Line**) sind hingegen nur als Endknoten erlaubt.

In einem JavaFX-Szenegraphen können also drei Sorten von Knoten auftreten:



13.3 Anwendung mit All-In-One - Architektur

Wir werden das im Abschnitt 13.2 vorgestellte Beispielprogramm zur Anwesenheitsüberwachung gleich *zweimal* realisieren:

- Im aktuellen Abschnitt 13.3 wird das Programm durch eine einzige Klasse realisiert, die sich auch um die Bedienoberfläche kümmert und diese auf traditionelle Weise durch Anweisun-

gen erstellt. Diese Vorgehensweise macht einige Merkmale der JavaFX-Anwendungsarchitektur deutlich und ist daher in der Lernphase angemessen.

- Im Abschnitt 13.4 lernen wir die für größere Projekte empfehlenswerte Vorgehensweise kennen, die sich moderner Techniken bedient (z. B. Layout-Deklaration per FXML) und eine Aufgabenverteilung auf mehrere Klassen vornimmt.

Im Beispielprogramm ist für den Wurzelknoten des Szenengraphen die von IntelliJ (in der Datei **attendance.fxml**) gewählte Klasse **GridPane** gut geeignet. Wir werden allerdings im aktuellen Abschnitt das Layout durch *Anweisungen* definieren, um einen Einblick in die JavaFX-Technik zu erhalten. Daher ignorieren wir die Datei **attendance.fxml** und machen in der Methode **start()** einen Layout-Neustart, wobei **GridPane** als Klasse des Wurzelknotens beibehalten wird::

```
public void start(Stage primaryStage) {
    GridPane root = new GridPane();
    . . .
}
```

Mit der **Region**-Methode **setPadding()** wird ein freizuhaltender Innenrahmen für den Container vereinbart, und über die **GridPane**-Methoden **setHgap()** bzw. **setVgap()** ein horizontaler bzw. vertikaler Abstand zwischen den Kindelementen (siehe Abschnitt 13.6.1.2):

```
double dist = 10.0;
root.setPadding(new Insets(dist, dist, dist, dist));
root.setHgap(dist); root.setVgap(dist);
```

Zur Anzeige von Beschriftungen erstellen wir zwei Komponenten aus der Klasse **Label** im Paket **javafx.scene.control**:

```
Label lblName = new Label("Name:");
Label lblPresent = new Label("Anwesend:");
```

Zur Aufnahme der Namen von Neuankömmlingen ist ein Objekt der Klasse **TextField** aus dem Paket **javafx.scene.control** zuständig:

```
TextField tfName = new TextField();
```

Über zwei Befehlsschalter aus der Klasse **Button** sollen Personen in die Anwesenheitsliste aufgenommen bzw. aus dieser Liste entfernt werden:

```
Button btnAdd = new Button("Angekommen");
Button btnRemove = new Button("Gegangen");
```

Nun kommen wir zum „technischen Glanzstück“ der verhältnismäßig simplen Anwendung, einem Objekt der Klasse **ListView<String>**, das die Elemente eines Objekts vom Typ **ObservableList<String>** sortiert anzeigt und dynamisch auf Änderungen in der Zusammensetzung der beobachtbaren Liste reagiert:

```
String[] anwesend = new String[] {"Willi", "Otto", "Theo", "Irma", "Doro",
                                  "Heiner", "Michael", "Ludger", "Ben"};
ObservableList<String> persons = FXCollections.observableArrayList(anwesend);
SortedList<String> perSorted = new SortedList<>(persons,
                                                Comparator.naturalOrder());
ListView<String> lvPersons = new ListView<>(perSorted);
```

Mit beobachtbaren Kollektionen, die bei einer Änderung ihrer Zusammensetzung eine Mitteilung an registrierte Beobachter versenden, werden wir uns später noch beschäftigen (siehe Abschnitt 13.5.3.3). Im Beispiel wird die generische Fabrikmethode **observableArrayList()** der Klasse **FXCollections** dazu verwendet, ein Objekt aus einer Klasse zu erzeugen, die das Interface **ObservableList<String>** implementiert und einen Array zur Aufbewahrung ihrer Elemente verwendet. Auf die Angabe des Elementtyps **String** kann beim Aufruf der generischen Methode dank Typinferenz verzichtet werden.

Für die Sortierung sorgt eine Verpackung der **ObservableList<String>** durch ein Objekt der Klasse **SortedList<String>**, die ebenfalls das Interface **ObservableList<String>** implementiert. Dem Konstruktor muss über seinen zweiten Parameter ein **Comparator<? super String>** zur Verfügung gestellt werden. Wir lassen von der statischen Methode **naturalOrder()** der Schnittstelle **Comparator<T>** ein Objekt der Klasse **Comparator<String>** erstellen. Das resultierende Objekt der Klasse **SortedList<String>**...

- ist eine beobachtbare Kollektion
- und hält die Elemente im sortierten Zustand.

Dem **ListView<String>** - Steuerelement wird dieses Objekt per Konstruktorparameter bekanntgegeben, sodass sich das Steuerelement dort als Beobachter registrieren kann. Im Ergebnis erhalten wir ein Steuerelement, das automatisch die aktuellen Listenelemente im sortierten Zustand anzeigt.

Beim Aufnahmeschalter wird durch einen **setOnAction()** - Aufruf eine per Lambda-Ausdruck realisierte Ereignisbehandlungsmethode registriert:

```
btnAdd.setOnAction(event -> {
    String s = tfName.getText();
    if (s.length() > 0  &&  !persons.contains(s))
        persons.add(s);
});
```

Wenn im Textfeld `tfName` ein Eintrag (mit Länge > 0) vorhanden ist, und sich diese Zeichenfolge noch nicht in der beobachtbaren Liste befindet, dann wird der Name per **add()** - Aufruf in die beobachtbare Liste aufgenommen.

Analog erhält der Entlassungsschalter eine Klickbehandlungsmethode, die das im **ListView<String>** gewählte Element ermittelt und es per **remove()** aus der beobachtbaren Liste entfernt:

```
btnRemove.setOnAction(event ->
        persons.remove(lvPersons.getSelectionModel().getSelectedItem()));
```

Bei jeder Änderung der beobachtbaren Liste aktualisiert das **ListView<String>** - Objekt automatisch die Anzeige.

Über die **GridPane**-Methode **add()** mit Parametern für die nullbasierten Spalten- und Zeilennummern platziert man die Bedienelemente in die passenden Matrixzellen, wobei der Spaltenindex zuerst anzugeben ist:

```
root.add(lblName, 0, 0);
root.add(tfName, 1, 0);
root.add(btnAdd, 2, 0);
root.add(lblPresent, 0, 1);
root.add(lvPersons, 1, 1);
root.add(btnRemove, 2, 1);
```

Abschließend sorgen wir noch dafür, dass die **GridPane**-Komponente den verfügbaren Platz im Fenster stets vollständig nutzt, wobei in horizontaler Richtung das Texteingabefeld und in vertikaler Richtung die Liste von einem wachsenden Platzangebot profitieren sollen:

```
GridPane.setHgrow(tfName, Priority.ALWAYS);
GridPane.setVgrow(lvPersons, Priority.ALWAYS);
```

Bei der Gestaltung eines **GridPane**-Containers kann man vorübergehend Gitterlinien aktivieren, um das Design zu erleichtern:

```
root.setGridLinesVisible(true);
```

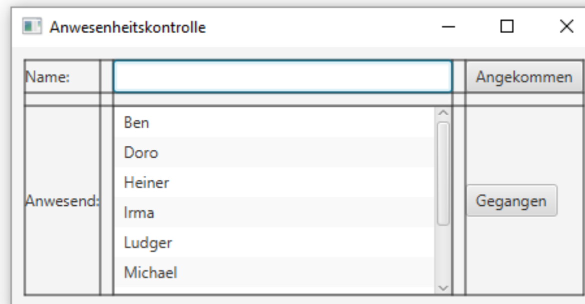
Im **Scene**-Konstruktor stellen wir eine passende initiale Fenstergröße ein:

```
primaryStage.setScene(new Scene(root, 450, 200));
```

Die Bühne erhält einen Titel und den Auftrag, mit der Vorstellung zu beginnen:

```
primaryStage.setTitle("Anwesenheitskontrolle");
primaryStage.show();
```

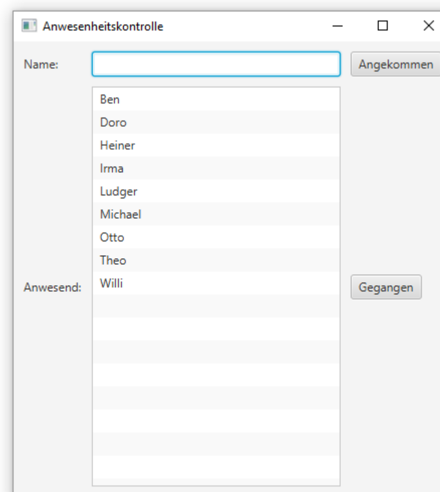
Das war‘s schon. Unser Programm ist einsatzbereit (hier noch mit Gitterlinien zur Layout-Begutachtung):



Eine initiale Fenstergröße festzulegen, ist nicht unbedingt erforderlich. Bei Verwendung eines einparametrischen **Scene**-Konstruktors

```
primaryStage.setScene(new Scene(root));
```

startet das Beispielprogramm so (ohne Gitterlinien zur Layout-Begutachtung):



Mit den **Stage**-Methoden **setMinWidth**() und **setMinHeight**() kann man den Benutzer daran hindern, das Fenster unbenutzbar klein zu machen, z. B.:

```
primaryStage.setMinWidth(400);
primaryStage.setMinHeight(150);
```

Die Anwendungsklasse ist (trotz der GUI-Gestaltung durch Anweisungen) noch gut zu überblicken:

```
package de.uni_trier.zimk.attendance;

import javafx.application.Application;
. . .
import java.util.Comparator;

public class Attendance extends Application {
    @Override
    public void start(Stage primaryStage) {
        double dist = 10.0;
        GridPane root = new GridPane();
        root.setPadding(new Insets(dist, dist, dist, dist));
```

```
        root.setHgap(dist); root.setVgap(dist);

        Label lblName = new Label("Name:");
        Label lblPresent = new Label("Anwesend:");
        TextField tfName = new TextField();
        Button btnAdd = new Button("Angekommen");
        Button btnRemove = new Button("Gegangen");

        String[] anwesend = new String[] {"Willi", "Otto", "Theo", "Irma", "Doro",
                "Heiner", "Michael", "Ludger", "Ben"};
        ObservableList<String> persons = FXCollections.observableArrayList(anwesend);
        SortedList<String> perSorted = new SortedList<>(persons,
                Comparator.naturalOrder());
        ListView<String> lvPersons = new ListView<>(perSorted);

        btnAdd.setOnAction(event -> {
            String s = tfName.getText();
            if (s.length() > 0  &&  !persons.contains(s))
                persons.add(s);
        });

        btnRemove.setOnAction(event ->
                persons.remove(lvPersons.getSelectionModel().getSelectedItem()));

        root.add(lblName, 0, 0);
        root.add(tfName, 1, 0);
        root.add(btnAdd, 2, 0);
        root.add(lblPresent, 0, 1);
        root.add(lvPersons, 1, 1);
        root.add(btnRemove, 2, 1);

        GridPane.setHgrow(tfName, Priority.ALWAYS);
        GridPane.setVgrow(lvPersons, Priority.ALWAYS);

        primaryStage.setScene(new Scene(root, 450, 200));
        primaryStage.setTitle("Anwesenheitskontrolle");
        primaryStage.setMinWidth(400);
        primaryStage.setMinHeight(150);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Bei komplexeren Bedienoberflächen ist die deklarative FXML-Alternative allerdings gegenüber der programmatischen GUI-Definition zu bevorzugen. Wir haben die FXML-basierte Technik schon im Abschnitt 4.9 verwendet und werden gleich im Abschnitt 13.4 weitere Erfahrungen damit sammeln.

Die **fxml**-Datei zur GUI-Deklaration und die Controller-Klasse sind im aktuellen Projekt überflüssig und können gelöscht werden. Ebenso kann die **throws**-Klausel zur **start**() - Methode entfallen, weil kein fehleranfälliges Laden einer **fxml**-Datei stattfindet.

13.4 Anwendung mit Model-View-Controller - Architektur (MVC)

13.4.1 Das Model-View-Controller - Konzept

Ein objektorientiertes Programm muss ...

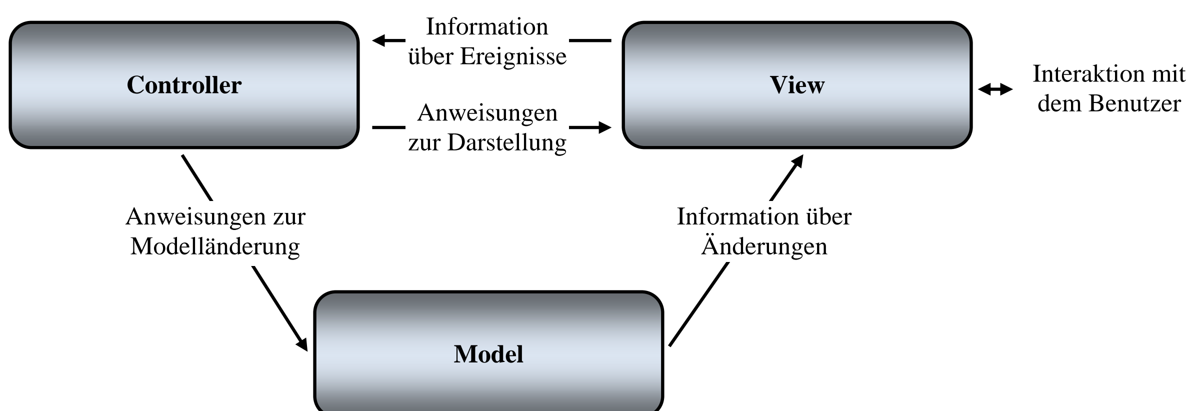
- die den Zustand des modellierten Systems repräsentierenden Daten unter Beachtung von Regeln der Geschäftslogik durch Anwendung von mehr oder weniger komplexen Algorithmen verwalten,
- mit den Benutzern interagieren, die meist mit Hilfe einer grafischen Bedienoberfläche Daten einsehen und verändern wollen.

Es ist allgemeiner Konsens, dass in einer objektorientierten Software die Verwaltung und Transformation der Daten nach den Regeln der Geschäftslogik getrennt werden sollte von der Präsentation der Daten und der Benutzerinteraktion (siehe z. B. Sharan 2015, S. 419ff). Seit ca. 30 Jahren werden Vorschläge zur Architektur von objektorientierter Software und zur Aufgabenverteilung entwickelt, wobei das zusammen mit der Programmiersprache Smalltalk eingeführte *Model-View-Controller* (*MVC*) - *Konzept* sehr oft, aber leider mit ziemlich variabler Bedeutung genannt wird (Lahres & Rayman 2009, Abschnitt 8.2).

Wir übernehmen von Hommel (2014) die folgenden Erläuterungen zu den Begriffen *Model*, *View* und *Controller*:

- **Model**
  Unter dem Model einer Anwendung soll die Sammlung der Klassen bzw. Objekte verstanden werden, welche die Daten verwalten und die anwendungsspezifischen Algorithmen realisieren. Sie veröffentlichen Daten (z. B. durch Mitteilungen an registrierte Beobachter) und erlauben eine kontrollierte Datenmodifikation durch Controller-Objekte (siehe unten).
- **View**
  Die View-Komponente einer JavaFX-Anwendung besteht aus den (meist per FXML deklarierten) Fenstern (Szenen). Als Knoten in den Szenegraphen der Fenster sind Bedienelemente vorhanden (z. B. aus den Klassen **ListView<E>**, **TextField**, **Button**), die eigenständig mit dem Benutzer interagieren, registrierte Controller-Objekte über eingetretene Ereignisse informieren (z. B. Mausklick auf einen Befehlsschalter) und von Model-Objekten übermittelte Daten darstellen.
- **Controller**
  Jedem Fenster (jeder Szene) wird eine Controller-Klasse mit Ereignisbehandlungsmethoden zugeordnet, welche sich von View-Knoten über Ereignisse informieren lassen, diese bewerten und ggf. Veränderungen bei Model-Objekten vornehmen. Neben der indirekten Beeinflussung von View-Knoten über die Model-Daten kommen auch direkte Modifikationen in Frage (z. B. das (De)aktivieren von View-Knoten).

In der folgenden Abbildung sind die Anwendungsbestandteile und ihre Kommunikationsbeziehungen dargestellt:



13.4.2 Projekt anlegen

Wir erstellen nun das im Abschnitt 13.2 beschriebene Beispielprogramm zur Anwesenheitskontrolle erneut ...

- unter Verwendung der View-Deklaration per FXML
- und mit Beachtung der MVC-Anwendungsarchitektur.

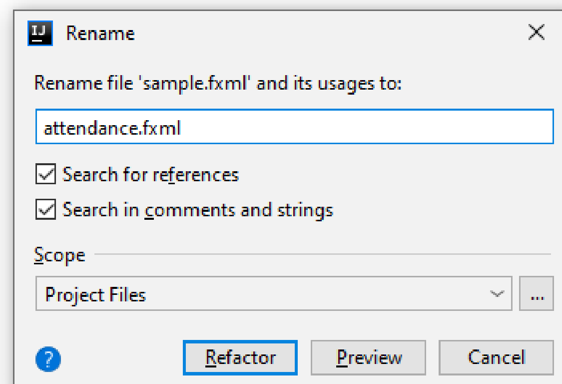
Die ersten Schritte beginnend mit dem Menübefehl

File > New > Project

laufen analog zu Abschnitt 13.2.1 ab:

- Das Projekt erhält den Namen `AttendanceMVC`.
- Das vom Assistenten angelegte Paket `sample` wird per Refaktorisierung umbenannt in `de.uni_trier.zimk.attendance`.
- Die vom Assistenten angelegte Klasse `Main` erhält den Namen `Attendance`, sodass in der **Run Configuration** der Name der **Main class** anzupassen ist.

Die vom Assistenten angelegte GUI-Deklarationsdatei **sample.fxml** sollte in **attendance.fxml** umbenannt werden:

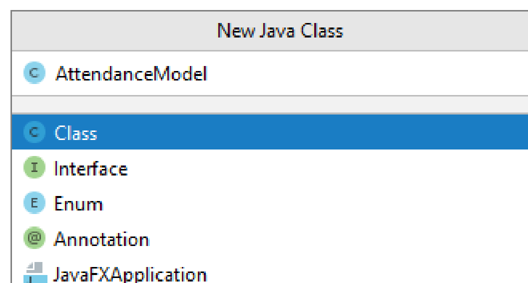


13.4.3 Model

Im Beispielprogramm soll das Model-Objekt ...

- zur Verwaltung der anwesenden Personen eine sortierte, beobachtbare Liste mit Elementen vom Typ **String** verwenden (siehe Abschnitt 13.5.3.3),
- eine nichtmodifizierbare Sicht auf die Anwesenheitsliste über die öffentliche Methode `getSortedList()` als Objekt vom Typ **ObservableList<String>** anbieten,
- über öffentliche Methoden namens `add()` und `remove()` das Einfügen bzw. Entfernen von Listenelementen erlauben, wobei das Einfügen nicht zu Dubletten führen darf.

Wir öffnen im **Project**-Fenster das Kontextmenü zum Paket `de.uni_trier.zimk.attendance` und wählen das Item **New > Java Class**, um die Klasse `AttendanceModel` anzulegen:



Die folgende Implementation der Klasse `AttendanceModel` kommt im Wesentlichen mit den im Abschnitt 13.3 erläuterten Techniken aus:

```
package de.uni_trier.zimk.attendance;

import java.util.Comparator;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.collections.transformation.SortedList;

public class AttendanceModel {
   private ObservableList<String> persons;
   private SortedList<String> perSorted;

   public AttendanceModel() {
      String[] anwesend = new String[] {"Willi", "Otto", "Theo", "Irma", "Doro",
                                        "Heiner", "Michael", "Ludger", "Ben"};
      persons = FXCollections.observableArrayList(anwesend);
      perSorted = new SortedList<>(persons, Comparator.naturalOrder());
   }

   public ObservableList<String> getSortedList() {
      return FXCollections.unmodifiableObservableList(perSorted);
   }

   public void add(String s) {
      if (!persons.contains(s))
         persons.add(s);
   }

   public void remove(String s) {
      persons.remove(s);
   }
}
```

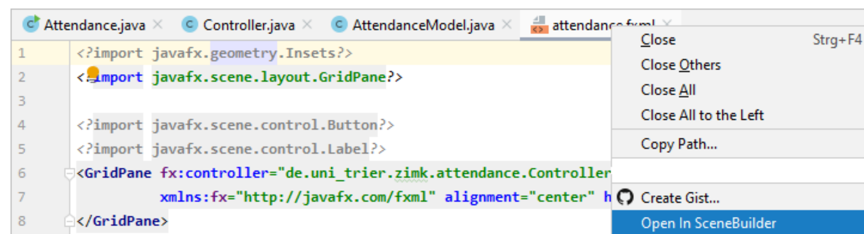
Das **ListView<String>** - Steuerelement der Bedienoberfläche benötigt eine Referenz auf die Anwesenheitsliste, um sich dort als Beobachter registrieren lassen zu können. Damit über diese Referenz keine Änderungen an der Anwesenheitsliste möglich sind, liefert `getSortedListe()` eine nicht-modifizierbare Sicht, die von der statischen Methode **unmodifiableObservableList**() der Klasse **FXCollections** erstellt wird:

```
public ObservableList<String> getSortedList() {
   return FXCollections.unmodifiableObservableList(perSorted);
}
```

13.4.4 GUI-Gestaltung per Scene Builder

Zum Öffnen der FXML-Datei **attendance.fxml** im Scene Builder bestehen zwei Möglichkeiten:

- Öffnen im IntelliJ-Rahmen
  Dazu wechselt man im Editor bei geöffneter Datei **attendance.fxml** zur Registerkarte **Scene Builder**.
- Öffnen als selbständiges Programm
  Diese Variante führt zu einer reichhaltigeren Scene Builder - Bedienoberfläche (z. B. mit Menü). Wir bevorzugen diese Option und wählen daher aus dem Kontextmenü der Editor-Registerkarte zur FXML-Datei das Item **Open in SceneBuilder**:



Wir markieren in der **Document**-Zone (unten links) des Scene Builders den vorhandenen **GridPane** - Wurzel-Container und setzen über das **Layout**-Segment der **Inspector**-Zone (am rechten Rand) seine bevorzugte Breite bzw. Höhe (**Pref Width** bzw. **Pref Height**) auf 450 bzw. 200. Diese Größen werden vom Anwendungsfenster übernommen, wenn wir in der Anwendungsklasse für das **Stage**-Objekt und damit für die Szene mit dem **GridPane**-Container als Wurzelknoten keine bevorzugte Größe angeben (siehe Abschnitt 13.4.7).

Über das Kontextmenü zum Wurzelknoten fügen wir 2 Zeilen und 3 Spalten ein:

GridPane > Add Row Above (2 mal)
Grid Pane > Add Column Before (3 mal)

Es ist zu beachten, dass in der **Editing**-Zone (in der Mitte) die Zeilen-Spalten-Struktur des **GridPane**-Containers erst dann angezeigt wird, wenn mindestens eine Zeile und mindestens eine Spalte vorhanden sind.

Aus dem **Controls**-Segment der **Library**-Zone befördern wir zwei **Label**-Komponenten, zwei **Button**-Komponenten, eine **ListView**-Komponente und eine **TextField**-Komponente in die gewünschte Gitterzelle.

Die **Label**- und **Button**-Komponenten werden jeweils nach einem Doppelklick beschriftet.

Für die markierte **TextField**-Komponente wird im **Layout**-Segment der **Inspector**-Zone die Eigenschaft **Hgrow** auf den Wert **Always** gesetzt. Für die markierte **ListView**-Komponente wird im **Layout**-Segment der **Inspector**-Zone die Eigenschaft **Vgrow** auf den Wert **Always** gesetzt. So wird dafür gesorgt, dass von einer wachsender Fensterbreite das Texteingabefeld und von einer wachsenden Fensterhöhe die Liste profitiert.

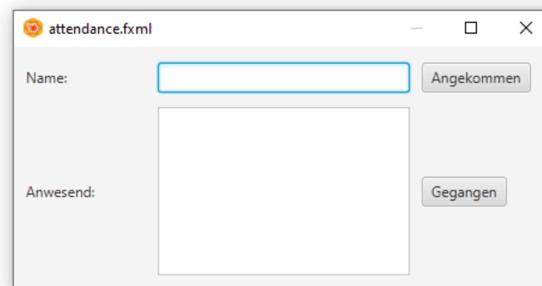
Um die Elemente im **GridPane**-Container vom Rand fernzuhalten, tragen wir bei markiertem Wurzelknoten im **Layout**-Segment der **Inspector**-Zone den Wert 10 in das erste Feld der **Padding**-Zeile ein und klicken dann auf den unmittelbar rechts danebenstehenden Pfeil, um diesen Wert für die restlichen Seiten zu übernehmen:

Denselben Wert 10 tragen wir auch für die **GridPane**-Layout-Eigenschaften **Hgap** und **Vgap** ein, um die Steuerelemente auf Abstand zu halten.

Wenn der Scene Builder als eigenständiges Programm läuft, kann man sich nach dem Menübefehl

Preview > Show Preview in Window

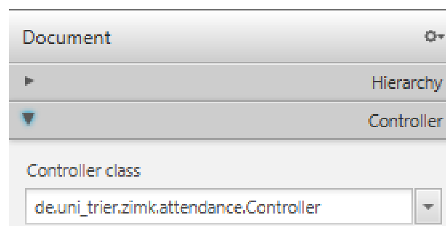
vom gelungenen Design überzeugen:



Um ein Steuerelement aus dem Szenegraphen später (z. B. zum Zweck der Ereignisbehandlung) mit einer Instanzvariablen der Controller-Klasse verknüpfen zu können, muss man dem Steuerelement eine Kennung, d.h. einen Wert für das FXML-Attribute **fx:id** zuordnen, was im **Code**-Segment der **Inspector**-Zone möglich ist. Wir vergeben die folgenden Kennungen:

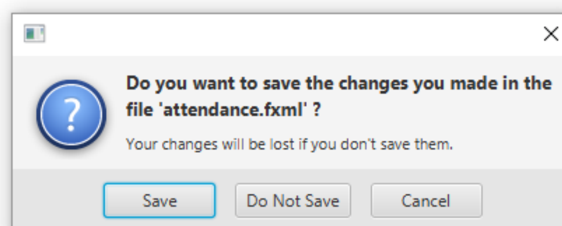
TextField-Komponente	`tfName`
ListView-Komponente	`lvPersons`
Button „Angekommen“	`btnAdd`
Button „Gegangen“	`btnRemove`

Wie das **Controller**-Segment der **Document**-Zone zeigt, ist die Controller-Klasse zum gerade entstehenden Fenster bereits eingetragen:



13.4.5 FXML

Wenn der Scene Builder als eigenständiges Programm verwendet wurde, beenden wir es nun und sichern unsere Arbeit:



Im FXML-Editor von IntelliJ wird der aktuelle Stand der FXML-Datei angezeigt:[1]

[1] Im **Run**-Fenster von IntelliJ erscheint beim Anwendungsstart eine Warnung, weil in der vom Assistenten erstellten FXML-Datei das JavaFX-API 11.0.1 avisiert, zum Starten aber das OpenJFX-SDK 8.0.232 verwendet wird:

```
WARNING: Loading FXML document with JavaFX API of version 11.0.1 by JavaFX runtime of version
8.0.242-ojdkbuild
```

Die Warnung unterbleibt, wenn in der FXML-Datei eine kompatible JavaFX-API - Version angegeben wird:

```
xmlns="http://javafx.com/javafx/8.0"
```

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.ListView?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.ColumnConstraints?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.RowConstraints?>

<GridPane alignment="CENTER" hgap="10" prefHeight="200.0" prefWidth="450.0" vgap="10"
          xmlns="http://javafx.com/javafx/11.0.1" xmlns:fx="http://javafx.com/fxml/1"
          fx:controller="de.uni_trier.zimk.attendance.Controller">
   <rowConstraints>
      <RowConstraints minHeight="10.0" prefHeight="30.0" />
      <RowConstraints minHeight="10.0" prefHeight="30.0" />
   </rowConstraints>
   <columnConstraints>
      <ColumnConstraints minWidth="10.0" prefWidth="100.0" />
      <ColumnConstraints minWidth="10.0" prefWidth="100.0" />
      <ColumnConstraints minWidth="10.0" prefWidth="100.0" />
   </columnConstraints>
   <children>
      <Label text="Name:" />
      <Label text="Anwesend:" GridPane.rowIndex="1" />
      <TextField fx:id="tfName" GridPane.columnIndex="1" GridPane.hgrow="ALWAYS" />
      <ListView fx:id="lvPersons" prefHeight="200.0" prefWidth="200.0"
                GridPane.columnIndex="1" GridPane.rowIndex="1" GridPane.vgrow="ALWAYS" />
      <Button fx:id="btnAdd" mnemonicParsing="false" text="Angekommen"
              GridPane.columnIndex="2" />
      <Button fx:id="btnRemove" mnemonicParsing="false" text="Gegangen"
              GridPane.columnIndex="2" GridPane.rowIndex="1" />
   </children>
   <padding>
      <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
   </padding>
</GridPane>
```

Die Datei startet mit sogenannten *XML-Verarbeitungsinstruktionen* (engl.: *processing Instructions*), die Anweisungen an den XML-Parser enthalten:

```
<? ... ?>
```

Auf eine Zeile mit der XML-Versions- und Kodierungsangabe folgen Importdeklarationen für Java-Klassen analog zu Abschnitt 3.1.7, wobei auch der Joker-Stern (*) erlaubt ist.

Das FXML-Wurzelelement namens **GridPane** enthält neben den im Scene Builder vorgenommenen Einstellungen noch XML-Namensraumvereinbarungen:

```
<GridPane alignment="CENTER" hgap="10" prefHeight="200.0" prefWidth="450.0" vgap="10"
          xmlns="http://javafx.com/javafx/11.0.1" xmlns:fx="http://javafx.com/fxml/1"
          fx:controller="de.uni_trier.zimk.attendance.Controller">
 . . .
</GridPane>
```

Weil die Vereinbarung eines umlaufenden Innenrandes für den **GridPane**-Container nicht per XML-Attribut erledigt werden kann, erscheint das eingeschachtelte Element **padding**:

```
<padding>
    <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
</padding>
```

Im **children**-Element werden die im **GridPane**-Container enthaltenen Knoten aufgelistet:

```
<children>
  . . . .
</children>
```

Die Elemente zu den Knotenobjekten tragen den Namen der realisierenden Java-Klasse. Im späteren Programmablauf werden die Bedienelemente über den parameterfreien Konstruktor ihrer Klasse erzeugt und dann konfiguriert.

Wie im folgenden Beispiel treten Attributnamen mit (z. B. **GridPane.rowIndex**) und ohne (z. B. **text**) Klassenpräfix auf:

```
<Button fx:id="btnRemove" mnemonicParsing="false" text="Gegangen"
        GridPane.columnIndex="2" GridPane.rowIndex="1" />
```

Bei Attributen ohne Klassenpräfix wird eine Instanzmethode der Steuerelementklasse zum Setzen einer Eigenschaftsausprägung aufgerufen (ein Property-Setter, vgl. Abschnitt 13.5). Bei Attributen mit Klassenpräfix wird eine statische Methode der Layoutmanager-Klasse aufgerufen (Horstmann 2014, S. 87), z. B.

```
GridPane.setColumnIndex(btnRemove, 2);
```

Weitere Hinweise zum FXML-Format finden sich z. B. bei Weaver et al (2014). Wie eine FXML-Datei beim Programmstart geladen wird, erfahren Sie im Abschnitt 13.4.7.

13.4.6 Controller

Die Ereignisbehandlung in unserem Projekt soll durch ein Objekt der Klasse `Controller` erledigt werden, die der IntelliJ-Assistent für neue Projekte bereits angelegt hat:

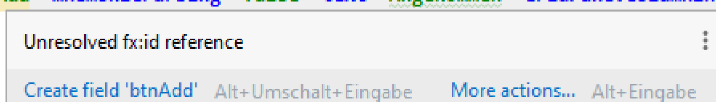
```
package de.uni_trier.zimk.attendance;

public class Controller {
}
```

Im FXML-Code haben einige GUI-Komponenten eine Kennung (**fx:id**) erhalten, damit sie von der Controller-Klasse angesprochen werden können. Um diese Ansprache zu ermöglichen, müssen in der Controller-Klasse noch passend benannte Instanzvariablen angelegt werden. Beim Laden der FXML-Datei (siehe Abschnitt 13.4.7) entsteht ein Objekt der Controller-Klasse, und es wird versucht, die in der FXML-Datei deklarierten und mit einer Kennung (**fx:id**) ausgestatteten GUI-Objekte in die Controller-Klasse zu „injizieren“ (Horstmann 2014, S, 88).

Gehen Sie folgendermaßen vor, um von IntelliJ eine Unterstützung bei der Deklaration von `Controller`-Instanzvariablen zu den GUI-Komponenten mit Kennung zu erhalten:

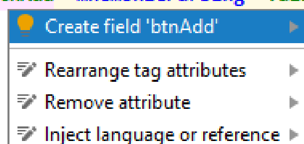
- Im Editorfenster der FXML-Datei sind die noch nicht zugeordneten Steuerelemente an der Hintergrundfarbe zu erkennen, z. B.:

```
<Button fx:id="btnAdd" mnemonicParsing="false" text="Angekommen" GridPane.columnIndex="2" />
```



- Setzen Sie die Einfügemarke auf die Kennung `btnAdd`, betätigen Sie die Tastenkombination **Alt+Enter**,

```
<Button fx:id="btnAdd" mnemonicParsing="false" text="Angekommen" GridPane.columnIndex="2" />
```



und lassen Sie das `Controller`-Feld zum Steuerelement durch Übernahme des ersten Korrekturvorschlags erstellen.

- Akzeptieren Sie für das neu angelegte Feld der `Controller`-Klasse

  ```
  public Button btnAdd;
  ```

  per **Enter** den vorgeschlagenen Datentyp.
- Verfahren Sie analog mit den GUI-Komponenten bzw. Feldern `btnRemove`, `tfName` und `lvPersons`.

Dass IntelliJ *öffentliche* Instanzvariablen verwendet, verstößt gegen das Prinzip der Datenkapselung:

```
public class Controller {
    public Button btnAdd;
    public Button btnRemove;
    public TextField tfName;
    public ListView lvPersons;
}
```

Wir wählen die empfehlenswerte Schutzstufe **private** und sorgen durch die Annotation **@FXML** dafür, dass die GUI-Komponenten über *private* Instanzvariablen der Controller-Klasse angesprochen werden können:

```
public class Controller {
   @FXML
   private Button btnAdd;
   @FXML
   private Button btnRemove;
   @FXML
   private TextField tfName;
   @FXML
   private ListView<String> lvPersons;
}
```

Außerdem sollte unbedingt der von IntelliJ vorgeschlagenen Rohtyp **ListView** durch den parametrisierten Typ **ListView<String>** ersetzt werden.

Wir ergänzen in der `Controller`-Klasse eine Instanzvariable vom Typ `AttendanceModel` (vgl. Abschnitt 13.4.3), weil der Controller mit dem Model kommunizieren soll:

```
private AttendanceModel am;
```

Zur Initialisierung des `Controller`-Objekts wird die parameterfreie Methode **initialize()** implementiert und mit **@FXML** annotiert, sodass sie beim Laden der FXML-Datei automatisch ausgeführt wird:[1]

---

[1] In JavaFX 2.2 wurde von einer Controller-Klasse noch gefordert, das Interface **Initializable** zu implementieren (siehe z. B. Horstmann 2014, S. 88). In der aktuellen OpenJFX-Dokumentation zu **Initializable** (https://openjfx.io/javadoc/13/javafx.fxml/javafx/fxml/Initializable.html) heißt es dazu:

> This interface has been superseded by automatic injection of `location` and `resources` properties into the controller. FXMLLoader will now automatically call any suitably annotated no-arg `initialize()` method defined by the controller.

Ab JavaFX 8 sollte also das im Beispiel demonstrierte Muster verwendet werden: Die Controller-Klasse besitzt eine parameterfreie Methode **initialize()** mit **void**-Rückgabe und **@FXML**-Annotation (siehe z. B. Weaver et al 2014, S. 96).

```
@FXML
public void initialize() {
   am = Attendance.getModel();

   lvPersons.setItems(am.getSortedList());

   btnAdd.setOnAction(event -> {
      String s = tfName.getText();
      if (s.length() != 0)
         am.add(s);
   });

   btnRemove.setOnAction(event ->
         am.remove(lvPersons.getSelectionModel().getSelectedItem()));
}
```

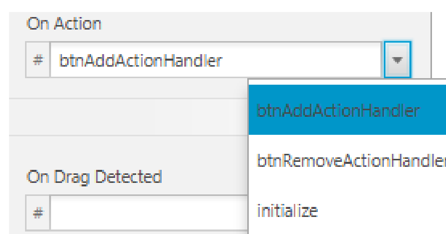
In **initialize()** ermittelt der Controller mit der (noch zu implementierenden) statischen `Attendance`-Methode `getModel()` das Model-Objekt und informiert die **ListView<String>** - Komponente durch einen **setItems()** - Aufruf darüber, welche Daten dargestellt werden sollen. Außerdem werden **ActionEvent**-Behandlungsmethoden für die beiden Befehlsschalter vereinbart, wobei die `AttendanceModel`-Instanzmethoden `add()` und `remove()` zum Einsatz kommen.

Eine Besonderheit der Controller-Klasse im Vergleich zu unseren bisherigen Erfahrungen mit Java besteht darin, dass in der Methode **initialize()** die Felder mit Referenzen zu den Bedienelementen verwendet werden, die vom JavaFX-Framework (per Injektion) initialisiert worden sind.

Im Abschnitt 4.9.4 haben wir einen Befehlsschalter mit seiner **ActionEvent**-Behandlungsmethode verknüpft, indem wir ...

- in der Controller-Klasse eine Instanzmethode erstellt
- und diese per Scene Builder als Wert des FXML-Attributs **onAction** festgelegt haben.

Wir könnten im aktuellen Beispiel analog vorgehen und den beiden **Button**-Steuerelementen im **Code**-Segment der **Inspector**-Zone des Scene Builders eine **ActionEvent**-Behandlungsmethode der Controller-Klasse zuweisen, z. B.:



Im FXML-Code würde die Vereinbarung der Ereignisbehandlungsmethoden so umgesetzt werden:

```
<Button fx:id="btnAdd" mnemonicParsing="false" onAction="#btnAddActionHandler"
        text="Angekommen" GridPane.columnIndex="2" />
<Button fx:id="btnRemove" mnemonicParsing="false" onAction="#btnRemoveActionHandler
        text="Gegangen" GridPane.columnIndex="2" GridPane.rowIndex="1" />
```

An dieser Vorgehensweise ist jedoch die unvollständige Trennung von Programmlogik und Präsentation zu bemängeln (Inden 2015, S. 165f). Wir haben daher die Zuordnung in der `initialize()` - Methode der Controller-Klasse vorgenommen.

13.4.7 Anwendungsklasse

In der Anwendungsklasse wird ein statisches Feld definiert, das auf ein Objekt der Model-Klasse zeigen soll:

```
static private AttendanceModel am;
```

In der **start()** - Methode der Anwendungsklasse `Attendance` wird ...

- ein Objekt der Model-Klasse angelegt
  ```
  am = new AttendanceModel();
  ```
  Bei der späteren Initialisierung des Controller-Objekts wird auf das Model-Objekt zugegriffen.
- in einer **try-catch** - Anweisung die FXML-Datei geladen
  Der zuständige **FXMLLoader** liefert als Rückgabe eine Referenz vom Typ **Parent** (siehe Klassenhierarchie im Abschnitt 13.2.3) auf das Wurzelelement des Szenegraphen:
  ```
  Parent root = null;
  try {
      root = FXMLLoader.load(getClass().getResource("attendance.fxml"));
  } catch (Exception e) {
       System.out.println("Fehler beim Lesen der Layout-Spezifikation");
       e.printStackTrace();
       Platform.exit();
  }
  ```
  Weil in der FXML-Datei eine Controller-Klasse eingetragen ist, wird versucht, ein Objekt dieser Klasse zu erzeugen und FXML-Elemente, die über eine **fx:id** verfügen, mit den gleichnamigen Instanzvariablen des Controller-Objekts zu verbinden.
- die JavaFX-typische Initialisierung einer Szene auf der primären Bühne vorgenommen:
  ```
  primaryStage.setScene(new Scene(root));
  primaryStage.setTitle("Anwesenheitskontrolle");
  primaryStage.setMinWidth(400);
  primaryStage.setMinHeight(150);
  primaryStage.show();
  ```
  Weil der **Scene**-Konstruktoraufruf keine Aktualparameterwerte zur Fenstergröße enthält, kommen Voreinstellungen zum Einsatz, die sich an den gewünschten Ausdehnungen der im Fenster enthaltenen Komponenten orientieren.

Außerdem benötigt die Anwendungsklasse noch eine statische Methode namens `getModel()`, mit der sich das Controller-Objekt in seiner Methode **initialize()** eine Referenz zum Model-Objekt besorgen kann:

```
static public AttendanceModel getModel() {
    return am;
}
```

Die **main()** - Methode einer JavaFX-Anwendung darf fehlen, wird aber in der Regel doch kodiert:

```
public static void main(String[] args) {
    launch(args);
}
```

Die Anwendungsklasse im Überblick:

```
package de.uni_trier.zimk.attendance;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Attendance extends Application {
    static private AttendanceModel am;

    @Override
    public void start(Stage primaryStage) {
        am = new AttendanceModel();
        Parent root = null;
        try {
            root = FXMLLoader.load(getClass().getResource("attendance.fxml"));
        } catch (Exception e) {
            System.out.println("Fehler beim Lesen der Layout-Spezifikation");
            e.printStackTrace();
            Platform.exit();
        }
        primaryStage.setScene(new Scene(root));
        primaryStage.setTitle("Anwesenheitskontrolle");
        primaryStage.setMinWidth(400);
        primaryStage.setMinHeight(150);
        primaryStage.show();
    }

    static public AttendanceModel getModel() {
        return am;
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Das auf den folgenden vier Dateien basierende Programm ist nun einsatzfähig:

- **AttendanceModel.java** mit der Model-Klasse
- **attendance.fxml** mit der View-Deklaration
  Darauf basiert wesentlich das in **start()** erstellte **Scene**-Objekt, das an die **Stage**-Methode **setScene()** übergeben wird. Die View-Komponente ist ein komplexes Gebilde unter Beteiligung des JavaFX-Frameworks, das z. B. die Bedienelemente erstellt und deren Adressen in die Controller-Klasse injiziert.
- **Controller.java** mit der Controller-Klasse
- **Attendance.java** mit der Anwendungsklasse

Bei der im Abschnitt 13.4 vorgeschlagenen MVC-Lösung wurde das in FXML enthaltenen Controller-Konzept (siehe z. B. das Attribute **fx:controller** im Element **GridPane**) übernommen. Es sind viele andere Realisationen der MVC-Idee möglich, insbesondere bei einer GUI-Erstellung per Programm (siehe z. B. Sharan 2015, Kapitel 11).

13.5 Properties mit Änderungssignalisierung und automatischer Synchronisation

JavaFX-Komponenten (z. B. für die Steuerelemente und Zeichnungselemente in einer grafischen Bedienoberfläche) verwenden für ihre öffentlich zugänglichen Eigenschaften (z. B. Text in einem Eingabefeld, aktueller Wert eines Schiebereglers) Objekte aus speziellen Property-Klassen, die wir

uns nun ansehen. In der Dokumentation zu einer JavaFX-Komponente erscheinen die Eigenschaften an prominenter Stelle (ganz oben) und oft in enormer Anzahl, z. B. bei der **Label**-Komponente (vgl. Abschnitt 13.7.1):

Property Summary

Properties

Type	Property	Description
ObjectProperty<Node>	labelFor	A Label can act as a label for a different Control or Node.

Properties inherited from class javafx.scene.control.Labeled

alignment, contentDisplay, ellipsisString, font, graphic, graphicTextGap, labelPadding, lineSpacing, mnemonicParsing, textAlignment, textFill, textOverrun, text, underline, wrapText

Properties inherited from class javafx.scene.control.Control

contextMenu, skin, tooltip

Properties inherited from class javafx.scene.layout.Region

background, border, cacheShape, centerShape, height, insets, maxHeight, maxWidth, minHeight, minWidth, opaqueInsets, padding, prefHeight, prefWidth, scaleShape, shape, snapToPixel, width

Properties inherited from class javafx.scene.Parent

needsLayout

Properties inherited from class javafx.scene.Node

accessibleHelp, accessibleRoleDescription, accessibleRole, accessibleText, blendMode, boundsInLocal, boundsInParent, cacheHint, cache, clip, cursor, depthTest, disabled, disable, effectiveNodeOrientation, effect, eventDispatcher, focused, focusTraversable, hover, id, inputMethodRequests, layoutBounds, layoutX, layoutY, localToParentTransform, localToSceneTransform, managed, mouseTransparent, nodeOrientation, onContextMenuRequested, onDragDetected, onDragDone, onDragDropped, onDragEntered, onDragExited, onDragOver, onInputMethodTextChanged, onKeyPressed, onKeyReleased, onKeyTyped, onMouseClicked, onMouseDragEntered, onMouseDragExited, onMouseDragged, onMouseDragOver, onMouseDragReleased, onMouseEntered, onMouseExited, onMouseMoved, onMousePressed, onMouseReleased, onRotate, onRotationFinished, onRotationStarted, onScrollFinished, onScroll, onScrollStarted, onSwipeDown, onSwipeLeft, onSwipeRight, onSwipeUp, onTouchMoved, onTouchPressed, onTouchReleased, onTouchStationary, onZoomFinished, onZoom, onZoomStarted, opacity, parent, pickOnBounds, pressed, rotate, rotationAxis, scaleX, scaleY, scaleZ, scene, style, translateX, translateY, translateZ, viewOrder, visible

Die Property-Klassen von JavaFX gehen in ihrer Funktionalität weit über die traditionelle Realisation von Komponenten-Eigenschaften hinaus. Insbesondere lassen sich JavaFX-Properties auf elegante Weise miteinander verknüpfen, sodass eine automatische Synchronisation ihrer Werte stattfindet.

JavaFX-Properties mit ihrer Fähigkeit zur automatischen Synchronisation lassen sich nicht nur in Steuerelementen nutzen, sondern in beliebigen Klassen.

13.5.1 Properties

Unter der Bezeichnung Namen *Properties* hat JavaFX beobachtbare Werte eingeführt. Sie werden realisiert durch Klassen im Paket **javafx.beans.property**, deren Objekte einen Wert kapseln und registrierte Interessenten informieren, wenn sich der Wert geändert hat oder ungültig geworden ist.

13.5.1.1 Traditionelle JavaBean-Eigenschaften

Wir wissen seit Beginn des Kurses, dass in objektorientierten Programmen die Eigenschaften von Objekten eine wichtige Rolle spielen:

- Sie repräsentieren die Zustandsdaten.
- Die letztlich zugrunde liegenden Instanzvariablen sind im Sinne der Datenkapselung vor dem direkten Zugriff durch fremde Klassen geschützt.
- Über öffentliche Zugriffsmethoden, die oft als *getter* bzw. *setter* bezeichnet werden, sind Lese und/oder Schreibzugriffe möglich.

Als *JavaBeans* werden in Java seit langer Zeit Klassen bezeichnet, die für Eigenschaften öffentliche Zugriffsmethoden mit genormten Namen besitzen:

- Auf die Einleitung durch *get* bzw. *set* folgt der Name der Eigenschaft (z. B. `getNumerator()`, `setNumerator()`).
- Bei Eigenschaften mit dem Datentyp **boolean** wird die Abfragemethode mit *is* eingeleitet (z. B. `isSelected()`).

Als Rückgabetyp für die setter hat sich **void** eingebürgert.[1]

Neben den Namenskonventionen besteht die JavaBeans-Komponententechnologie aus einem API mit etlichen Klassen und Schnittstellen im Paket **java.beans**. Mit Hilfe der Klasse **PropertyChangeSupport** kann man z. B. eine Liste von Beobachtern verwalten, die durch **PropertyChangeEvents** über Veränderungen bei den Eigenschaften einer JavaBeans-Komponente informiert werden sollen.

Es resultieren *Komponenten*, die sich gut für die Wiederverwendung und für die Unterstützung durch Entwicklungswerkzeuge eignen. Typische Beispiele für JavaBeans-Komponenten sind die Steuerelementen im GUI-Framework Swing. Weiterführende Informationen zur JavaBeans-Komponententechnologie bietet z. B. das Java Tutorial (Oracle 2019a).[2]

Traditionelle JavaBean-Eigenschaften machen einigen Aufwand bei der Verwaltung von Beobachtern, den man sich mit den Property-Klassen von JavaFX ersparen kann.

13.5.1.2 Property-Klassen von JavaFX

JavaFX-Komponenten (z. B. für die Steuerelemente in einer grafischen Bedienoberfläche) speichern Eigenschaften mit der Fähigkeit zum Versand von Veränderungsmitteilungen in Objekten von speziellen Property-Klassen. Auch für JavaFX-Properties werden in der Regel Zugriffsmethoden (getter und setter) unter Beachtung der traditionellen Namenskonventionen für JavaBeans-Eigenschaften implementiert. Zusätzlich bietet eine JavaFX-Komponente zu einem Property-Objekt eine Zugriffsmethode mit einer Referenz auf dieses Objekt als Rückgabe. Diese Rückgabe wird z. B. benötigt, um bei einem Property-Objekt einen Beobachter für Wertveränderungen zu registrieren.

Für eine JavaFX-Komponentenklasse gelten folgende Namensregeln:

- Die Instanzvariable zu einem Property-Objekt trägt einen Namen gemäß dem Camel Casing - Prinzip (siehe Abschnitt 3.3.2), der anschließend als *Eigenschaftsname* bezeichnet werden soll (z. B. `caretPosition`).
- In den Namen der Methoden für den lesenden bzw. schreibenden Zugriff folgt (wie bei JavaBeans) auf *get* bzw. *set* der Eigenschaftsname mit einem groß geschriebenen Anfangsbuchstaben (z. B. `getCaretPosition()` bzw. `setCaretPosition()`).
- Im Namen der Methode zum Erfragen der Property-Referenz folgt auf den Eigenschaftsnamen der Zusatz *Property* (z. B. `caretPositionProperty`).

Im Paket **javafx.beans.property** befinden sich für die Datentypen **boolean**, **double**, **float**, **int**, **long**, **Object**, **String**, **List<E>**, **Map<K,V>** und **Set<E>** jeweils vier Klassen, z. B. beim Typ **int**:

---

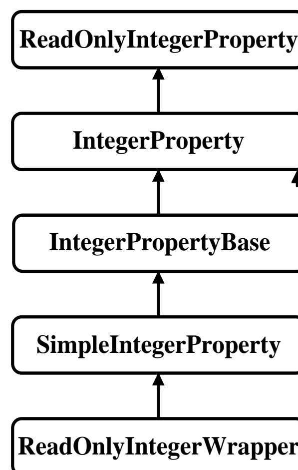
[1] Das Einführungsbeispiel im Abschnitt 1.1 enthält getter- und setter-Methoden, die sich wenig an den JavaBeans-Konventionen orientieren:

- Es werden deutsche Namen verwendet (z. B. `setzeNenner()`).
- Die Methode `setzeNenner()` verwendet nicht den Rückgabetyp **void**, sondern signalisiert durch eine Rückgabe vom Typ **boolean**, ob die gewünschte Wertzuweisung durchgeführt wurde. Fremde Programmierer ignorieren eventuell den unkonventionellen Rückgabewert. Bei einer für die Öffentlichkeit gedachten Klasse sollte als Warnung vor einer nicht ausgeführten Wertzuweisung besser eine Ausnahme geworfen werden vgl. Kapitel 11).

[2] http://download.oracle.com/javase/tutorial/javabeans

- **IntegerProperty**
  Diese abstrakte Klasse eignet sich als Datentyp für **int**-Properties.
- **SimpleIntegerProperty**
  Diese konkrete, von **IntegerProperty** abstammende Klasse wird zum Instanzieren von **int**-Properties verwendet, wenn der Lese- *und* der Schreibzugriff möglich sein sollen.
- **ReadOnlyIntegerProperty**
  Diese Basisklasse von **IntegerProperty** wird verwendet, wenn eine Klasse eine **int**-Property für den ausschließlich lesenden Zugriff durch fremde Klassen anbieten möchte.
- **ReadOnlyIntegerWrapper**
  Diese Klasse stammt von **SimpleIntegerProperty** ab und stellt *Anbieter-intern* eine **int**-Property zur Verfügung. Über die Methode **getReadOnlyProperty()** liefert ein Wrapper ein Objekt der Klasse **ReadOnlyIntegerProperty**, das automatisch mit dem internen **SimpleIntegerProperty**-Objekt synchronisiert wird.

Weil in der Auflistung die Abstammungsverhältnisse der Klassen nicht leicht zu überblicken waren, folgt noch eine grafische Darstellung:



Ein **IntegerProperty**-Objekt beherrscht u.a. die folgenden Methoden:

- **public int get()**
  Diese Methode liefert den gekapselten Wert vom Typ **int**.
- **public void set(int** *value***)**
  Mit dieser Methode verändert man den gekapselten Wert.
- **public String getName()**
  Über diese Methode erfährt man den Namen der Property.
- **public Object getBean()**
  JavaFX-Properties können per Konstruktor einem Objekt (einer Komponente) zugeordnet werden (siehe unten) und per **getBean()** nach diesem Objekt befragt werden. Nicht zugeordnete Properties antworten mit der Rückgabe **null**.

Die Methoden **addListener()** und **removeListener()** zum Registrieren von Beobachtern werden im Abschnitt 13.5.2 behandelt, und über die Methoden **bind()**, **unbind()**, **bindBidirectional()** und **unbindBidirectional()** zur automatischen Synchronisation von Property-Objekten berichtet Abschnitt 13.5.3.

Wir betrachten nun die Verwendung von JavaFX-Properties im Rahmen einer simplen Klasse namens `Person`, die folgende Properties enthält:

- Für den Vor- und den Nachnamen sind die **StringProperty**-Objekte `firstName` und `lastName` vorhanden.
- Für das Alter ist die **IntegerProperty** age vorhanden.
- Für die Personalnummer ist der **ReadOnlyIntegerWrapper** `id` vorhanden. Auf die automatisch inkrementierten Werte dieser Eigenschaft soll die Außenwelt nur lesend zugreifen können.

Im folgenden Quellcode

```
public class Person {
   private static int count = 0;
   private ReadOnlyIntegerWrapper id = new ReadOnlyIntegerWrapper(this, "id", ++count);
   private StringProperty firstName = new SimpleStringProperty(this, "firstName", "");
   private StringProperty lastName = new SimpleStringProperty(this, "lastName", "");
   private IntegerProperty age = new SimpleIntegerProperty(this, "age", -99);

   public Person(String first, String last, int a) {
      firstName.setValue(first);
      lastName.setValue(last);
      age.setValue(a);
   }
  . . .
}
```

kommen Property-Konstruktoren mit drei Parametern zum Einsatz:

- Im ersten Parameter (**Object** *bean*) wird festgelegt, zu welchem Objekt die Property gehört. Im Beispiel wird jeweils das gerade erzeugte `Person`-Objekt eingetragen.
- Im zweiten Parameter (**String** *name*) erhält die Property einen Namen.
- Im dritten Parameter wird der initiale Wert festgelegt.

Es sollten auch die bei JavaBean-Eigenschaften üblichen Zugriffsmethoden unter Beachtung der im Abschnitt 13.5.1.1 beschriebenen Namensregeln erstellt werden, z. B. für die Property `firstName`:

```
public final String getFirstName() {
   return firstName.get();
}
public final void setFirstName(String first) {
   firstName.set(first);
}
```

Durch das Finalisieren wird verhindert, dass die Methoden in abgeleiteten Klassen überschrieben werden.

Die folgende Methode liefert eine Referenz auf das **StringProperty**-Objekt `firstName`:

```
public final StringProperty firstNameProperty() {
   return firstName;
}
```

Eine Property-Referenz wird benötigt, um ...

- Veränderungs- bzw. Invalidierungs-Listener zu registrieren (siehe Abschnitt 13.5.2)
- Bindungen vorzunehmen (siehe Abschnitt 13.5.3).

Bei der Read-Only - Property `id` wird keine Setter-Methode implementiert, und der Referenz-Getter **idProperty()** liefert unter Verwendung der **ReadOnlyIntegerWrapper**-Methode **getReadOnlyProperty()** ein Objekt der Klasse **ReadOnlyIntegerProperty**, das automatisch mit der intern verfügbaren **IntegerProperty** synchronisiert wird:

```
public final int getId() {
   return id.get();
}
public final ReadOnlyIntegerProperty idProperty() {
   return id.getReadOnlyProperty();
}
```

Das folgende Programm zeigt die Verwendung der `Person`-Properties, kann aber noch nicht demonstrieren, wozu der im Vergleich zu gewöhnlichen gekapselten Instanzvariablen höhere Property-Aufwand taugt:

```
class PersonDemo {
   public static void main(String[] args) {
      Person p = new Person("Otto", "Rempremerding", 89);
      System.out.println(p.getId() + "\n" + p.getFirstName()
            + "\n" + p.getLastName() + "\n" + p.getAge());
      p.setFirstName("Ludwig");
      p.setLastName("Thoma");
      p.setAge(76);
//    p.setId(2); //verboten
   }
}
```

Während der Lesezugriff bei allen Properties möglich ist, kann das **ReadOnlyIntegerProperty**-Objekt im Unterschied zu den **StringProperty**-Objekten nicht verändert werden. In `Person`-Methoden ist aber eine Veränderung des synchronen **ReadOnlyIntegerWrapper**-Objekts durchaus erlaubt.

Im weiteren Verlauf von Abschnitt 13.5 werden die Fähigkeiten von Property-Objekten zum Versand von Änderungsbenachrichtigungen und zur automatischen Synchronisation beschrieben.

13.5.1.3 Vermeidung von überflüssigen Objektkreationen

Für die Komfortfunktionen der JavaFX-Properties muss man mit zahlreichen Objekt-Kreationen bezahlen. Wenn das zum Problem wird, lassen sich Objektkreationen mit einem simplen Trick hinauszögern und vielfach vermeiden. Man wickelt zu einer JavaFX-Property die Speicherung sowie Lese und Schreibzugriffe wie bei einer traditionellen JavaBean-Eigenschaft zunächst über eine gewöhnliche Instanzvariable ab. Das Property-Objekt wird erst dann erstellt, wenn eine Referenz auf dieses Objekt angefordert wird.

Anschließend wird das Verfahren mit einer abgespeckten Version der im Abschnitt 13.5.1.2 vorgestellten Klasse `Person` vorgeführt:

```
import javafx.beans.property.*;

public class Person {
   private StringProperty firstName;
   private String firstNameB = "";

   public final String getFirstName() {
      if (firstName == null)
         return firstNameB;
      else
         return firstName.get();
   }
```

```
    public final void setFirstName(String first) {
       if (firstName == null)
          firstNameB = first;
       else
          firstName.set(first);
    }

    public final StringProperty firstNameProperty() {
       if (firstName == null) {
          firstName = new SimpleStringProperty(this, "firstName", firstNameB);
          System.out.println("Property created");
       }
       return firstName;
    }
 }
```

Erst ein Aufruf der Methode `firstNameProperty()` führt zur Objektkreation:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>
class LacyPropertyCreation {
  public static void main(String[] args) {
    Person p = new Person();
    p.setFirstName("Otto");
    System.out.printf("Vorname: %s\n\n", p.getFirstName());
    p.firstNameProperty().addListener((obs, old, nev) ->
        System.out.println("Vorname geändert in: " + nev));
    p.setFirstName("Otti");
  }
}
</pre></td><td><pre>
Vorname: Otto

Property created
Vorname geändert in: Otti
</pre></td></tr>
</table>

13.5.2 Invalidierungs- und Veränderungsereignisse

Während bei einer JavaBean-Eigenschaft die Verwaltung von Interessenten für Veränderungsmitteilungen einigen Aufwand verursacht (siehe z. B. Epple 2015, Abschnitt 3.1), sind JavaFX-Properties für diese Aufgabe gut vorbereitet:

- Alle JavaFX-Properties implementieren das Interface **Observable**. Mit **addListener**()bzw. **removeListener**() kann ein Objekt aus einer Klasse, die das Interface **InvalidationListener** erfüllt, in die Liste der Interessenten für Invalidierungsnachrichten aufgenommen bzw. daraus entfernt werden. Die **InvalidationListener**-Methode **invalidated**() wird aufgerufen, wenn die beobachtete Eigenschaft ungültig geworden ist und erhält dabei als Aktualparameter eine Referenz auf die beobachtete Eigenschaft.
- Alle JavaFX-Properties implementieren das von **Observable** abstammende Interface **ObservableValue<T>**. Mit **addListener**() bzw. **removeListener**() kann ein Objekt aus einer Klasse, die das Interface **ChangeListener<? super T>** erfüllt, in die Liste der Interessenten für Veränderungsnachrichten aufgenommen bzw. daraus entfernt werden. Die **ChangeListener<T>** - Methode **changed**() wird bei jeder Wertveränderungen aufgerufen und erhält dabei als Aktualparameter eine Referenz auf die beobachtete Eigenschaft sowie den alten und den neuen Wert:

  public void changed (ObservableValue<? extends T> *observable*, **T** *oldVal*, **T** *newVal***)**

  Bei den Property-Klassen zur Verwaltung eines primitiven Werts (z. B. **IntegerProperty**) ist **T** notwendigerweise ein Referenztyp (z. B. **Integer**).

Nach der Kreation ist ein Property-Objekt im gültigen Zustand. Bei einem Property-Objekt in gültigem Zustand hat eine Wertveränderung folgende Effekte:

- Das Objekt wechselt in den ungültigen Zustand.
- Es feuert ein Invalidierungsereignis.

Bei einem ungültigen Property-Objekt haben weitere Wertveränderungen keine Invalidierungsereignisse zur Folge. Eine Wertabfrage per **get()** - Aufruf bringt ein ungültiges Property-Objekt wieder in den gültigen Zustand.

Das beschriebene Verhalten ist im folgenden Programm zu beobachten:

Quellcode

```
import javafx.beans.property.*;

class InvalidationListener {
  public static void main(String[] args) {
    IntegerProperty inum = new SimpleIntegerProperty();
    inum.addListener(obs->System.out.println("Invalidated"));
    System.out.println("Wert nach Konstruktor: "+inum.get());
    inum.set(1);
    inum.set(2);
    System.out.println("Wert = " + inum.get());
    inum.set(3);
  }
}
```

Ausgabe

```
Wert nach Konstruktor: 0
Invalidated
Wert = 2
Invalidated
```

Wie das folgende Programm zeigt, informiert ein Property-Objekt registrierte Change-Listener über *jede* Wertveränderung:

Quellcode

```
import javafx.beans.property.*;

public class ChangeListener {
  public static void main(String[] args) {
    IntegerProperty inum = new SimpleIntegerProperty();
    inum.addListener((obs, old, nev) ->
        System.out.println("Changed to: " + nev));
    System.out.println("Wert nach Konstruktor: "+inum.get());
    inum.set(1);
    inum.set(2);
  }
}
```

Ausgabe

```
Wert nach Konstruktor: 0
Changed to: 1
Changed to: 2
```

13.5.3 Automatische Synchronisation von Property-Objekten

Ein Invalidation- bzw. Change-Listener kann in seiner Methode **invalidated()** bzw. **changed()** beliebige Aktionen ausführen. Oft soll aber auf eine Wertveränderung bei einer Property lediglich mit einer Wertanpassung bei anderen Properties reagiert werden. Das lässt sich durch eine uni- oder bidirektionale Bindung leichter realisieren als durch Listener:

- Ein Property-Objekt kann uni- oder bidirektional mit einem anderen Property-Objekt vom gleichen Typ verbunden werden.
- Ein Property-Objekt kann an ein Berechnungsergebnis gebunden werden, zu dem mehrere Objekte vom Typ **ObservableValue<T>** (z. B. mehrere Property-Objekte) beitragen.

13.5.3.1 Uni- und bidirektionale Synchronisation von Property-Objekten

JavaFX-Properties können aneinander gebunden werden, sodass eine (uni- oder bidirektionale) Synchronisation ihrer Werte stattfindet, ohne dass sich Programmierer abmühen müssen. Alle JavaFX-Properties implementieren das von **ReadOnlyProperty<T>** und **WritableValue<T>** ab-

stammende Interface **Property<T>**, das folgende Methoden zur Unterstützung der Datenbindung vorschreibt:

- **public void bind(ObservableValue<? extends T>** *observable***)**
  Mit dieser Methode wird ein Property-Objekt unidirektional an ein Objekt vom Typ **ObservableValue<? extends T>** gebunden (z. B. an ein anderes Property-Objekt), das als Quelle fungiert. Das angesprochene (und gebundene) Objekt übernimmt sofort den Wert der Quelle. Direkte Schreibzugriffe auf das Zielobjekt der Bindung sind verboten.
- **public void unbind()**
  Mit dieser Methode wird eine unidirektionale Verbindung aufgehoben.
- **public void bindBidirectional(Property<T>** *other***)**
  Mit dieser Methode wird ein Property-Objekt bidirektional mit einem anderen Property-Objekt vom selben Typ verbunden. Das angesprochene Objekt übernimmt sofort den Wert des Parameterobjekts.
- **public void unbindBidirectional(Property<T>** *other***)**
  Mit dieser Methode wird eine bidirektionale Verbindung aufgehoben.
- **public boolean isBound()**
  Diese Methode informiert darüber, ob das angesprochene Property-Objekt gebunden ist.

Im folgenden Programm werden zwei Objekte vom Typ **SimpleIntegerProperty** bidirektional verbunden:

```
import javafx.beans.property.*;

class PropBiSync {
  public static void main(String[] args) {
    IntegerProperty i1 = new SimpleIntegerProperty(1);
    IntegerProperty i2 = new SimpleIntegerProperty(2);

    i1.addListener((obs, old, nev) -> System.out.println("i1 nun gleich "+i1.get()));
    i2.addListener((obs, old, nev) -> System.out.println("i2 nun gleich "+i2.get()));

    i1.bindBidirectional(i2);

    System.out.println("\nÄnderungen nach bidirektionaler Verbindung:");
    i1.set(11);
    i2.set(22);

    System.out.println("\nBidirektionale Verbindung aufgehoben:");
    i1.unbindBidirectional(i2);
    i1.set(111);
  }
}
```

Über die Change-Listener kann die (ausbleibende) Fortpflanzung von Änderungen beobachtet werden:

```
i1 nun gleich 2

Änderungen nach bidirektionaler Verbindung:
i1 nun gleich 11
i2 nun gleich 11
i2 nun gleich 22
i1 nun gleich 22

Bidirektionale Verbindung aufgehoben:
i1 nun gleich 111
```

In der nächsten Programmvariante sorgt eine unidirektionale Verbindung

```
i1.bind(i2);
```

dafür, dass nur Wertveränderungen (`i2` → `i1`) propagiert werden:

```
import javafx.beans.property.*;

class PropUniSync {
  public static void main(String[] args) {
    IntegerProperty i1 = new SimpleIntegerProperty(1);
    IntegerProperty i2 = new SimpleIntegerProperty(2);

    i1.addListener((obs, old, nev) -> System.out.println("i1 nun gleich "+i1.get()));
    i2.addListener((obs, old, nev) -> System.out.println("i2 nun gleich "+i2.get()));

    i1.bind(i2);
    System.out.println("\nÄnderung nach unidirektionaler Verbindung:");
//    i1.set(11); //verboten
    i2.set(22);

    System.out.println("\nUnidirektionale Verbindung aufgehoben:");
    i1.unbind();
    i2.set(222);
  }
}
```

Änderungen bei `i2` werden auf `i1` übertragen:

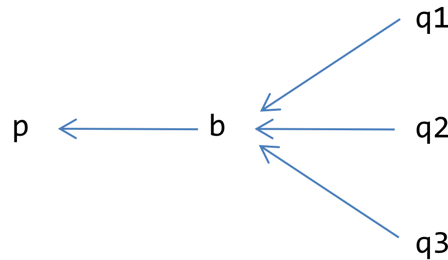
```
i1 nun gleich 2

Änderung nach unidirektionaler Verbindung:
i1 nun gleich 22
i2 nun gleich 22

Unidirektionale Verbindung aufgehoben:
i2 nun gleich 222
```

Direkte Schreibzugriffe auf die gebundene Property `i1` führen zu einem Ausnahmefehler.

13.5.3.2 Property-Objekt an ein Berechnungsergebnis binden

Es ist möglich, den Wert eines Property-Objekts an ein Berechnungsergebnis zu binden, zu dem mehrere Objekte vom Typ **ObservableValue<T>** (z. B. mehrere Property-Objekte) beitragen. Dazu verwendet man ein Objekt einer **Binding-Klasse**, die aufgrund ihres Stammbaums und ihrer Interface-Verpflichtungsverträge die Kompetenz besitzt, aus mehreren Quellen, die auch als ihre *Abhängigkeiten* (engl.: *dependencies*) bezeichnet werden, einen Funktionswert zu berechnen. Im Paket **javafx.beans.binding** befinden sich für die Funktionswertdatentypen **boolean**, **double**, **float**, **int**, **long**, **Object**, **String**, **List<E>**, **Map<K,V>** und **Set<E>** die Klassen **BooleanBinding**, **DoubleBinding**, **FloatBinding**, **IntegerBinding**, **LongBinding**, **ObjectBinding**, **StringBinding**, **ListBinding**, **MapBinding** und **SetBinding**, die alle das Interface **Binding<T>** implementieren. Der Typformalparameter **T** steht natürlich für einen Referenztyp, und die Klasse **DoubleBinding** implementiert z. B. das Interface **Binding<Number>**.

Wenn eine Property (z. B. `p`) an ein Binding-Objekt (z. B. `b`) mit mehreren Abhängigkeiten bzw. Quellen (z. B. `q1`, `q2`, `q3`) gebunden werden soll, kommen natürlich nur *unidirektionale* Bindungen in Betracht:



Ein Binding-Objekt verwendet im Hintergrund Invalidation-Listener für all seine Quellen und wird ungültig, sobald eine Quelle ungültig wird. In dieser Situation löst das Binding-Objekt sein eigenes Invalidierungs-Ereignis aus. Die Neuberechnung des Wertes erfolgt aus Performanzgründen erst bei Bedarf (engl.: *lacy execution*). Ist ein Change-Listener bei einem Binding-Objekt registriert, ist nach eingetretener Ungültigkeit die sofortige Neuberechnung des Wertes erforderlich (engl.: *eager execution*).

Wer über die anschließenden kurz gefassten Erläuterungen hinaus weitere Informationen zum Binding-API benötigt, findet diese z. B. in Sharan (2015, S. 62ff).

13.5.3.2.1 High-Level Binding-API

Das High-Level - Binding-API in JavaFX deckt häufige Anwendungsfälle auf bequeme Weise ab. Zur Demonstration greifen wir die Klasse `Person` aus Abschnitt 13.5.1.2 wieder auf und ergänzen die Eigenschaft `yearOfBirth`:

```
private IntegerProperty yearOfBirth = new SimpleIntegerProperty(this, "yearOfBirth", -99);
private IntegerProperty age = new SimpleIntegerProperty(this, "age", -99);
```

Nun soll allerdings die Eigenschaft `age` nicht mehr separat geführt, sondern an die Eigenschaft `yearOfBirth` gebunden werden. Im High-Level Binding-API stehen dazu zwei Lösungen bereit:

- Fluent API
  In den Property- und den Binding-Klassen stehen Methoden bereit, die aus beobachtbaren Variablen und konstanten Werten ein neues Binding-Objekt erstellen. Man spricht vom *Fluent API*, weil die Methoden ein flüssiges Programmieren durch Verketten von Aufrufen erlauben. Im Beispiel soll zur Berechnung des Alters vom aktuellen Jahr (ermittelt über die statische Methode **now()** der Klasse **Year**) das Geburtsjahr abgezogen werden, was mit Hilfe der Methoden **subtract()** und **negate()** gelingt:[1]
  ```
  IntegerBinding ib = yearOfBirth.subtract(Year.now().getValue()).negate();
  ```
  Es resultiert ein Objekt der Klasse **IntegerBinding**. An dieses Objekt wird die Property `age` gebunden:
  ```
  age.bind(ib);
  ```

[1] Der Einfachheit halber wird nur das Geburts*jahr* erfasst, sodass die Altersberechnung nicht ganz korrekt ist.

- Statische Methoden der Klasse **Bindings**
  Ein Binding-Objekt, an das eine Property gebunden werden kann, lässt sich auch über statische Methoden der Klasse **Bindings** erstellen, z. B.:[1]
  ```
  age.bind(Bindings.subtract(Year.now().getValue(), yearOfBirth));
  ```

Im folgenden Programm

```
class PersonDemo {
   public static void main(String[] args) {
      Person p = new Person("Otto", "Rempremerding", 1990);

      System.out.println(p.getId() + ", " + p.getFirstName() +
                         " " + p.getLastName() + ", " + p.getAge());

      p.ageProperty().addListener(obs -> System.out.println("\nAlter unbekannt"));
      p.setYearOfBirth(1990);

      p.ageProperty().addListener((obs, old, nev) ->
                                    System.out.println("\nAlter geändert auf " + nev));
      p.setYearOfBirth(1995);
   }
}
```

wird die automatische Altersberechnung demonstriert:

```
1, Otto Rempremerding, 30

Alter unbekannt

Alter geändert auf 25
```

Nach einem Schreibzugriff auf die Quelle (Property `yearOfBirth`) ist die Property `age` ungültig, was die Neuberechnung nach Bedarf demonstriert (lacy execution). Ist bei der Property `age` ein Change-Listener registriert, führt die veränderte Quelle zur sofortigen Neuberechnung (eager execution).

13.5.3.2.2 Low-Level Binding-API

Nicht immer genügen zur Definition einer Bindung die im High-Level - Binding-API unterstützten Verknüpfungen der Quellen durch arithmetische Operationen. Mit dem Low-Level Binding-API lässt sich jede beliebige Bindungsdefinition realisieren. Man benötigt eine Binding-Klasse mit passender Berechnung des Funktionswerts aus den Abhängigkeiten und muss dazu aus einer Basisklasse mit dem korrekten Funktionswertdatentyp eine eigene Klasse ableiten, um dort die Methode **computeValue()** zu überschreiben.

Wir betrachten als Beispiel eine von **IntegerBinding** abstammende Klasse, deren Objekte für zwei beobachtete **IntegerProperty**-Quellen den größten gemeinsamen Teiler als Funktionswert liefern.

---

[1] Die statische Methode **subtract()** der Klasse **Bindings** liefert ein Objekt vom Typ **NumberBinding**, der u.a. den Typ **ObservableValue<Number>** erweitert. Die Methode **bind()** der Klasse **IntegerProperty** verlangt für ihren Parameter den Typ **ObservableValue<? extends Number>**, sodass im Beispiel der **bind()** - Aufruf in Ordnung geht. Dabei wird ein Objekt vom Typ der abstrakten Klasse **IntegerProperty** an ein Objekt vom Typ der Schnittstelle **NumberBinding** gebunden, wobei nötigenfalls eine Typumwandlung stattfindet. Das ist im Beispiel aber kaum zu erwarten, weil **Year.now().getValue()** einen **int**-Wert liefert, und `yearOfBirth` vom Typ **IntegerProperty** ist. In der folgenden Anweisung liefert die Methode **subtract()** aber ein Ergebnis vom Typ **DoubleBinding**:
```
age.bind(Bindings.subtract(10e200, yearOfBirth));
```
Die daran gebundene **IntegerProperty** erhält als Ergebnis den maximal möglichen **int**-Wert (2^{31}-1 = 2147483647), wobei kein alarmierendes Ausnahmeobjekt geworfen wird.

Einem häufig verwendeten Entwurfsmuster folgend (siehe z. B. Sharan 2015, S. 77ff) realisieren wir die Ableitung als anonyme Klasse. Bei der Instanzierung eines Objekts aus der anonymen Bindungsklasse muss die **IntegerBinding**-Methode **bind()** aufgerufen werden, um Abhängigkeiten zu registrieren:

protected final void bind(Observable... *dependencies***)**

Zur Initialisierung neuer Objekte können bei einer anonymen Klasse mangels Klassenname keine Konstruktoren verwendet werden. Als Ersatz bieten sich die ansonsten selten benötigten **Instanzinitialisierer** an (vgl. Abschnitt 4.4.4).

Das folgende Beispielprogramm definiert und nutzt die anonyme **IntegerBinding**-Ableitung im Rahmen der **main()** - Methode:

```
import javafx.beans.binding.IntegerBinding;
import javafx.beans.property.*;

class LowLevelBinding {
  public static void main(String[] args) {
    IntegerProperty i1 = new SimpleIntegerProperty(1);
    IntegerProperty i2 = new SimpleIntegerProperty(8);

    IntegerBinding ggtBnd = new IntegerBinding() {
      // Instanzinitialisierer
      {
          bind(i1, i2);
      }
      @Override
      protected int computeValue() {
        int rest, a = i1.get(), b = i2.get();
        do {
          rest = a % b;
          a = b;
          b = rest;
        } while (rest > 0);
        return Math.abs(a);
      }
    };

    IntegerProperty ggt = new SimpleIntegerProperty();
    ggt.bind(ggtBnd);
    System.out.println("Initialer Wert: " + ggt.getValue());
    ggt.addListener((obs, old, nev) ->
                System.out.println(" Neuer GGT: " + nev));

    System.out.println("Automatische Wertanpassung bei ggt:");
    i1.set(22);
    i1.set(12);
    i1.set(24);
    i1.set(32);
    i1.set(33);
  }
}
```

Die anonyme Klasse überschreibt die abstrakte Basisklassenmethode **computeValue()** und realisiert dabei die Modulo-Variante des Euklidischen Verfahrens zur Berechnung des größten gemeinsamen Teilers (siehe Übungsaufgabe auf Seite 195). Dass bei einer anonymen Klassendefinition auf die lokalen Variablen der umgebenden Methode zugegriffen werden kann (vgl. Abschnitt 12.1.1.2), erweist sich im Beispiel als sehr praktisch. Das Programm liefert die folgende Ausgabe:

```
Initialer Wert: 1
Automatische Wertanpassung bei ggt:
 Neuer GGT: 2
 Neuer GGT: 4
 Neuer GGT: 8
 Neuer GGT: 1
```

Das Property-Objekt `ggt` erhält einen (per Lambda-Ausdruck realisierten) Change-Listener, sodass die Effekte der anschließend durchgeführten Änderungen bei der Quell-Property `i1` beobachtet werden können. Deren Ausgangswert von 1 wird mehrfach mit bzw. ohne Auswirkung auf den größten geneinsamen Teiler von `i1` und `i2` verändert.

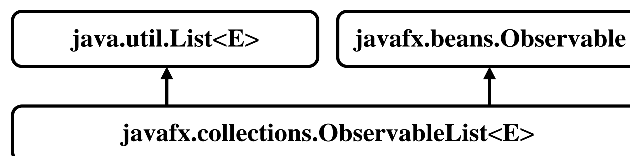
13.5.3.3 Beobachtbare Listen

Eine beobachtbare Liste, die bei einer Änderung ihrer Zusammensetzung eine Mitteilung an registrierte Beobachter versendet, haben wir schon in einem JavaFX-Beispielprogramm als Datenmodell für ein **ListView<E>** - Steuerelement verwendet und dabei als „technisches Glanzstück“ bezeichnet (siehe Abschnitt 13.2). In diesem Abschnitt folgen noch einige ergänzenden Erläuterungen zu beobachtbaren Listen.

Diese implementieren das Interface **ObservableList<E>**, das sich wie alle anderen im Zusammenhang mit beobachtbaren Kollektionen relevanten Typen im Paket **javafx.collections** befindet. Mit beobachtbaren Mengen (vom Typ **ObservableSet<E>**) und beobachtbaren Abbildungen (vom Typ **ObservableMap<K,V>**) können wir uns aus Zeitgründen nicht beschäftigen. Eine ausführliche Beschreibung von beobachtbaren Kollektionen (Listen, Mengen und Abbildungen) findet sich z. B. bei Sharan (2015, S. 83ff).

Während es z. B. sehr leicht fällt, ein **ListView<E>** - Steuerelement mit einem **ObservableList<E>** - Objekt zu verbinden, sind beim Zugriff auf beliebige Ereignisse einer beobachtbaren Liste etliche Details beteiligt. Wer aktuell keine Informationen zur flexiblen Verarbeitung von Listenereignissen benötigt (z. B. die Anzahl der ergänzten Listenelemente übersteigt eine Schwelle, irgendein Element wird auf einen bestimmten Wert gesetzt), sollte sich die technischen Details im weiteren Verlauf von Abschnitt 13.5.3.3 besser vorläufig ersparen.

Das Interface **ObservableList<E>** erweitert die Schnittstellen **List<E>** und **Observable**,



sodass eine implementierende Klasse neben den Funktionalitäten einer Liste auch die **Observable**-Methoden **addListener()** und **removeListener()** anbieten muss, um Interessenten für Invalidierungsereignisse zu verwalten:

- **public void addListener(InvalidationListener** *listener***)**
- **public void removeListener(InvalidationListener** *listener***)**

Außerdem verlangt das Interface **ObservableList<E>** auch Überladungen der Methoden **addListener()** und **removeListener()** zur Verwaltung von Interessenten für Listenveränderungen:

- **public void addListener(ListChangeListener<? super E>** *listener*)
- **public void removeListener(ListChangeListener<? super E>** *listener*)

Implementierende Objekte zu den Schnittstellen für beobachtbare Kollektionen sind über statische Fabrikmethoden der Klasse **FXCollections** verfügbar. Von der generischen Methode **observableArrayList()** erhält man z. B. ein Objekt aus einer Klasse, die das Interface **ObservableList<E>** implementiert und einen Array zur Aufbewahrung ihrer Listenelemente verwendet. Im folgenden Beispiel kann beim Aufruf der generischen Methode dank Typinferenz auf die Angabe des Elementtyps **String** verzichtet werden:

```
ObservableList<String> ols = FXCollections.observableArrayList();
```

13.5.3.3.1 Zusammensetzung und Anordnung der Listenelemente beobachten

Um von der eben erstellten Liste vom Typ **ObservableList<String>** über Veränderungen im Detail informiert zu werden, registriert man per **addListener()** ein Objekt aus einer das Interface **ListChangeListener<String>** implementierenden Klasse. Im folgenden Codesegment wird dazu eine anonyme Klasse definiert:

```
ols.addListener(new ListChangeListener<String>() {
    @Override
    public void onChanged(ListChangeListener.Change<? extends String> c) {
        int change = 0;
        while (c.next()) {
            change++;
            if (c.wasRemoved())
                System.out.println("Removed-Veränderung " + change +
                                   ", entfernt: " + c.getRemoved());
            if (c.wasAdded())
                System.out.println("Added-Veränderung " + change +
                        ", betroffen: von " + c.getFrom() + " bis " + (c.getTo()-1) +
                        ", ergänzt: " + c.getAddedSubList());
        }
    }
});
```

Ein **ListChangeListener<E>** muss die Methode **onChanged()** implementieren, die bei einer Listenveränderung aufgerufen wird und dabei ein Parameterobjekt vom Typ der innerhalb von **ListChangeListener<E>** definierten statischen Mitgliedsklasse **Change<E>** erhält. Der im **Change<E>** - Objekt enthaltene Bericht kann mehrere Teile umfassen, wenn von einer Veränderung mehrere, nicht hintereinander liegende Elemente der beobachteten Liste betroffen sind. Im Beispiel wird die Veränderungsverarbeitung in einer **while**-Schleife so lange fortgesetzt, bis der **next()** - Aufruf im Schleifenkopf durch den Rückgabewert **false** signalisiert, dass im **Change<E>** - Objekt keine weiteren Teilberichte vorhanden sind.

Über die (bei allen Teilberichten in einem **Change<E>** - Objekt identische) Veränderungsart informieren die folgenden **Change**-Methoden:

- **public boolean wasAdded()**
  Es wurden Elemente aufgenommen.
- **public boolean wasPermutated()**
  Die Reihenfolge der Elemente wurde verändert.
- **public boolean wasRemoved()**
  Es wurden Elemente entfernt.
- **public boolean wasReplaced()**
  Es wurden Elemente ersetzt, d.h. es wurden alte Elemente entfernt und neue aufgenommen. Wurde das Entfernen und die Aufnahme von Elementen verarbeitet, muss man sich *nicht* zusätzlich um die Ersetzungen kümmern.
- **public boolean wasUpdated()**
  Bei einer beobachtbaren Liste mit beobachtbaren Elementen kann eine Wertveränderung gemeldet werden (siehe Abschnitt 13.5.3.3.2).

Welche Listenelemente von einer Veränderung betroffen sind, ist über die folgenden **Change<E>** - Methoden zu ermitteln:

- **public int getFrom()**
  Liefert der Startindex (inklusive)
- **public int getTo()**
  Liefert der Endindex (exklusive)

Weitere **Change<E>** - Methoden:

- **public ObservableList<E> getList()**
  Liefert eine Referenz auf die beobachtete Liste
- **public int getAddedSize()**
  Liefert die Anzahl der ergänzten Elemente
- **public List<E> getAddedSubList()**
  Liefert eine Liste mit den ergänzten Elementen
- **protected int[] getPermutation()**
  Der gelieferte **int**-Array beschreibt eine Permutation.
- **public int getRemovedSize()**
  Liefert die Anzahl der entfernten Elemente
- **public List<E> getRemoved()**
  Liefert eine Liste mit den entfernten Elementen

Aus den im bisherigen Verlauf des Abschnitts 13.5.3.3 präsentierten Codesegmenten resultiert ein Objekt vom Typ **ObservableList<String>**, bei dem ein Interessent für Veränderungen registriert ist. Nun werden an der beobachtbaren Liste einige Veränderungen vorgenommen:

```
System.out.println("Elemente aufnehmen:");
ols.addAll("A", "B", "C", "D", "E", "F");

System.out.println("\nHintereinander liegende Elemente entfernen:");
ols.removeAll("D", "E", "F");

System.out.println("\nEin Element ersetzen:");
ols.set(0, "A1");

System.out.println("\nNicht hintereinander liegende Elemente entfernen:");
ols.removeAll("A1", "C");
```

Bei der Aufnahme von 6 Elementen sowie beim Löschen von drei hintereinander liegenden Elementen resultiert jeweils ein einteiliger Veränderungsbericht:

```
Elemente aufnehmen:
Added-Veränderung 1, betroffen: von 0 bis 5, ergänzt: [A, B, C, D, E, F]

Hintereinander liegende Elemente entfernen:
Removed-Veränderung 1, entfernt: [D, E, F]
```

Das ist auch beim Ersetzen eines Listenelementes der Fall, wobei aber diesmal die Methoden **wasRemoved()** und **wasAdded()** beide die Rückgabe **true** liefern:

```
Ein Element ersetzen:
Removed-Veränderung 1, entfernt: [A]
Added-Veränderung 1, betroffen: von 0 bis 0, ergänzt: [A1]
```

Werden Listenelemente gelöscht, die *nicht* hintereinander liegen, dann enthält das **Change**-Objekt einen mehrteiligen Veränderungsbericht:

```
Nicht hintereinander liegende Elemente entfernen:
Removed-Veränderung 1, entfernt: [A1]
Removed-Veränderung 2, entfernt: [C]
```

13.5.3.3.2 Wertveränderungen bei Listenelementen beobachten

Sollen auch Wertveränderungen bei einzelnen Listenelementen beobachtet werden, dann muss für jedes Element ein Array mit Änderungsmeldern vom Typ **Observable** benannt werden, die auf Veränderungen beim Element abgehört werden sollen. Um Update-Ereignisse feuern zu können, überwacht eine beobachtbare Liste für jedes ihrer Elemente einen Array von **Observable**-Objekten.

Diese komplexe, aber unvermeidliche Konstruktion wird von Sharan (2015, S. 93) so erklärt: Eine Liste enthält Elemente mit Referenztyp, und jedes Element kann an diversen Stellen im Programm verändert werden, ohne dass die Liste davon erfährt. Daher muss die Liste zu jedem ihrer Elemente alle Objekte kennen, die über eine Änderung des Listenelements informieren können.

Den zu einem Listenelement gehörigen **Observable**-Array erfragt eine beobachtbare Liste bei einem Objekt, das die Schnittstelle **Callback<E,Observable[]>** implementiert. Eine beobachtbare Liste kann nur dann Update-Ereignisse liefern, wenn der **FXCollections**-Fabrikmethode ein passendes Objekt übergeben worden ist, z. B.:

public static <E> ObservableList<E> observableArrayList(Callback<E, Observable[]> *extractor***)**

Das Objekt mit dem Parameternamen *extractor* beherrscht eine Methode namens **call()**, die als Parameter ein Listenelement erhält und den zugehörigen Array mit Objekten vom Typ **Observable** zu liefern hat. Diese Methode **call()** wird von der beobachtbaren Liste aufgerufen, sobald ein neues Element eingefügt wird.

Wenn die Listenelemente selbst beobachtbar sind, ist der erforderliche **Observable**-Array zu einem Element leicht zu beschaffen, indem das Element in einen einelementigen Array vom Typ **Observable[]** verpackt wird. Wir ersetzen im Beispiel aus Abschnitt 13.5.3.3.1 die Klasse **String** durch die Klasse **StringProperty**. Das **Callback<E,Observable[]>** - Parameterobjekt für die **FXCollections**-Methode **observableArrayList()** wird durch einen Lambda-Ausdruck realisiert, der sein Argument in einen einelementigen, per Initialisierungsliste erstellten **Observable[]** - Array verpackt:

```
ObservableList<StringProperty> ols =
  FXCollections.observableArrayList(e -> new Observable[] {e});
```

Im folgenden Beispielprogramm erhält das **ObservableList<StringProperty>** - Objekt nach dem im Abschnitt 13.5.3.3.1 beschriebenen Strickmuster einen **ListChangeListener<StringProperty>**. Diesmal werden in der Methode **onChanged()** auch Update-Ereignisse verarbeitet. Man erkennt sie am Rückgabewert **true** der **wasUpdated()** - Anfrage an das in **onChanged()** verfügbare Parameterobjekt vom Typ **ListChangeListener.Change<E>**:

```
import javafx.beans.Observable;
import javafx.beans.property.*;
import javafx.collections.*;

class ObservableListUpdates {
  public static void main(String[] args) {
    ObservableList<StringProperty> ols =
      FXCollections.observableArrayList(e -> new Observable[] {e});

    ols.addListener(new ListChangeListener<StringProperty>() {
      @Override
      public void onChanged(ListChangeListener.Change<? extends StringProperty> c) {
        while (c.next()) {
          if (c.wasAdded())
            System.out.println("Ergänzt: " + c.getAddedSubList());
          if (c.wasUpdated()) {
            int i = c.getFrom();
            System.out.println("Neuer Wert von Element " + i + ": " +
                               c.getList().get(i).get());
          }
        }
      }
    });
```

```
        StringProperty spa = new SimpleStringProperty("A");
        StringProperty spb = new SimpleStringProperty("B");

        System.out.println("\nElemente aufnehmen:");
        ols.addAll(spa, spb);

        System.out.println("\nEin Element ändern:");
        ols.get(1).set("B1");
    }
}
```

Wenn ein Listenelement einen neuen Wert erhält, feuert die beobachtbare Liste ein Update-Ereignis:

```
Elemente aufnehmen:
Ergänzt: [StringProperty [value: A], StringProperty [value: B]]

Ein Element ändern:
Neuer Wert von Element 1: B1
```

13.6 Layoutmanager

Die Spezialisierungen der Klasse **javafx.scene.layout.Pane** sind in der Lage, den verfügbaren Platz sinnvoll auf die enthalten Kindelemente zu verteilen und insbesondere bei einer Größenänderung die Ausdehnungs- und Anordnungswünsche der enthaltenen Kindelemente zu berücksichtigen. Wie Sie aus Abschnitt 13.3 wissen, kann man z. B. festlegen, welche Komponente eine horizontale oder vertikale Größenzunahme konsumieren soll.

Anschließend wird zur Demonstration einiger JavaFX-Layoutmanager (**Grid**-, **Anchor**- und **BorderPane**) die All-In-One - Variante des Beispielprogramms zur Anwesenheitskontrolle verwendet (siehe Abschnitt 13.3). Um den Layoutmanager mit minimalem syntaktischem Aufwand austauschen zu können, werden Instanzvariablen für die Steuerelemente angelegt:[1]

```
private Label lblName = new Label("Name:");
private Label lblPresent = new Label("Anwesend:");
private TextField tfName = new TextField();
private ListView<String> lvPersons;
private Button btnAdd = new Button("Angekommen");
private Button btnRemove = new Button("Gegangen");
```

In der **start()** - Methode verbleiben die Layout-neutralen Programmbestandteile und ein Methodenaufruf zum jeweils verwendeten Layoutmanager, z. B.:

---

[1] Das IntelliJ-Projekt zur Demonstration diverser Layoutmanager ist im folgenden Ordner zu finden:
...\BspUeb\JavaFX\Layoutmanager\GridAnchorBorderPane

```
@Override
public void start(Stage primaryStage) {
    String[] anwesend = new String[] {"Willi", "Otto", "Theo", "Irma", "Doro",
                                      "Heiner", "Michael", "Ludger", "Ben"};

    ObservableList<String> persons = FXCollections.observableArrayList(anwesend);
    SortedList<String> perSorted = new SortedList<>(persons,
                                                     Comparator.naturalOrder());
    lvPersons = new ListView<>(perSorted);

    btnAdd.setOnAction(event -> {
        String s = tfName.getText();
        if (s.length() > 0  &&  !persons.contains(s))
            persons.add(s);
    });

    btnRemove.setOnAction(event ->
            persons.remove(lvPersons.getSelectionModel().getSelectedItem()));

    Pane root = makeGridPane();

    primaryStage.setTitle("Anwesenheitskontrolle");
    primaryStage.setScene(new Scene(root, 450, 200));
    primaryStage.setMinWidth(400);
    primaryStage.setMinHeight(150);
    primaryStage.show();
}
```

Von den drei Optionen zur Konfiguration der Layoutmanager

- Java-Anweisungen
- Direkte FXML-Deklaration
- Indirekte FXML-Deklaration via Scene Builder

kann aus Zeitgründen nur die *erste* berücksichtigt werden. Immerhin lernt man so die objektorientierten Grundlagen (das API) von JavaFX kennen. API-Kenntnisse sind unverzichtbar, wenn im laufenden Programm die Bedienoberfläche modifiziert werden soll.

13.6.1 GridPane

Der **GridPane**-Layoutmanager, mit dem wir schon etliche Erfahrungen gesammelt haben (siehe z. B. die Abschnitte 4.9, 13.3 und 13.4), ordnet die Kindelemente in einer Matrix an, wobei sich ein Element auch über mehrere Zellen ausdehnen darf. Er ist der flexibelste JavaFX-Layoutmanager und wird daher zu Recht von IntelliJ als Voreinstellung für JavaFX-Projekte verwendet.

13.6.1.1 Beispiel

Im folgenden Quellcode-Segment

```
double dist = 10.0;

GridPane root = new GridPane();
root.setPadding(new Insets(dist, dist, dist, dist));
root.setHgap(dist); root.setVgap(dist);

root.add(lblName, 0, 0); root.add(tfName, 1, 0); root.add(btnAdd, 2, 0);
root.add(lblPresent, 0, 1); root.add(lvPersons, 1, 1); root.add(btnRemove, 2, 1);

GridPane.setHalignment(btnAdd, HPos.RIGHT);
GridPane.setHalignment(btnRemove, HPos.RIGHT);
```

```
GridPane.setHgrow(tfName, Priority.ALWAYS);
GridPane.setVgrow(lvPersons, Priority.ALWAYS);
```

wird ein **GridPane**-Objekt erzeugt

```
GridPane root = new GridPane();
```

und befüllt:

```
root.add(lblName, 0, 0); root.add(tfName, 1, 0); root.add(btnAdd, 2, 0);
root.add(lblPresent, 0, 1); root.add(lvPersons, 1, 1); root.add(btnRemove, 2, 1);
```

Wie das Beispiel zeigt, wird die Zahl der Spalten und Zeilen nicht beim Erstellen eines **GridPane**-Objekts erwartet, sondern aus der tatsächlichen Verwendung ermittelt.

13.6.1.2 GridPane-Eigenschaften zur Gestaltung von Abständen

Ein **GridPane**-Container kann über seine Eigenschaften **padding**, **hgap** und **vgap** dafür sorgen, dass seine Kindelemente zum Container-Rand sowie untereinander einen einstellbaren Abstand halten:

- **padding**
  Um zu verhindern, dass die in einem Container enthaltenen Elemente gegen den Rand stoßen, kann mit der **Region**-Methode **setPadding()** unter Verwendung eines Parameterobjekts der Klasse **Insets** ein freizuhaltender Innenrahmen für den Container vereinbart werden, z. B.:
  ```
  root.setPadding(new Insets(dist, dist, dist, dist));
  ```
- **hgap**, **vgap**
  Über die Methoden **setHgap()** und **setVgap()** lässt sich ein Abstand zwischen den Elementen in einem **GridPane**-Container vereinbaren, z. B.:
  ```
  root.setHgap(dist); root.setVgap(dist);
  ```

Seit Java 8 Update 60 passt JavaFX unter Windows Positions- und Größenangaben automatisch an die Bildschirmauflösung an. Alle Positions- und Größenangaben verwenden die Maßeinheit Pixel bei einer zugrunde gelegten Bildschirmauflösung von 96 dpi. Hat ein Display z. B. die doppelte Auflösung von 192 dpi, dann werden alle Pixel-Werte automatisch verdoppelt.

Bei der Gestaltung eines **GridPane**-Containers kann man vorübergehend Gitterlinien aktivieren, um das Design zu erleichtern:

```
root.setGridLinesVisible(true);
```

13.6.1.3 Anzeigeeinstellungen für Kindelemente (layout constraints)

Ein **GridPane**-Layoutmanager unterstützt mehrere Anzeigeeinstellungen für die enthaltenen Knoten, die über *statische* **GridPane**-Methoden gesetzt werden. So wird z. B. mit der Methode **setVgrow()** vereinbart, welches Steuerelement bei einer vertikalen Größenzunahme des Containers profitieren soll:

public static void setVgrow (Node *child*, **Priority** *value***)**

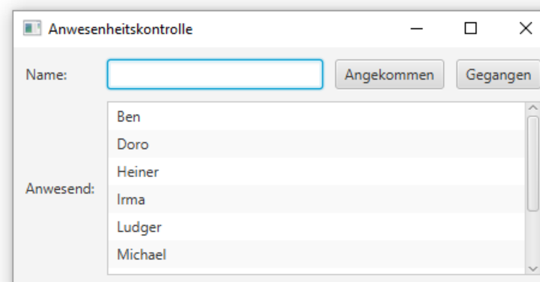
Der natürliche Ansprechpartner für einen derartigen Darstellungswunsch ist eher das konkret betroffene Layoutmanager-Objekt als seine Klasse. Nach Horstmann (2014, S. 83) ist das ungewöhnliche, in vielen Layoutmanager-Klassen von JavaFX realisierte Muster entstanden, um die Verarbeitung von Layout-Deklarationen in FXML-Dateien unterstützen zu können.

In der API-Dokumentation werden die über statische Layoutmanager-Methoden konfigurierbaren Eigenschaften der Kindelemente als **layout constraints** bezeichnet. Ein Kindelement sammelt die ihm vom Layoutmanager zugeordneten Anzeigeeinstellungen in einer Hashtabelle (Typ

ObservableMap<Object, Object>, abrufbar über die Methode **getProperties()**). Ein Layoutmanager liest und berücksichtigt die Anzeigeeinstellungen (layout constraints) der enthaltenen Knoten.[1]

Die Flexibilität des **GridPane**-Layoutmanagers artikuliert sich auch durch eine umfangreiche Liste von Anzeigeeinstellungen für Kindelemente:

- **columnIndex**, **rowIndex**
  Bei diesen Einstellungen geht es um die Spalten- bzw. Zeilenposition eines Kindelements, z. B.:
  ```
  root.getChildren().add(lblName);
  GridPane.setColumnIndex(lblName, 0);
  GridPane.setRowIndex(lblName, 0);
  ```
  Die initiale Positionierung eines Kindelements wird meist schon bei der Aufnahme durch die **GridPane**-Instanzmethode **add()** vorgenommen, z. B.:
  ```
  root.add(lblName, 0, 0);
  ```
- **columnSpan**, **rowSpan**
  Über diese Eigenschaften wird für ein Kindelement festgelegt, über wie viele Spalten bzw. Zeilen es sich erstecken soll. In der folgenden Variante des Beispielprogramms zur Anwesenheitskontrolle



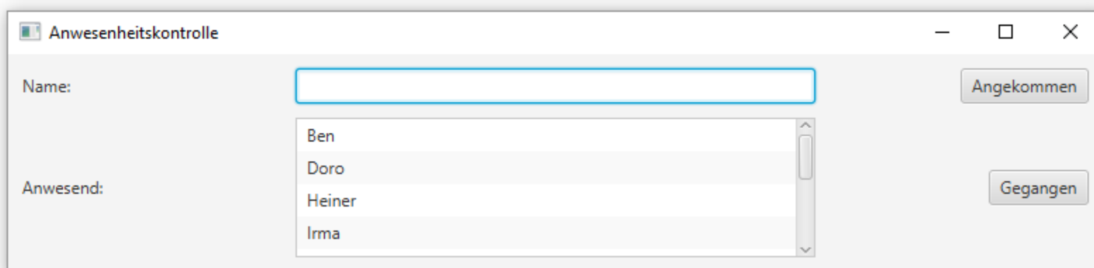
  erstreckt sich das **ListView<String>** - Bedienelement über 3 Spalten, was sich durch den folgenden Methodenaufruf vereinbaren lässt:
  ```
  GridPane.setColumnSpan(lvPersons, 3);
  ```
  Alternativ lässt sich die Erstreckung über mehr als eine Spalte bzw. Zeile auch schon bei der Aufnahme eines Kindelements durch eine **add()** - Überladung erreichen, z. B.:
  ```
  root.add(lvPersons, 1, 1, 3, 1);
  ```
- **halignment**, **valignment**
  Hier geht es um die horizontale bzw. vertikale Ausrichtung eines Kindelements innerhalb seiner Zelle bzw. Zone. Durch die folgende Anweisung wird dafür gesorgt, dass im Beispielprogramm zur Anwesenheitskontrolle das **Button**-Objekt `btnRemove` innerhalb seiner Zelle eine rechtsbündige horizontale Ausrichtung erhält:
  ```
  GridPane.setHalignment(btnRemove, HPos.RIGHT);
  ```
- **hgrow** bzw. **vgrow**
  Es kann ein Kindelement bestimmt werden, das von einem horizontalen bzw. vertikalen Größenzuwachs des Containers profitieren soll, z. B.:
  ```
  GridPane.setHgrow(tfName, Priority.ALWAYS);
  GridPane.setVgrow(lvPersons, Priority.ALWAYS);
  ```

[1] Weitere Details sind hier zu finden:
https://softwareengineering.stackexchange.com/questions/316643/why-does-javafx-gridpane-attach-properties-of-the-layout-to-the-components

- **margin**
  Für ein Kindelement lässt sich per **Insets**-Objekt ein Außenrand definieren (mit einer individuellen oberen, rechten, unteren und linken Breite), z. B.:
  ```
  GridPane.setMargin(lvPersons, new Insets(dist, dist, dist, dist));
  ```

13.6.1.4 Dynamische Platzverteilung auf mehrere Spalten bzw. Zeilen

Wenn bei horizontaler (vertikaler) Größenzunahme des Containers *mehrere* Spalten (Zeilen) profitieren sollen und dabei feste Größenrelationen beibehalten werden sollen, verwendet man Objekte der Klasse **ColumnConstraints** (**RowConstraints**). In der folgenden Variante des Beispielprogramms zur Anwesenheitskontrolle wachsen bei einer horizontalen Größenzunahme alle drei Spalten, wobei die prozentuale Größenverteilung 25:50:25 beibehalten wird:



Um dieses Ziel zu erreichen, werden drei Objekte der Klasse **ColumnConstraints** erstellt, über die Methode **setPercentWidth()** mit dem passenden Größenanteil versorgt und über die Methode **addAll()** in die Liste mit allen **ColumnConstraints** aufgenommen:

```
ColumnConstraints col1 = new ColumnConstraints();
col1.setPercentWidth(25);
ColumnConstraints col2 = new ColumnConstraints();
col2.setPercentWidth(50);
ColumnConstraints col3 = new ColumnConstraints();
col3.setPercentWidth(25);
root.getColumnConstraints().addAll(col1, col2, col3);
```

13.6.2 AnchorPane

Ein Kindelement dieses Layoutmanagers wird an 1, 2, 3 oder 4 Seiten des Containers in einem bestimmten Abstand verankert, der bei einer Änderung der Container-Größe erhalten bleibt. In der folgenden **AnchorPane**-Layoutdefinition für das Beispielprogramm zur Anwesenheitskontrolle

```
AnchorPane root = new AnchorPane();
root.getChildren().addAll(lblName,lblPresent,tfName,lvPersons,btnAdd,btnRemove);

double bd = 10.0; double ldc2 = 100.0; double rdc2 = 120.0; double tdr2 = 50.0;
AnchorPane.setTopAnchor(lblName, bd); AnchorPane.setLeftAnchor(lblName, bd);
AnchorPane.setTopAnchor(lblPresent, tdr2); AnchorPane.setLeftAnchor(lblPresent,bd);
AnchorPane.setTopAnchor(btnAdd, bd); AnchorPane.setRightAnchor(btnAdd, bd);
AnchorPane.setTopAnchor(btnRemove, tdr2); AnchorPane.setRightAnchor(btnRemove, bd);
AnchorPane.setTopAnchor(tfName, bd); AnchorPane.setLeftAnchor(tfName, ldc2);
AnchorPane.setRightAnchor(tfName, rdc2);
AnchorPane.setTopAnchor(lvPersons, tdr2); AnchorPane.setBottomAnchor(lvPersons,bd);
AnchorPane.setLeftAnchor(lvPersons,ldc2);AnchorPane.setRightAnchor(lvPersons,rdc2);
```

wird ein Objekt der Klasse **AnchorPane** erstellt

```
AnchorPane root = new AnchorPane();
```

und mit den Steuerelementen befüllt:

```
root.getChildren().addAll(lblName,lblPresent,tfName,lvPersons,btnAdd,btnRemove);
```

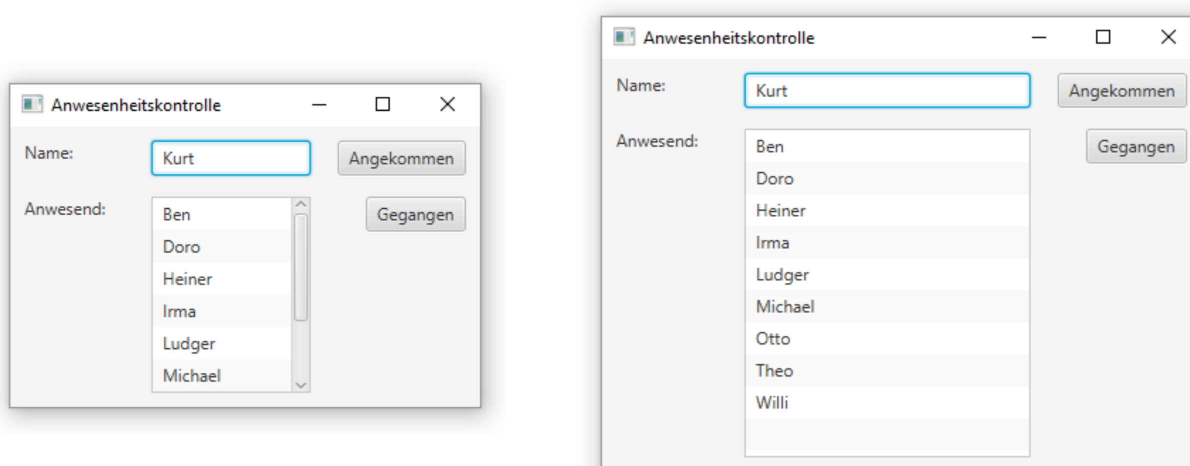
Anschließend sorgen Aufrufe von statischen **AnchorPane**-Methoden zum Setzen von layout constraints (vgl. Abschnitt 13.6.1.3)

- **public static void setTopAnchor (Node** *child*, **Double** *value***)**
- **public static void setBottomAnchor (Node** *child*, **Double** *value***)**
- **public static void setLeftAnchor (Node** *child*, **Double** *value***)**
- **public static void setRightAnchor (Node** *child*, **Double** *value***)**

dafür, dass die Steuerelemente passende Positionierung und Dynamisierungen erhalten. Das **TextField**-Steuerelement zur Aufnahme des Namens eines Neuankömmlings erhält z. B. ...

- zum oberen Container-Rand den Abstand 10
- zum linken Container-Rand den Abstand 100
- zum rechten Container-Rand den Abstand 120

Bei einer Änderung der Container-Größe behält das oben, links und rechts verankerte **TextField**-Steuerelement seine vertikale Position bei und passt seine horizontale Größe an den verfügbaren Platz an:

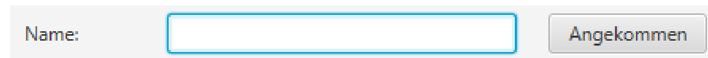


13.6.3 HBox und VBox

Ein Layoutmanager aus der Klasse **HBox** bzw. **VBox** ordnet die Kindelemente neben- bzw. untereinander an. Er unterstützt die folgenden Anzeigeeinstellungen (layout constraints, siehe Abschnitt 13.6.1.3) für Kindelemente, die über statische Methoden verwaltet werden:

- **hgrow** bzw. **vgrow**
  Es kann ein Kindelement bestimmt werden, das von einem horizontalen bzw. vertikalen Größenzuwachs des Containers profitieren soll, z. B.:
  ```
  HBox.setHgrow(tfName, Priority.ALWAYS);
  ```
- **margin**
  Für ein Kindelement lässt sich per **Insets**-Objekt ein Außenrand definieren (mit einer individuellen oberen, rechten, unteren und linken Breite), z. B.:
  ```
  double dist = 10.0;
  Insets insets = new Insets(dist,dist,dist,dist);
  HBox.setMargin(tfName, insets);
  ```

Wir verwenden im Beispielprogramm zur Anwesenheitskontrolle ein **HBox**-Objekt als separaten Layoutmanager für die drei Steuerelemente in der oberen „Zeile“ zur späteren Verwendung als eingeschachteltes Element in einem **BorderPane**-Layout (siehe Abschnitt 13.6.4):



Im folgenden Quellcode-Segment

```
double dist = 10.0;
double lblw = 70.0;
double btnw = 100.0;
Insets insets = new Insets(dist,dist,dist,dist);

HBox top = new HBox();
top.setAlignment(Pos.BASELINE_LEFT);
HBox.setMargin(lblName, insets);
lblName.setPrefWidth(lblw);
HBox.setMargin(btnAdd, insets);
btnAdd.setPrefWidth(btnw);
HBox.setMargin(tfName, insets);
HBox.setHgrow(tfName, Priority.ALWAYS);
top.getChildren().addAll(lblName, tfName, btnAdd);
```

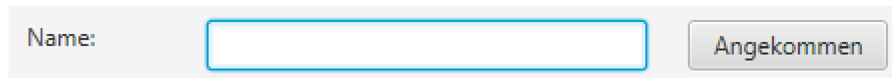
wird ein **HBox**-Objekt erzeugt

```
HBox top = new HBox();
```

und befüllt:

```
top.getChildren().addAll(lblName, tfName, btnAdd);
```

Neben den beschriebenen layout constraints für Kindelemente werden im Beispielprogramm die folgenden Eigenschaften modifiziert:

- **alignment**
  Für den **HBox**-Container wird über seine Eigenschaft **alignment** eine vertikale Ausrichtung der Elemente an den Element*beschriftungen* veranlasst:
  ```
  top.setAlignment(Pos.BASELINE_LEFT);
  ```
  Ohne diese Maßnahme stehen z. B. die **Label**- und die **Button**-Beschriftung nicht auf derselben Höhe:
  

- **prefWith**
  Das **Label**- und das **Button**-Objekt erhalten über die **Region**-Eigenschaft **prefWidth** eine Wunschbreite, um für eine Ausrichtung mit anderen Steuerelementen zu sorgen.

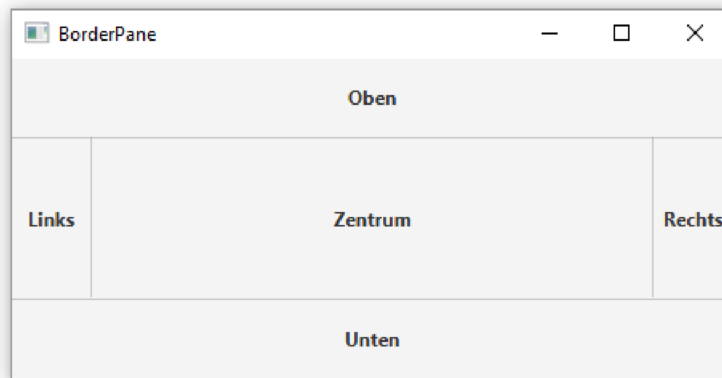
Per Konstruktorüberladung mit Abstandsparameter sorgt man bequem für einen generellen horizontalen Abstand (bei **HBox**) bzw. für einen generellen vertikalen Abstand (bei **VBox**) zwischen allen Kindelementen, z. B.:

```
HBox top = new HBox(10.0);
```

13.6.4 BorderPane

Dieser Layoutmanager sich z. B. als Wurzelknoten für Fenster mit einem Layout bestehend aus einem Menü am oberen Fensterrand, einer Statuszeile am unteren Fensterrand, einem Dokumentenbereich in der Mitte, der optional noch von Bedienelementen am linken und rechten Fensterrand flankiert wird.

Das folgende Programm demonstriert die **BorderPane**-Platzaufteilung:



Ein **BorderPane**-Layoutmanager unterstützt die folgenden Anzeigeeinstellungen (layout constraints, siehe Abschnitt 13.6.1.3) für Kindelemente, die über statische Methoden verwaltet werden:

- **alignment**
  Hier geht es um die Ausrichtung eines Kindelements innerhalb seiner Zone. Durch die folgende Anweisung wird dafür gesorgt, dass im Beispielprogramm zur Anwesenheitskontrolle das **Button**-Objekt `btnRemove` innerhalb seiner Zone (Rechts) eine rechtsbündige horizontale Ausrichtung und eine zentrierte vertikale Ausrichtung erhält:
  ```
  BorderPane.setAlignment(btnRemove, Pos.CENTER_RIGHT);
  ```
- **margin**
  Für ein Kindelement lässt sich per **Insets**-Objekt ein Außenrand definieren (mit einer individuellen oberen, rechten, unteren und linken Breite), z. B.:
  ```
  double dist = 10.0;
  Insets insets = new Insets(dist,dist,dist,dist);
  BorderPane.setMargin(btnRemove, insets);
  ```

Im folgenden Quellcode-Segment wird der im Abschnitt 13.6.3 erstellte **HBox**-Container mit den drei oberen Bedienelementen des Beispielprogramms einbezogen:

```
double dist = 10.0;
double lblw = 70.0;
double btnw = 100.0;
Insets insets = new Insets(dist,dist,dist,dist);

HBox top = new HBox();
 . . .
top.getChildren().addAll(lblName, tfName, btnAdd);

BorderPane root = new BorderPane();
BorderPane.setMargin(lvPersons, insets);
BorderPane.setMargin(lblPresent, insets);
BorderPane.setMargin(btnRemove, insets);
lblPresent.setPrefWidth(lblw);
btnRemove.setPrefWidth(btnw);
BorderPane.setAlignment(lblPresent, Pos.CENTER_LEFT);
BorderPane.setAlignment(btnRemove, Pos.CENTER_RIGHT);

root.setTop(top);
root.setLeft(lblPresent);
root.setCenter(lvPersons);
root.setRight(btnRemove);
```

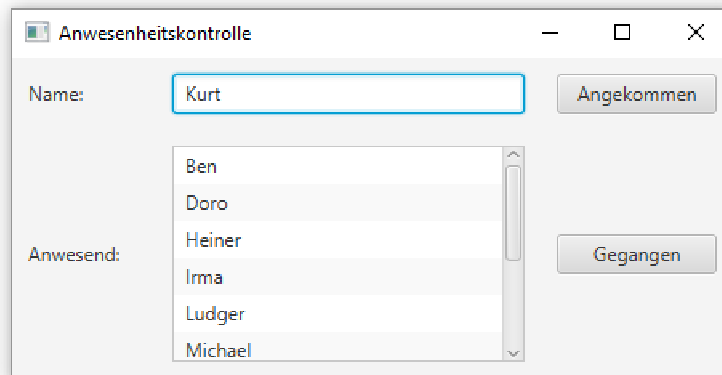
Es wird ein **BorderPane**-Objekt erzeugt

```
BorderPane root = new BorderPane();
```

und befüllt:

```
root.setTop(top);
root.setRight(btnRemove);
root.setLeft(lblPresent);
root.setCenter(lvPersons);
```

Das **Label**-Objekt `lblPresent` und das **Button**-Objekt `btnRemove` erhalten per **setPrefWidth()** eine Wunschbreite, um für eine Ausrichtung mit anderen Steuerelementen zu sorgen. So gelingt ein akzeptables Layout, wobei jedoch im Vergleich zur Verwendung des **GridPane**-Layoutmanagers (siehe Abschnitt 13.6.1) etwas mehr Aufwand erforderlich ist:



Wie zu Beginn des Abschnitts erwähnt, wurde die **BorderPane**-Klasse für ein anderes Anwendungs-Layout konzipiert.

13.6.5 FlowPane

Je nach Orientierung eines **FlowPane**-Containers werden die Knoten neben bzw. untereinander angeordnet und nach Erreichen des Randes umgebrochen.

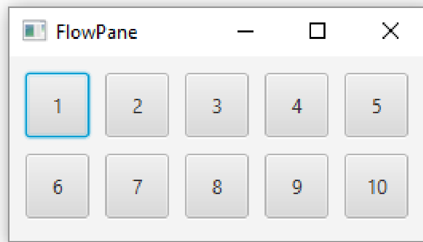
In der folgenden **start()** - Methode eines JavaFX-Programms nimmt ein **FlowPane**-Layoutmanager mit der voreingestellten horizontalen Orientierung 10 **Button**-Objekte auf:

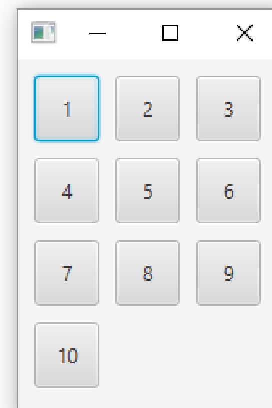
```
public void start(Stage primaryStage) {
    double dist = 10.0;
    int nob = 10;
    double csize = 40.0;

    FlowPane root = new FlowPane();
    root.setPadding(new Insets(dist, dist, dist, dist));
    root.setHgap(dist); root.setVgap(dist);

    Button[] buttons = new Button[nob];
    for (int i = 0; i < nob; i++) {
        buttons[i] = new Button(String.valueOf(i + 1));
        buttons[i].setPrefWidth(csize); buttons[i].setPrefHeight(csize);
        buttons[i].setAlignment(Pos.CENTER);
    }
    root.getChildren().addAll(buttons);

    primaryStage.setScene(new Scene(root, 250, 250));
    primaryStage.setTitle("FlowPane");
    primaryStage.show();
}
```

Der Layoutmanager erhält per **setPadding()** einen Innenrand. Mit **setHgap()** bzw. **setVgap()** wird ein horizontaler bzw. vertikaler Abstand zwischen den enthaltenen Knoten vereinbart:

```
root.setPadding(new Insets(dist, dist, dist, dist));
root.setHgap(dist); root.setVgap(dist);
```

Per Voreinstellung werden die Kindelemente linksbündig angeordnet und nach Erreichen der Container-Breite umgebrochen:



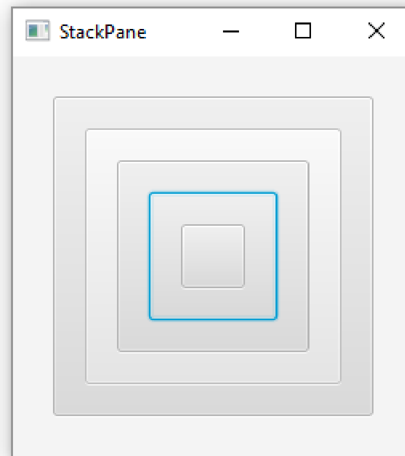


13.6.6 StackPane

Von einem **StackPane**-Layoutmanager werden die Kindelemente übereinander gestapelt (senkrecht zur Display-Ebene), sodass unten liegende Elemente (teilweise) überdeckt werden. In der folgenden **start**() - Methode eines JavaFX-Programms nimmt ein **StackPane**-Layoutmanager 5 quadratische **Button**-Objekte mit sukzessiv schrumpfender Kantenlänge auf:

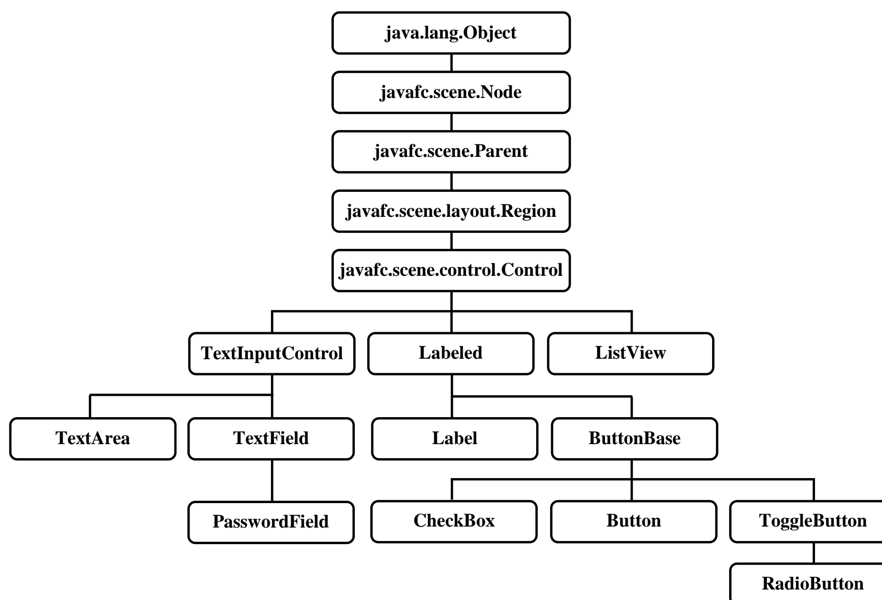
```
public void start(Stage primaryStage) {
    int nob = 5;
    double csize = 40.0;

    StackPane root = new StackPane();

    Button[] buttons = new Button[nob];
    for (int i = 0; i < nob; i++) {
        buttons[i] = new Button();
        buttons[i].setPrefSize(csize*(nob-i), csize*(nob-i));
    }
    root.getChildren().addAll(buttons);

    primaryStage.setTitle("StackPane");
    primaryStage.setScene(new Scene(root, 250, 250));
    primaryStage.show();
}
```

Die resultierende Schalterkombination kommt vielleicht in einer speziellen Anwendung als innovatives Bedienelement in Frage:



13.7 Elementare Steuerelemente

Aus der stattlichen Sammlung von JavaFX-Steuerelementen werden im Manuskript einige besonders häufig benötigte Exemplare näher beschrieben. Deren Einordnung in die Klassenhierarchie zeigt die folgende Abbildung:



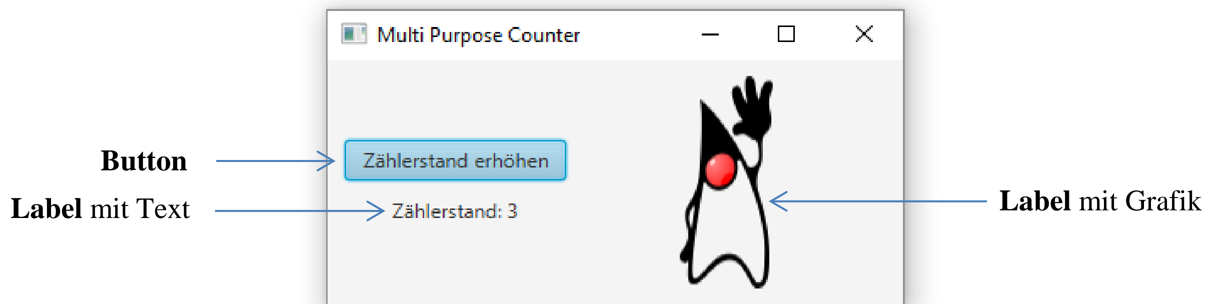
Von den drei Optionen zur Konfiguration der Steuerelemente

- Java-Anweisungen
- Direkte FXML-Deklaration
- Indirekte FXML-Deklaration via Scene Builder

kann aus Zeitgründen nur die *erste* berücksichtigt werden. Immerhin lernt man so die objektorientierten Grundlagen (das API) von JavaFX kennen. API-Kenntnisse sind unverzichtbar, wenn im laufenden Programm die Bedienoberfläche modifiziert werden soll.

13.7.1 Label

Mit Komponenten der Klasse **Label** aus dem Paket **javafx.scene.control** realisiert man Bedienungshinweise in Schrift- und/oder Bildform. Im folgenden Beispielprogramm zum Zählen von Vorkommnissen und Objekten aller Art



befindet sich unter dem Befehlsschalter ein Label zur Anzeige des aktuellen Zählerstands, das folgendermaßen instanziert wird:

```
String lblPrefix = "Zählerstand: ";
Label lblText = new Label(lblPrefix + "0");
```

Für Textänderungen im Programmablauf verwendet man die **Label**-Methode **setText**(), z. B.:

```
lblText.setText(lblPrefix + numClicks);
```

Das zweite Label im aktuellen Einstiegsbeispiel dient zur Anzeige von Bilddateien (Java-Maskottchen Duke in verschiedenen Posen), die von **Image**-Objekten repräsentiert werden. Bevor ein **Image**-Objekt mit der **Label**-Methode **setGraphic**() seiner Verwendung zugeführt werden kann, muss es noch in ein **ImageView**-Objekt verpackt werden:

```
Image[] icons = new Image[3];
icons[0] = new Image(getClass().getResourceAsStream("duke.png"));
icons[1] = new Image(getClass().getResourceAsStream("fight.gif"));
icons[2] = new Image(getClass().getResourceAsStream("snooze.gif"));
ImageView imageView = new ImageView(icons[0]);
Label lblIcon = new Label();
lblIcon.setGraphic(imageView);
```

Als Grafikformate unterstützt die Klasse **Image**:

- BMP (*Bitmap*)
- GIF (*Graphics Interchange Format*)
- JPEG (*Joint Photographic Experts Group*)
- PNG (*Portable Network Graphics*).

Um das Icon im Programmablauf auszutauschen, erhält das **ImageView**-Objekt mit der Methode **setImage**() eine neue Füllung, z. B.:

```
imageView.setImage(icons[iconInd]);
```

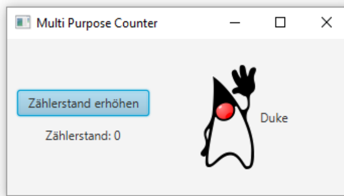
Neben **Label**-Objekten lassen sich auch diverse andere JavaFX-Komponenten mit Bildern verschönern (z. B. Befehlsschalter).

Text und Grafik können auch gemeinsam auftreten, wobei die Eigenschaft **ContentDisplay** über die relative Anordnung von Text und Grafik entscheidet. Im folgenden Codesegment

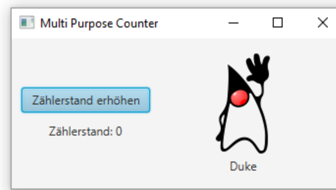
```
Label lblIcon = new Label("Duke");
lblIcon.setGraphic(imageView);
lblIcon.setContentDisplay(ContentDisplay.TOP);
```

sorgt ein Aufruf der Methode **setContentDisplay**() mit dem Parameterwert **ContentDisplay.TOP** dafür, dass die Grafik *über* dem Text erscheint. Anschließend sind einige Anordnungen zu sehen:

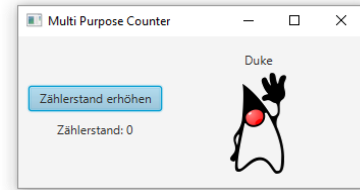
ContentDisplay.LEFT
(= Voreinstellung)



ContentDisplay.TOP



ContentDisplay.BOTTOM



Mit der Methode

public final void setGraphicTextGap(double *value*)

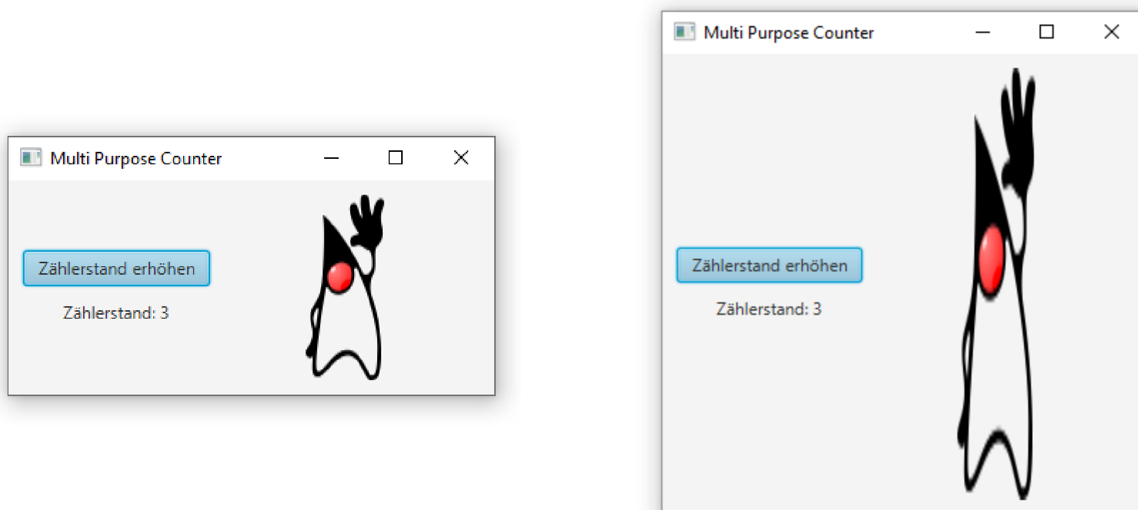
lässt sich der Abstand zwischen Text und Grafik verändern (Voreinstellung: 4), z. B.:

```
lblIcon.setGraphicTextGap(10);
```

Mit der im Abschnitt 13.5.3.1 beschriebenen Technik zum Verbinden von JavaFX-Eigenschaften kann im Beispiel dafür gesorgt werden, dass die Höhe des **ImageView**-Objekts (seine Eigenschaft **fitHeight**) an die Höhe des **BorderPane**-Containers (an dessen Eigenschaft **height**) gebunden wird:

```
imageView.fitHeightProperty().bind(root.heightProperty().subtract(2*dist));
```

Im Beispiel sorgt die Methode **subtract()** aus dem High-Level Binding-API (siehe Abschnitt 13.5.3.2.1) dafür, dass der Innenrand des Containers berücksichtigt wird. Es resultiert eine Grafik mit dynamisch angepasster Höhe:



13.7.2 Button

Befehlsschalter werden in JavaFX durch die Klasse **Button** aus dem Paket **javafx.scene.control** realisiert. Die Syntax zum Deklarieren bzw. Erzeugen eines Schalters mit Beschriftung bietet keinerlei Überraschungen:

```
Button btnAdd = new Button("Zählerstand erhöhen");
```

Das im Abschnitt 13.7.1 vorgestellte Mehrzweckzählprogramm besitzt einen Schalter, der nach einem Klickereignis den Zählerstand erhöht, die Anzeige des Text-Labels aktualisiert und als Prävention gegen Müdigkeit des Benutzers das **Image**-Objekt austauscht. Diese Arbeiten verrichtet aber nicht das **Button**-Objekt selbst, sondern ein dort per **setOnAction**() registriertes Objekt vom Typ **EventHandler<ActionEvent>**. Seit Java 8 kann man den Ereignisempfänger per Lambda-Ausdruck realisieren (siehe Abschnitt 12.1):

```
btnAdd.setOnAction(event -> {
   numClicks++;
   lblText.setText(lblPrefix + numClicks);
   if (iconInd < icons.length-1)
      iconInd++;
   else
      iconInd = 0;
   imageView.setImage(icons[iconInd]);
});
```

Ein Aufruf der Methode **setMnemonicParsing()** mit dem Parameterwert **true** sorgt bei **Labeled**-Objekten (also insbesondere auch bei **Button**-Objekten) dafür, dass der erste Unterstrich in der Beschriftung eine besondere Bedeutung erhält:

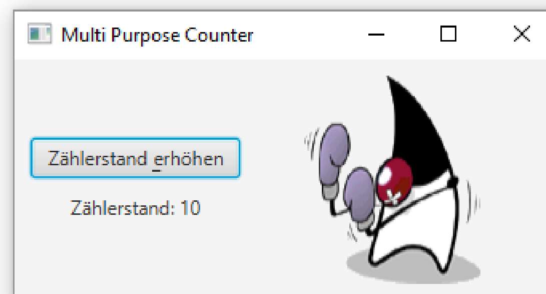
- Für das auf den ersten Unterstrich folgende Zeichen wird eine **Alt**-Tastenkombination vereinbart.
- Der erste Unterstrich wird im Programm nicht angezeigt.

Ist eine **Alt**-Tastenkombination vereinbart, wird im Programmablauf nach Betätigung der **Alt**-Taste das auf den ersten Unterstrich folgende Zeichen unterstrichen. Solange dieser Zustand nicht eine weitere Betätigung der **Alt**-Taste aufgehoben wird, hat die Eingabe des unterstrichenen Zeichens denselben Effekt wie ein Mausklick auf das Steuerelement.

Im folgenden Code-Segment

```
Button btnAdd = new Button("Zählerstand _erhöhen");
btnAdd.setMnemonicParsing(true);
```

wird für einen Befehlsschalter mit Unterstrich im Beschriftungstext die (per Voreinstellung inaktive) Kurzwahldefinition aktiviert:

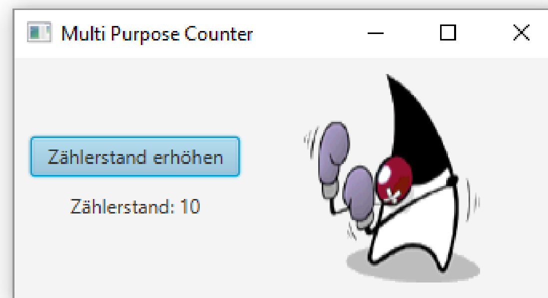


Besitzt ein Schalter den Eingabefokus (erkennbar an einem glimmenden Rand, siehe Beispiel), dann kann per Voreinstellung sein Klickereignis mit der Leertaste ausgelöst werden. Im Beispielprogramm hat der Schalter den Fokus sicher, weil keine weiteren fokussierbaren Steuerelemente vorhanden sind.

Für *einen* Schalter pro Fenster kann man zur weiteren Bedienungserleichterung das Auslösen per **Enter**-Taste ermöglichen, indem man ihn durch einen Aufruf der Methode **setDefaultButton()** zum **Default Button** ernennt, z. B.:

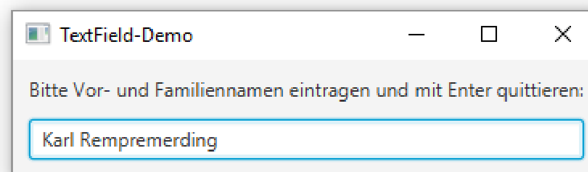
```
btnAdd.setDefaultButton(true);
```

Während (bei Verwendung des voreingestellten JavaFX-Designs) der aktiv gewählte Schalter an einem glimmenden Rand zu erkennen ist (siehe vorheriges Bildschirmfoto), ist beim Default Button der Innenraum blau gefärbt. Aktuell sind im Beispielprogramm beide Rollen (Fokusinhaber, Default Button) und dementsprechend beide Hervorhebungen aktiv:

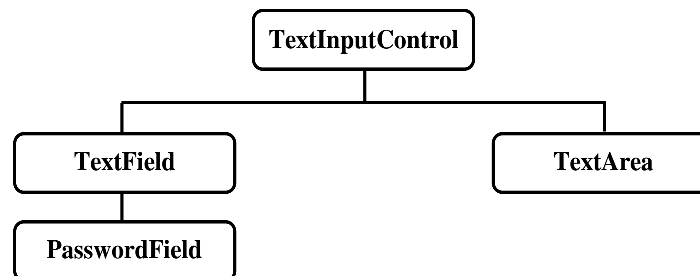


13.7.3 Einzeiliges Texteingabefeld

Um in einem JavaFX-Fenster eine einzelne Zeile mit Text zu erfassen, verwendet man die Klasse **TextField** , z. B.:

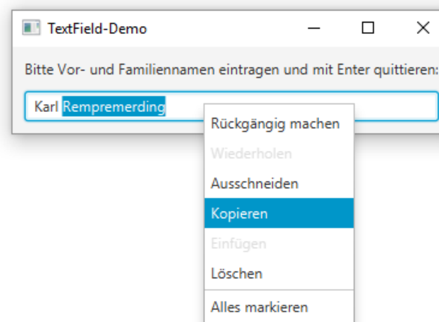


Sie stammt wie die (im Manuskript nicht behandelte) Klasse **TextArea** zur Erfassung von *mehrzeiligem* Text von der Klasse **TextInputControl** ab:



Eine **TextInputControl**-Komponente besitzt ab Fabrik einige Interaktionskompetenzen:

- Unterstützung der Zwischenablage
- Undo-Funktion (unter Windows z. B. über die Tastenkombination **Strg-Z**)
- Ein Kontextmenü, z. B.:

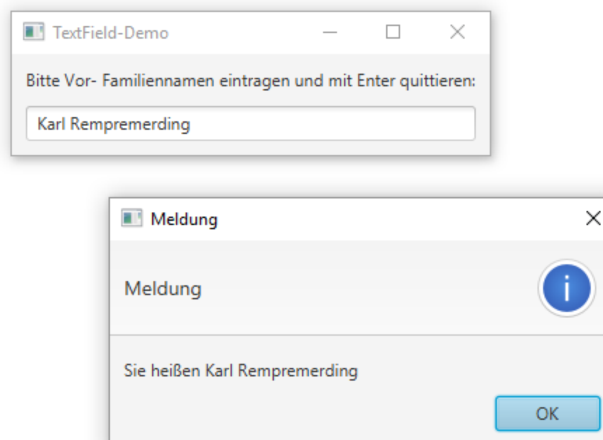


Der Quellcode des Beispielprogramms:

```
import javafx.application.Application;
 . . .
import javafx.stage.Stage;

public class TextFieldDemo extends Application {
  @Override
  public void start(Stage stage) {
    double dist = 10.0;
    VBox root = new VBox(dist);
    root.setPadding(new Insets(dist, dist, dist, dist));

    Label label=new Label("Bitte Vor- und Familiennamen eintragen und mit Enter quittieren:");
    TextField name = new TextField();

    root.getChildren().addAll(label, name);
    root.setAlignment(Pos.CENTER);

    name.setOnAction(e -> {
        Alert alert = new Alert(AlertType.INFORMATION, "Sie heißen " + name.getText());
        alert.showAndWait();
        });

    stage.setScene(new Scene(root));
    stage.setTitle("TextField-Demo");
    stage.show();
  }

  public static void main(String[] args) {
    launch(args);
  }
}
```

Das Tabulatorzeichen wird von einer **TextField**-Komponente *nicht* entgegengenommen, weil die Tabulatortaste in der Regel den Eingabefokus zum nächsten Steuerelement bewegt. Ist hingegen eine **TextArea**-Komponente der Fokusinhaber, dann wird durch dieselbe Taste ein Tabulatorzeichen in den Text eingefügt.

Drückt der Benutzer die **Enter**-Taste, während eine **TextField**-Komponente den Eingabefokus besitzt, dann wird ein **ActionEvent** ausgelöst. Im Beispiel präsentiert der per Lambda-Ausdruck realisierte Event Handler daraufhin einen Benachrichtigungs-Standarddialog mit dem erfassten Text, den er über die **TextInputControl**-Methode **getText()** ermittelt:



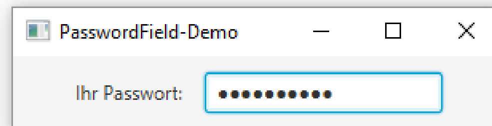
Statt der voreingestellten und im Beispiel angemessenen linksbündigen Ausrichtung des Textfeldinhalts kann mit der **TextField**-Methode **setAlignment()** unter Verwendung eines Parameterobjekts vom Aufzählungstyp **Pos** (aus dem Paket **javafx.geometry**) auch eine zentrierte oder rechtsbündige Ausrichtung gewählt werden, z. B.:

```
name.setAlignment(Pos.BASELINE_RIGHT);
```

Rechtsbündige Textfelder sind bei der Erfassung von Zahlen zu bevorzugen.

Mit **setEditable(false)** wird für eine **TextField**-Komponente festgelegt, dass ihr Text vom Benutzer *nicht* geändert werden darf.

Zum Erfassen von **Passwörtern** steht die von **TextField** abstammende Klasse **PasswordField** bereit, die im Unterschied zu ihrer Basisklasse für jedes eingegebene Zeichen einen Punkt anzeigt, z. B.:

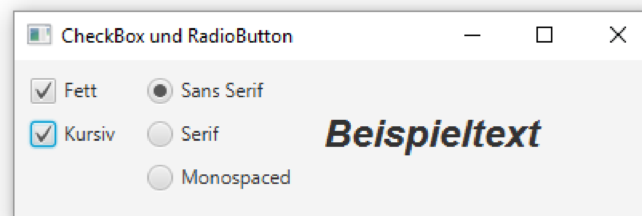


13.7.4 Umschalter

In diesem Abschnitt werden zwei Klassen für Umschalter vorgestellt:

- Für Kontrollkästchen, die jeweils einen Zustand (de)aktivieren, steht die Klasse **CheckBox** zur Verfügung.
- Für eine Gruppe von mehreren Schaltern, von denen zur Wahl *einer* Alternative aus mehreren Optionen genau einer eingerastet werden kann, verwendet man Komponenten vom Typ **RadioButton**.

Im folgenden Programm kann für den Text einer **Label**-Komponente über zwei Kontrollkästchen die Schriftauszeichnung und über eine Radioschaltergruppe die Schriftart gewählt werden:



Das IntelliJ-Projekt zum Programm finden Sie im Ordner:

…\BspUeb\JavaFX\Steuerelemente\Umschalter

13.7.4.1 Kontrollkästchen

Im Beispielprogramm erhalten die beiden **CheckBox**-Komponenten im Konstruktoraufruf eine Beschriftung:

```
private CheckBox cbBold = new CheckBox("Fett"),
                 cbItalic = new CheckBox("Kursiv");
```

Per Voreinstellung sind als Zustände eines Kontrollkästchens die beiden Werte der booleschen Eigenschaft **selected** relevant. Über die Methode **setAllowIndeterminate()** lässt sich zusätzlich die boolesche Eigenschaft **indeterminate** aktivieren, sodass ein Kontrollkästchen drei Zustände annehmen kann, die nacheinander per Mausklick erreicht werden:

Zustand	Wert der Eigenschaft **indeterminate**	Wert der Eigenschaft **selected**	Anzeige
unbestimmt	**true**	**false**	⊟
gewählt	**false**	**true**	☑
nicht gewählt	**false**	**false**	☐

Im Beispiel wird auf den dritten Zustand verzichtet.

Aus Layout-Gründen werden die beiden Kontrollkästchen in einem eigenen **VBox**-Container untergebracht, der am linken Rand des Fensters Platz nehmen soll. Als Top-Level-Container wird ein **GridPane**-Objekt mit einer Zeile und drei Spalten verwendet:

```
GridPane root = new GridPane();
VBox vboxCheck = new VBox(dist);
root.add(vboxCheck, 0, 0);
vboxCheck.getChildren().addAll(cbBold, cbItalic);
```

Bei den **selected**-Eigenschaften der beiden **CheckBox**-Objekte registrieren wir einen per Methodenreferenz (siehe Abschnitt 12.1.3.1) realisierten Interessenten für Veränderungsereignisse:

```
cbBold.selectedProperty().addListener(this::cbChanged);
cbItalic.selectedProperty().addListener(this::cbChanged);
```

In der realisierenden Methode `cbChanged()` wird überprüft, zu welchem **CheckBox**-Objekt eine **selected**-Änderung gemeldet wird. Dann wird in Anhängigkeit vom gemeldeten neuen Wert (**true** oder **false**) die Instanzvariable `fontWeight` (Typ **FontWeight**, speichert die Fettauszeichnung) bzw. die Instanzvariable `fontPosture` (Typ **FontPosture**, speichert die Kursivauszeichnung)

```
private FontWeight fontWeight = FontWeight.NORMAL;
private FontPosture fontPosture = FontPosture.REGULAR;
```

auf den neuen Wert gesetzt:

```
private void cbChanged(ObservableValue<? extends Boolean> obs,
                       Boolean old, Boolean nev) {
   if (obs.equals(cbBold.selectedProperty()))
      if (nev == true)
         fontWeight = FontWeight.BOLD;
      else
         fontWeight = FontWeight.NORMAL;
   else
      if (nev == true)
         fontPosture = FontPosture.ITALIC;
      else
         fontPosture = FontPosture.REGULAR;
   lblBeispiel.setFont(Font.font(lblBeispiel.getFont().getFamily(),
                       fontWeight, fontPosture, fontSize));
}
```

Schließlich erhält die **Label**-Komponente per **setFont()** eine neue Schriftart. Das benötigte **Font**-Objekt wird mit der statischen **Font**-Methode **font()** produziert. Wir verwenden eine **font()** - Überladung mit vier Parametern:

- **String** *family*
  Von den im lokalen System vorhandenen Schriftartfamilien wird die am besten passende gewählt. Im Beispiel wird die vom **Label**-Objekt aktuell verwendete Schriftart mit **getFont()** ermittelt und dann mit **getFamily()** nach ihrer Familienzugehörigkeit befragt.
- **javafx.scene.text.FontWeight** *weight*
  Die Werte im Enumerationstyp **FontWeight** stehen für aufsteigend geordnete Schriftstärken (**THIN**, **EXTRA_LIGHT**, **LIGHT**, **NORMAL**, **MEDIUM**, **SEMI_BOLD**, **BOLD**,

EXTRA_BOLD, **BLACK**). Mit Java 8 unter Windows sind aber offenbar nur zwei Schriftstärken realisiert (**NORMAL**, **BOLD**).

- **javafx.scene.text.FontPosture** *posture*
  Die Werte im Enumerationstyp **FontWeight** sind **REGULAR** und **ITALIC**.
- **double** *size*
  Die Schriftgröße wird in der Einheit Punkt (= 1/72 Zoll) angegeben.

13.7.4.2 Radioschalter

Die drei **RadioButton**-Objekte des Umschalter-Beispielprogramms (siehe Einstieg von Abschnitt 13.7.4) erhalten per Konstruktor eine Beschriftung:

```
private RadioButton rbSans = new RadioButton("Sans Serif"),
                    rbSerif = new RadioButton("Serif"),
                    rbMono = new RadioButton("Monospaced");
```

Im Beispielprogramm sind die Optionsschalter in einem eigenen **VBox**-Container untergebracht, der sich in der mittleren Spalte des **GridPane** - Root-Containers befindet:

```
GridPane root = new GridPane();
VBox vboxRadio = new VBox(dist);
root.add(vboxRadio, 1, 0);
vboxRadio.getChildren().addAll(rbSans, rbSerif, rbMono);
```

Damit von den drei **RadioButton**-Objekten maximal eines ausgewählt sein kann, werden sie einem Objekt aus der Klasse **ToggleGroup** zugeordnet:

```
ToggleGroup rbGroup = new ToggleGroup();
rbGroup.getToggles().addAll(rbSans, rbSerif, rbMono);
rbSans.setSelected(true);
```

Über die Methode **getToggles()** erhält man eine beobachtbare Liste (vgl. Abschnitt 13.5.3.3), in die per **addAll()** die Optionsschalter aufgenommen werden. Mit der **RadioButton**-Methode **setSelected()** wird im Beispielprogramm dafür gesorgt, dass beim Programmstart der Radioschalter zur schnörkellosen Schriftart ausgewählt ist.

Das ausgewählte Element erfährt man von einem **ToggleGroup**-Objekt über seine Eigenschaft **selectedToggle**, bei der wir einen per Methodenreferenz (siehe Abschnitt 12.1.3.1) implementierten Change-Listener registrieren:

```
rbGroup.selectedToggleProperty().addListener(this::rbChanged);
```

In der realisierenden Methode `rbChanged()` wird passend zum ausgewählten Element die Schriftfamilie des **Label**-Objekts neu festgelegt:

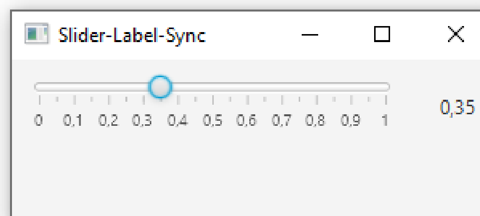
```
private String ffSans = "Arial", ffSerif = "Times New Roman", ffMono = "Courier New";
 . . .
private void rbChanged(ObservableValue<? extends Toggle> obs,
                      Toggle old, Toggle nev) {
   String family = null;
   if (nev == rbMono) {
      family = ffMono;
   } else if (nev == rbSerif) {
      family = ffSerif;
   } else if (nev == rbSans) {
      family = ffSans;
   }
   lblBeispiel.setFont(Font.font(family, fontWeight, fontPosture, fontSize));
}
```

13.8 Übungsaufgaben zum Kapitel 13

1) Welche der folgenden Aussagen sind richtig bzw. falsch?

1. Im Szenegraphen eines JavaFX-Fensters ist *genau ein* Layoutmanager vorhanden.
2. In einem **GridPane**-Layoutmanager kann ein Steuerelement auch *mehr* als eine Zelle belegen.
3. Auch in der Startklasse einer JavaFX-Anwendung ist die **main()** - Methode unverzichtbar.
4. Jede JavaFX-Anwendung benötigt mindestens eine FXML-Datei.
5. Die Property-Objekte von JavaFX halten sich an das erprobte Entwurfsmuster von JavaBeans-Eigenschaften und unterstützen außerdem leistungsfähige Techniken zur Änderungssignalisierung und zur Datenbindung.

2) Erstellen Sie ein Programm mit einem Schieberegler (Klasse **Slider** im Paket **javafx.scene.control**), dessen Wert per unidirektionaler Bindung von einem **Label** angezeigt wird:



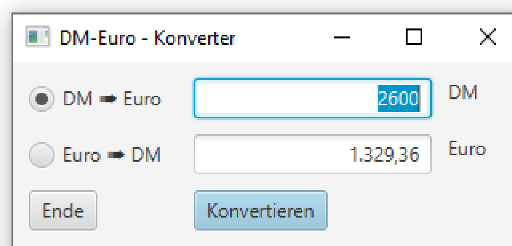
Hinweise:

- Im **Slider**-Konstruktor wählt man das Minimum, das Maximum und den Startwert.
- Den Abstand der Hauptunterteilungspunkte legt man mit der **Slider**-Methode **setMajorTickUnit**() fest.
- Die Anzahl der Nebenunterteilungspunkte zwischen zwei Hauptunterteilungspunkten wählt man mit der **Slider**-Methode **setMinorTickCount**().
- Mit der **Slider**-Methode **setSnapToTicks**() sorgt man dafür, dass nur die zu (Neben)unterteilungspunkten gehörigen Werte einstellbar sind.
- Mit der folgenden Anweisung

  ```
  label.textProperty().bind(slider.valueProperty().asString("%4.2f"));
  ```

  bindet man die **text**-Eigenschaft des Labels an die **value**-Eigenschaft des Schiebereglers, wobei der Reglerwert unter Beachtung einer Formatvorschrift in ein **String**-Objekt gewandelt wird. Dank Property-Binding kommt man ohne Ereignisbehandlung aus.

3) Erstellen Sie einen DM-Euro - Konverter:



Hinweise:

- Die untere **TextField**-Komponente soll nur zur Ausgabe dienen. Daher sollte per **setEditable(false)** festgelegt werden, dass sie vom Benutzer nicht geändert werden kann.
- Bei dem horizontalen Pfeil in den Beschriftungen der Optionsschalter handelt es sich um das Unicode-Zeichen mit der Nummer 0x27A0. Über eine Unicode-Escape-Sequenz (siehe Abschnitt 3.3.11.4) lassen sich beliebige Unicode-Zeichen in einem Java-Programm verwenden, z. B.:

```
dm2euro = new RadioButton("DM "+'\u27a0'+" Euro");
```

14 Ein- und Ausgabe über Datenströme

Bisher haben wir Daten nur in den zu einer Methode, zu einem Objekt oder zu einer Klasse gehörigen Variablen gespeichert. Zwar ist der lesende und schreibende Zugriff auf Variablen im Hauptspeicher bequem und schnell zu realisieren, doch spätestens beim Verlassen des Programms gehen alle Variableninhalte verloren. In diesem Kapitel behandeln wir elementare Verfahren zum **sequentiellen Datenaustausch** zwischen den Variablen eines Java-Programms und externen Datenquellen bzw. -senken. Es werden Verfahren beschrieben, um Zeichenfolgen, primitive Werte (Typ **byte**, **int**, **double** etc.) oder ganze Objekte in eine Datei auf der Festplatte zu schreiben bzw. von dort zu lesen. Außerdem werden wir uns mit der Verwaltung von Dateien und Verzeichnissen beschäftigen.

Ausblick auf zwei verwandte Themen:[1]

- Bei der Beschäftigung mit **Netzwerkverbindungen** werden Sie weitere, außerordentlich wichtige Datenquellen bzw. -senken kennenlernen und dabei von Ihren Kenntnissen über die sequentielle Datenstromtechnik profitieren.
- Bei der Suche nach einer professionellen Persistenzlösung spielt die **Datenbankprogrammierung** eine wichtige Rolle. Ihre leistungsfähigen Datenverwaltungstechniken bewähren sich im Netzwerk- und im Mehrbenutzerkontext. Dabei überlässt man den direkten Kontakt mit Dateien einer speziellen Software, dem Datenbankmanagement-System (DBMS).

Mit dem Vorsatz, komplexe Dateiverwaltungsaufgaben einem DBMS zu überlassen, verzichten wir in diesem Manuskript auf die Behandlung des **wahlfreien Zugriffs** auf Dateiinhalte (siehe z. B. Krüger & Hansen 2014, Kapitel 21) und beschränken uns auf Techniken, die externe Datenquellen bzw. -senken unidirektional im Vorwärtsgang bearbeiten.

Wir beschränken uns außerdem auf die am Datenstrommodell (siehe Abschnitt 14.1.1) orientierte Ein-/Ausgabetechnik. Die als Ergänzung zur Datenstromtechnik für Anwendungen mit intensivem Datentransfer konzipierte **Channel-Technik** kommt nicht *explizit* zum Einsatz, wird allerdings von etlichen API-Klassen im Hintergrund verwendet.[2]

Unsere Beispielprogramme verwenden regelmäßig Klassen aus den Paketen **java.io** und **java.nio.file**, sodass sich der Import dieser Pakete meistens lohnt:

```
import java.io.*;
import java.nio.file.*;
```

14.1 Grundlagen

14.1.1 Datenströme

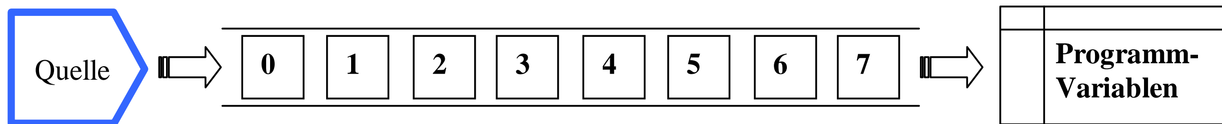
In Java wird die sequentielle Datenein- und -ausgabe über sogenannte *Ströme* (engl. *streams*) abgewickelt. Ein Programm liest Bytes[3] aus einem **Eingabestrom**, der aus einer Datenquelle (z. B. Datei, Array, Eingabegerät, Netzwerkverbindung) gespeist wird:

---

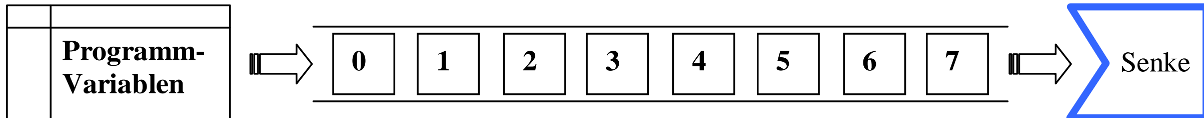
[1] Leider sind die Kapitel über Netzwerk- und Datenbankprogrammierung mangels Zeit für die erforderliche Aktualisierung derzeit nicht im Manuskript enthalten.

[2] Die Channel-Technik setzt verstärkt auf innovative Ein-/Ausgabe - Kompetenzen des zugrundeliegenden Betriebssystems. Ein **FileChannel**-Objekt transportiert in der Regel Daten zwischen einer Datei und einem **ByteBuffer**-Objekt, das als Programm-interner Zwischenspeicher dient. Wenn über die statischen Methoden **newOutputStream()**, **newInputStream()**, **newBufferedWriter()** oder **newBufferedReader()** der Klasse **Files** ein Datenstromobjekt angefordert wird, dann kommt im Hintergrund die Channel-Technik zum Einsatz.

[3] Wenn im Kapitel 14 der Namensteil *Byte* auftaucht, ist keine Java-Wrapper-Klasse gemeint, sondern eine 8 Bit umfassende Informationseinheit der Datenverarbeitung.



Ein Programm **schreibt** Bytes in einen **Ausgabestrom**, der die Werte von Programmvariablen zu einer Datensenke befördert (z. B. Datei, Array, Ausgabegerät, Netzverbindung):



In der Regel kommen *externe* Quellen bzw. Senken zum Einsatz (Dateien, Geräte, Netzwerkverbindungen). Gelegentlich werden aber programminterne Objekte per Datenstromtechnik angesprochen (z. B. **byte**-Arrays, **String**-Objekte).

Mit dem Datenstromkonzept wird bezweckt, Anweisungen zur Ein- oder Ausgabe von Daten möglichst unabhängig von den Besonderheiten konkreter Datenquellen und -senken formulieren zu können.

Ein- bzw. Ausgabeströme werden in Java-Programmen durch Objekte aus Klassen des Pakets **java.io** repräsentiert. Dort finden sich auch Datenstromklassen zum Transport von höheren Datentypen, die intern einen Byte-Strom mit direktem Kontakt zur Quelle bzw. Senke verwenden.

Datenströme, die von Beginn an zum Java-API gehören, haben zwar eine konzeptionelle Verwandtschaft mit den (z. B. auf den Elementen von Kollektionen basierenden) Strömen, die in Java 8 zur Unterstützung der funktionalen Programmierung und der Parallelverarbeitung eingeführt wurden (siehe Abschnitt 12.2), doch weichen die technischen Realisierungen stark voneinander ab.

14.1.2 Beispiel

Das folgende Programm schreibt einen **byte**-Array in eine Datei und liest die Daten anschließend wieder zurück:

Quellcode

```java
import java.io.*;
import java.nio.file.*;

class IOIntro {
   public static void main(String[] args) throws IOException {
      byte[] arro = {0, 1, 2, 3, 4, 5, 6, 7};
      byte[] arri = new byte[8];
      Path file = Paths.get("demo.bin");

      try (OutputStream os = Files.newOutputStream(file)) {
         os.write(arro);
      }

      try (InputStream is = Files.newInputStream(file)) {
         is.read(arri);
      }

      for (int i : arri)
         System.out.println(i);
   }
}
```

Ausgabe

```
0
1
2
3
4
5
6
7
```

Zum Schreiben wird über die statische Methode **newOutputStream()** der Klasse **Files** im Paket **java.nio.file** das Ausgabestromobjekt `os` mit einem von der Basisklasse **OutputStream** abstammenden Typ erzeugt und mit der vom **Path**-Parameterobjekt `file` repräsentierten Datei verbunden. Nachdem das Schreiben durch die Methode **write()** erledigt ist, wird die Datei geschlossen. Dies geschieht über die seit Java 7 verfügbare **try**-Anweisung mit automatischer Ressourcenfreigabe (vgl. Abschnitt 11.8.2).

Zum Lesen wird auf analoge Weise das Eingabestromobjekt `is` mit einem von der Basisklasse **InputStream** abstammenden Typ erzeugt und mit der zuvor gefüllten Datei verbunden. Eine **try**-Anweisung mit automatischer Ressourcenfreigabe sorgt wieder dafür, dass nach dem Lesen per **read()** - Methode die nicht mehr benötigte Datei schnell und garantiert (unter allen Umständen) freigegeben wird (durch einen **close()** - Aufruf hinter den Kulissen).

Bei der Konstruktion eines Datenstromobjekts sowie beim Lesen bzw. Schreiben von Daten kann es zu einer von **IOException** abstammenden Ausnahme kommen, die entweder in einer **try-catch** - Anweisung abgefangen oder im Definitionskopf der betroffenen Methode deklariert werden muss (vgl. Abschnitt 11.4.2). Das Beispielprogramm beschränkt sich der Einfachheit halber auf eine Deklaration der Ausnahmeklasse **IOException** und wird infolgedessen im Fehlerfall nach einer Stack Trace - Ausgabe von der JVM beendet (vgl. Abschnitt 11.1).

In realen Anwendungen werden statt Bytes in der Regel höhere Datentypen (z. B. Unicode-Zeichen, **double**-Werte, beliebige Objekte) geschrieben oder gelesen. Trotzdem ist das Beispiel nicht überflüssig, weil es die Verwendung von Byte-orientierten Datenströmen vorführt, auf denen die Ströme für höhere Datentypen basieren.

14.1.3 Klassifikation der Stromverarbeitungsklassen

Das Paket **java.io** enthält vier abstrakte Klassen, von denen die für uns relevanten Stromverarbeitungsklassen abstammen:

- **InputStream** und **OutputStream**
  Die Klassen aus diesen beiden Hierarchien lesen bzw. schreiben **Ströme mit Bytes**. Byte-Ströme transportieren **binäre Daten** (Werte mit einem primitiven Datentyp oder komplette Objekte).
- **Reader** und **Writer**
  Die Klassen aus diesen beiden Hierarchien lesen bzw. schreiben **Ströme aus Zeichen** in einer bestimmten Kodierung (z. B. UTF-8 oder ANSI). Wird ein Zeichenstrom in eine Datei geleitet, kann diese anschließend mit jedem Texteditor bearbeitet werden, der die verwendete Kodierung versteht.

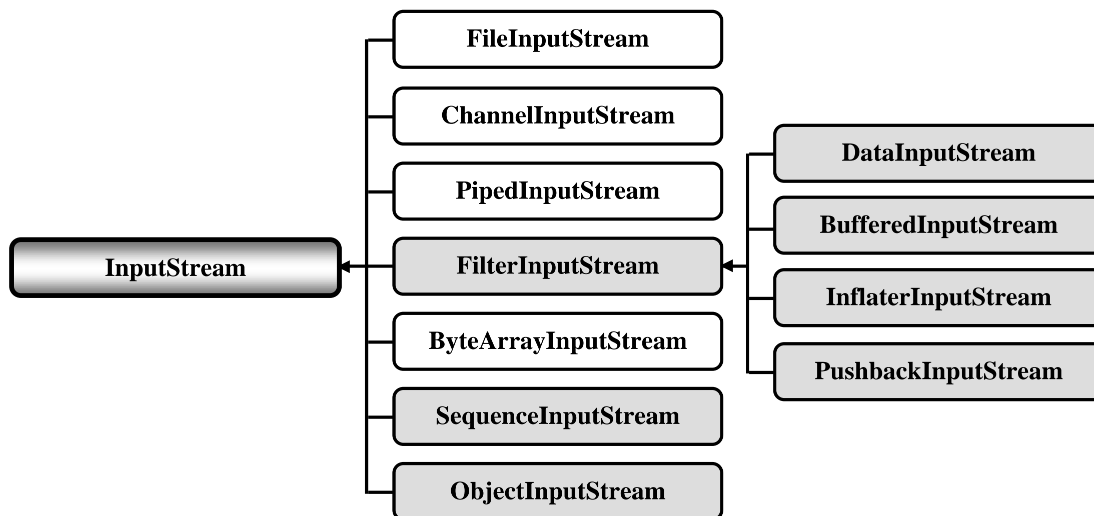
Die Byte-orientierten Klassen gehörten schon zur ersten Java-Generation und sollten ursprünglich auch zur Verarbeitung von Texten dienen. Zur Lösung von Problemen bei der Internationalisierung von Java-Programmen wurden mit Java 1.1 die zeichenorientierten Klassen hinzugefügt. Wo alte und neue Lösungen zur Verarbeitung von Textdaten konkurrieren, sollten die zeichenorientierten Klassen den Vorzug erhalten.

Bei den Abkömmlingen der vier abstrakten Klassen sind nach der *Funktion* zu unterscheiden:

- **Ein- bzw. Ausgabeklassen**
  Sie haben **direkten Kontakt zu einer Datenquelle bzw. -senke**. Sollen z. B. Bytes aus einer Datei gelesen bzw. in eine Datei geschrieben werden, kommen die Klassen **FileInputStream** bzw. **FileOutputStream** zum Einsatz.

- **Eingabe-** bzw. **Ausgabetransformationsklassen**
  Sie dienen zum **Transformieren** von Strömen und werden oft auch als *Filterklassen* bezeichnet. Sollen z. B. Werte mit beliebigem primitivem Datentyp (**int**, **double**, etc.) aus einer Datei gelesen werden, schaltet man einen Filterstrom und einen Eingabestrom hintereinander:
  - Ein Objekt der Eingabestromklasse **FileInputStream** ist mit der Datei verbunden und besorgt dort Byte-Sequenzen.
  - Ein Objekt der Filterstromklasse **DataInputStream** setzt Byte-Sequenzen zu den angeforderten primitiven Werten zusammen.
  - Wir richten unsere Anforderungen an den Filterstrom.

Den Hierarchien zu den vier Basisklassen werden später eigene Abschnitte gewidmet. Vorab werfen wir schon einmal einen Blick auf die **InputStream**-Hierarchie. In der folgenden Abbildung sind die Eingabeklassen mit einen weißen, und die Eingabetransformationsklassen mit einem grauen Hintergrund dargestellt:



14.1.4 Aufbau und Verwendung der Transformationsklassen

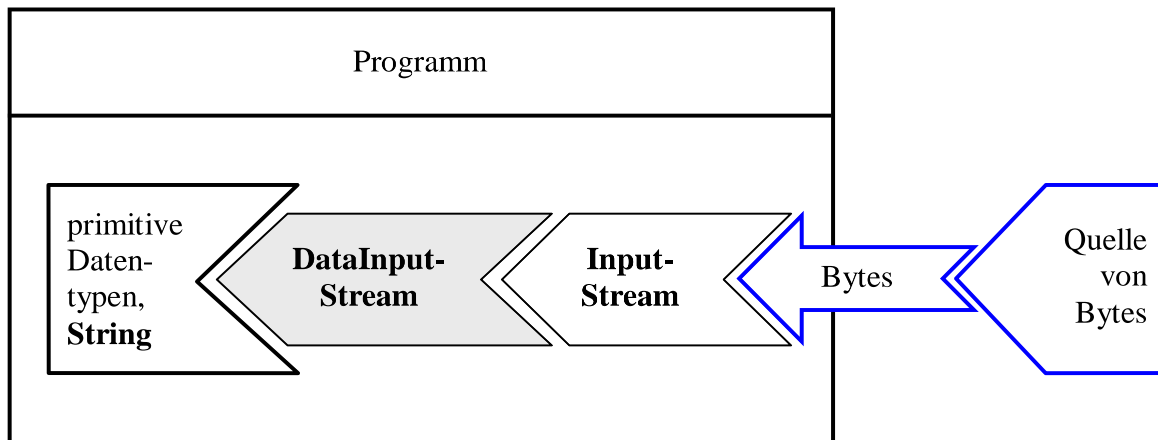
Eine Transformations- bzw. Filterklasse baut auf einer anderen Datenstromklasse auf und stellt Methoden für eine erweiterte Funktionalität zur Verfügung. Wie diese Zusammenarbeit organisiert wird, betrachten wir am Beispiel der Eingabetransformationsklasse **DataInputStream** aus der **InputStream**-Hierarchie. Diese Klasse besitzt …

- eine Instanzvariable vom Typ **InputStream**,
- in ihrem Konstruktor einen Parameter vom Typ **InputStream**, dessen Wert der **InputStream**-Instanzvariablen zugewiesen wird.

Folglich muss beim Erstellen eines **DataInputStream**-Objekts die Referenz auf ein Objekt aus einer von **InputStream** abstammenden Klasse übergeben werden. Im folgenden Beispiel wird das **InputStream**-Objekt von der statischen **Files**-Methode **newInputStream()** geliefert, die als Parameter ein **Path**-Objekt erhält, das eine Datei repräsentiert (zu den Klassen **Files** und **Path** siehe Abschnitt 14.2.1):

```
DataInputStream dis = new DataInputStream(Files.newInputStream(file))
```

Die Transformationsleistung eines **DataInputStream**-Objekts besteht darin, Werte primitiver Datentypen aus einer **byte**-Sequenz passender Länge zusammenzusetzen. Der Filterstrom nimmt also Bytes entgegen und liefert z. B. **int**-Werte (zusammengesetzt aus jeweils 4 Bytes) ab. Aus dem elementaren Byte-Strom wird ein Strom, dem Daten von primitivem Typ entnommen werden können:



Wird für das **DataInputStream**-Objekt die **close()** - Methode aufgerufen (vgl. Abschnitt 14.1.5), dann leitet es diese Botschaft an das verbundene **InputStream**-Objekt weiter.

Im folgenden Beispielprogramm kooperieren ein **DataInputStream**-Objekt und ein **FileInputStream**-Objekt dabei, **int**-Werte aus einer Datei zu lesen. Zuvor werden diese **int**-Werte in dieselbe Datei geschrieben, wobei ein Objekt der Ausgabetransformationsklasse **DataOutputStream** und ein Objekt der Ausgabeklasse **FileOutputStream** kooperieren. Hier zerlegt der Filter die **int**-Werte in einzelne Bytes und schiebt sie in den Ausgabestrom.

Quellcode

```
import java.io.*;
import java.nio.file.*;

class Filterklassen {
  public static void main(String[] args) throws IOException {
    Path file = Paths.get("demo.dat");
    int[] iar = {1024, 2048, 4096, 8192};

    try (DataOutputStream dos = new DataOutputStream(Files.newOutputStream(file))) {
      for(int el : iar)
        dos.writeInt(el);
    }

    try (DataInputStream dis = new DataInputStream(Files.newInputStream(file))) {
      for(int i = 0; i < iar.length; i++)
        System.out.println(iar[i] = dis.readInt());
    }
  }
}
```

Ausgabe

```
1024
2048
4096
8192
```

Am Beispiel **DataInputStream** sollen noch einmal wichtige Merkmale einer Transformations- bzw. Filterklasse zusammengefasst werden:

- Die Filterklasse **DataInputStream** besitzt eine im Konstruktor initialisierte Instanzvariable vom Typ **InputStream**, über die der Kontakt zu einem Eingabestromobjekt hergestellt wird.
- Die **DataInputStream**-Eingabemethoden beauftragen den eingebundenen **InputStream**, Bytes in hinreichender Zahl zu beschaffen. Diese werden dann zu Werten eines primitiven Datentyps zusammengesetzt.
- **DataInputStream**-Objekte können mit jedem **InputStream**-Objekt kooperieren. Beim Lesen von primitiven Datenwerten aus einer *Datei* kann man die Eingabeklasse **FileInputStream** aus der Paket **java.io** verwenden.
- Ein Aufruf der **DataInputStream**-Methode **close()** wird an das verbundene **InputStream**-Objekt durchgereicht.

14.1.5 Zum guten Schluss

Ist ein Datenstromobjekt mit einer externen Quelle oder Senke verbunden, dann ist eine Ressource (z. B. Datei oder Netzwerkverbindung) belegt, die für andere Prozesse nicht mehr (uneingeschränkt) zur Verfügung steht. Nach der Programmbeendigung sind die belegten Ressourcen zwar auf jeden Fall wieder frei, doch sollte man die Benutzer oder andere Prozesse nicht ohne Grund so lange warten lassen. Zudem können geöffnete Dateien auch programminterne Arbeiten blockieren (z. B. das Umbenennen).

Außerdem setzen viele Datenstromklassen aus der **OutputStream**- oder **Writer**-Hierarchie **Zwischenspeicher** ein, die unbedingt vor dem Entfernen der Datenstromobjekte geleert werden müssen, z. B. durch einen **close()** - Aufruf. Anderenfalls gehen die gepufferten Daten verloren.

Alle Datenstromobjekte aus dem Java-API beherrschen die Methode **close()**, die ggf. Zwischenspeicher entleert, den Strom schließt und die assoziierten Ressourcen freigibt. Danach ist das Strombjekt zum Lesen bzw. Schreiben von Daten nicht mehr zu gebrauchen.

Manche API-Klassen überschreiben die von **Object** geerbte und vom Garbage Collector ausgeführte Methode **finalize()** und rufen dort die Methode **close()** auf, z. B. **FileOutputStream** bis zur Version Java 8:

```
protected void finalize() throws IOException {
   if (fd != null) {
      if (fd == FileDescriptor.out || fd == FileDescriptor.err) {
         flush();
      } else {
         close();
      }
   }
}
```

Es gibt jedoch zwingende Gründe für den expliziten **close()** - Aufruf:

- Es nicht keinesfalls sicher, ob die **finalize()** - Methode tatsächlich aufgerufen wird, weil der Garbage Collector nur bei Hauptspeicherbedarf zum Einsatz kommt. Erst recht sind Zeitpunkt und Reihenfolge der Aufrufe für verschiedene Objekte ungewiss. Im Java-Tutorial (Oracle 2019a), das für Java 8 geschrieben wurde, heißt es dazu unmissverständlich:[1]

  > The `finalize()` method *may be* called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you.

- Viele puffernde Ausgabeklassen (z. B. **BufferedOutputStream**, **OutputStreamWriter**) überschreiben die von **java.lang.Object** geerbte **finalize()** - Methode *nicht*. Weil das Erbstück einen leeren Anweisungsblock besitzt, wird **close**() dort nicht aufgerufen (siehe z. B. Abschnitt 14.3.1.5).

Um allen Problemen aus dem Weg zu gehen, sorgt man dafür, dass jeder Strom so früh wie möglich durch einen expliziten **close()** - Aufruf geschlossen wird. Bei manchen mit *programminternen* Quellen oder Senken verbundenen Datenströmen (z. B. **ByteArrayOutputStream**) ist die **close()** - Methode überflüssig und wirkungslos, aber nicht schädlich.

[1] https://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html

Ein Transformationsobjekt gibt einen **close()** - Aufruf an den zugrunde liegenden Datenstrom weiter, sodass bei Datenstromkopplungen von beliebiger Komplexität normalerweise ein **close()** - Aufruf an das oberste Objekt genügt.[1]

Seit Java 7 stehen zwei Möglichkeiten zur Verfügung, für die garantierte Ausführung eines **close()** - Aufrufs (auch bei Ausnahmefehlern) zu sorgen (vgl. Abschnitt 11.8). Die ältere Technik besteht darin, kritische Ein- bzw. Ausgabemethoden im überwachten Block einer **try-catch-finally** - Anweisung aufzurufen und die erforderlichen **close()** - Aufrufe im **finally**-Block vorzunehmen (vgl. Abschnitt 11.8.1). Beim korrekten Schließen von Ressourcen kommt es im Handbetrieb leicht zu Fehlern, und Evans & Flanagan (2015, S. 294) berichten von einer Einschätzung durch Software-Entwickler der Firma Oracle, wonach in der ursprünglichen JDK-Version 6 ca. 60% des Codes zur Behandlung von Ressourcen fehlerhaft war.

Seit Java 7 lässt sich das Schließen der in einem **try**-Block benötigten Ressourcen automatisieren (*try with resources*). Dazu deklariert man Objekte, die baldmöglichst zu schließende Ressourcen repräsentieren, nach dem Schlüsselwort **try** zwischen runden Klammern in einer ein- oder mehrelementigen Liste (siehe Abschnitt 11.8.2).[2] Beteiligte Klassen müssen das Interface **AutoCloseable** implementieren, was bei den Datenstromklassen im Java-API (also bei den Ableitungen der Klassen **InputStream**, **OutputStream**, **Reader** und **Writer**) der Fall ist. Der Compiler sorgt dafür, dass erforderliche **close()** - Aufrufe hinter den Kulissen unter allen Umständen automatisch und garantiert erfolgen.

Im folgenden Beispiel werden in der Methode `mean()` mit Hilfe eines Objekts vom Typ **DataInputStream**, das intern ein Objekt vom **InputStream** verwendet, **double**-Werte aus einer Datei gelesen, um den Mittelwert daraus zu berechnen:

```
import java.io.*;
import java.nio.file.*;

class TryWithResources {
   static double mean(Path file) throws IOException {
      double sum = 0.0;
      int n = 0;
      try (DataInputStream dis = new DataInputStream(Files.newInputStream(file))) {
         while (dis.available() > 0) {
            n++;
            sum += dis.readDouble();
         }
      }
      return sum/n;
   }

   public static void main(String args[]) throws IOException {
      Path file = Paths.get("werte.dat");
      System.out.println(mean(file));
   }
}
```

Die Vorteile der try-with-resources - Lösung sind:

[1] Es kann allerdings der (mehr oder weniger unwahrscheinliche) Fall auftreten, dass nach dem erfolgreichen Öffnen eines Ein- bzw. Ausgabestroms ein geplantes Filterstromobjekt *nicht* zustande kommt. In dieser Lage hätte ein **close()** - Aufruf an das nicht existente Filterobjekt eine **NullPointerException** zur Folge, und der Ein- bzw. Ausgabestrom bliebe eventuell offen.

[2] In der Programmiersprache C# bietet die **using**-Anweisung eine analoge Funktionalität (siehe z. B. Baltes-Götz 2019, Abschnitt 16.2.3).

- Die Ressourcen werden automatisch in der richtigen Reihenfolge geschlossen.
- Falls es in der Methode `mean()` zu einer **IOException** kommt, sorgt der Compiler dafür, dass die ursprüngliche Ausnahme an den Aufrufer weitergeleitet wird und nicht eine eventuell beim Schließen der Ressourcen aufgetretene Ausnahme, was beim manuellen Schließen per **finally**-Klausel passieren kann.[1]
- Der Sichtbarkeitsbereich der Ressourcen ist auf die **try**-Anweisung beschränkt, sodass der Gefahrenbereich für (fehlerhafte) Zugriffe möglichst klein gehalten wird.

14.2 Verwaltung von Dateien und Verzeichnissen

Bis Java 6 (alias 1.6) war für den Umgang mit Dateien und Verzeichnissen (z. B. Erstellen, auf Existenz prüfen, Löschen, Attribute lesen und setzen) die Klasse **File** aus dem Paket **java.io** zuständig. Seit Java 7 (alias 1.7) bietet das Paket **java.nio.file** eine bessere Unterstützung. Weil die Möglichkeiten des schon seit Java 1.4 vorhandenen Pakets **java.nio** in der Version 7 stark verbessert wurden, spricht man vom **NIO.2 - API**.

Bei neuen Projekten, die nicht auf Kompatibilität mit Java 6 angewiesen sind, sollten statt der Klasse **File** die moderneren Typen aus dem aus dem Paket **java.nio.file** (z. B. **Path**, **Paths** und **Files**) verwendet werden (siehe Abschnitt 14.2.1). Einige Nachteile der veralteten Methoden sind:

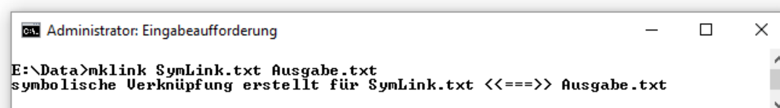
- Kommunikation von Fehlern per Rückgabewert statt über Ausnahmen (vgl. Kapitel 11)
  Wenn etwa das Löschen einer Datei über die **File**-Instanzmethode **delete()** misslingt, erhält der Aufrufer den Rückgabewert **false**. Er erfährt jedoch nichts über die Ursache des Problems (z. B. Datei nicht vorhanden, fehlende Rechte). Demgegenüber kommuniziert die statische **Files**-Methode **delete()** über Ausnahmeobjekte und kann daher den Aufrufer im Fehlerfall detailliert informierten.
- Keine Unterstützung für symbolische Links
  Dateisystemeinträge, die auf eine Datei oder ein Verzeichnis verweisen, werden von UNIX/Linux seit jeher und in Windows seit der Version Vista (bzw. Server 2008) für das NTFS-Dateisystem unterstützt.[2]

Eine Verwendung der mit Java 6 kompatiblen Technik kommt bei älteren, noch weiter zu pflegenden Projekten in Frage. Außerdem ist die Klasse **File** z. B. als Parameterdatentyp bei vielen relevanten Datenstromklassen weiter im Spiel. Daher wird im Manuskript auch die ältere Technik beschrieben (siehe Abschnitt 14.2.2).

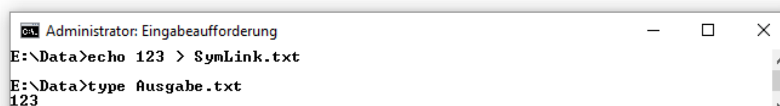
---

[1] https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

[2] Unter Windows ist zu beachten, dass die hier verbreiteten Verknüpfungen (mit der Dateinamenserweiterung **.lnk**) *keine* symbolischen Links sind. Dies sind gewöhnliche Dateien, die vom Windows-Explorer speziell behandelt werden. Man kann ab Windows Vista bzw. Windows Server 2008 auf einem Datenträger mit dem Dateisystem NTFS mit dem Kommando MKLNK einen symbolischen Link erstellen, wobei administrative Rechte erforderlich sind, z. B.:



Erfolgt nach diesem Kommando ein Schreibzugriff auf **SymLink.txt**, landen die Daten in **Ausgabe.txt**:



14.2.1 Dateisystemzugriffe über das NIO.2 - API

Wir beschränken uns auf die Typen **Path**, **Paths** und **Files** aus dem Paket **java.nio.file**.

14.2.1.1 Repräsentation von Dateisystemeinträgen

Der Typ **Path** im Paket **java.nio.file** repräsentiert einen Eintrag im hierarchischen Dateisystem eines Rechners. Ein **absoluter Pfad** ...

- beginnt mit dem Wurzelknoten (z. B. / bei Linux oder **C:** bei Windows),
- enthält optional eine Serie von Zwischenknoten,
- und endet mit dem Zielknoten (Datei, Verzeichnis oder symbolischer Link).

Fehlt der Wurzelknoten ist der Pfad **relativ** und nur in einem bestimmten Kontext (aktuellen Verzeichnis) eine korrekte Ortsangabe.

Path wurde als *Interface* realisiert, sodass es keinen Konstruktor zu diesem Typ gibt. In diese Presche springt die Klasse **Paths** mit der statischen Methode **get()**: mit den folgenden Überladungen

public static Path get(URI *uri***)**
public static Path get(String *first***, String...** *more***)**

In der Aktualparameterliste der zweiten Überladung darf dem obligatorischen *ersten* Knotennamen eine beliebig lange Liste weiterer Knotennamen folgen (zum Serienparameter siehe Abschnitt 4.3.1.4.3). Im resultierenden Objekt (aus einer das Interface **Path** implementierenden Klasse) landet also eine Serie von Knotennamen. Bei Verwendung dieser Syntax taucht das plattformabhängige Knotentrennzeichen *nicht* auf, z. B. im folgenden absoluten Pfad:

```
Path p0 = Paths.get("C:","Users","otto","Documents","java","io","ausgabe.txt");
```

Das Beispiel stammt offenbar von einem Windows-Rechner, und die ersten drei Knotennamen bezeichnen zusammen das Heimatverzeichnis des Benutzers `otto`. Mit Hilfe der statischen **System**-Methode **getProperty()** lässt sich das Heimatverzeichnis des angemeldeten Benutzers unabhängig vom konkreten Benutzernamen und von der Plattform ansprechen, z. B.:

```
Path p0 = Paths.get(System.getProperty("user.home"),"Documents","java","io","ausgabe.txt");
```

Es ist auch erlaubt, beim **get()** - Aufruf einen kompletten Pfad in *einem* **String**-Objekt unterzubringen, wobei das plattformspezifische Trennzeichen zu verwenden ist. Unter Windows sind der Rückwärtsschrägstrich (verdoppelt zur Unicode-Escape-Sequenz) und der gewöhnliche Schrägstrich erlaubt, z. B.:[1]

```
Path p1 = Paths.get("U:\\Eigene Dateien\\Java\\io\\ausgabe.txt");
Path p1 = Paths.get("U:/Eigene Dateien/Java/io/ausgabe.txt");
```

Mit den folgenden Instanzmethoden, die eine das Interface **Path** implementierende Klasse beherrscht, kann man sich über ein **Path**-Objekt informieren oder ein durch Anwendung einer Funktion daraus resultierendes **Path**-Objekt (z. B. das elterliche) anfordern:

- **public Path getFileName()**
  Liefert das relative **Path**-Objekt zum Zielknoten, z. B.:

Quellcodesegment	**toString()** - Ergebnis
`p1.getFileName()`	`ausgabe.txt`

  Das klappt auch, wenn der Zielknoten ein Ordner ist.

[1] Weil der Rückwärtsschrägstrich in der Java-Syntax die Escape-Sequenzen einleitet, muss der Wurzelknoten eines Windows-Laufwerks mit einem doppelten Rückwärtsschrägstrich (z. B. U:\\) oder mit einem Vorwärtsschrägstrich (z. B. U:/) notiert werden.

- **public Path getParent()**
  Liefert das übergeordnete **Path**-Objekt, z. B.:

Quellcodesegment	**toString()** - Ergebnis
`p1.getParent()`	`U:\Eigene Dateien\Java\io`

- **public int getNameCount()**
  Liefert die Anzahl der Namenssegmente (ohne Wurzelknoten), z. B.:

Quellcodesegment	Ergebnis
`p1.getNameCount()`	`4`

- **public Path getRoot()**
  Liefert das **Path**-Objekt zum Wurzelknoten, z. B.:

Quellcodesegment	**toString()** - Ergebnis
`p1.getRoot()`	`U:\`

- **public Path getName(int** *index***)**
  Liefert das relative **Path**-Objekt zu einem Knoten über einen nullbasierten Index, wobei die Nummer 0 zum Nachbarn des Wurzelknotens gehört, z. B.:

Quellcodesegment	**toString()** - Ergebnis
`p1.getName(1)`	`Java`

- **public Path subpath(int** *startIndex*, **int** *endIndex***)**
  Liefert eine Teilstrecke des Pfades (inklusive Startindex, exklusive Endindex), z. B.:

Quellcodesegment	**toString()** - Ergebnis
`p1.subpath(0, 2)`	`Eigene Dateien\Java`

- **public boolean isAbsolute()**
  Informiert darüber, ob ein absoluter (mit einem Wurzelknoten startender) Pfad vorliegt, z. B.:

Quellcodesegment	**toString()** - Ergebnis
`p1.isAbsolute()` `p1.getName(1).isAbsolute()`	`true` `false`

- **public Path resolve(Path** *other***)**
  public Path resolve(String *other***)**
  Eine nützliche Anwendung der Methode **resolve()** ergibt sich dann, wenn ....
  - das angesprochene **Path**-Objekt einen Wurzelknoten enthält und auf einen Ordner zeigt,
  - und das **resolve()** - Parameterobjekt einen Dateinamen enthält.

  Dann resultiert ein absoluter Pfad, der auf die Datei zeigt, z. B.:

Quellcodesegment	**toString()** - Ergebnis
`Path dir = Paths.get("U:/Eigene Dateien");` `Path datei = dir.resolve("Ausgabe.txt");`	 `U:\Eigene Dateien\Ausgabe.txt`

- **public Path resolveSibling(Path** *other***)**
  public Path resolveSibling(String *other***)**
  Im Unterschied zu **resolve()** bezieht sich bei **resolveSibling()** die Auflösung auf das übergeordnete **Path**-Objekt. Eine nützliche Anwendung ergibt sich z. B. dann, wenn ...
  - das angesprochene **Path**-Objekt einen Wurzelknoten enthält und auf eine Datei zeigt
  - und das Parameterobjekt einen Dateinamen enthält.

  Dann resultiert ein absoluter Pfad mit dem Ordner aus dem angesprochenen **Path**-Objekt und dem Dateinamen aus dem Parameterobjekt, z. B.:

Quellcodesegment	**toString()** - Ergebnis
`Path d1=Paths.get("U:/Eigene Dateien/a.txt");` `Path d2=d1.resolveSibling("b.txt");`	`U:\Eigene Dateien\b.txt`

Die **Path**-Methode **toFile()**

public File toFile()

liefert das dem **Path**-Objekt entsprechende **File**-Objekt (vgl. Abschnitt 14.2.2).

Von der Methode **toUri()** erhält man ein **URI**-Objekt (*Uniform* ***R****esource* ***I****dentifier*) zum angesprochenen **Path**-Objekt, das sich z. B. zum Öffnen einer Datei durch einen WWW-Browser verwenden lässt, z. B.:

Quellcodesegment	**toString()** - Ergebnis
`p1.toUri()`	`file:///U:/Eigene%20Dateien/Java/io/ausgabe.txt`

Mit **compareTo()** befragt, äußert sich ein **Path**-Objekt zu seiner Sortierungspriorität in Bezug auf einen Vergleichspfad. Insbesondere wird mit der Rückgabe 0 die Identität gemeldet, wobei in Abhängigkeit von der Zielplattform (z. B. unter Windows) die Groß-/Kleinschreibung für das Vergleichsergebnis irrelevant ist, z. B.:

Quellcodesegment	Ausgabe

`Path p1 = Paths.get("U:/Eigene Dateien/Java/io/Ausgabe.txt");`
`Path p2 = Paths.get("U:/eigene dateien/java/io/ausgabe.txt");`
`System.out.println(p2.compareTo(p1));`
	`0`

Damit redundante Bestandteile in der Namenssequenz eines **Path**-Objekts einen Vergleich nicht stören, sollte man diese per **normalize()** - Methode entfernen, z. B.:

Quellcodesegment	Ausgabe

`Path p1 = Paths.get("U:/Eigene Dateien/Java/io/ausgabe.txt");`
`Path p2 = Paths.get("U:/eigene dateien/java/../java/io/ausgabe.txt");`
`System.out.println(p2.compareTo(p1));`
`System.out.println(p2.normalize().compareTo(p1));`
	`-27`
	`0`

Weitere **Path**-Methoden werden anschließend im Zusammenhang mit ihrer typischen Verwendung beschrieben.

14.2.1.2 Existenzprüfung

Mit der statischen **Files**-Methode **exists()** findet man für ein **Path**-Objekt heraus, ob es bereits einen Dateisystemeintrag (Datei, Verzeichnis, symbolischer Link) mit diesem Pfad gibt, z. B.:

```
if (Files.exists(dir))
   System.out.println(dir + " existiert bereits.");
else
   if (Files.notExists(dir))
      System.out.println(dir + " existiert noch nicht");
   else
      System.out.println(dir + " hat einen unbekannten Status.");
```

Als Ursache für den **exist()** - Rückgabewert **false** kommt auch ein Zugriffsproblem in Frage.

Dass zum Zeitpunkt der Abfrage kein Dateisystemeintrag mit dem fraglichen Pfad vorhanden war, beweist die Rückgabe **true** der statischen **Files**-Methode **notExists()**.

Wie im Java-Tutorial (Oracle 2019a) zu Recht betont wird, sollte sich ein Programm anschließend (z. B. nach dem Verstreichen von etlichen Millisekunden) *nicht* auf das Existenzprüfergebnis verlassen, weil ein TOCTTOU-Fehler droht (*Time of check to time of use*).

14.2.1.3 Verzeichnis anlegen

Um das Verzeichnis

U:\Eigene Dateien\Java\io

anzulegen, erzeugen wir zunächst ein passendes **Path**-Objekt (vgl. Abschnitt 14.2.1.1):

```
Path dir = Paths.get("U:", "Eigene Dateien", "Java", "io");
```

Mit der statischen **Files**-Methode **createDirectory()**, die ein **Path**-Objekt als Parameter erwartet, lässt sich ein neues Verzeichnis in einem bereits vorhandenen Ordner anlegen. Sollen nötigenfalls auch erforderliche Zwischenstufen automatisch angelegt werden, ist die Methode **createDirectories()** zu verwenden, z. B.:

```
try {
   Files.createDirectories(dir);
} catch (FileAlreadyExistsException ae) {
   System.err.println(dir + " existiert, ist aber kein Verzeichnis.");
   System.exit(1);
}
```

Von beiden Methoden sind die folgenden geprüften Ausnahmen zu erwarten (**catch**-Block bzw. Deklaration erforderlich):

- eine allgemeine **IOException**
- die **IOException**-Spezialisierung **FileAlreadyExistsException**
  Diese Ausnahme tritt auf, wenn bereits ein Dateisystemeintrag mit dem gewünschten Namen existiert, der aber kein Verzeichnis ist. Die Aufforderung, ein bereits vorhandenes Verzeichnis anzulegen, hat *keine* Ausnahme zur Folge.

Der Exception Handler im obigen Code-Segment schreibt eine Fehlermeldung in den sogenannten *Fehlerausgabestrom*. Dieser vom Laufzeitsystem verwaltete und per Voreinstellung zur Konsole kanalisierte Standardstrom ist in Java über die statische Referenzvariable **err** der Klasse **System** anzusprechen. **System.err** zeigt wie der Standardausgabestrom **System.out** auf ein Objekt der Klasse **PrintStream** (siehe Abschnitt 14.3.1.6). Beide Standardströme werden automatisch zur Verfügung gestellt und müssen weder geöffnet noch geschlossen werden.

14.2.1.4 Datei explizit erstellen

Zwar wird z. B. beim Erzeugen eines **FileOutputStream**-Objekts eine benötigte Datei bei Bedarf automatisch erstellt, doch ergeben sich auch Anlässe, eine Datei (mit einem Inhalt von 0 Bytes) explizit anzulegen, wozu die statische **Files**-Methode **createFile()** bereitsteht. Das als Parameter benötigte **Path**-Objekt zur Datei kann man z. B. aus einem vorhanden **Path**-Objekt zum Verzeichnis und einem Dateinamen über die **Path**-Methode **resolve()** gewinnen:

```
Path file = dir.resolve("Ausgabe.dat");
try {
   Files.createFile(file);
} catch (FileAlreadyExistsException ae) {
   System.err.println(file + " existiert bereits.");
}
```

14.2.1.5 Attribute von Dateisystemobjekten ermitteln

Mit diversen statischen Methoden der Serviceklasse **Files**, die allesamt eine Ausnahme vom Typ **IOException** werfen können, lassen sich einzelne Attribute von Dateisystemobjekten ermitteln. z. B.:

- **public static FileTime getLastModifiedTime(Path** *path*, **LinkOption...** *options*) **throws IOException**
  Für das durch den Pfad bezeichnete Dateisystemobjekt erfährt man über die Rückgabe vom Typ **FileTime** den Zeitpunkt der letzten Änderung. Über die Methode **toString()** befragt, liefert das **FileTime**-Objekt eine Zeitangabe in GMT nach der Norm ISO 8601, z. B.:
  ```
  2020-02-08T18:21:29.17708Z
  ```
  Zum Parameter **LinkOption** folgt eine Erläuterung hinter der Auflistung.
- **public static long size(Path** *path*) **throws IOException**
  Bei einer regulären Datei erhält man die Größe in Bytes. Bei anderen Dateisystemobjekten (Verzeichnis oder symbolischer Link) ist die Rückgabe implementationsabhängig.
- **public static boolean isRegularFile(Path** *path*, **LinkOption...** *options*)
  public static boolean isDirectory(Path *path*, **LinkOption...** *options*)
  public static boolean isSymbolicLink(Path *path*)
  Diese Methoden informieren darüber, ob das durch den Pfad bezeichnete Dateisystemobjekt eine reguläre Datei, ein Verzeichnis oder ein symbolischer Link ist.

Mit dem bei einigen Methoden vorhandenen Serienparameter (vgl. Abschnitt 4.3.1.4.3) vom Enumerationstyp **LinkOption** legt man fest, wie **symbolische Links**, die auf eine Datei oder ein Verzeichnis verweisen, behandelt werden sollen. Per Voreinstellung wird ein Link aufgelöst, sodass die ermittelten Attributausprägungen vom Verweisziel stammen. Mit dem Parameterwert **LinkOption.NOFOLLOW_LINKS** unterbleibt die Auflösung, sodass die Attributausprägungen vom Link stammen.

In den folgenden Anweisungen werden Informationen über eine Datei gesammelt:

```
Path dir = Paths.get("U:", "Eigene Dateien", "Java", "io");
Path file = dir.resolve("Ausgabe.dat");
 . . .
System.out.println("Eigenschaften von    " + file);
System.out.println(" Größe in Bytes:     " + Files.size(file));
System.out.println(" Letzte Änderung:    " + Files.getLastModifiedTime(file));
System.out.println(" Datei:              " + Files.isRegularFile(file));
System.out.println(" Verzeichnis:        " + Files.isDirectory(file));
System.out.println(" Symb. Link:         " + Files.isSymbolicLink(file));
```

Ausgabe:

```
Eigenschaften von    U:\Eigene Dateien\Java\io\Ausgabe.dat
 Größe in Bytes:     3
 Letzte Änderung:    2020-02-08T18:21:29.17708Z
 Datei:              true
 Verzeichnis:        false
 Symb. Link:         false
```

Statt für mehrere Attribute eines Dateisystemobjekts jeweils eine zeitaufwändige Anfrage an das Dateisystem zu richten, sollte man über die **Files**-Methode **readAttributes()** ein Informationsbündel mit allen Basis-, DOS- oder POSIX-Attributen eines Dateisystemobjekts anfordern, z. B.:

```
BasicFileAttributes attr = Files.readAttributes(file, BasicFileAttributes.class);
```

Durch den zweiten Parameter mit dem geforderten Typ **Class<? extends BasicFileAttributes>** wird der gewünschte Rückgabetyp vereinbart. Gibt man (wie im Beispiel) das **Class**-Objekt zur Schnittstelle **BasicFileAttributes** (im Paket **java.nio.file.attribute**) an, dann erhält man ein Objekt,

das elementare, von vielen Dateisystemen unterstützte Attribute kapselt. Vom Rückgabeobjekt sind später die Attribute ohne Dateisystemzugriffe zu erfahren, z. B.:

```
System.out.println(" Größe in Bytes:   " + attr.size());
```

14.2.1.6 Zugriffsrechte für Dateien ermitteln

Mit den statischen Methoden **isReadable()**, **isWritable()** und **isExecutable()** der Klasse **Files** kann man feststellen, ob das aktive Programm (bzw. die JVM) eine Datei lesen, schreiben oder ausführen darf:

- **public static boolean isReadable(Path** *path***)**
  Bedeutung der Rückgabewerte:
  - **true**
    Die Datei existiert, und es bestehen Leserechte.
  - **false**
    Entweder ist die Datei nicht vorhanden, oder es bestehen keine Leserechte, oder die Leserechte sind nicht feststellbar.
- **public static boolean isWritable(Path** *path***)**
  Bedeutung der Rückgabewerte:
  - **true**
    Die Datei existiert, und es bestehen Schreibrechte.
  - **false**
    Entweder ist die Datei nicht vorhanden, oder es bestehen keine Schreibrechte, oder die Schreibrechte sind nicht feststellbar.
- **public static boolean isExecutable(Path** *path***)**
  Bedeutung der Rückgabewerte:
  - **true**
    Die Datei existiert und kann vom aktiven Programm ausgeführt werden.
  - **false**
    Entweder ist die Datei nicht vorhanden, oder es bestehen keine Ausführungsrechte, oder die Ausführungsrechte sind nicht feststellbar.

Streng kann man sich schon nach kurzer Zeit nicht mehr darauf verlassen, dass die ermittelten Zugriffsrechte noch bestehen.

Beispiel:

Quellcodesegment	Ausgabe
`System.out.println(Files.isWritable(file));`	`true`

14.2.1.7 Attribute ändern

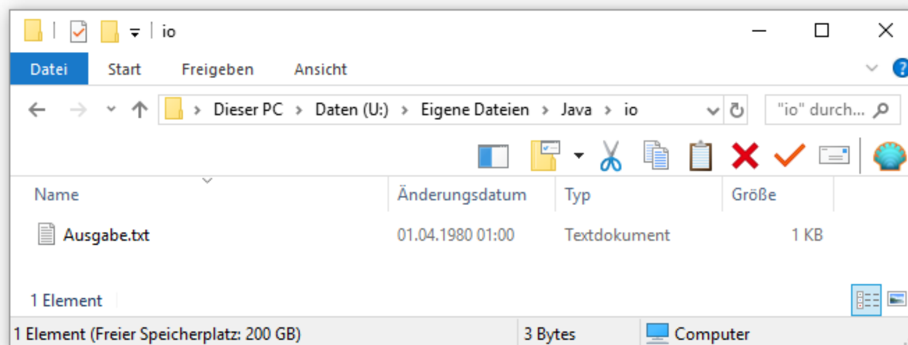
Über die Klasse **Files** lassen sich einige Attribute von Dateisystemobjekten ändern. Wir beschränken uns auf das Datum der letzten Modifikation:

public static Path setLastModifiedTime(Path *path***, FileTime...** *time***)**
throws IOException

Zum Erstellen des benötigten **FileTime**-Objekts ist die statische Methode **fromMillis()** der Klasse **FileTime** (im Paket **java.nio.file.attribute**) geeignet. Um deren Parameter über vertraute Zeiteinheiten festlegen zu können, wird im folgenden Vorschlag ein Objekt der Klasse **Calendar** verwendet:

```
Calendar cal = Calendar.getInstance();
cal.set(1980, Calendar.APRIL, 1, 0, 0, 0);
Files.setLastModifiedTime(file, FileTime.fromMillis(cal.getTimeInMillis()));
```

So gelingt der Sprung zurück in die Zeit vor dem ersten IBM-PC:



Wie das Beispiel zeigt, wird für die übergebene Zeitangabe die Zeitzone GMT angenommen. Eine große forensische Aussagekraft sollte man dem Letztzugriffsdatum einer Datei offenbar nicht zumessen. Neben dem Kreationsdatum können auch diverse DOS- bzw. POSIX-Dateiattribute (z. B. Hidden, Owner) gesetzt werden (siehe Java-Tutorial, Oracle 2019a).[1]

14.2.1.8 Über Verzeichniseinträge iterieren

Zu einem Ordner liefert die statische **Files**-Methode **newDirectoryStream()** ein Objekt aus einer Klasse, welche u.a. die Interfaces **AutoClosable** und **Iterable<Path>** beherrscht.[2] Für das geöffnete Verzeichnis werden Ressourcen belegt, sodass ein möglichst frühes Schließen erforderlich ist, was am besten in einer **try**-Anweisung mit automatischer Ressourcenfreigabe geschieht. Im folgenden Beispiel wird das **DirectoryStream<Path>** - Objekt in einer erweiterten **for**-Schleife (siehe Abschnitt 3.7.3.2) dazu verwendet, um über die Einträge im Verzeichnis zu iterieren:

```
try (DirectoryStream<Path> ds = Files.newDirectoryStream(dir)) {
   for (Path path: ds)
      System.out.println(path.getFileName());
}
```

14.2.1.9 Datei und Ordner kopieren

Zum Kopieren von Dateien wurde vor Java 7 häufig ein Gespann aus einem **FileInputStream** und einem **FileOutputStream** mit Zwischenspeicherung in einem **byte**-Array verwendet (siehe Beispiel im Abschnitt 14.3.1.2). Mit der in drei Überladungen vorhandenen statischen **Files**-Methode **copy()** lässt sich deutlich bequemer eine meist flottere Lösung erstellen, z. B.:

```
import java.io.IOException;
import java.nio.file.*;
class FilesCopy {
   public static void main(String[] args) throws IOException{
      Path quelle = Paths.get("quelle.dat");
      Path ziel = Paths.get("ziel.dat");
      Files.copy(quelle, ziel, StandardCopyOption.REPLACE_EXISTING);
   }
}
```

---

[1] http://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html

[2] Das Rückgabeobjekt beherrscht aber weder das Interface **Collection<T>**, noch das Interface **Stream<T>**. Es ist also weder eine Kollektion im Sinne von Kapitel 10, noch ein Strom im Sinne von Kapitel 12.

Im Beispiel kommt die folgende **copy()** - Überladung mit **Path**-Parametern für Quelle und Ziel zum Einsatz:

public static Path copy(Path *source*, **Path** *target*, **CopyOption...** *options*)
throws IOException

Der optionale Parameter vom Interface-Typ **CopyOption** akzeptiert eine Serie von Werten, wobei die folgenden Konstanten der Enumerationen **StandardCopyOption** und **LinkOption** erlaubt sind:[1]

- **StandardCopyOption.REPLACE_EXISTING**
  Bei einer bereits existenten Zieldatei wird das Überschreiben erlaubt. Anderenfalls wird ggf. eine Ausnahme vom Typ **FileAlreadyExistsException** geworfen.
- **StandardCopyOption.COPY_ATTRIBUTES**
  Die Attribute der Quelle sollen auf das Ziel übertragen werden, sofern dies vom Betriebs- bzw. Dateisystem unterstützt wird.[2]
- **LinkOption.NOFOLLOW_LINKS**
  Ist die Quelle ein symbolischer Link, dann wird per Voreinstellung das Verweisziel kopiert. Mit der Option **NOFOLLOW_LINKS** wird stattdessen der Link kopiert, wobei unter Windows Administratorrechte erforderlich sind.

Man kann auch Ordner kopieren, wobei allerdings die enthaltenen Dateisystemobjekte *nicht* einbezogen werden. Ist der Zielordner bereits vorhanden, wird eine **FileAlreadyExistsException** geworfen. Sind bei Verwendung der **StandardCopyOption.REPLACE_EXISTING** im Zielordner bereits Objekte vorhanden, wird eine Ausnahme vom Typ **DirectoryNotEmptyException** geworfen.

14.2.1.10 Umbenennen und Verschieben

Mit der statischen **Files**-Methode **move()** lässt sich ein Dateisystemobjekt umbenennen oder verschieben:

public static Path move(Path *source*, **Path** *target*, **CopyOption...** *options*)
throws IOException

Statt die Methode komplett zu beschreiben (siehe Java-Tutorial, Oracle 2019a), beschränken wir uns auf zwei Beispiele.[3]

Soll eine Datei *umbenannt* werden, gibt man eine Zieldatei im *selben* Ordner an, wobei das benötigte **Path**-Objekt bequem über die **Path**-Methode **resolveSibling()** zu erstellen ist, z. B.:

```
Files.move(file, file.resolveSibling("Umbenannt.dat"));
```

Auf die Angabe von Optionen wird in diesem Beispiel verzichtet, was bei einem Serienparameter erlaubt ist (vgl. Abschnitt 4.3.1.4.3).

Soll eine Datei *verschoben* werden, gibt man eine Zieldatei in einem *anderen* Ordner an. Im folgenden Beispiel wird die Quelldatei in das übergeordnete Verzeichnis verschoben und dabei auch noch umbenannt:

---

[1] Wie bei einem Serienparameter üblich, darf man die Aktualparameterliste auch komplett weglassen (vgl. Abschnitt 4.3.1.4.3). Die Enumerationen **StandardCopyOption** und **LinkOption** implementieren das Interface **CopyOption**.

[2] Unter Windows 10 mit dem Dateisystem NTFS hat die Option **StandardCopyOption.COPY_ATTRIBUTES** wohl *keinen* Effekt:
- Elementare Attribute (z. B. Änderungsdatum, Schreibschutz) werden auf jeden Fall (auch *ohne* **StandardCopyOption.COPY_ATTRIBUTES**) übertragen.
- Zugriffsrechte (ACLs) werden auch *mit* **StandardCopyOption.COPY_ATTRIBUTES** *nicht* übertragen.

[3] http://docs.oracle.com/javase/tutorial/essential/io/move.html

```
Files.move(file, dir.getParent().resolve("Verschoben.dat"),
           StandardCopyOption.REPLACE_EXISTING);
```

Das **Path**-Objekt zu einem (vom Wurzelknoten verschiedenen) Verzeichnis liefert über die Methode **getParent()** das übergeordnete Verzeichnis (siehe Abschnitt 14.2.1.1). Soll beim Verschieben in den übergeordneten Ordner der Dateiname beibehalten werden, bildet man den Zielpfad mit Hilfe der **Path**-Methode **getFileName**():

```
Files.move(file, dir.getParent().resolve(file.getFileName()),
           StandardCopyOption.REPLACE_EXISTING);
```

Im **CopyOption**-Serienparameter sind bei der Methode **move()** die beiden folgenden Konstanten der Enumeration **StandardCopyOption** erlaubt:

- **StandardCopyOption.REPLACE_EXISTING**
  Eine am Zielort vorhandene gleichnamige Datei soll überschrieben werden.
- **StandardCopyOption.ATOMIC_MOVE**
  Die Verschiebung wird als *atomare* Operation deklariert, sodass entweder *beide* Teilaufgaben (Anlegen am neuen Ort, Löschen am alten Ort) ausgeführt werden, oder gar keine Änderung stattfindet. Ist diese Option gesetzt, werden alle anderen ignoriert, was derzeit nur der Option **REPLACE_EXISTING** passieren kann. Wenn keine atomare Ausführung möglich ist, wird eine Ausnahme vom Typ **AtomicMoveNotSupportedException** geworfen.

14.2.1.11 Löschen

Mit der statischen **Files**-Methode **delete()** lässt sich ein Dateisystemobjekt (Datei, Ordner oder symbolischer Link) löschen:

public void delete(Path *path*) **throws IOException**

Im folgenden Beispiel werden alle Dateisystemobjekte in einem Ordner über ein Objekt der Klasse **DirectoryStream<Path>** (vgl. Abschnitt 14.2.1.8) aufgesucht und gelöscht:

```
try (DirectoryStream<Path> ds = Files.newDirectoryStream(dir)) {
   for (Path path: ds)
      Files.delete(path);
} catch (IOException ioe) {
   ioe.printStackTrace();
}
```

Damit ein Ordner gelöscht werden kann, muss er leer sein. Wird ein symbolischer Link gelöscht, bleibt sein Verweisziel unangetastet.

Ist ein zu löschendes Dateisystemobjekt *nicht* vorhanden, wirft **delete()** eine **NoSuchFileException**. Soll das Programm stattdessen kommentarlos weiterarbeiten, verwendet man statt **delete()** die Methode **deleteIfExists()**.

14.2.1.12 Informationen über Dateisysteme ermitteln

Mit Hilfe der Klassen **FileSystem**, **FileSystems** und **FileStore** kann man sich über die Dateisysteme des lokalen Rechners informieren. Ein Windows-Rechner verfügt in der Regel nur über *ein* Dateisystem, und das repräsentierende **FileSystem**-Objekt erhält man von der statischen Methode **getDefault()** der Klasse **FileSystems**, z. B.:

```
FileSystem fs = FileSystems.getDefault();
```

Genaugenommen sind die Klassen **FileSystem** und **FileStore** abstrakt, und das von **getDefault()** gelieferte Objekt gehört zu einer **FileSystem**-Ableitung, die wir nicht näher kennen müssen.

In Abhängigkeit vom Betriebssystem enthält ein Dateisystem unterschiedliche Speichereinheiten (z. B. Partitionen oder Laufwerke), die im NIO.2 - API durch Objekte der Klasse **FileStore** repräsentiert werden. Die zu einem Dateisystem gehörigen Speichereinheiten erhält man über die **FileSystem**-Methode **getFileStores()** als Objekt einer Klasse, welche das Interface **Iterable<FileStore>** implementiert, z. B.:

```
try {
   for (FileStore store: fs.getFileStores())
      getSpaceOfStore(store);
} catch (IOException ioe) {
   ioe.printStackTrace();
}
```

Unter Windows resultiert eine Liste mit den Laufwerken (inkl. Netzwerk).

Mit den folgenden Methoden der Klasse **FileStore** erhält man Informationen über die gesamte bzw. verfügbare Kapazität einer Speichereinheit:

- **public long getTotalSpace() throws IOException**
  Man erhält die Gesamtkapazität in Bytes.
- **public long getUsableSpace() throws IOException**
  Man erhält die verfügbare Kapazität in Bytes, wobei die Exaktheit der Auskunft laut API-Dokumentation nicht garantiert ist. Mit zunehmender Zeitdistanz seit der Abfrage schwindet die Genauigkeit der Auskunft ohnehin wegen der laufenden Dateisystemaktivitäten.

Im obigen Codesegment wird die folgende Methode zur Ausgabe von Kapazitätsdaten aufgerufen:

```
static void getSpaceOfStore(FileStore store) throws IOException {
   final long MEGA = 1024*1024;
   long gesamt = store.getTotalSpace()/MEGA;
   long belegt = (store.getTotalSpace()-store.getUsableSpace())/MEGA;
   long frei = store.getUsableSpace()/MEGA;
   System.out.printf("%-20s %12d %12d %12d\n", store, gesamt, belegt, frei);
}
```

Eine typische Ausgabe (Windows-Rechner mit einer SSD, einer Festplatte, einer eingelegten DVD und einer verbundenen Netzfreigabe):

```
Laufwerk               Gesamt (MB)  Belegt (MB)    Frei (MB)
System (C:)                 953368       194202       759166
DESINFECT (D:)                5711         5711            0
Daten (E:)                  753865       670792        83073
SYSVOL (V:)                  49907         2550        47357
```

14.2.1.13 Weitere Optionen

Aus Zeitgründen können einige attraktive Neuerungen im NIO.2 - API nur erwähnt werden (siehe Kapitel *Basic I/O* im Java-Tutorial, Oracle 2019a):[1]

- Rekursives Durchwandern eines Verzeichniszweigs
- Suche nach Dateinamen, die ein Muster erfüllen
- Überwachung eines Dateisystemordners auf Veränderungen (siehe z. B. Krüger & Hansen 2014, S. 461ff)

[1] http://docs.oracle.com/javase/tutorial/essential/io/

14.2.2 Dateisystemzugriffe über die Klasse File aus dem Paket java.io

In Java 6 wird der Umgang mit Dateien und Verzeichnissen (z. B. Erstellen, auf Existenz prüfen, Löschen, Attribute lesen und setzen) durch die Klasse **File** aus dem Paket **java.io** unterstützt. Viele Methoden dieser Klasse werden im weiteren Verlauf des aktuellen Abschnitts anhand von Codefragmenten aus einem Beispielprogramm mit dem folgenden Rahmen vorgestellt:

```
import java.io.*;
class FileDemo {
   public static void main(String[] args) {
      byte[] arr = {1, 2, 3};
       . . .
   }
}
```

14.2.2.1 Verzeichnis anlegen

Zunächst legen wir das Verzeichnis

U:\Eigene Dateien\Java\FileDemo\AusDir

an:

```
String ordner = "U:/Eigene Dateien/Java/FileDemo/AusDir";
File dir = new File(ordner);
if (dir.exists()) {
   if (dir.isDirectory())
      System.out.println("Das Verzeichnis " + ordner + " existiert bereits.");
   else {
      System.out.println(ordner + " existiert, ist aber kein Verzeichnis.");
      System.exit(1);
   }
} else
   if (dir.mkdirs())
      System.out.println("Verzeichnis " + ordner + " erstellt");
   else {
      System.out.println("Verzeichnis " + ordner + " konnte nicht erstellt werden.");
      System.exit(1);
   }
```

Im **File**-Konstruktor kann ein absoluter (z. B. **U:/Eigene Dateien/Java/FileDemo/AusDir**) oder ein relativer, vom aktuellen Verzeichnis ausgehender, Pfad (z. B. **AusDir**) angegeben werden.

Weil der Rückwärtsschrägstrich in Java eine Escape-Sequenz einleitet, muss unter Windows zwischen Pfadbestandteilen entweder der Vorwärtsschrägstrich (/) oder ein verdoppelter Rückwärtsschrägstrich gesetzt werden (z. B.: **U:\\Eigene Dateien\\Java\\FileDemo\\AusDir**). In der Konstanten **File.pathSeparatorChar** findet sich das für die aktuelle Plattform gültige Trennzeichen zwischen Pfadbestandteilen.

Mit der **File**-Methode **exists()** lässt sich die Existenz eines Ordners oder einer Datei überprüfen. Ihr boolescher Rückgabewert ist genau dann **true**, wenn die Suche erfolgreich war.

Ob es sich bei einem Verzeichniseintrag um ein Unterverzeichnis handelt, stellt man mit der Methode **isDirectory()** fest.

Um ein neues Verzeichnis anzulegen, verwendet man die Methode **mkdir()**. Sollen dabei ggf. auch erforderliche Zwischenstufen automatisch angelegt werden, ist die Methode **mkdirs()** zu verwenden (siehe Beispiel).

14.2.2.2 Dateien explizit erstellen

Zwar wird z. B. beim Erzeugen eines **FileOutputStream**-Objekts eine benötigte Datei bei Bedarf automatisch erstellt, doch ergeben sich auch Anlässe, eine Datei explizit anzulegen, wozu die Methode **createNewFile()** der Klasse **File** bereitsteht:

```
String name = ordner + "/Ausgabe.dat";
File f = new File(name);
if (!f.exists()) {
   try {
      f.createNewFile();
      System.out.println("Datei " + name + " erstellt");
   } catch (Exception e) {
      System.out.println("Fehler beim Erstellen der Datei " +name);
   }
}
```

Das Erzeugen eines **File**-Objekts führt *nicht* zum Erstellen einer Datei mit dem als Konstruktor-Parameter verwendeten Namen. Ebenso wird eine bereits vorhandene Datei *nicht* geöffnet, wenn ihr Name als Aktualparameter in einem **File**-Konstruktor auftritt.

14.2.2.3 Informationen über Dateien und Ordner

Neben **isDirectory**() kennen **File**-Objekte noch weitere Informationsmethoden, z. B.:

- **public String getAbsolutePath()**
  Ermittelt den absoluten Pfadnamen
- **public long lastModified()**
  Ermittelt den Zeitpunkt der letzten Änderung, gemessen in Millisekunden seit dem 1. Januar 1970 (00:00:00 GMT)
- **public long length()**
  Stellt die Größe einer Datei in Bytes fest
- **public boolean canWrite()**
  Prüft, ob das Programm (die JVM) schreibend auf eine Datei zugreifen darf
- **public long getUsableSpace()**
  Schätzt das in einem Verzeichnis (also in der zugehörigen Partition) durch den aktuellen Anwender (unter Berücksichtigung seiner Schreibrechte) nutzbare Speichervolumen in Bytes

Hier werden die Anfragen an ein **File**-Objekt gerichtet, das eine Datei repräsentiert:

```
System.out.println("Eigenschaften der Datei " + f.getName());
System.out.println("  Vollst. Pfad:       " + f.getAbsolutePath());
DateFormat df = DateFormat.getInstance();
String time = df.format(new Date(f.lastModified()));
System.out.println("  Letzte Änderung:    " + time);
System.out.println("  Größe in Bytes:     " + f.length());
System.out.println("  Schreiben möglich:  " + f.canWrite()+"\n");
```

Ausgabe:

```
Eigenschaften der Datei Ausgabe.dat
  Vollst. Pfad:       U:\Eigene Dateien\Java\FileDemo\AusDir\Ausgabe.dat
  Letzte Änderung:    08.02.20 23:26
  Größe in Bytes:     3
  Schreiben möglich:  true
```

Für die formatierte Ausgabe der **lastModified()** - Rückgabe unter Berücksichtigung der lokalen Zeitzone sorgen ein **Date**- und ein **DateFormat**-Objekt.

14.2.2.4 Atribute ändern

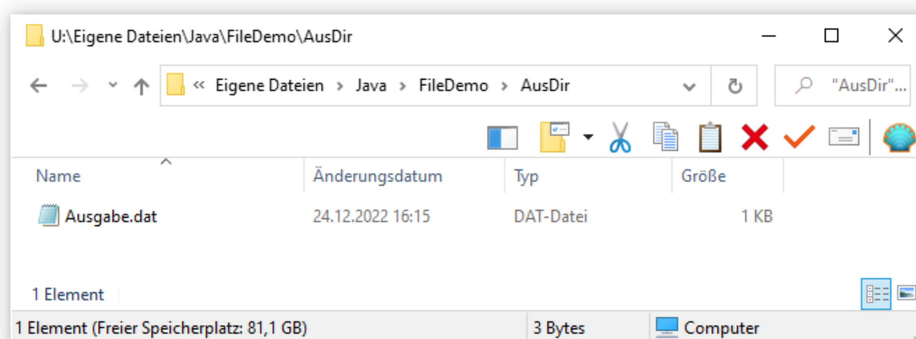
Man kann etliche Attribute von Dateien oder Ordnern ändern, z. B.:

- **boolean setLastModified(long** *time***)**
  Legt für eine Datei oder einen Ordner den Zeitpunkt der letzten Änderung neu fest
- **boolean setWritable(boolean** *writable***)**
  Setzt oder entfernt den Schreibschutz

Im folgenden Beispiel werden die Anforderungen an ein **File**-Objekt gerichtet, das eine Datei repräsentiert:

```
Date d = null;
DateFormat df = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
try {
   d = df.parse("24.12.2022 16:15:00");
} catch (Exception e) {
   System.err.println("Fehler bei der Datumsinterpretation");
   d = new Date();
}
f.setLastModified(d.getTime());
f.setWritable(false);
```

Windows hat nichts dagegen, das Änderungsdatum in die Zukunft zu verlegen:



14.2.2.5 Verzeichnisinhalte auflisten

Im folgenden Codefragment wird das **File**-Objekt `curDir` mit der Botschaft **listFiles()** beauftragt, für jeden Eintrag im aktuellen Verzeichnis ein Element im **File**-Array `files` anzulegen:

```
File curDir = new File(".");
File[] files = curDir.listFiles();
System.out.println("Einträge im aktuellen Verzeichnis:");
for (File fi : files)
   System.out.println(" " + fi.getName());
```

Anschließend werden die Datei- oder Verzeichnisnamen mit Hilfe der **File**-Methode **getName()** ausgegeben:

```
Dateisystemobjekte im akt. Verzeichnis:
 .idea
 Java 6.iml
 out
 src
```

Eine alternative **listFiles()** - Überladung liefert eine *gefilterte* Liste mit **File**-Verzeichniseinträgen, z. B.:

```
files = curDir.listFiles(new FNFilter("iml"));
System.out.println("\nEinträge im aktuellen Verzeichnis mit Extension .iml:");
for (File fi : files)
   System.out.println(" " + fi.getName());
```

Sie benötigt dazu ein Objekt aus einer Klasse, die das Interface **FilenameFilter** implementiert. Im Beispiel wird dazu die Klasse `FileFilter` definiert:

```
import java.io.*;

class FNFilter implements FilenameFilter {
  private String ext;

  public FileFilter(String ext_) {ext = ext_;}

  @Override
  public boolean accept(File dir, String name)  {
    return name.toLowerCase().endsWith("." + ext);
  }
}
```

Um den **FilenameFilter** - Interface-Vertrag zu erfüllen, muss `FNFilter` die Methode **accept()** implementieren. Im Beispiel resultiert die folgende Ausgabe:

```
Einträge im aktuellen Verzeichnis mit Extension .java:
 Java 6.iml
```

14.2.2.6 Umbenennen

Mit der **File**-Methode **renameTo()** lässt sich eine Datei oder ein Verzeichnis umbenennen, wobei als Parameter ein **File**-Objekt mit dem neuen Namen zu übergeben ist:

```
File fn = new File(ordner + "/Rausgabe.txt");
if (f.renameTo(fn))
   System.out.println("\nDatei " + f.getName() + " umbenannt in " + fn.getName());
else
   System.out.println("Fehler beim Umbenennen der Datei " + f.getName());
```

Beim Umbenennen wie beim anschließend zu beschreibenden Löschen einer Datei darf diese nicht geöffnet sein.

14.2.2.7 Löschen

Mit der **File**-Methode **delete()** löscht man eine Datei oder einen Ordner, z. B.:

```
if (fn.delete())
   System.out.println("Datei " + fn.getName() + " gelöscht");
else
   System.out.println("Fehler beim Löschen der Datei " + fn.getName());

if (dir.delete())
   System.out.println("Verzeichnis " + dir.getName() + " gelöscht");
else
   System.out.println("Fehler beim Löschen des Ordners " + dir.getName());
```

Damit ein Verzeichnis gelöscht werden kann, muss es leer sein.

14.3 Klassen zur Verarbeitung von Byte-Strömen

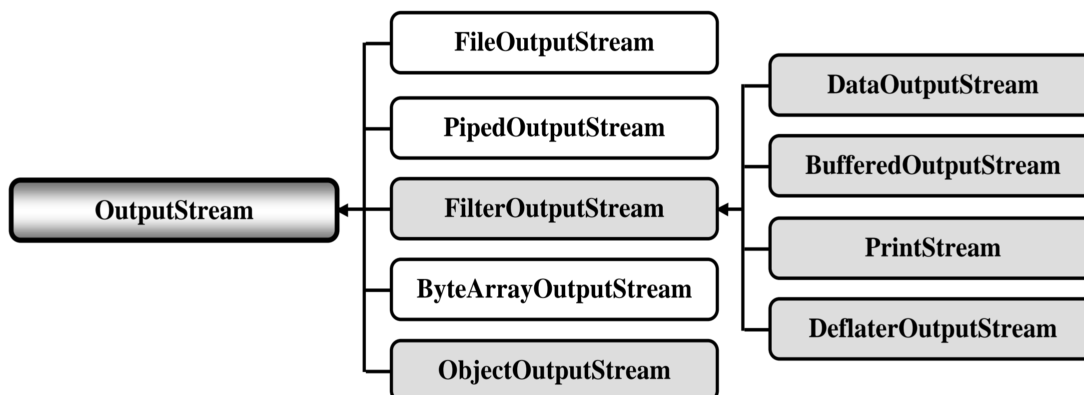
In Java 1.0 stammten *alle* Ein-/Ausgabeklassen von **InputStream** oder **OutputStream** ab. Diese Klassen haben sich zur Ein- bzw. Ausgabe von Bytes, primitiven Datenwerten und Objekten bewährt, aber bei der Behandlung von Unicode-Zeichen und vor allem beim Internationalisieren von Java-Software Probleme bereitet. Mit Java 1.1 wurden daher zur Verarbeitung von Textdaten die neuen Basisklassen **Reader** und **Writer** mit ihren Klassenhierarchien eingeführt. Für die Ein- bzw. Ausgabe von Bytes, primitiven Datenwerten und Objekten sind aber nach wie vor die von **InputStream** bzw. **OutputStream** abstammenden, Byte-orientierten Klassen adäquat.

An einigen Stellen haben alte Lösungen zur Zeichenverarbeitung überlebt, z. B. die von **OutputStream** (indirekt) abstammende und bei der Standard(fehler)ausgabe eines Java-Programms beteiligte Klasse **PrintStream** (siehe Abschnitt 14.3.1.6).

14.3.1 Die OutputStream-Hierarchie

14.3.1.1 Überblick

In der folgenden Abbildung sehen Sie den für uns relevanten Teil der Klassenhierarchie zur Basisklasse **OutputStream**, wobei die Ausgabeklassen (in direktem Kontakt mit einer Datensenke) mit einem weißen Hintergrund und die Ausgabe*transformations*klassen mit einem grauen Hintergrund dargestellt sind:



Im weiteren Verlauf des aktuellen Abschnitts werden wichtige Vertreter dieser Hierarchie behandelt.

Bei einigen Klassen erfolgt die Behandlung in späteren Abschnitten:

- Durch ein Tandem aus einem **PipedOutputStream** und einem verbundenen **PipedInputStream** lässt sich ein unidirektionaler Datentransfer zwischen zwei Threads (Ausführungsfäden) realisieren. Der erste Thread schreibt Bytes in den **PipedOutputStream** und der zweite Thread liest aus dem verbundenen **PipedInputStream**. Im Abschnitt 15.2.4.2 wird die Realisation einer Produzenten-Konsumenten - Kooperation mit der Pipeline-Technik beschrieben.
- Mit der Transformationsklasse **ObjectOutputStream** können komplette Objekte in einen Byte-orientierten Ausgabestrom geschrieben werden. Sie wird zusammen mit ihrem Gegenstück **ObjectInputStream** im Abschnitt 14.6 über die Objekt(de)serialisation behandelt.

Mit den folgenden Klassen werden wir uns im Manuskript *nicht* näher beschäftigen:

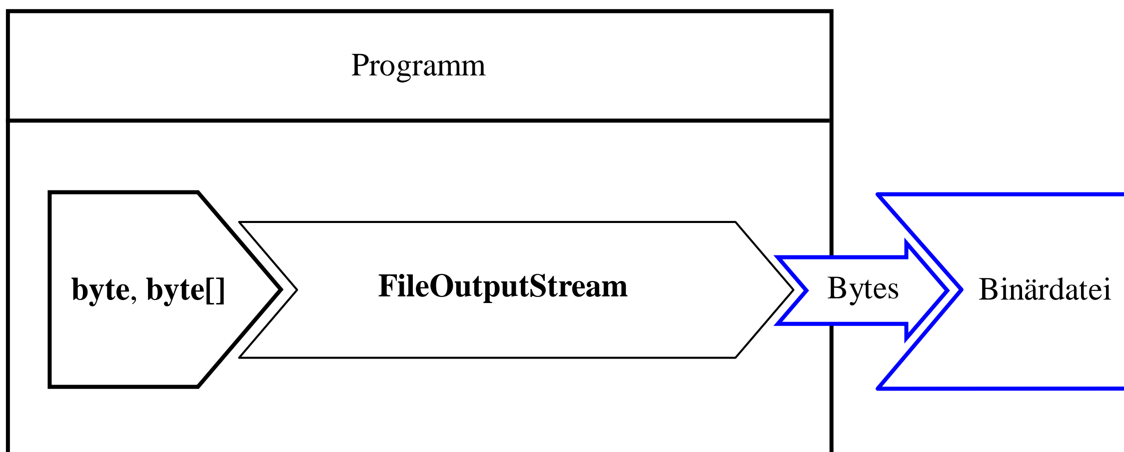
- Objekte der Ausgabeklasse **ByteArrayOutputStream** schreiben Bytes in einen **byte**-Array, also in eine programminterne Senke. Man kann sie z. B. zusammen mit dem Gegenstück **ByteArrayInputStream** dazu verwenden, um per Objekt(de)serialisation eine tiefe Objektkopie (inkl. referenzierter Member-Objekte) zu erstellen (siehe z. B. Krüger & Hansen 2014, S. 881ff).
- Objekte der Filterklasse **DeflaterOutputStream** aus dem Paket **java.util.zip** komprimieren einen Ausgabestrom. Mit der von **DeflaterOutputStream** abgeleiteten Klasse **ZipOutputStream** lassen sich neue Einträge im einem ZIP-Archiv erstellen (siehe z. B. Krüger & Hansen 2014, S. 425f).

Als Ausgabemethoden stehen zur Verfügung:

- **public void write(int** *b***)　throws IOException**
  Es wird *ein* Byte geschrieben, wobei von den 4 Bytes des Aktualparameters nur das niederwertigste Byte verwendet wird.
- **public void write(byte[]** *b***)　throws IOException**
  Der komplette **byte**-Array wird geschrieben.
- **public void write(byte[]** *b***, int** *off***, int** *len***)　throws IOException**
  Aus dem **byte**-Array werden *len* Bytes beginnend mit der Position *off* geschrieben.

14.3.1.2 FileOutputStream

Ein **FileOutputStream**-Objekt ist mit einer Datei verbunden, die vom Konstruktor im Schreibmodus geöffnet und nötigenfalls automatisch erstellt wird. Die in drei Überladungen vorhandene **write()** - Methode befördert die Inhalte von **byte**-Variablen oder -Arrays in die Ausgabedatei:



In den **FileOutputStream**-Konstruktoren wird die anzusprechende Datei über ein **File**-Objekt (siehe Abschnitt 14.2.2) oder über einen **String** spezifiziert:

- **public FileOutputStream(File** *file***)**
- **public FileOutputStream(File** *file***, boolean** *append***)**
- **public FileOutputStream(String** *name***)**
- **public FileOutputStream(String** *name***, boolean** *append***)**

Die Konstruktoren werfen eine geprüfte Ausnahme vom Typ **FileNotFoundException**, wenn …

- das im ersten Parameter angegebene Dateisystemobjekt ein *Ordner* ist,
- die Ausgabedatei vorhanden ist, aber nicht zum Schreiben geöffnet werden kann,
- das automatische Erstellen der nicht vorhandenen Ausgabedatei misslingt.

Mit dem *append*-Aktualparameterwert **true** sorgt man dafür, dass die Ausgaben bei einer vorhandenen Datei am Ende *angehängt* werden. Anderenfalls wird eine vorhandene Ausgabedatei überschrieben.

Soll ein **FileOutputStream** unter Verwendung einer per **Path**-Objekt identifizierten Datei instanziert werden, bietet sich die **Path**-Methode **toFile()** an, die zu einem **Path**-Objekt ein korrespondierendes **File**-Objekt liefert (siehe Abschnitt 14.2.1.1).

Weil **FileOutputStream**-Objekte nur **byte**-Variablen oder -Arrays befördern können (über die von OutputStream geerbten Methoden, siehe Abschnitt 14.3.1.1), werden sie oft mit Filterobjekten (z. B. aus der Klasse **DataOutputStream**) kombiniert, die flexiblere Ausgabemethoden bieten (siehe Abschnitt 14.3.1.4). Im folgenden Beispielprogramm ist diese Einschränkung jedoch irrelevant. Es demonstriert, welchen früher üblichen Aufwand beim Kopieren von Dateien man sich heute durch die Verwendung der **Files**-Methode **copy()** sparen kann (vgl. Abschnitt 14.2.1.9). Während das Programm nicht mehr als Muster für das Kopieren von Dateien taugt, demonstriert es doch die weiterhin relevante Verwendung eines **FileOutputStream**-Objekts zum Schreiben in eine Binärdatei. Außerdem wird auch gleich die Verwendung eines **FileInputStream**-Objekts zum Lesen aus einer Binärdatei vorgeführt (vgl. Abschnitt 14.3.2.2):

```
import java.io.*;

class FileCopy {
  final static String QUELLE = "quelle.dat", ZIEL = "ziel.dat";
  final static int BUFLEN = 1048576; // 1 Megabyte (1024*1024 Bytes) als Puffergröße

  public static void main(String[] args) throws IOException {
    byte[] buffer = new byte[BUFLEN];
    int nread;
    long zeit, total = 0;
    try (FileInputStream  fis = new FileInputStream(QUELLE);
         FileOutputStream fos = new FileOutputStream(ZIEL)) {
      zeit = System.currentTimeMillis();
      System.out.println("Kopieren von " + QUELLE + " in " + ZIEL + " gestartet:");
      for(int i = 1; ; i++) {
        nread = fis.read(buffer, 0, Math.min(BUFLEN, fis.available()));
        if (nread == 0)
          break;
        else {
          fos.write(buffer, 0, nread);
          total += nread;
          if (total >= BUFLEN) {
            String s = i + " Megabyte";
            for (int j = 0; j < s.length(); j++)
              System.out.print("\b");
            System.out.print(s);
          }
        }
      }
      zeit = System.currentTimeMillis() - zeit;
      System.out.println("\nEs wurden " + total + " Bytes kopiert. "+
          "(Benötigte Zeit: " + zeit + " Millisekunden.)");
    }
  }
}
```

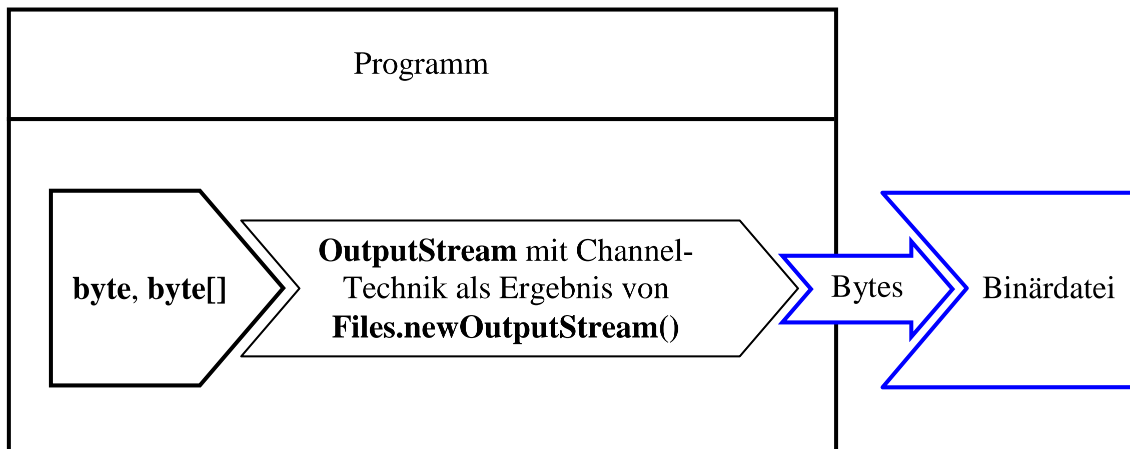
In der **for**-Schleife (ohne Terminierungsbedingung) wird mit der **FileInputStream**-Methode **read()** aus der Quelldatei jeweils ein Megabyte oder aber die per **available()** - Aufruf ermittelte Restmenge (vgl. Abschnitt 14.3.2.2) gelesen und anschließend von der **FileOutputStream**-Methode **write()** in die Zieldatei befördert. Per Rückgabewert informiert die **FileInputStream**-Methode **read()** darüber, wie viele Bytes tatsächlich gelesen wurden. Das Programm protokolliert den Arbeitsfortschritt, wobei durch die Ausgabe einer hinreichenden Zahl von Backspace-Steuerzeichen (**\b**) dafür

gesorgt wird, dass alle Fortschrittsmeldungen in derselben Konsolenzeile erscheinen.[1] Nach Abschluss des Kopiervorgangs werden die Transportleistung und die benötigte Zeit protokolliert, z. B.:

```
Kopieren von quelle.dat in ziel.dat gestartet:
1663 Megabyte
Es wurden 1743651037 Bytes kopiert. (Benötigte Zeit: 38228 Millisekunden.)
```

14.3.1.3 OutputStream mit Dateianschluss per NIO.2 - API

Von der Klasse **Files** im Paket **java.nio** erhält man über die statische Methode **newOutputStream()** einen **OutputStream**, der mit einer binären Ausgabedatei verbunden ist:



Man übergibt der Methode ein **Path**-Objekt mit dem Dateibezug und optionale Angaben zum Öffnungsmodus (vgl. Abschnitt 14.7.1):

public static OutputStream newOutputStream(Path *path*, **OpenOption...** *options*)
throws IOException

Wird kein **OpenOption**-Parameter angegeben, sind aus der Enumeration **StandardOpenOption** die folgenden Werte in Kraft: **CREATE**, **TRUNCATE_EXISTING** und **WRITE**. Folglich wird eine fehlende Datei erstellt und eine vorhandene Datei zunächst entleert.

Im Vergleich zu einem **FileOutputStream** - Objekt bestehen folgende Vorteile:

- Im Vergleich zum **FileOutputStream**-Konstruktor bietet die Methode **newOutputStream()** differenzierte Öffnungsoptionen für die Datei (siehe Abschnitt 14.7.1).
- Es kommt die Channel-Technik zum Einsatz. Eine Inspektion des API-Quellcodes (in den Klassen **Files**, **FileSystemProvider** und **Channels**) zeigt, dass im Hintergrund bei den Dateizugriffen die Channel-Technik zum Einsatz kommt, wobei ein Geschwindigkeitsvorteil möglich, aber nicht garantiert ist.[2] Bei manchen Anforderungsprofilen kann die Channel-Technik sogar ein schlechteres Leistungsverhalten zeigen als die traditionelle Datenstromtechnik.
- Die gleichzeitige Nutzung durch mehrere Threads (Ausführungsfäden, siehe Kapitel 15) ist erlaubt.

Ein Einsatzbeispiel für die Methode **newOutputStream()** war schon im Abschnitt 14.1.2 zu sehen.

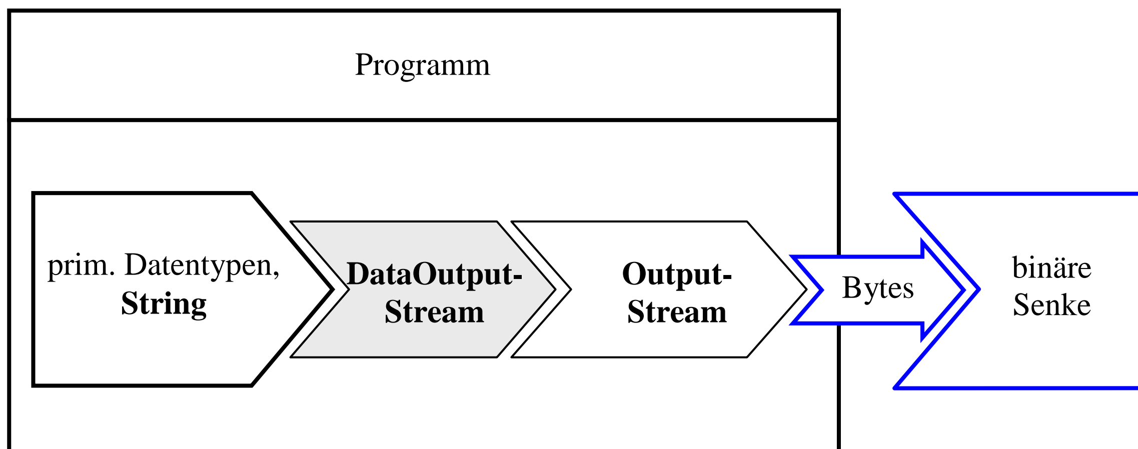
---

[1] Der Trick stammt von: https://stackoverflow.com/questions/7939802/how-can-i-print-to-the-same-line

[2] Mit **getClass()** befragt, liefert ein mit **Files.newOutputStream()** erzeugtes Objekt die Klasse **java.nio.channels.Channels$1** (OpenJDK 8 unter Windows 10).

14.3.1.4 DataOutputStream

Mit einem Objekt aus der Transformationsklasse **DataOutputStream** lassen sich die Werte primitiver Datentypen sowie **String**-Objekte über einen **OutputStream** in eine binäre Datensenke befördern:



Ein **DataOutputStream** beherrscht diverse Methoden zum Schreiben primitiver Datenwerte (z. B. **writeInt**(), **writeDouble**()). Mit **writeUTF()** steht auch eine Methode zur Ausgabe von Zeichen bereit, wobei eine *modifizierte* Variante der UTF-8 - Kodierung (vgl. Abschnitt 14.4.1.2) zum Einsatz kommt. Diese Methode ist angemessen, sofern die resultierenden Zeichen später mit der **DataInputStream**-Methode **readUTF()** wieder eingelesen werden sollen (vgl. Abschnitt 14.3.2.4). Für universell verwendbare Textdateien ist die Klasse **OutputStreamWriter** aus der **Writer**-Hierarchie mit einstellbarer und normkonformer Kodierung weit besser geeignet (siehe Abschnitt 14.4.1.2).

Im folgenden Beispielprogramm wird ein **DataOutputStream** auf einen **OutputStream** aufgesetzt und dann beauftragt, Daten vom Typ **int**, **double** und **String** zu schreiben. Das **OutputStream**-Objekt wird von der statischen **Files**-Methode **newOutputStream()** geliefert, die als Parameter ein **Path**-Objekt erhält, das eine Datei repräsentiert:

```java
import java.io.*;
import java.nio.file.*;

class DataOutputStreamDemo {
  public static void main(String[] args) throws IOException{
    Path file = Paths.get("demo.dat");

    try (DataOutputStream dos = new DataOutputStream(Files.newOutputStream(file))) {
      dos.writeInt(4711);
      dos.writeDouble(Math.PI);
      dos.writeUTF("DataOutputStream-Demo");
    }

    try (DataInputStream dis = new DataInputStream(Files.newInputStream(file))) {
      System.out.println("readInt() - Ergebnis:    " + dis.readInt() +
                         "\nreadDouble() - Ergebnis: " + dis.readDouble() +
                         "\nreadUTF() - Ergebnis:    " + dis.readUTF());
    }
  }
}
```

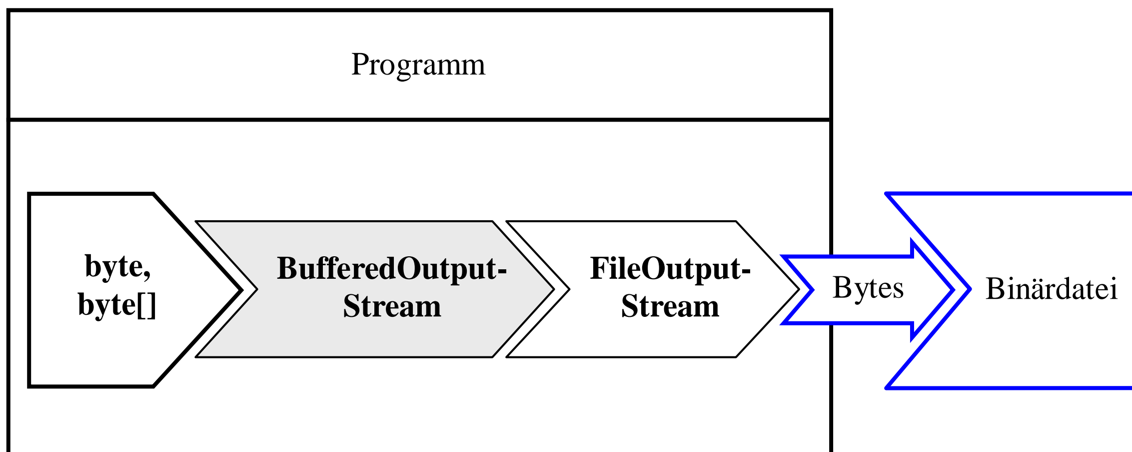
Ein **DataInputStream** holt in Kooperation mit einem **InputStream** die Werte zurück (vgl. Abschnitt 14.3.2):

```
readInt() - Ergebnis:    4711
readDouble() - Ergebnis: 3.141592653589793
readUTF() - Ergebnis:    DataOutputStream-Demo
```

14.3.1.5 BufferedOutputStream

Zur Beschleunigung von Ein- oder Ausgaben setzt man oft Transformationsklassen ein, die durch das Zwischenspeichern von Daten die Anzahl der (meist langsamen) Zugriffe auf Datenquellen oder -senken reduzieren. Diese Transformationsklassen kooperieren mit Ein- bzw. Ausgabeklassen, die in direktem Kontakt mit einer Datenquelle oder -senke stehen.

Ein **BufferedOutputStream**-Objekt nimmt Bytes entgegen und leitet diese in geeigneten Portionen an einen **OutputStream** weiter (z. B. an einen **FileOutputStream**):



Im **BufferedOutputStream**-Konstruktor ist obligatorisch ein **OutputStream**-Objekt zu übergeben (vgl. Abschnitt 0). Optional kann die voreingestellte Puffergröße von 8192 Bytes geändert werden:

- **public BufferedOutputStream**(**OutputStream** *out*)
- **public BufferedOutputStream**(**OutputStream** *out*, **int** *size*)

Zur Ausgabe vom **byte**-Werten und **byte**-Arrays stehen die **write()** - Überladungen der Basisklasse **OutputStream** zur Verfügung.

Das folgende Beispielprogramm schreibt in 10.000 **write()** - Aufrufen jeweils ein einzelnes Byte in eine Datei, zunächst ungepuffert, dann unter Verwendung eines **BufferedOutputStream**-Objekts:

```
import java.io.*;
import java.nio.file.*;

class BufferedOutputStreamFile {
   private final static int ANZAHL = 10_000;
   private static Path AUSGABE = Paths.get("Ausgabe.dat");

   public static void main(String[] args) throws IOException {
      long time;

      try (OutputStream fos = Files.newOutputStream(AUSGABE)) {
         time = System.currentTimeMillis();
         for (int i = 1; i <= ANZAHL; i++)
            fos.write(i);
         System.out.println("Zeit für die ungepufferte Ausgabe: " +
               (System.currentTimeMillis() - time));
      }
```

```
         try (BufferedOutputStream bos = new BufferedOutputStream(
               Files.newOutputStream(AUSGABE))) {
            time = System.currentTimeMillis();
            for (int i = 1; i <= ANZAHL; i++)
               bos.write(i);
            System.out.println("Zeit für die gepufferte Ausgabe:   " +
                  (System.currentTimeMillis() - time));
         }
      }
   }
```

Durch den Einsatz des **BufferedOutputStream**-Objekts (mit einer Puffergröße von 8192 Bytes) kann der Zeitaufwand beim Schreiben erheblich reduziert werden, was sich schon beim Schreiben auf eine lokale Festplatte zeigt (Angaben in Millisekunden):[1]

```
Zeit für die ungepufferte Ausgabe: 212
Zeit für die gepufferte Ausgabe:   2
```

Noch drastischer ist der Zeitgewinn durch die gepufferten Ausgabe, wenn sich die Ausgabedatei im Netzwerk befindet:[2]

```
Zeit für die ungepufferte Ausgabe: 436594
Zeit für die gepufferte Ausgabe:   45
```

Wegen des erheblichen Performanzvorteils sollte also ein Ausgabepuffer eingesetzt werden, wenn zahlreiche Schreibvorgänge mit jeweils kleinem Volumen stattfinden. Das `FileCopy`-Beispielprogramm im Abschnitt 14.3.1.2 wird hingegen *nicht* profitieren, weil das dortige **FileOutputStream**-Objekt nur sehr große Datenblöcke zur Ausgabe erhält.

Ein **BufferedOutputStream** muss unbedingt vor seinem Ableben (z. B. am Ende des Programms) per **flush()** entleert werden, weil sonst die zwischengelagerten Daten verfallen. Das Entleeren kann auch über die Methode **close()** erfolgen, die **flush()** aufruft und anschließend den zugrunde liegenden **OutputStream** schließt. Durch die im Beispiel verwendete **try-with-resources** - Technik wird der **close()** - Aufruf automatisiert (siehe Abschnitt 11.8.2).

Die von **Object** geerbte **finalize()** - Methode wird weder von **FilterOutputStream** noch von **BufferedOutputStream** überschreiben, sodass beim Terminieren eines **BufferedOutputStream**-Objekts per Garbage Collector *kein* **close()** - und insbesondere *kein* **flush()** - Aufruf erfolgt. Auf die **finalize()** - Methode sollte man sich ohnehin generell *nicht* verlassen, weil ihr Aufruf nicht garantiert ist (vgl. Abschnitt 14.1.5).

14.3.1.6 PrintStream

Die Transformationsklasse **PrintStream** dient dazu, Werte beliebigen Typs in einer für Menschen lesbaren Form auszugeben, z. B. auf der Konsole. Während ein **DataOutputStream** dazu dient, Variablen beliebigen Typs in eine *Binärdatei* zu schreiben, eignet sich ein **PrintStream** zur Ausgabe solcher Daten in eine *Textdatei*. Nach Abschnitt 14.1.3 sind bei der Zeichenstromverarbeitung allerdings die Klassen aus der später ins Java-API aufgenommenen **Writer**-Hierarchie zu bevorzugen. Diese haben bei der Textausgabe (Datentypen **String**, **char**) den Vorteil, dass der Java-intern verwendete Unicode in die bevorzugte Textkodierung umgesetzt werden kann (siehe Abschnitt 14.4.1.2). Trotzdem ist die Klasse **PrintStream** *nicht* überflüssig, weil z. B. der per **System.out**

---

1 Rechner mit Intel Core i3 550

2 Zur Netzwerkverbindung gehörte eine DSL-Strecke mit 10 MBit/s Upstream-Tempo.

ansprechbare Standardausgabestrom ein **PrintStream**-Objekt ist. Dies gilt auch für den Standard-*fehler*ausgabestrom, der über die Klassenvariable **System.err** ansprechbar ist.[1]

Ein **PrintStream**-Objekt kann mit Hilfe seiner vielfach überladenen Methoden **print**() und **println**() Daten von beliebigem Typ ausgeben, z. B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class PrintStreamConsole {
   public static void main(String[] args) {
      PrintStreamConsole wob = new PrintStreamConsole();
      System.out.println("Ein PrintStream kann Variablen " +
                "beliebigen Typs verarbeiten," +
                "\nz. B. die double-Zahl\n" + "   " + Math.PI +
                "\noder auch das Objekt\n" + "   " + wob);
   }
}
```

</td><td>

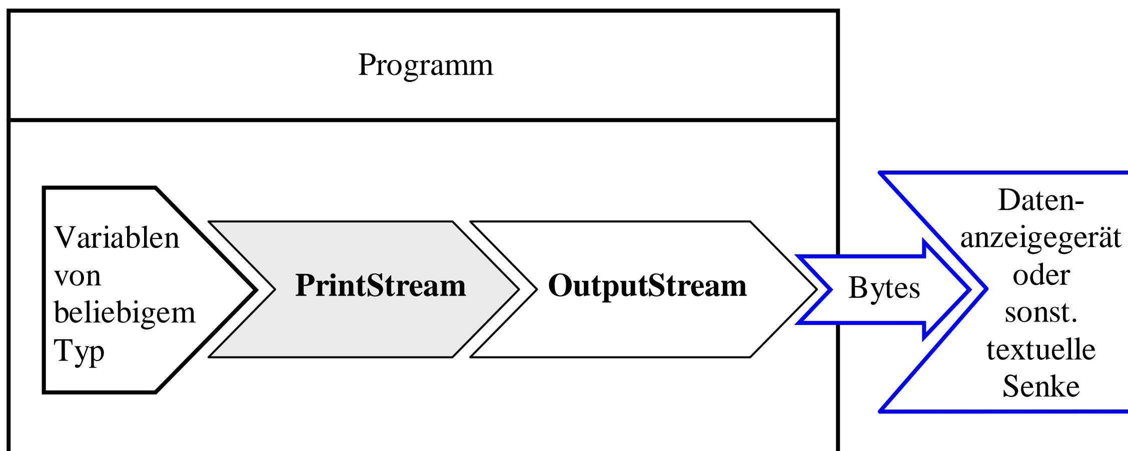
```
Ein PrintStream kann Variablen
bel. Typs verarbeiten,
z. B. die double-Zahl
   3.141592653589793
oder auch das Objekt
   PrintStreamConsole@16f0472
```

</td></tr>
</table>

Seit Java 5.0 (alias 1.5) ist auch die **PrintStream**-Methode **printf**() (alias **format**()) zur formatierten Ausgabe verfügbar, die schon im Abschnitt 3.2.2 vorgestellt wurde.

Im Unterschied zu den Methoden anderer **OutputStream**-Ableitungen werfen die **PrintStream**-Methoden *keine* **IOException**. Stattdessen setzen sie ein Fehlersignal, das mit **checkError**() abgefragt werden kann. Es wäre in der Tat sehr umständlich, jeden Aufruf der Methode **System.out.println**() in einen überwachten **try**-Block zu setzen.

Generell kann man die **PrintStream**-Arbeitsweise folgendermaßen darstellen:



Im nächsten Beispiel ist zu sehen, wie mit Hilfe der Transformationsklasse **PrintStream** Werte primitiver Datentypen in eine *Textdatei* geschrieben werden (über einen **FileOutputStream**):

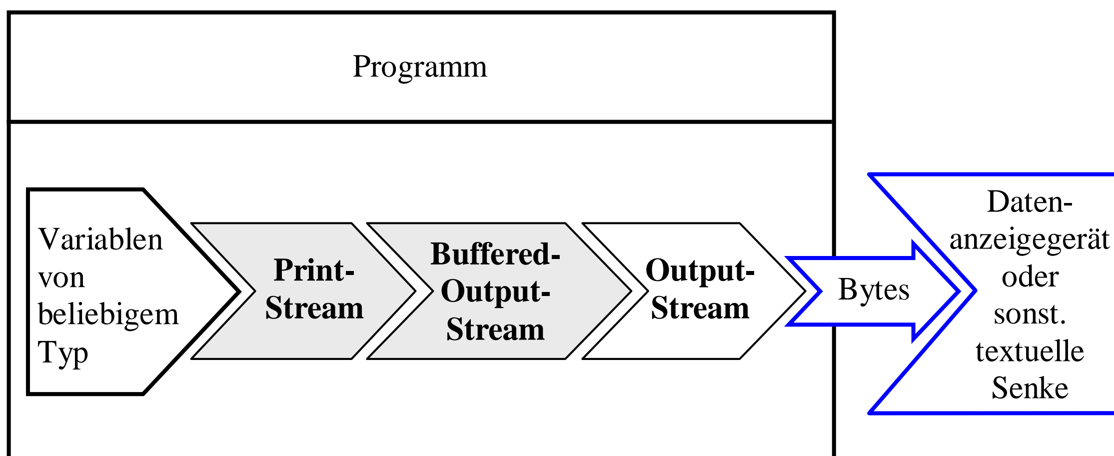
```
FileOutputStream fos = new FileOutputStream("ps.txt");
PrintStream ps = new PrintStream(fos);
ps.println(64798 + "   " + Math.PI);
```

In der Ausgabedatei **ps.txt** landen die Zeichenkettenrepräsentationen des **int**- und des **double**-Werts:

```
64798   3.141592653589793
```

Wenn ein **PrintStream**-Objekt zwecks Geschwindigkeitsoptimierung in einen **BufferedOutputStream** schreibt, sind zwei Transformationsobjekte nacheinander geschaltet:

[1] Wird z. B. ein Ausnahmeobjekt über die Methode **printStackTrace**() beauftragt, die Aufrufsequenz auszugeben, dann landet diese im Fehlerausgabestrom.



In dieser Situation müssen Sie unbedingt dafür sorgen, dass vor dem Terminieren des **PrintStream**-Objekts der Puffer geleert wird. Dazu stehen mehrere Möglichkeiten bereit:

- Aufruf der **PrintStream**-Methode **flush()**
  Dieser Aufruf wird an den angekoppelten **BufferedOutputStream** durchgereicht, wo die Pufferung stattfindet. Per **flush()** - Aufruf kann man jederzeit dafür sorgen, dass die Senke durch Entleeren des Zwischenspeichers auf den aktuellen Stand gebracht wird.
- Aufruf der **PrintStream**-Methode **close()**
  Dabei wird auch die **close()** - Methode des angekoppelten **BufferedOutputStream**-Objekts aufgerufen, die wiederum einen **flush()** - Aufruf enthält (siehe Abschnitt 14.3.1.5).
- Impliziter Aufruf der **PrintStream**-Methode **close()** durch Verwendung einer **try**-Anweisung mit automatischer Ressourcenfreigabe
- **PrintStream**-Konstruktor mit **autoFlush**-Parameter wählen und diesen auf **true** setzen
  Damit wird der Puffer in folgenden Situationen automatisch geleert:
  - nach dem Schreiben eines **byte**-Arrays
  - nach der Ausgabe eines Newline-Zeichens (**\n**)
  - nach der Ausführung einer **println()** - Methode

Weil die Klasse **PrintStream** die von **java.lang.Object** geerbte **finalize()** - Methode *nicht* überschreibt, findet bei der Beseitigung eines **PrintStream**-Objekts per Garbage Collector *kein* **close()** - oder **flush()** - Aufruf und damit keine Pufferentleerung statt.

Ein Beispiel für die Kombination aus einem **PrintStream** und einem **BufferedOutputStream** ist der per **System.out** ansprechbare Standardausgabestrom, der analog zum folgenden Codefragment initialisiert wird:

```
FileOutputStream fdout =
        new FileOutputStream(FileDescriptor.out);
BufferedOutputStream bos =
        new BufferedOutputStream(fdout, 128);
PrintStream ps =
        new PrintStream(bos, true);
System.setOut(ps);
```

Mit der statischen Variablen **out** der Klasse **FileDescriptor** wird der Bezug zur Konsole hergestellt. Im **PrintStream**-Konstruktor wird der **autoFlush**-Parameter auf den Wert **true** gesetzt. Über die **System**-Methode **setOut()** kann ein selbst entworfener Strom als Standardausgabe in Betrieb genommen werden.

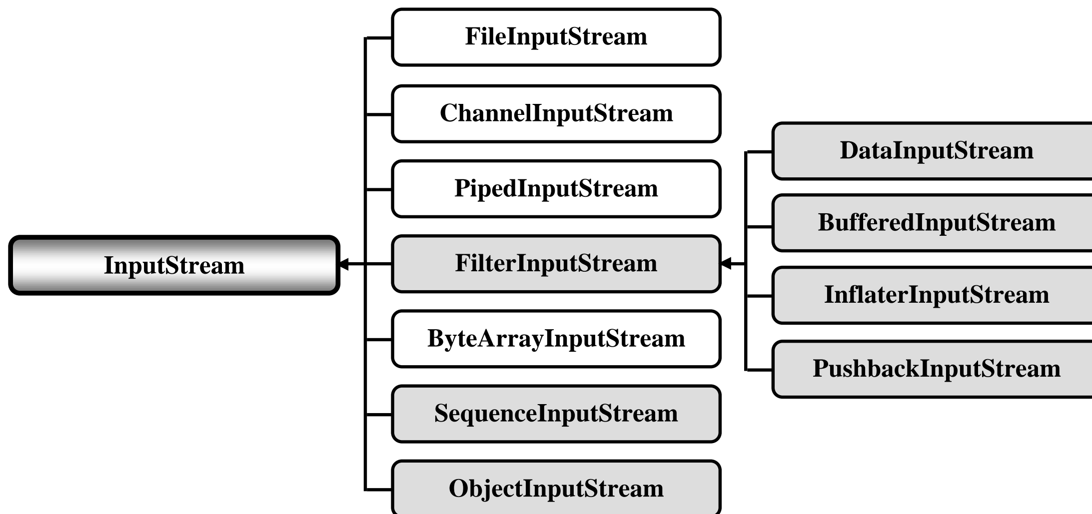
Bei der Textausgabe in eine Datei ist in der Regel die modernere und flexiblere Klasse **PrintWriter** zu bevorzugen (siehe Abschnitt 14.4.1.5). Vermutlich werden also der per **System.out** ansprechbare

Standardausgabestrom und der per **System.err** ansprechbare Standardfehlerausgabestrom die einzigen **PrintStream**-Objekte in Ihren Java-Programmen bleiben.

14.3.2 Die InputStream-Hierarchie

14.3.2.1 Überblick

Um Ihnen das Blättern zu ersparen, wird die schon im Abschnitt 14.1.3 gezeigte Abbildung zur **InputStream**-Hierarchie wiederholt (Eingabeklassen mit weißem Hintergrund und Eingabe*transformations*klassen mit grauem Hintergrund):



Im weiteren Verlauf des aktuellen Abschnitts werden wichtige Vertreter dieser Hierarchie behandelt.

Bei einigen Klassen erfolgt die Behandlung in späteren Abschnitten:

- Mit der Transformationsklasse **ObjectInputStream** werden wir uns im Abschnitt 14.6 näher beschäftigen. Sie ermöglicht das Einlesen kompletter Objekte aus einem Byte-Strom.
- Durch ein Tandem aus einem **PipedInputStream** und einem verbundenen **PipedOutputStream** lässt sich ein unidirektionaler Datentransfer zwischen zwei Threads (Ausführungsfäden) einrichten. Der erste Thread schreibt Bytes in den **PipedOutputStream** und der zweite Thread liest aus dem verbundenen **PipedInputStream**. Im Abschnitt 15.2.4.2 wird die Realisation einer Produzenten-Konsumenten - Kooperation mit der Pipeline-Technik beschrieben.

Die folgenden Klassen können nur kurz vorgestellt werden:

- **ByteArrayInputStream**
  Objekte dieser Eingabeklasse lesen Bytes aus einem **byte**-Array. Man kann sie z. B. zusammen mit dem Gegenstück **ByteArrayOutputStream** dazu verwenden, um per Objekt(de)serialisation eine tiefe Objektkopie (inkl. referenzierter Member-Objekte) zu erstellen (siehe z. B. Krüger & Hansen 2014, S. 881ff).
- **SequenceInputStream**
  Mit Hilfe dieser Transformationsklasse kann man eine geordnete Kollektion von **InputStream**-Objekten bilden und als einen einzigen Strom behandeln. Das Lesen startet mit dem ersten Eingabestrom. Ist das Ende eines Eingabestroms erreicht, wird er geschlossen, und der nächste Strom in der Sequenz wird geöffnet. Auf diese Weise kann man z. B. aus einer Serie von Dateien *einen* gemeinsamen Eingabestrom erstellen.

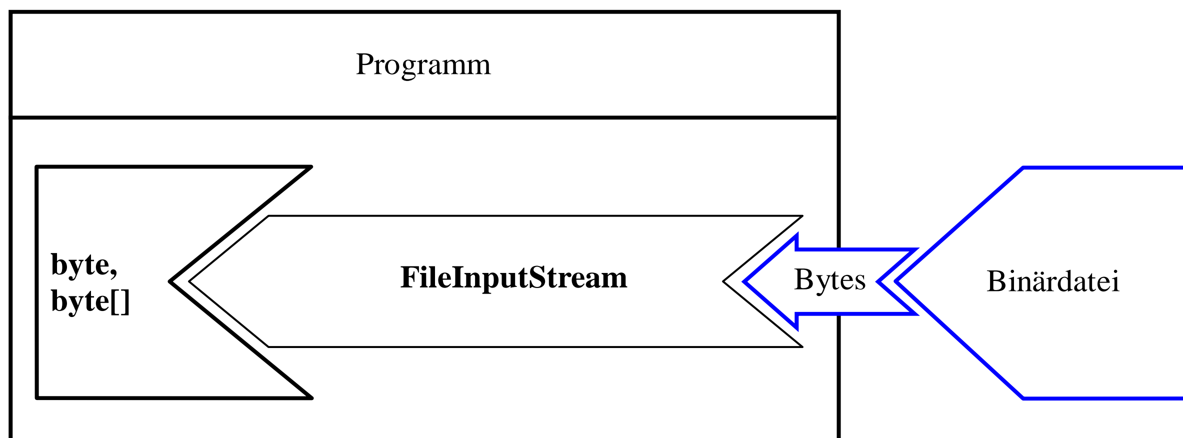
- **BufferedInputStream**
  Analog zum **BufferedOutputStream** (siehe Abschnitt 14.3.1) realisiert diese Eingabetransformationsklasse einen Zwischenspeicher, um das Lesen aus einem Eingabestrom zu beschleunigen.
- Objekte der Filterklasse **InflaterInputStream** aus dem Paket **java.util.zip** dekomprimieren einen Eingabestrom. Mit der von **InflaterInputStream** abgeleiteten Klasse **ZipInputStream**. kann man Dateien aus einem ZIP-Archiv lesen (siehe z. B. Krüger & Hansen 2014, S. 433f).
- **PushbackInputStream**
  Diese Transformationsklasse bietet Methoden, um aus einem Eingabestrom entnommene Bytes wieder zurückzustellen, was z. B. dann sinnvoll ist, wenn nach einer vorausschauenden Prüfung die eigentliche Verarbeitung durch ein anderes Stromobjekt erfolgen soll.

Ein **InputStream**-Objekt beherrscht die folgenden Methoden zum Lesen von Daten:

- **public int read() throws IOException**
  Als Rückgabewert (vom Typ **int**!) erhält man das nächste Byte (mögliche Werte von 0 bis 255) oder den Wert -1, falls das Dateiende erreicht ist.
- **public int read(byte[]** *b***) throws IOException**
  Diese Methode überträgt maximal **b.length** Bytes aus der Eingabedatei in den per Parameter angegebenen **byte**-Array. Als Rückgabewert erhält man die Anzahl der gelesenen Bytes oder -1, falls das Ende des Eingabestroms erreicht ist.
- **public int available() throws IOException**
  Laut API-Dokumentation schätzt **available()**, wie viele Bytes eine Methode aus dem Strom lesen kann, ohne auf Daten warten zu müssen.

14.3.2.2 FileInputStream

Mit einem **FileInputStream** kann man Bytes aus einer Datei lesen:



Ein Beispielprogramm mit **FileInputStream**-Beteiligung war schon im Abschnitt 14.3.1.2 (über den **FileOutputStream**) zu sehen.

Im **FileInputStream**-Konstruktor kann die anzusprechende Datei über ein **File**-Objekt (siehe Abschnitt 14.2.2) oder über einen **String** spezifiziert werden:

- **public FileInputStream(File** *file*)
- **public FileInputStream(String** *name***)**

Scheitert das Öffnen wegen einer nicht-existenten Datei oder aus einem anderen Grund, dann werfen die Konstruktoren eine geprüfte Ausnahme vom Typ **FileNotFoundException**.

Soll ein **FileInputStream** unter Verwendung einer per **Path**-Objekt identifizierten Datei instanziert werden, bietet sich die **Path**-Methode **toFile()** an, die zu einem **Path**-Objekt ein korrespondierendes **File**-Objekt liefert (siehe Abschnitt 14.2.1.1).

Im folgenden Codefragment werden die drei im Abschnitt 14.3.2.1 beschriebenen Methoden zum Lesen von Daten verwendet:

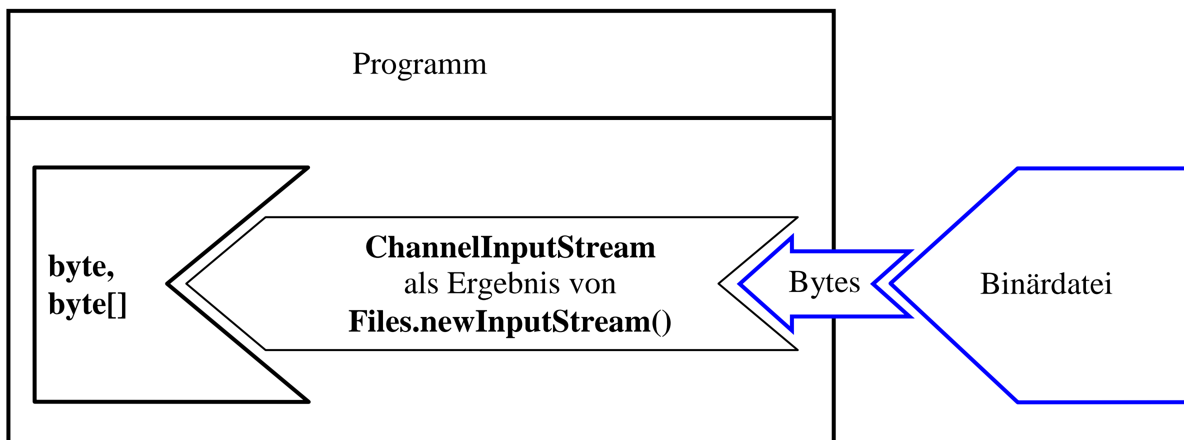
```
int anfang, gelesen;
byte[] rest = new byte[10];
try (FileInputStream fis = new FileInputStream("demo.bin")) {
    System.out.println("Verfügbar in Datei:    " + fis.available());
    anfang = fis.read();
    System.out.println("Verfügbar nach read(): " + fis.available());
    gelesen = fis.read(rest);
    System.out.print("Eingelesen:            " + anfang);
    for (int i = 0; i < gelesen; i++)
        System.out.print(rest[i]);
}
```

Mit einer binären Eingabedatei, die 8 Bytes mit den Werten 0 bis 7 enthält, kommt die folgende Ausgabe zustande:

```
Verfügbar in Datei:    8
Verfügbar nach read(): 7
Eingelesen:            01234567
```

14.3.2.3 InputStream mit Dateianschluss per NIO.2 - API

Von der Klasse **Files** im Paket **java.nio** erhält man über die statische Methode **newInputStream()** einen **InputStream**, der mit einer binären Eingabedatei verbunden ist:



Man übergibt der Methode ein **Path**-Objekt mit dem Dateibezug und optionale Angaben zum Öffnungsmodus (vgl. Abschnitt 14.7.1):

> **public static InputStream newInputStream(Path** *path*, **OpenOption...** *options*)
> **throws IOException**

Wird kein **OpenOption** - Parameter angegeben, ist der Enumerationswert **StandardOpenOption.READ** in Kraft.

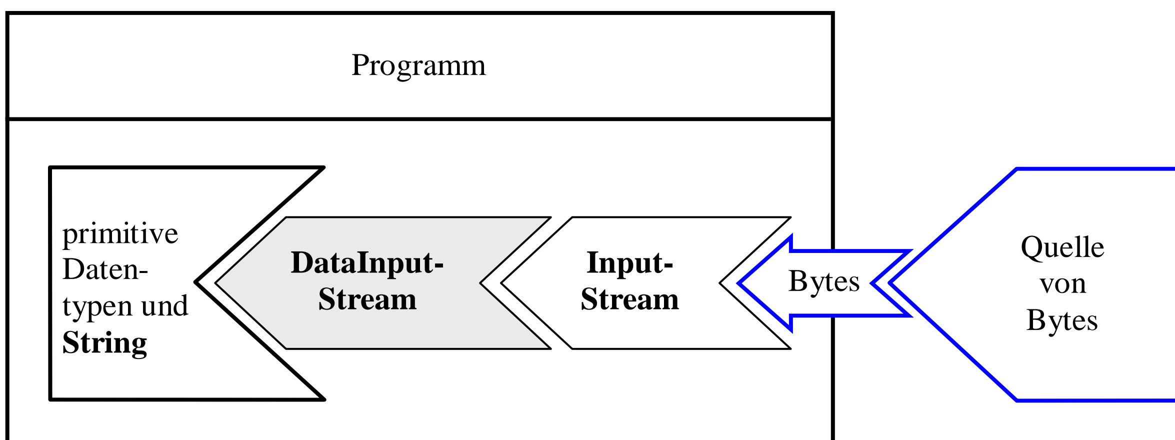
Im Vergleich zu einem **FileInputStream** - Objekt bestehen folgende Vorteile:

- Im Vergleich zum **FileInputStream**-Konstruktor bietet die Methode **newInputStream()** differenzierte Öffnungsoptionen für die Datei (siehe Abschnitt 14.7.1).
- Es kommt die Channel-Technik zum Einsatz. Über die Methode **getClass()** befragt, nennt das Rückgabeobjekt als seinen Typ die von **InputStream** abstammende Klasse **ChannelInputStream** im Paket **sun.nio.ch**. Im Hintergrund kommt bei den Dateizugriffen die Channel-Technik zum Einsatz, wobei ein Geschwindigkeitsvorteil möglich, aber nicht garantiert ist. Bei manchen Anforderungsprofilen kann die Channel-Technik sogar ein schlechteres Leistungsverhalten zeigen als die traditionelle Datenstromtechnik.
- Die gleichzeitige Nutzung durch mehrere Threads (Ausführungsfäden, siehe Kapitel 15) ist erlaubt.

Ein Einsatzbeispiel für die Methode **newInputStream()** war schon im Abschnitt 14.1.2 zu sehen.

14.3.2.4 DataInputStream

Die Transformationsklasse **DataInputStream** liest Werte mit einem primitiven Datentype sowie **String**-Objekte aus einem Bytestrom und ist uns zusammen mit ihrem Gegenstück **DataOutputStream** schon im Abschnitt 14.3.1.3 begegnet.



Im **DataInputStream**-Konstruktor ist der zugrunde liegende Eingabestrom anzugeben:

public DataInputStream(**InputStream** *in*)

Erläuterungen zu den diversen Lesemethoden (z. B. **readInt()**, **readDouble()**) finden Sie in der API-Dokumentation. Mit **readUTF()** steht auch eine Methode zum Lesen von Zeichen bereit, wobei eine *modifizierte* Variante der UTF-8 - Kodierung (vgl. Abschnitt 14.4.1.2) vorausgesetzt wird. Diese Methode ist nur dann angemessen, wenn die Zeichen mit der **DataOutputStream**-Methode **writeUTF()** geschrieben worden sind (vgl. Abschnitt 14.3.1.4).

14.4 Klassen zur Verarbeitung von Zeichenströmen

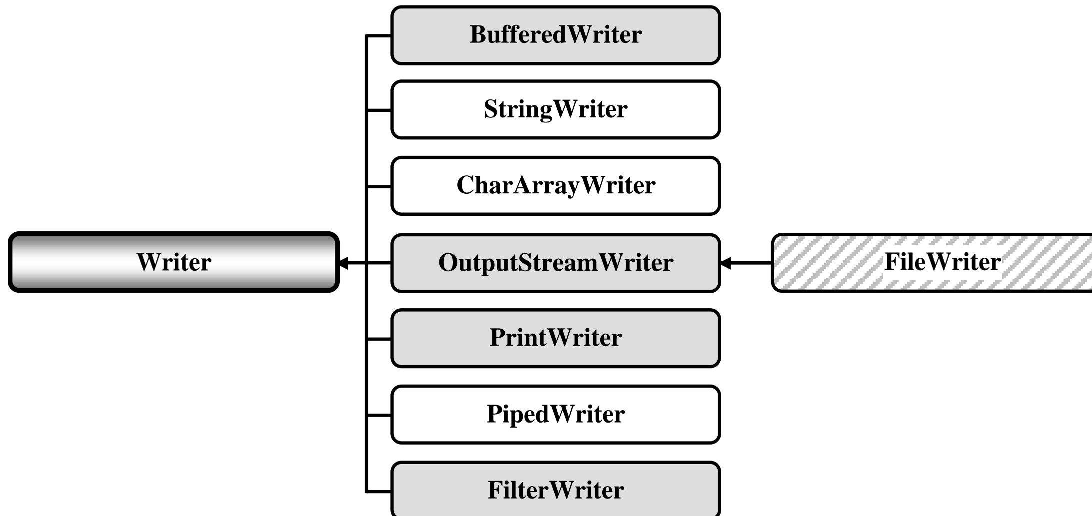
Java verwendet intern zur Repräsentation von Zeichen den Unicode, und die Klassen zur Verarbeitung von Zeichenströmen müssen beim Lesen und Schreiben zwischen der internen Repräsentation und der extern benötigten (z. B. der lokalüblichen) Zeichenkodierung vermitteln.

In diesem Abschnitt werden die mit Java 1.1 eingeführten Klassen zur Verarbeitung von Zeichenströmen behandelt, die von den beiden abstrakten Basisklassen **Writer** bzw. **Reader** abstammen. Sie sind bei der Verarbeitung von Textdaten vor allem wegen der unproblematischen Internationalisierung gegenüber den älteren Bytestromklassen **PrintStream** und **DataInputStream** zu bevorzugen.

14.4.1 Die Writer-Hierarchie

14.4.1.1 Überblick

In der folgenden Darstellung der **Writer**-Hierarchie sind Ausgabeklassen (in direktem Kontakt mit einer Senke) mit weißem Hintergrund dargestellt, Ausgabe*transformations*klassen mit grauem Hintergrund:



Weil die von **OutputStream**Writer abgeleitete Klasse **FileWriter** mit einer Datei verbunden ist *und* eine Transformationsfunktion besitzt, ist sie mit schraffiertem Hintergrund dargestellt.

Bei den folgenden **Writer**-Ableitungen beschränken wir uns auf kurze Hinweise:

- **StringWriter** und **CharArrayWriter**
  Ein **StringWriter** schreibt in einen dynamisch wachsenden **StringBuffer** (siehe Abschnitt 5.2.2). Im folgenden Beispiel werden die auszugebenden Zeichen von einem **PrintWriter** (siehe Abschnitt 14.4.1.5) geliefert:

Quellcode	Ausgabe

<pre>import java.io.*;
 	
class StringWriterDemo {
   public static void main(String[] args) {
      StringWriter sw = new StringWriter();
      PrintWriter pw = new PrintWriter(sw);
      for (int i = 1; i <= 5; i++)
         pw.println("Zeile " + i);
      System.out.println(sw.toString());
   }
}</pre>
	<pre>Zeile 1
	Zeile 2
	Zeile 3
	Zeile 4
	Zeile 5</pre>

  Ein **CharArrayWriter** schreibt in einen **char**-Array, der bei Bedarf mit Hilfe der statischen **Arrays**-Methode **copyOf()** durch ein größeres Exemplar ersetzt wird.

- **PipedWriter**
  Diese Klasse ist das zeichenorientierte Analogon zu Klasse **PipedOutputStream**.

- **FilterWriter**
  Diese abstrakte Basisklasse bietet sich dazu an, eigene Transformationsklassen für zeichenorientierte Ausgabeströme abzuleiten.

Zur Ausgabe stellt ein **Writer**-Objekt die folgenden **write()** - Überladungen zur Verfügung:

- **public void write(int** *c***)** **throws IOException**
  Die beiden niederwertigen Bytes des Parameters legen die Unicode-Nummer des auszugebenden Zeichens fest.
- **public void write(char[]** *cbuf***)** **throws IOException**
  Es wird ein Array mit Elementen vom Typ **char** komplett ausgegeben.
- **public void write(char[]** *cbuf***, int** *off***, int** *len***)** **throws IOException**
  Vom **char**-Array im ersten Parameter werden beginnend mit dem Zeichen an der Position *off* insgesamt *len* Zeichen ausgegeben.
- **public void write(String** *s***)** **throws IOException**
  Es wird ein **String** komplett ausgegeben.
- **public void write(String** *s***, int** *off***, int** *len***)** **throws IOException**
  Vom **String**-Objekt im ersten Parameter werden beginnend mit dem Zeichen an der Position *off* insgesamt *len* Zeichen ausgegeben.

14.4.1.2 Brückenklasse OutputStreamWriter

Die Klasse **OutputStreamWriter** überträgt die von Java intern verwendeten Unicode-Zeichen in einen Byte-Strom, wobei unterschiedliche Kodierungsschemata (engl. *encodings*) zum Einsatz kommen können. Weil ein **OutputStreamWriter** einen Zeichenstrom in einen Bytestrom überführt, spricht die API-Dokumentation hier von einer *Brückenklasse*.

Es folgt eine kleine Auswahl der insgesamt 40 in Java 13 unterstützten Kodierungen (*Basic Encoding Set*, enthalten im Modul **java.base**):[1]

<table>
<tr><th>Name für java.nio</th><th>Name für java.io und java.lang</th><th>Beschreibung</th></tr>
<tr><td>US-ASCII</td><td>ASCII</td><td>American Standard Code for Information Interchange
Es werden nur 7 Bit verwendet. Bei den Unicode-Zeichen \u0000 bis \u007F wird das niederwertige Byte ausgegeben, ansonsten ein Fragezeichen (0x3F).</td></tr>
<tr><td>ISO-8859-1</td><td>ISO8859_1</td><td>Erweiterter ASCII-Code (ISO Latin-1)
Bei den Unicode-Zeichen \u0000 bis \u00FF wird das niederwertige Byte ausgegeben, ansonsten ein Fragezeichen (0x3F).</td></tr>
<tr><td>UTF-8</td><td>UTF8</td><td>Bei diesem Schema werden die Unicode-Zeichen durch eine variable Anzahl von Bytes kodiert. So können alle Unicode-Zeichen ausgegeben werden, ohne die platzverschwenderische Anhäufung von Null-Bytes bei den ASCII-Zeichen in Kauf nehmen zu müssen:
<table>
<tr><th colspan="2">Unicode-Zeichen</th><th rowspan="2">Anzahl Bytes</th></tr>
<tr><th>von</th><th>bis</th></tr>
<tr><td>\u0000</td><td>\u007F</td><td>1</td></tr>
<tr><td>\u0080</td><td>\u07FF</td><td>2</td></tr>
<tr><td>\u0800</td><td>\uFFFF</td><td>3</td></tr>
</table>
Bei den ersten 128 Unicode-Zeichen liefern die Kodierungen US-ASCII, ISO-8859-1 und UTF-8 identische Ergebnisse.</td></tr>
<tr><td>UTF-16BE</td><td>UnicodeBigUnmarked</td><td>Für alle Unicode-Zeichen werden 16 Bit in Big-Endian - Reihenfolge ausgegeben: Das höherwertige Byte zuerst.
In Java ist diese Reihenfolge voreingestellt (auch bei anderen Datentypen). Beim großen griechischen Delta (\u0394) wird z. B.: ausgegeben: 03 94</td></tr>
</table>

[1] Die Angaben stammen von der Webseite
https://docs.oracle.com/en/java/javase/13/intl/supported-encodings.html
Dort werden noch weitere unterstützte Kodierungen aufgelistet.

Name für **java.nio**	Name für **java.io** und **java.lang**	Beschreibung
UTF-16LE	**UnicodeLittleUnmarked**	Für alle Unicode-Zeichen werden 16 Bit in *Little-Endian* - Reihenfolge ausgegeben: Das niederwertige Byte zuerst. Beim großen griechischen Delta (\u0394) wird z. B.: ausgegeben: 94 03
Windows-1252	**Cp1252**	Windows Latin-1 (ANSI) Im Unterschied zu ISO-8859-1 werden die Codes von 0x80 bis 0x9F (in ISO-8859-1 reserviert für Steuerzeichen) mit „höheren" Unicode-Zeichen belegt. Z. B. wird das Eurozeichen (Unicode: \u20AC) auf den Code 0x80 abgebildet.
IBM850	**Cp850**	MS-DOS Latin-1 MS-DOS-Codepage zur Verwendung in Westeuropa

Dass die Klassen aus dem Paket **java.io** (z. B. **OutputStreamWriter**) für die Kodierungen andere Namen benutzen als die Klassen aus den Paketen **java.nio.*** (z. B. **Files**), ist bedauerlich, aber nicht tragisch.

Bei den folgenden Überladungen des **OutputStreamWriter**-Konstruktors kann die gewünschte Kodierung über ihren Namen oder über ein Objekt der Klasse **Charset** (aus dem Paket **java.nio.charset**) angegeben werden:

- **public OutputStreamWriter(OutputStream** *out*, **String** *charsetName*)
  throws UnsupportedEncodingException
- **public OutputStreamWriter(OutputStream** *out*, **Charset** *cs*)

Wird im **OutputStreamWriter**-Konstruktor *keine* Kodierung angegeben,

public OutputStreamWriter(OutputStream *out*)

dann entscheidet die Systemeigenschaft **file.encoding**. Ihr Wert kann mit der **System**-Methode **getProperty()** ermittelt werden, z. B.:[1]

Quellcodesegment	Ausgabe
`String denc = System.getProperty("file.encoding");` `System.out.println(denc);`	`UTF-8`

Die voreingestellte Kodierung wird beim Start der JVM festgelegt und ist damit vom Programm nicht zu beeinflussen.[2] In der Regel wird per Voreinstellung die Kodierung UTF-8 verwendet.

Zur Ausgabe von Textdaten stehen die von der Basisklasse **Writer** geerbten (und teilweise überschriebenen) **write()** - Überladungen zur Verfügung.

Im folgenden Programm werden die oben beschriebenen Kodierungen nacheinander dazu verwendet, um einen kurzen Text mit dem Umlaut „ä" (\u00E4) in eine Datei zu schreiben. Dabei kommt die von der statischen **System**-Methode **lineSeparator()** gelieferte Plattform-spezifische Zeilenschaltung zum Einsatz:

[1] Beobachtet unter Windows 10 mit Java 9

[2] https://stackoverflow.com/questions/361975/setting-the-default-java-character-encoding

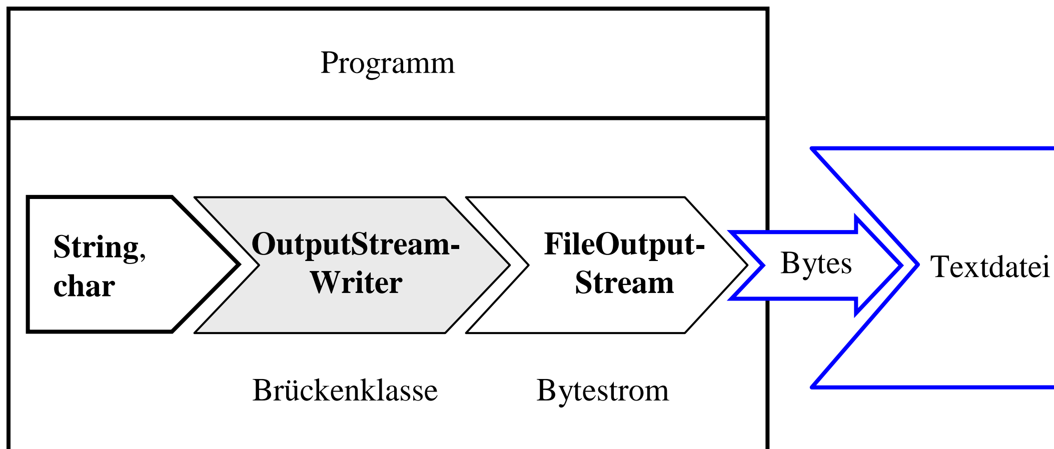
```
import java.io.*;
import java.nio.charset.Charset;
import java.nio.file.*;

class OutputStreamWriterDemo {
  public static void main(String[] args) throws IOException {
    String[] encodings = {"US-ASCII", "ISO-8859-1", "UTF-8",
                          "UTF-16BE", "UTF-16LE", "Windows-1252", "IBM850"};
    Files.deleteIfExists(Paths.get("test.txt"));
    for (int i = 0; i < encodings.length; i++) {
      try (OutputStreamWriter osw = new OutputStreamWriter(
             new FileOutputStream("test.txt", true), Charset.forName(encodings[i]))) {
        osw.write(encodings[i] + " ae = ä" + System.lineSeparator());
      }
    }
  }
}
```

Im **FileOutputStream**-Konstruktor hat der Parameter **append** den Wert **true** erhalten, damit die Ausgaben den bisherigen Dateiinhalt nicht ersetzen, sondern erweitern.

Für das Unicode-Zeichen \u00E4 wird jeweils ausgegeben:

Kodierung (Name für **java.nio**)	Byte(s) in der Ausgabe (Hexadezimal)
US-ASCII	3F
ISO-8859-1	E4
UTF-8	C3 A4
UTF-16BE	00 E4
UTF-16LE	E4 00
Windows-1252	E4
IBM850	84

Das Beispielprogramm arbeitet mit folgender Datenstromarchitektur:



Datenstromobjekte aus der Klasse **OutputStreamWriter** (und auch aus der Ableitung **FileWriter**, siehe Abschnitt 14.4.1.3) sammeln die per Unicode-Wandlung entstandenen Bytes zunächst in einem internen Puffer (Größe: 8192 Bytes), den ein Objekt der Klasse **StreamEncoder** aus dem Paket **sun.nio.cs** verwaltet.[1] Daher muss auf jeden Fall vor dem Ableben eines **OutputStreamWriter**-Objekts (z. B. beim Programmende) der Puffer geleert werden. Dazu stehen mehrere Möglichkeiten bereit:

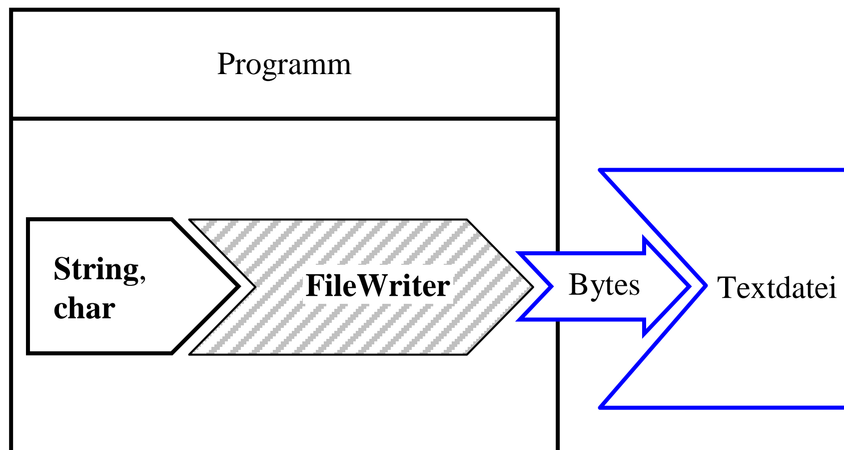
[1] Die technischen Details wurden aus dem OpenJDK 13 - Quellcode ermittelt.

- Aufruf der Methode **flush()**
  Dieser Aufruf wird an den eingebauten **StreamEncoder** durchgereicht.
- Aufruf der Methode **close()**
  Sie sorgt dafür, dass der Puffer des eingebauten **StreamEncoder**s vor dem Schließen geleert wird.
- Impliziter Aufruf der Methode **close()** durch Verwendung einer **try**-Anweisung mit automatischer Ressourcen-Freigabe

Weil die Klassen **OutputStreamWriter** und **FileWriter** die von **java.lang.Object** geerbte **finalize()** - Methode *nicht* überschreiben, findet bei der Beseitigung eines Objekts aus diesen Klassen per Garbage Collector kein **close()** - bzw. **flush()** - Aufruf und somit keine Pufferentleerung statt.

14.4.1.3 FileWriter

Die von **OutputStreamWriter** abgeleitete Klasse **FileWriter** im Paket **java.io** eignet sich zur Ausgabe von **String**- und **char**-Variablen in eine Textdatei, wenn die voreingestellte Kodierung (siehe Abschnitt 14.4.1.2) akzeptabel ist:



Die Klasse **FileWriter** wird mit einem schraffierten Hintergrund dargestellt (siehe auch die **Writer**-Hierarchie im Abschnitt 14.4.1.1), weil ihre Objekte …

- mit einer Datei in Kontakt stehen (über ein Instanzobjekt aus der Klasse **FileOutputStream**), sodass sie als Ausgabestrom arbeiten können,
- den eingehenden Zeichenstrom in einen Bytestrom transformieren, sodass sie als Filterobjekte bezeichnet werden können.

Wichtiger als die akademische Bemerkung zur korrekten Klassifikation der Klasse **FileWriter** sind ihre Konstruktoren. Das per **String**- oder **File**-Objekt bestimmte Ausgabeziel wird zum Schreiben geöffnet, wobei der optionale zweite Parameter darüber entscheidet, ob ein vorhandener Dateianfang erhalten bleibt:

- **public FileWriter(File** *file*) **throws IOException**
- **public FileWriter(File** *file*, **boolean** *append*) **throws IOException**
- **public FileWriter(String** *fileName*) **throws IOException**
- **public FileWriter(String** *fileName*, **boolean** *append*) **throws IOException**

Soll ein **FileWriter** unter Verwendung einer per **Path**-Objekt identifizierten Datei instanziert werden, bietet sich die **Path**-Methode **toFile()** an, die zu einem **Path**-Objekt ein korrespondierendes **File**-Objekt liefert (siehe Abschnitt 14.2.1.1).

Zur Ausgabe von Textdaten stehen die von **Writer** bzw. **OutputStreamWriter** geerbten **write()** - Überladungen zur Verfügung. Das folgende Programm schreibt ein einzelnes Zeichen und eine Zei-

chenfolge jeweils in eine eigene Zeile, und trennt die beiden Zeilen durch die Plattform-spezifische Zeilenschaltung:

```
import java.io.*;

class FileWriterDemo {
   public static void main(String[] args) throws IOException {
      try (FileWriter fw = new FileWriter("fw.txt")) {
         fw.write("ä");
         fw.write(System.lineSeparator() + "Zeile 2");
      }
   }
}
```

Ist das voreingestellte Kodierungsschema ungeeignet, muss man auf die **FileWriter**-Bequemlichkeit verzichten, stattdessen einen **OutputStreamWriter** mit passender Kodierung erstellen und einen **OutputStream** dahinter setzen (siehe Abschnitt 14.4.1.2).

14.4.1.4 BufferedWriter

Am Ende von Abschnitt 14.4.1.2 über die Klasse **OutputStreamWriter** wurde die implizite Speicherung von Bytes beschrieben, die aus der Wandlung von Unicode-Zeichen entstehen. Für die *explizite* Pufferung von *Zeichen* steht in der **Writer**-Hierarchie die Transformationsklasse **BufferedWriter** mit den folgenden Konstruktor-Überladungen zur Verfügung:

- **public BufferedWriter(Writer *out*)**
- **public BufferedWriter(Writer *out*, int *bufferSize*)**

In der zweiten Überladung kann die voreingestellte Puffergröße von 8192 Zeichen verändert werden.

Abweichend vom Aufbau der **OutputStream**-Hierarchie ist **BufferedWriter** übrigens *nicht* von **FilterWriter** abgeleitet.

Vom folgenden Programm werden mit Hilfe eines **FileWriter**s (siehe Abschnitt 14.4.1.3) zweimal jeweils 20.000.000 Zeilen in eine Textdatei geschrieben, zunächst ohne und dann mit zwischengeschaltetem **BufferedWriter**. Weil die erste Ausgabe unabhängig von der verwendeten Technik stets länger dauert, findet zunächst ein „Warmlaufen“ statt:

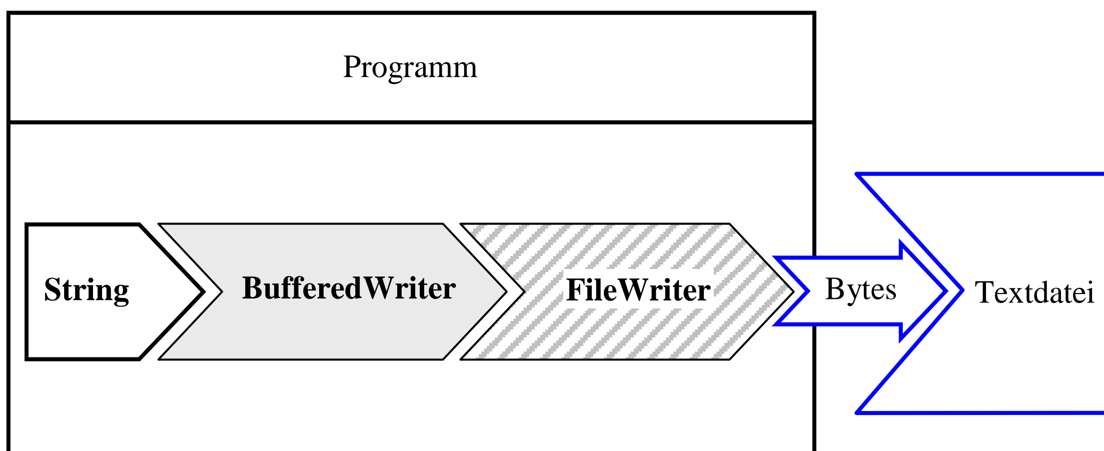
```
import java.io.*;
import java.nio.file.*;

class BufferedWriterDemo {
   final static int ANZAHL = 20_000_000;

   static void ohneBufferedWriter() throws IOException {
      for (int i = 0; i < 2; i++) {
         try (FileWriter fw = new FileWriter("test.txt")) {
            long time = System.currentTimeMillis();
            for (int j = 1; j <= ANZAHL; j++)
               fw.write("Zeile " + i + System.lineSeparator());
            if (i == 1)
               System.out.println("Benötigte Zeit ohne BufferedWriter:       "
                  + (System.currentTimeMillis() - time));
         }
         Files.delete(Paths.get("test.txt"));
      }
   }
```

```
    static void mitBufferedWriter() throws IOException {
       for (int i = 0; i < 2; i++) {
          try (BufferedWriter bw = new BufferedWriter(new FileWriter("test.txt"))) {
             long time = System.currentTimeMillis();
             for (int j = 1; j <= ANZAHL; j++)
                bw.write("Zeile " + i + System.lineSeparator());
             if (i == 1)
                System.out.println("Benötigte Zeit mit BufferedWriter:        "
                   + (System.currentTimeMillis() - time));
          }
          Files.delete(Paths.get("test.txt"));
       }
    }

    public static void main(String[] args) throws IOException {
       ohneBufferedWriter();
       mitBufferedWriter();
    }
 }
```

Dass der Zeitgewinn durch den **BufferedWriter** relativ bescheiden ausfällt, liegt vermutlich an der automatischen Byte-Pufferung durch den **FileWriter**:[1]

```
Benötigte Zeit ohne BufferedWriter:        3064
Benötigte Zeit mit BufferedWriter:         2081
```

Die etwas performantere Datenstrom-Pipeline hat den folgenden Aufbau:



14.4.1.5 PrintWriter

Die Transformationsklasse **PrintWriter** besitzt diverse **print()** - bzw. **println()** - Überladungen, um Variablen beliebigen Typs in Textform auszugeben. Sie wurde mit Java 1.1 als Nachfolger bzw. Ergänzung der älteren Klasse **PrintStream** eingeführt, die aber zumindest im Standardausgabestrom **System.out** und im Standardfehlerausgabestrom **System.err** weiterlebt (vgl. Abschnitt 14.3.1.6). Seit der Java-Version 5.0 (alias 1.5) beherrschen **PrintWriter**-Objekte auch die funktionsgleichen Methoden **printf()** und **format()** zur *formatierten* Ausgabe. Elementare Formatierungsoptionen wurde schon im Abschnitt 3.2.2 erläutert.

Bei Problemen mit dem Ausgabestrom oder mit der Formatierung werfen die **PrintWriter**-Methoden *keine* **IOException**, sondern setzen ein Fehlersignal, das mit der Methode **checkError**() abgefragt werden kann.

Wie die folgende Auswahl der **PrintWriter**-Konstruktoren zeigt, dürfen die angekoppelten Datenstromobjekte von den Basisklassen **OutputStream** oder **Writer** abstammen:

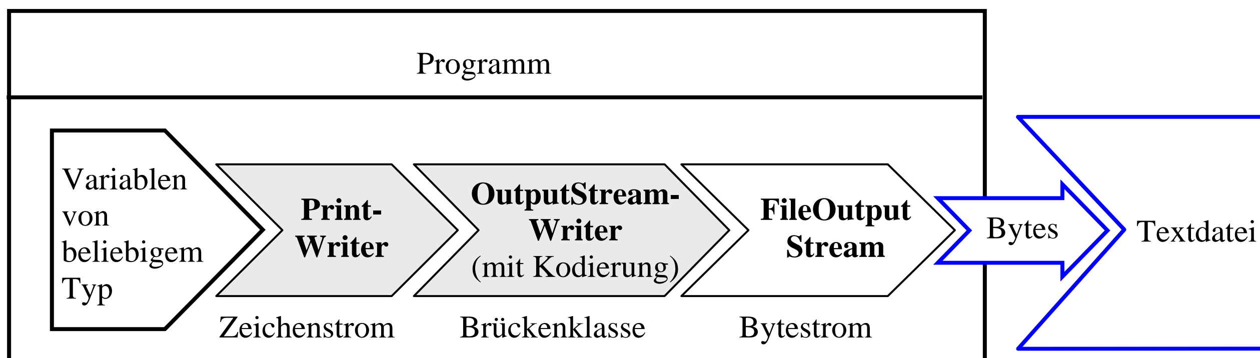
---

[1] Die Zeiten stammen von einem Rechner mit der Intel-CPU Core i3 (3,2 GHz).

- **public PrintWriter(OutputStream** *out*)
- **public PrintWriter(OutputStream** *out*, **boolean** *autoFlush*)
- **public PrintWriter(Writer** *out*)
- **public PrintWriter(Writer** *out*, **boolean** *autoFlush*)

Letztlich übergibt ein **PrintWriter** alle Ausgabedaten als Unicode-Zeichen an einen **OutputStreamWriter**, der die Zeichen in Abhängigkeit von einer Kodierung in Byte-Sequenzen übersetzt. Diese Bytes werden auf dem Weg zur Datensenke zwischengespeichert (vgl. Abschnitt 14.4.1.2). Erhält der **autoFlush**-Parameter eines **PrintWriter**-Konstruktors den Wert **true**, dann wird der Puffer bei jedem Aufruf der **PrintWriter**-Methoden **println()**, **printf()** oder **format()** entleert. Dies ist bei einer Konsolenausgabe sinnvoll, sollte aber bei einer Dateiausgabe aus Performanzgründen vermieden werden.

Gibt man im **PrintWriter**-Konstruktor *explizit* einen **OutputStreamWriter** an, kann man die Kontrolle über die Kodierung übernehmen:



Diese Möglichkeit stellt den entscheidenden Vorteil der Klasse **PrintWriter** gegenüber dem Vorgänger **PrintStream** dar (vgl. Abschnitt 14.4.1.2). Bei der Ausgabe von numerischen Daten in eine Textdatei spielt die Wahlfreiheit bei der Kodierung allerdings keine große Rolle, weil die hier beteiligten Zeichen von allen Kodierungen im Wesentlichen identisch in Bytes übersetzt werden. Insgesamt ist die Klasse **PrintWriter** bei der Ausgabe in Textdateien zu bevorzugen, weil sich damit alle Aufgaben gut bewältigen lassen.

Wird ein **PrintWriter** auf einen Ausgabestrom aus der **OutputStream**-Hierarchie gesetzt, dann nimmt der Konstruktor insgeheim einen **BufferedWriter**, der Zeichen zwischenspeichert (siehe Abschnitt 14.4.1.4), und einen **OutputStreamWriter** mit der voreingestellten Kodierung, der Bytes zwischenspeichert (siehe Abschnitt 14.4.1.2), in Betrieb. Die folgenden **PrintWriter**-Konstruktoren stammen aus dem OpenJDK 13 - Quellcode:

```
public PrintWriter(OutputStream out) {
   this(out, false);
}

public PrintWriter(OutputStream out, boolean autoFlush) {
   this(out, autoFlush, Charset.defaultCharset());
}

public PrintWriter(OutputStream out, boolean autoFlush, Charset charset) {
   this(new BufferedWriter(new OutputStreamWriter(out, charset)), autoFlush);
    . . .
}
```

Damit arbeitet insgesamt das folgende Gespann:

Vor dem Terminieren eines **PrintWriter**-Objekts müssen die Zwischenspeicher unbedingt entleert werden, was ab Java 7 am besten mit einer try-with-resources - Anweisung geschieht:

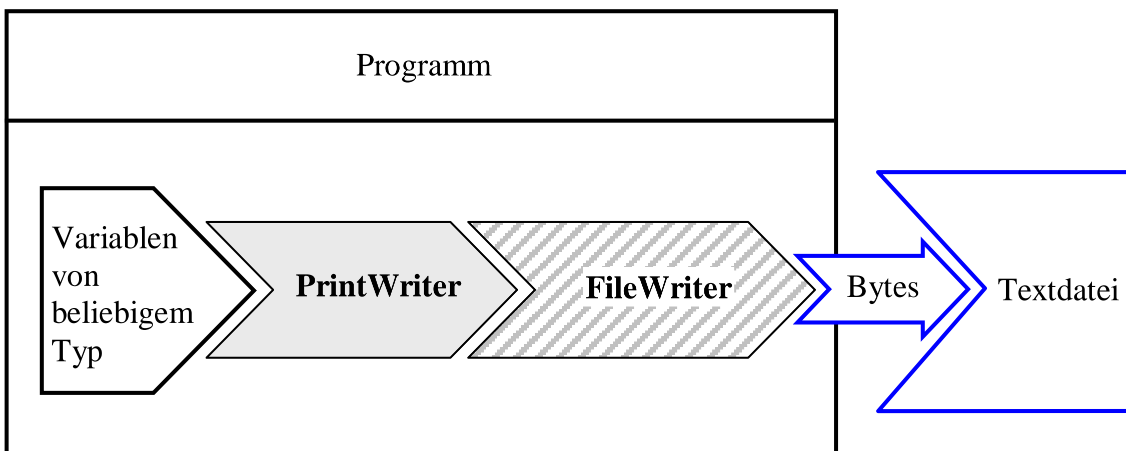
```
import java.io.*;
class PrintWriterDemo {
   public static void main(String[] args) throws IOException {
      try (PrintWriter pw = new PrintWriter(new FileOutputStream("pw.txt"))) {
         for (int i = 1; i <= 3000; i++)
            pw.println(i);
      }
   }
}
```

Ein **PrintWriter** reicht einen (impliziten oder expliziten) **close()** - Aufruf an die zugrunde liegenden Datenströme weiter, sodass im Beispiel auch die Dateiverbindung geschlossen wird.

In der folgenden Programmvariante unterbleibt die Pufferentleerung mit dem Effekt, dass in der Ausgabedatei ca. 1500 Zeilen fehlen:

```
import java.io.*;
class PrintWriterDemo {
   public static void main(String[] args) throws IOException {
      PrintWriter pw = new PrintWriter(new FileOutputStream("pw.txt"));
         for (int i = 1; i <= 3000; i++)
            pw.println(i);
   }
}
```

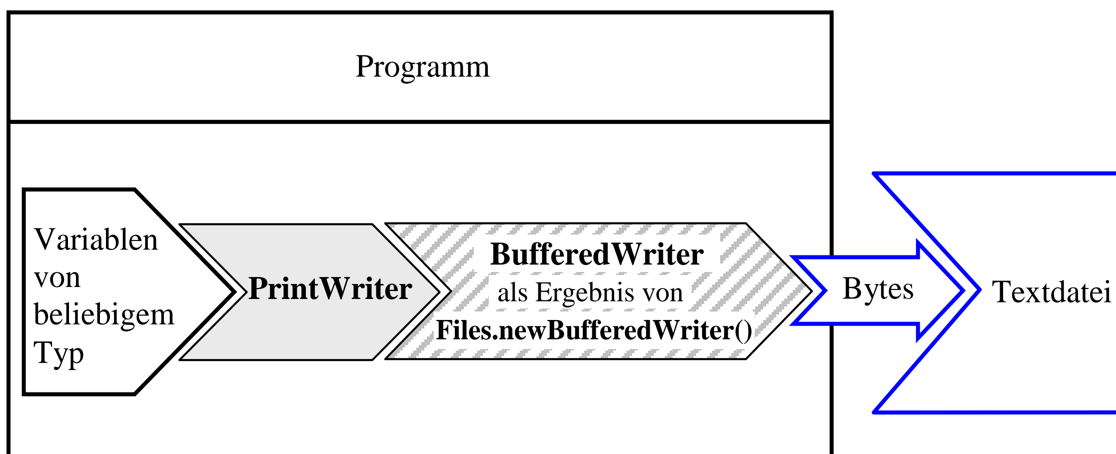
Wenn die voreingestellte Kodierung (siehe Abschnitt 14.4.1.2) akzeptabel ist, taugt auch ein **FileWriter** (siehe Abschnitt 14.4.1.3) als Verbindungsstück zwischen einem **PrintWriter** und einer Textdatei:



Während bei der **PrintWriter-OutputStream** - Konstruktion im Hintergrund ein **BufferedWriter** und ein **OutputStreamWriter** mit Pufferung beteiligt sind (siehe oben), puffert bei der **PrintWriter-FileWriter** - Konstruktion nur die **OutputStreamWriter**-Ableitung **FileWriter**. Wegen des beschränkten Effekts einer zusätzlichen **BufferedWriter**-Beteiligung spielt das keine große Rolle (siehe Abschnitt 14.4.1.4).

14.4.1.6 BufferedWriter mit Dateianschluss per NIO.2 - API

Wer beim gepufferten Schreiben von Zeichen die Verbindung zur Textdatei über das NIO.2 - API (vgl. Abschnitt 14.2.1) herstellen möchte, kann bei der Klasse **Files** im Paket **java.nio** über die statischen Methode **newBufferedWriter()** einen **BufferedWriter** anfordern, der mit einer Ausgabedatei verbunden ist:



Man übergibt der Methode ein **Path**-Objekt mit dem Dateibezug, ein **Charset**-Objekt zur Wahl der Kodierung und optionale Angaben zum Öffnungsmodus (vgl. Abschnitt 14.7.1):

public static BufferedWriter newBufferedWriter(Path *path*, **Charset** *cs*,
OpenOption... *options*)
throws IOExecption

Wird kein **OpenOption** - Parameter angegeben, sind aus der Enumeration **StandardOpenOption** die folgenden Werte in Kraft: **CREATE**, **TRUNCATE_EXISTING** und **WRITE**. Folglich wird eine fehlende Datei erstellt und eine vorhandene Datei zunächst entleert.

Wie ein Blick in den API-Quellcode der Klasse **Files** in Java 13 zeigt,

```
public static BufferedWriter newBufferedWriter(Path path, Charset cs,
                                               OpenOption... options)
        throws IOException {
    CharsetEncoder encoder = cs.newEncoder();
    Writer writer = new OutputStreamWriter(newOutputStream(path, options), encoder);
    return new BufferedWriter(writer);
}
```

erhält man einen **BufferedWriter**, der seinen Zeichenstrom an einen **OutputStreamWriter** mit der gewünschten Kodierung weitergibt, wobei dieses Brückenklassenobjekt mit einem **OutputStream** verbunden ist, den die **Files**-Methode **newOutputStream()** zum **Path**-Objekt mit den gewünschten Öffnungseinstellungen liefert.

Im Vergleich zu einem traditionell durch Konstruktoraufrufe erzeugten Gespann aus **BufferedWriter** und **FileWriter** bestehen folgende Vorteile:

- Vereinfachung der Syntax, weil zwei verschachtelte Konstruktoraufrufe durch einen Methodenaufruf ersetzt werden.
- Per **Charset**-Objekt kann eine Kodierung gewählt werden.
- Es lassen sich detaillierte Öffnungsoptionen für die Datei angeben (siehe Abschnitt 14.7.1).
- Über das im Hintergrund per **newOutputStream()** erzeugte Ausgabeobjekt kommt die Channel-Technik zum Einsatz, wobei ein Geschwindigkeitsvorteil möglich, aber nicht garantiert ist. Bei manchen Anforderungsprofilen kann die Channel-Technik sogar ein schlechteres Leistungsverhalten zeigen als die traditionelle Datenstromtechnik.

Zur Ausgabe in eine Textdatei mit UTF-8 - Kodierung eignet sich die folgende Überladung ohne **Charset**-Parameter:

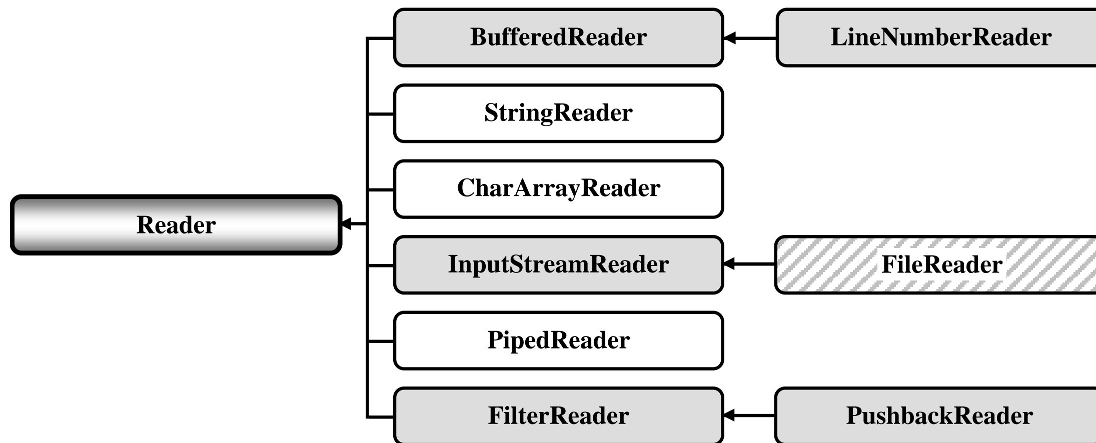
public static BufferedWriter newBufferedWriter(Path *path*, **OpenOption...** *options*)
throws IOException

14.4.2 Die Reader-Hierarchie

In der **Reader**-Hierarchie finden sich diverse Klassen zur Verarbeitung von zeichenorientierten *Eingabe*strömen.

14.4.2.1 Überblick

In der folgenden Abbildung sind Eingabeklassen (in direktem Kontakt mit einer Datenquelle) mit weißem Hintergrund dargestellt, Eingabe*transformations*klassen mit grauem Hintergrund:



Weil die Klasse **FileReader** mit einer Datei verbunden ist *und* eine Transformationsfunktion besitzt, ist sie mit schraffiertem Hintergrund dargestellt.

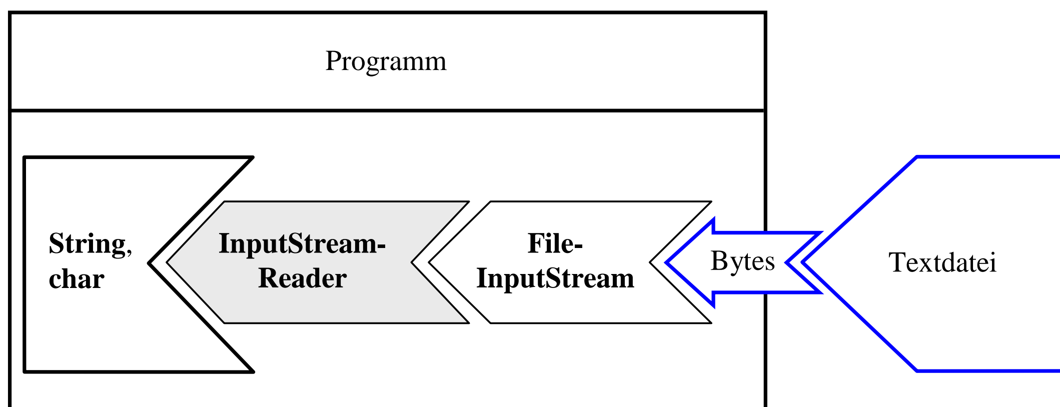
Bei den meisten **Reader**-Unterklassen beschränken wir uns auf kurze Hinweise:

- **LineNumberReader**
  Dieser gepufferte Zeicheneingabestrom erweitert seine Basisklasse **BufferedReader** um Methoden zur Verwaltung von Zeilennummern.
- **StringReader** und **CharArrayReader**
  Objekte dieser Klassen lesen aus einem **String** bzw. aus einem **char**-Array.
- **PipedReader**
  Objekte dieser Klasse lesen Zeichen aus einer Pipeline, die zur Kommunikation zwischen Threads dient.
- **FilterReader**
  Diese abstrakte Basisklasse bietet sich dazu an, eigene Transformationsklassen für zeichenbasierte Eingabeströme abzuleiten.
- **PushbackReader**
  Diese Klasse bietet Methoden, um die aus einem Eingabestrom entnommenen Zeichen wieder zurückzustellen, was z. B. dann sinnvoll ist, wenn nach einer vorausschauenden Prüfung die eigentliche Verarbeitung durch ein anderes Stromobjekt erfolgen soll.

Wer eine Möglichkeit zum komfortablen Einlesen von *numerischen* Daten aus Textdateien sucht, sollte sich im Abschnitt 14.5 die (unmittelbar aus **java.lang.Object** abgeleitete) Klasse **Scanner** ansehen.

14.4.2.2 Brückenklasse InputStreamReader

Die Brückenklasse **InputStreamReader** ist das Gegenstück zur Klasse **OutputStreamReader**, wandelt also Bytes unter Verwendung einer einstellbaren Kodierung (vgl. Abschnitt 14.4.1.2) in Unicode-Zeichen, sodass z. B. beim Lesen aus einer Textdatei die folgende Verarbeitungskette entsteht:



Wie beim **OutputStreamReader** findet zur Beschleunigung der Konvertierung automatisch eine Pufferung des Byte-Stroms statt.

14.4.2.3 FileReader und BufferedReader

Die Klasse **FileReader** ist das Gegenstück zur Klasse **FileWriter**. Sie wird mit einem schraffierten Hintergrund dargestellt (siehe z. B. **Reader**-Hierarchie im Abschnitt 14.4.2.1), weil ihre Objekte …

- mit einer Datei in Kontakt stehen (über ein Instanzobjekt aus der Klasse **FileInputStream**), sodass sie als Eingabestrom arbeiten können,
- als **InputStreamReader**-Abkömmlinge den eingehenden Byte-Strom in einen Zeichenstrom transformieren, sodass sie als Filterobjekte bezeichnet werden können.

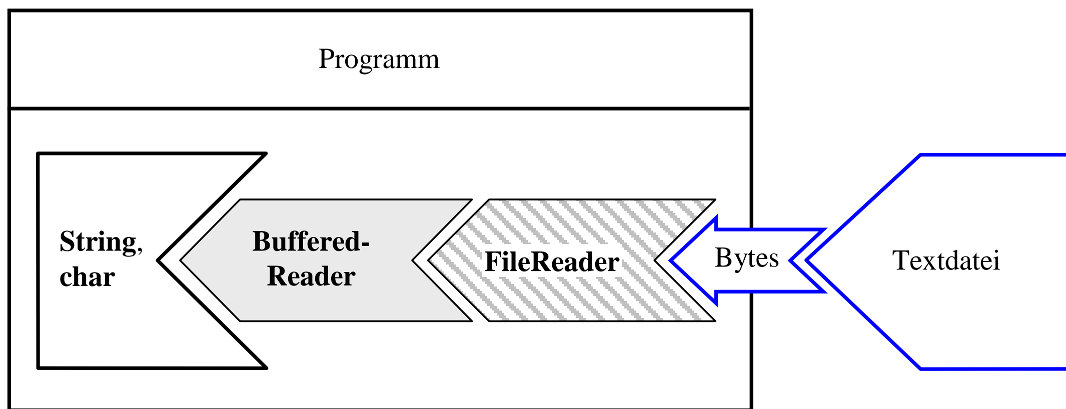
Bei der Wandlung von Bytes in Unicode-Zeichen kommt die voreingestellte Kodierung zum Einsatz (siehe Abschnitt 14.4.1.2). Wer eine alternative Kodierung benötigt, muss auf die **FileReader**-Bequemlichkeit verzichten, einen **InputStreamReader** mit passender Kodierung erstellen und mit einem **InputStream** kombinieren.

In der Regel setzt man vor den **FileReader** noch einen **BufferedReader**, der für eine Beschleunigung sorgt und außerdem die bei Dateien mit Zeilenstruktur sehr nützliche Methode **readLine()** zum Einlesen einer kompletten Zeile bietet, z. B.:

Quellcode	Ausgabe
(siehe Code unten)	`Zeile 100`

```
import java.io.*;
import java.util.*;

class BufferedFileReader {
  public static void main(String[] args) throws IOException {
    List<String> als = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader("fr.txt"))) {
      String line;
      while (true) {
        line = br.readLine();
        if (line != null)
          als.add(line);
        else
          break;
      }
      System.out.println(als.get(als.size()-1));
    }
  }
}
```

Das Beispielprogramm arbeitet mit der folgenden Datenstrom-Architektur:

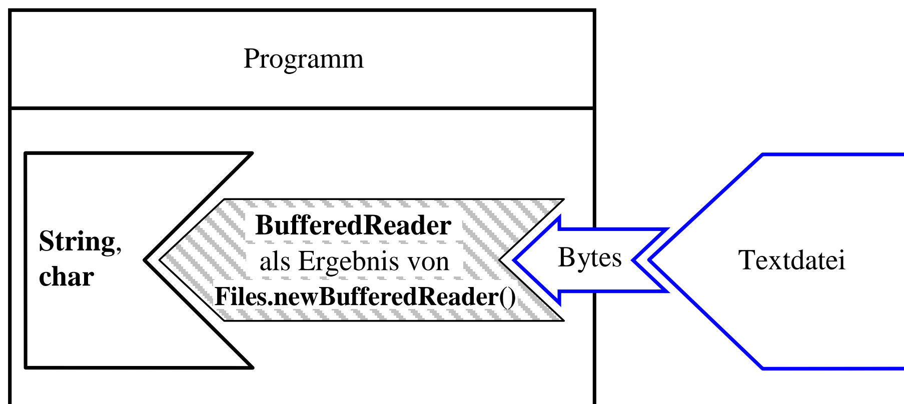


Die bei einem **BufferedReader** voreingestellte Puffergröße von 8192 Zeichen lässt sich per Konstruktorparameter ändern.

Soll ein **FileReader** unter Verwendung einer per **Path**-Objekt identifizierten Datei instanziert werden, bietet sich die **Path**-Methode **toFile()** an, die zu einem **Path**-Objekt ein korrespondierendes **File**-Objekt liefert (siehe Abschnitt 14.2.1.1).

14.4.2.4 BufferedReader mit Dateianschluss per NIO.2 - API

Wer beim gepufferten Lesen von Zeichen die Verbindung zur Textdatei über das NIO.2 - API (vgl. Abschnitt 14.2.1) herstellen möchte, kann bei der Klasse **Files** im Paket **java.nio** über die statische Methode **newBufferedReader()** einen **BufferedReader** anfordern, der mit einer Eingabedatei verbunden ist:



Man übergibt der Methode ein **Path**-Objekt mit dem Dateibezug und ein **Charset**-Objekt zur Wahl der Kodierung:

public static BufferedReader newBufferedReader(Path *path*, **Charset** *cs*)
throws IOException

Wie ein Blick in den OpenJDK 13 - Quellcode zeigt,

```
public static BufferedReader newBufferedReader(Path path, Charset cs)
        throws IOException {
    CharsetDecoder decoder = cs.newDecoder();
    Reader reader = new InputStreamReader(newInputStream(path), decoder);
    return new BufferedReader(reader);
}
```

erhält man einen **BufferedReader**, der seinen Zeichenstrom von einem **InputStreamReader** mit der gewünschten Kodierung bezieht, wobei dieses Brückenklassenobjekt mit einem **InputStream** verbunden ist, den die **Files**-Methode **newInputStream()** zum **Path**-Objekt liefert.

Im Vergleich zu einem traditionell durch Konstruktoraufrufe erzeugten Gespann aus **BufferedReader** und **FileReader** (vgl. Abschnitt 14.4.2.3) bestehen folgende Vorteile:

- Vereinfachung der Syntax, weil zwei verschachtelte Konstruktoraufrufe durch einen Methodenaufruf ersetzt werden.
- Per **Charset**-Objekt kann eine Kodierung gewählt werden.
- Über das im Hintergrund per **newInputStream()** erzeugte Eingabeobjekt kommt die Channel-Technik zum Einsatz, wobei ein Geschwindigkeitsvorteil möglich, aber nicht garantiert ist. Bei manchen Anforderungsprofilen kann die Channel-Technik sogar ein schlechteres Leistungsverhalten zeigen als die traditionelle Datenstromtechnik.

Für eine Textdatei mit UTF-8 - Kodierung eignet sich die folgende **newBufferedReader()** - Überladung ohne **Charset**-Parameter:

public static BufferedReader newBufferedReader(Path *path***)**
throws IOException

Das Beispielprogramm aus Abschnitt 14.4.2.3 lässt sich mit Hilfe der **Files**-Methode **newBufferedReader()** etwas vereinfachen:

```
import java.io.*;
import java.nio.file.*;
import java.util.*;

class BufferedReaderNio2 {
   public static void main(String[] args) throws IOException {
      List<String> als = new ArrayList<>();
      try (BufferedReader br = Files.newBufferedReader(Paths.get("inp.txt"))) {
         String line;
         while (true) {
            line = br.readLine();
            if (line != null)
               als.add(line);
            else
               break;
         }
         System.out.println(als.get(als.size()-1));
      }
   }
}
```

Aufgrund der Channel-Technik ist ein Leistungsvorteil möglich, aber nicht garantiert. Bei manchen Anforderungsprofilen zeigt die Channel-Technik sogar ein schlechteres Leistungsverhalten als die traditionelle Datenstromtechnik.

14.5 Zahlen und Zeichenfolgen aus einer Textdatei lesen

Seit Java 5.0 (alias 1.5) erleichtert die unmittelbar von **java.lang.Object** abstammende Klasse **Scanner** im Paket **java.util** das Einlesen von Zahlen und abgegrenzten Zeichenfolgen (z. B. Wörtern) aus Textdateien im Vergleich zu den früheren Lösungen.

Ein Scanner zerlegt den Eingabestrom aufgrund einer frei wählbaren Trennzeichenmenge in Bestandteile, *Tokens* genannt, auf die man mit diversen Methoden sequentiell zugreifen kann, z. B.:

- **public String next()**
  Die Methode holt das nächste Token.
- **public int nextInt()**
  public double nextDouble()
  Die Methoden versuchen, das nächste Token als **int**- bzw. **double**-Wert zu interpretieren, und werfen bei Misserfolg eine **InputMismatchException**.
- **public BigDecimal nextBigDecimal()**
  Die Methode versucht, das nächste Token als Dezimalzahl zu interpretieren und ein Objekt der Klasse **BigDecimal** daraus zu erstellen. Bei Misserfolg wird eine **InputMismatch-Exception** geworfen.

Ob noch ein Token vorhanden und vom gewünschten Typ ist, kann mit einer entsprechenden Methode festgestellt werden, z. B.:

- **public boolean hasNext()**
  Die Methode prüft, ob noch ein Token vorhanden ist.
- **public boolean hasNextInt()**
  public boolean hasNextDouble()
  Die Methode prüft, ob das nächste Token als **int**- bzw. **double**-Wert interpretierbar ist.
- **public boolean hasNextBigDecimal()**
  Die Methode prüft, ob das nächste Token als Dezimalzahl interpretierbar ist, um ein Objekt der Klasse **BigDecimal** daraus zu erstellen.

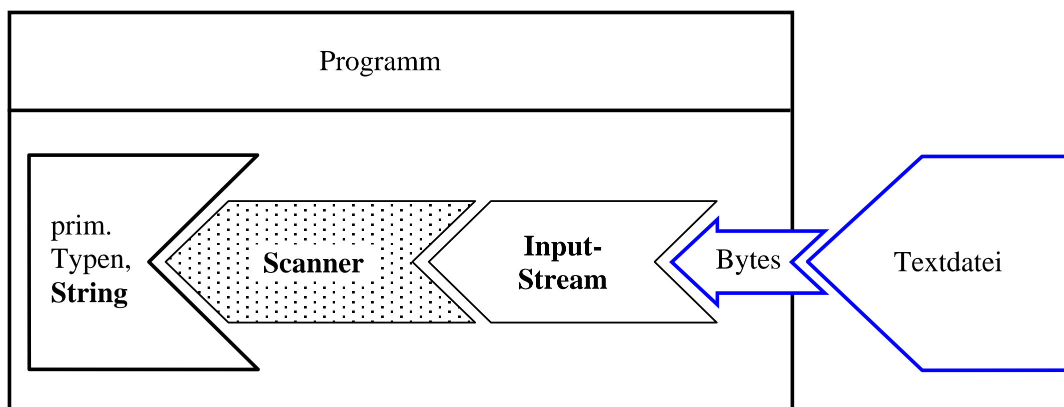
Als Trennzeichen für die Zerlegung des Eingabestroms in Tokens gelten per Voreinstellung alle *WhiteSpace*-Zeichen (z. B. Leerzeichen, Tabulator). Ob ein Zeichen zu dieser Menge gehört, lässt sich mit der statischen **Character**-Methode **isWhitespace()** feststellen. Für eine alternative Festlegung der Trennzeichen steht die **Scanner**-Methode **useDelimiter()** zur Verfügung.

In den diversen **Scanner**-Konstruktoren wird u.a. ein **File**-, **Path**- oder **InputStream**-Objekt als Datenquelle akzeptiert, wobei optional auch eine Kodierung (vgl. Abschnitt 14.4.1.2) angegeben werden kann:

- **public Scanner(File** *source*) **throws FileNotFoundException**
- **public Scanner(File** *source*, **String** *charsetName*) **throws FileNotFoundException**
- **public Scanner(Path** *source*) **throws IOException**
- **public Scanner(Path** *source*, **String** *charsetName*) **throws IOException**
- **public Scanner(InputStream** *source***)**
- **public Scanner(InputStream** *source*, **String** *charsetName*)

Weil der Parameter *charsetName* an die statische Methode **forName()** der Klasse **Charset** im Paket **java.nio.charset** übergeben wird, ist unter den für das **java.nio** - API vorgesehenen Kodierungsnamen zu wählen (siehe Abschnitt 14.4.1.2).

Obwohl die Klasse **Scanner** weder von **InputStream** noch von **Reader** abstammt, benutzen wir doch die im aktuellen Kapitel eingeführte Darstellung der Funktionsweise:



Das folgende Programm liest Zahlen und **String**-Objekte aus einer Textdatei:

```
import java.util.Scanner;
class ScannerFile {
   public static void main(String[] args) throws java.io.IOException {
      try (Scanner input = new Scanner(new java.io.FileInputStream("daten.txt"),
                                       "Windows-1252")) {
         while (input.hasNext())
            if (input.hasNextInt())
               System.out.println("int-Wert:    " + input.nextInt());
            else
               if (input.hasNextDouble())
                  System.out.println("double-Wert: " + input.nextDouble());
               else
                  System.out.println("Text:        " + input.next());
      }
   }
}
```

Mit den Eingabedaten

```
4711   3,1415926
Nicht übel!
13     9,99
```

in einer ANSI-kodierten Textdatei (vgl. Abschnitt 14.4.1.2 zur Kodierung) erhält man die Ausgabe:

```
int-Wert:         4711
double-Wert:      3.1415926
Text:             Nicht
Text:             übel!
int-Wert:         13
double-Wert:      9.99
```

In der Eingabedatei ist das lokalspezifische Dezimaltrennzeichen zu verwenden, bei uns also ein Komma. Zum Lesen einer vorhandenen Datei mit dem Punkt als Dezimaltrennzeichen muss man das Kultur- bzw. Gebietsschema anpassen, z. B.

```
try (Scanner input = new Scanner(
                     new java.io.FileInputStream("daten.txt"), "Windows-1252")) {
  input.useLocale(Locale.US);
   . . .
}
```

Wir haben den Einsatz der Klasse **Scanner** für die Datenerfassung via Konsole bereits im Abschnitt 3.4.1 erwogen. Das folgende Programm nimmt zwei reelle Zahlen *a* und *b* von der Standardeingabe (einem **InputStream**-Objekt) entgegen und berechnet die Potenz a^b mit Hilfe der statischen **Math**-Methode **pow()**:

```
import java.util.*;
class ScannerConsole {
  public static void main(String[] args) {
    double basis, exponent;
    Scanner input = new Scanner(System.in);
    System.out.print("Basis und Exponent Argumente (durch Leerzeichen getrennt):  ");
    try {
      basis = input.nextDouble();
      exponent = input.nextDouble();
      System.out.println(basis + " hoch " + exponent +
                         " = " + Math.pow(basis, exponent));
    } catch(InputMismatchException e) {
      System.err.println("Eingabefehler");
    }
  }
}
```

Nun sind Sie im Stande, die von der Klasse `Simput` (siehe Abschnitt 3.4.1) zur Verfügung gestellten Methoden zur Eingabe primitiver Datentypen via Tastatur komplett zu verstehen (und zu kritisieren). Als Beispiel wird die Methode `gint()` wiedergegeben:

```
package de.uni_trier.zimk.util.conio;

import java.util.*;
import java.io.*;

public class Simput {
   static private boolean error;
   static private String  errorDescription = "";

   static public int gint() {
      int eingabe = 0;
      @SuppressWarnings("resource")
      Scanner input = new Scanner(System.in);
      try {
         eingabe = input.nextInt();
         error = false;
         errorDescription = "";
      } catch(Exception e) {
         error = true;
         errorDescription = e.getMessage();
         System.out.println("Falsche Eingabe!\n");
      }
      return eingabe;
   }
  .  .  .
}
```

Das IntelliJ-Projekt mit der Klasse `Simput` als Bestandteil des Pakets `de.uni_trier.zimk.util.conio` im Modul `de.uni_trier.zimk.util` finden Sie im Ordner:

…\BspUeb\Pakete und Module\Bruchaddition\IntelliJ

14.6 Objektserialisierung

Wer objektorientiert programmiert, möchte natürlich auch ...

- Persistenzaufgaben objektorientiert lösen, also komplette Objekte in permanente Datenspeicher befördern und von dort einlesen,
- bei Netzwerktransfers zwischen Java-Programmen auf verschiedenen Rechnern komplette Objekte auf einfache Weise übertragen.

In Java können Objekte tatsächlich meist genauso einfach wie primitive Datentypen in einen Byte-Strom geschrieben bzw. von dort gelesen werden. Die Übersetzung eines Objektes (mit all seinen Instanzvariablen) in einen Byte-Strom bezeichnet man recht treffend als *Objektserialisierung*, den umgekehrten Vorgang als *Objektdeserialisierung*. Wenn Instanzvariablen auf andere Objekte zeigen, werden diese in die Sicherung und spätere Wiederherstellung einbezogen. Weil auch die referenzierten Objekte wieder Mitgliedsobjekte haben können, ist oft ein ganzer *Objektgraph* beteiligt. Ein mehrfach referenziertes Objekt wird dabei Ressourcen schonend nur *einmal* einbezogen, und Objektidentitäten bleiben gültig.

Eine häufig genutzte Anwendung der (De-)Serialisierung besteht darin, eine tiefe, den gesamten Objektgraphen einbeziehende Kopie eines Objekt zu erstellen (siehe z. B. Krüger & Hansen 2014, S. 881ff). Man arbeitet mit den Klassen **ByteArrayOutputStream** und **ByteArrayInputStream**, verwendet also einen **byte**-Array auf dem Heap zur Zwischenspeicherung. Eine tiefe Kopie sollte eine Klasse eigentlich durch eine passende Überschreibung der **Object**-Methode **clone()** ermöglichen, die jedoch bei vielen Klassen fehlt.

In die Serialisierung werden alle Instanzvariablen (unabhängig von der Sichtbarkeit) einbezogen, die nicht als **transient** deklariert sind (siehe Abschnitt 14.6.5). Statische Felder werden naheliegender Weise bei der Objektserialisierung ignoriert.

Leider hat sich herausgestellt, dass die komplexe (De-)Serialisierungstechnik in Java ein erhebliches Potential für Attacken auf Software enthält (Denial-of-Service, Ausführung von beliebigem Schad-Code). Das Deserialisieren von nicht-vertrauenswürdigen Byte-Strömen ist strikt zu unterlassen. Weitere Details zu Sicherheitsrisiken bei der Objektserialisierung werden im Abschnitt 14.6.1 behandelt.

Wer über die anschließenden Erläuterungen hinausgehende Informationen zur Serialisierung benötigt, findet sie z. B. in der *Java Object Serialization Specification* (Oracle 2010).[1]

14.6.1 Objektserialisierung und Sicherheit

Auffälligster Bestandteil in der API-Dokumentation zur Schnittstelle **Serializable** ist die folgende Warnung:[2]

> **Warning: Deserialization of untrusted data is inherently dangerous and should be avoided. Untrusted data should be carefully validated according to the "Serialization and Deserialization" section of the Secure Coding Guidelines for Java SE. Serialization Filtering describes best practices for defensive use of serial filters.**

Wer aus unsicherer Quelle stammende Byte-Ströme bedenkenlos deserialisiert, riskiert böswillige Angriffe aller Art durch manipulierte Ströme (siehe z. B. Bloch 2018, S. 339ff), z. B.:

---

1 https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html

2 https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/io/Serializable.html

- Remote-Code-Execution (RCE)
- Denial-of-Service (DoS)
  Dabei werden Ströme konstruiert, die einen sehr großen Deserialisierungsaufwand verursachen.

Bloch (2018) empfiehlt:

- Nach Möglichkeit sollte auf die Java-Serialisierung verzichtet werden. Durch die im Abschnitt 14.6.8 als Alternativen zur Java-Serialisierung zur Überwindung von Kompatibilitäts- und Performanz-Problemen beschriebenen Techniken (JSON, Protocol Buffers) lassen sich auch die Sicherheitsprobleme lösen.
- Wenn z. B. ein Framework zur Verwendung der Serialisierung zwingt, muss beim Deserialisieren ein hoher Sicherheitsaufwand betrieben werden.

14.6.2 Beispiel für eine serialisierbare Klasse

Voraussetzungen für die Serialisierbarkeit einer Klasse sind:

- Die Klasse muss das Marker-Interface **Serializable** implementieren.
  Diese Schnittstelle deklariert keinerlei Methoden, und das Implementieren ist als Einverständniserklärung zu verstehen.
- Enthält eine Klasse Instanzobjekte, dann müssen auch deren Klassen mit der Serialisierung einverstanden sein. Anderenfalls kommt es zu einer **NotSerializableException**, die allerdings verhindert werden kann, indem betroffene Instanzvariablen als **transient** deklariert werden (siehe Abschnitt 14.6.5).

Wie gleich im Abschnitt 14.6.3 zu sehen sein wird, führt die Nutzung der Serialisierung, also die Produktion von permanent gespeicherten Objekten dazu, dass die Weiterentwicklung einer Klasse mit Inkompatibilitätskosten verbunden sein kann. Dazu trägt vor allem die Tatsache bei, dass auch *private* Instanzvariablen in die Serialisierung einbezogen werden.

Wir demonstrieren die Serialisation mit Hilfe der folgenden Klasse `Kunde`:

```
import java.io.*;
import java.math.BigDecimal;

public class Kunde implements Serializable {
   private static final long serialVersionUID = 1L;
   private String vorname;
   private String name;
   private int stimmung;
   private int nkaeufe;
   private BigDecimal aussen;

   public Kunde(String vorname_, String name_, int stimmung_,
                int nkaeufe_, BigDecimal aussen_) {
      vorname = vorname_;
      name = name_;
      stimmung = stimmung_;
      nkaeufe = nkaeufe_;
      aussen = aussen_;
   }
   public void prot() {
      System.out.println("Name: " + vorname + " " + name);
      System.out.println("Stimmung: " + stimmung);
      System.out.println("Anz. Einkäufe: " + nkaeufe +
                         ", Außenstände: " + aussen+ "\n");
   }
}
```

Für die Summe der Außenstände eines Kunden wird zur Vermeidung von Rundungsungenauigkeiten an Stelle einer Gleitkommavariablen ein Objekt der Klasse **BigDecimal** verwendet. Diese Praxis ist bei allen finanzmathematischen Anwendungen zu empfehlen (vgl. Abschnitt 3.3.7.2).

14.6.3 Versionskontrolle und Kompatibilitätsprobleme

Durch Änderungen an der Klassendefinition kann es zu Inkompatibilitäten mit vorhandenen serialisierten Objekten kommen. Um damit zusammenhängende Probleme zu verhindern, wird beim Serialisieren stets eine Versionsangabe zur Klasse abgespeichert. Per Voreinstellung wird eine automatisch aus der Klassendefinition berechnete Versionsangabe verwendet. Weil sich diese Versionsangabe auch bei Serialisierungs-irrelevanten Modifikationen der Klassendefinition (z. B. durch die Ergänzung einer Methode) ändert, kommt es zu unnötigen Einschränkungen bei der Deserialisierung.

Um dies zu verhindern, definiert man eine private statische **long**-Variable mit dem Namen **serialVersionUID** und ändert deren Wert nur bei einer Serialisierungs-relevanten Modifikationen der Klassendefinition, z. B.:

```
public class Kunde extends Person implements Serializable {
   private static final long serialVersionUID = 1L;
  . . .
}
```

Über inkompatible und kompatible Änderungen der Klassendefinition informiert Oracle (2010) im Abschnitt 5.6. Beispiele für inkompatible Änderungen sind:

- Löschen von Instanzvariablen
  Wenn eine ältere Klasse beim Deserialisieren auf ein jüngeres Objekt trifft, wird eine fehlende Instanzvariable auf den typspezifischen Nullwert gesetzt, was eventuell zu Fehlverhalten führt.
- Änderung des primitiven Datentyps einer Instanzvariablen
  In diesem Fall scheitert das Deserialisieren mit einem Ausnahmefehler vom Typ **InvalidClassException**.

Bei einer inkompatiblen Änderung der Klassendefinition muss die **serialVersionUID** aktualisiert werden, um Fehler zu verhindern.

Beispiele für kompatible Änderungen sind:

- Ergänzung von Instanzvariablen
  Wenn eine jüngere Klasse beim Deserialisieren auf ein älteres Objekt trifft, wird eine fehlende Instanzvariable auf den typspezifischen Nullwert gesetzt. Damit daraus kein Fehlverhalten resultiert, kann die renovierte Klasse eine Methode namens **readObject**() implementieren und dort für eine passende Initialisierung sorgen (siehe Abschnitt 14.6.6).
- Änderung der Sichtbarkeit von Feldern
  Eine Änderung der Sichtbarkeit von Feldern (package, **private**, **protected**, **public**) bereitet bei der Deserialisierung keine Probleme.

Von Inden (2018, Abschnitt 10.3.3) wird ein Verfahren beschrieben, das die Versionskontrolle bei der Serialisierung komplett in Eigenregie ausführt und für den Preis eines deutlich höheren Aufwands z. B. auch Änderungen bei den Datentypen von Instanzvariablen verkraften kann.

Insgesamt ist festzuhalten, dass sich die Serialisierbarkeit einer Klasse bei der Weiterentwicklung durch Einschränkungen und/oder Aufwand negativ bemerkbar machen kann.

14.6.4 Objekte in eine Datei schreiben und von dort lesen

Für das Schreiben von Objekten ist die Byte-orientierte Ausgabetransformationsklasse **ObjectOutputStream** zuständig, für das Lesen die Byte-orientierte Eingabetransformationsklasse **ObjectInputStream**. Einen komplexen Objektgraphen in einen Byte-Strom zu verwandeln bzw. von dort zu rekonstruieren, ist eine anspruchsvolle Aufgabe, die aber in der Regel automatisiert und erfolgreich abläuft.

Im folgenden Beispielprogramm wird ein Objekt der serialisierbaren Klasse `Kunde` mit der **ObjectOutputStream-**Methode **writeObject()**

public final void writeObject (Object *obj***) throws IOException**

in eine Datei befördert und anschließend mit der **ObjectInputStream**-Methode **readObject()**

public final Object readObject() throws IOException, ClassNotFoundException

von dort zurückgeholt. Außerdem wird demonstriert, dass die beiden Klassen **ObjectOutputStream** bzw. **ObjectInputStream** auch Methoden zum Schreiben bzw. Lesen von primitiven Datentypen besitzen (z. B. **writeInt()** bzw. **readInt()**):

```
import java.io.*;

class Serialisierung {
   public static void main(String[] args) throws Exception {
      Kunde kunde = new Kunde("Fritz", "Orth", 1, 13,
                                new java.math.BigDecimal("426.89"));
      System.out.println("Zu sichern:\n");
      kunde.prot();
      try (ObjectOutputStream oos = new ObjectOutputStream(
                                       new FileOutputStream("test.ser"))) {
         oos.writeInt(1);
         oos.writeObject(kunde);
      }

      try (ObjectInputStream ois =  new ObjectInputStream(
                                       new FileInputStream("test.ser"))) {
         int anzahl = ois.readInt();
         Kunde unbekannt = (Kunde) ois.readObject();
         System.out.printf("Nummer des deserialisierten Falles: %d\n\n", anzahl);
         unbekannt.prot();
      }
   }
}
```

Das Programm liefert die folgende Ausgabe:

```
Zu sichern:

Name: Fritz Orth
Stimmung: 1
Anz. Einkäufe: 13, Außenstände: 426.89

Nummer des deserialisierten Falles: 1

Name: Fritz Orth
Stimmung: 1
Anz. Einkäufe: 13, Außenstände: 426.89
```

Selbstverständlich können auch mehrere Objekte in eine Datei gesichert werden, wobei die Reihenfolge beim Schreiben und Lesen identisch sein muss.

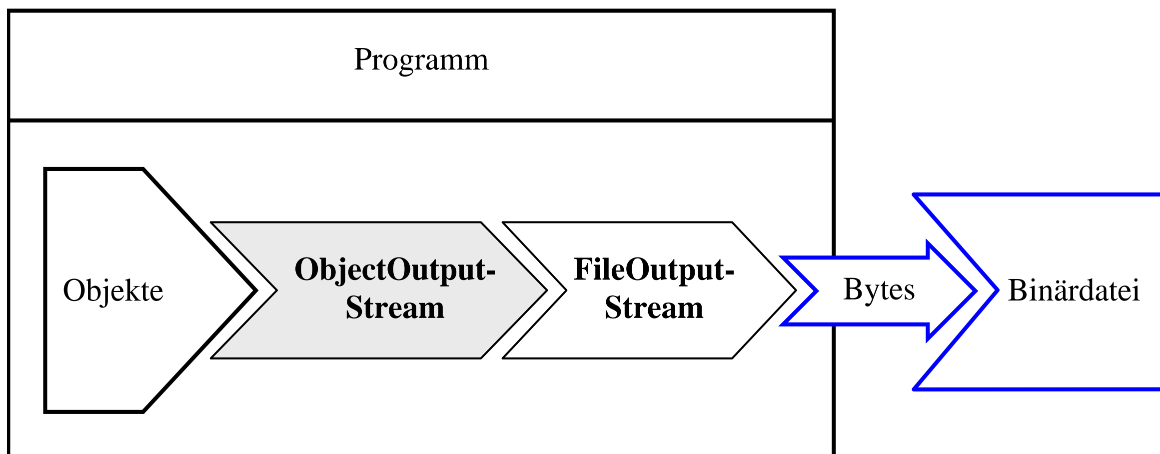
Beim Schreiben eines Objekts wird auch seine Klasse samt **serialVersionUID** festgehalten.

Beim Lesen eines Objekts wird seine Klasse festgestellt, und die JVM benötigt den Bytecode von allen Klassen im Objektgraphen. Ist die **serialVersionUID** identisch, wird das Objekt auf dem Heap angelegt, und die Instanzvariablen erhalten ihre rekonstruierten Werte. Dabei wird *kein* Konstruktor der serialisierbaren Klasse aufgerufen. Gegebenenfalls wird allerdings der Konstruktor der ersten nicht-serialisierbaren Oberklasse aufgerufen (siehe Abschnitt 14.6.7).

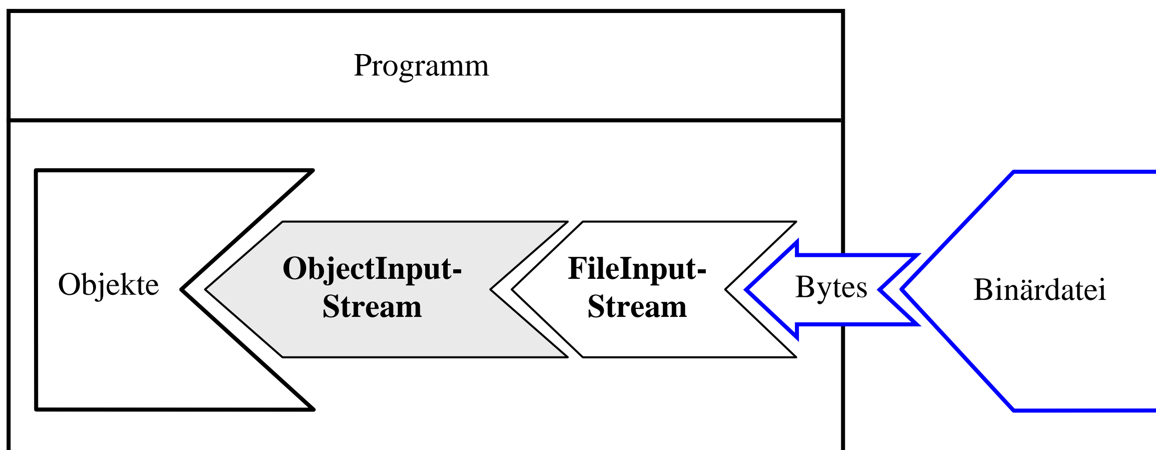
Weil **readObject()** den Rückgabetyp **Object** hat, ist in der Regel eine explizite Typumwandlung erforderlich.

Eine häufig verwendete, aber nicht strikt vorgeschriebene Namenserweiterung für Dateien mit serialisierten Objekten ist **.ser**.

In der folgenden Abbildung wird die Serialisierung von Objekten vom internen Format der JVM in eine binäre Datei skizziert:



Den umgekehrten Weg bei der Deserialisierung von Objekten aus einer binären Datei in das interne Format der JVM beschreibt die nächste Abbildung:



14.6.5 Von der Serialisierung ausgeschlossene Felder

In die Serialisierung werden alle Instanzvariablen (unabhängig von der Sichtbarkeit) einbezogen, die nicht als **transient** deklariert sind. Der Modifikator **transient** eignet sich z. B. für ...

- Variablen, die aus Datenschutzgründen nicht in einer Datei gespeichert werden sollen,
- Variablen vom Typ einer nicht-serialisierbaren Klasse (z. B. **Thread**, **Socket**),

- Rekonstruierbare Variablen,[1]
- Variablen mit temporären Inhalten, die auf jeden Fall nach dem Deserialisieren neu gesetzt werden müssen.

Wird in der Klasse `Kunde` die Instanzvariable `stimmung` als **transient** deklariert,

```
private transient int stimmung;
```

dann liefert das Beispielprogramm die folgende Ausgabe:

```
Zu sichern:

Name: Fritz Orth
Stimmung: 1
Anz. Einkaeufe: 13, Aussenstaende: 426.89

Nummer des deserialisierten Falles: 1

Name: Fritz Orth
Stimmung: 0
Anz. Einkaeufe: 13, Aussenstaende: 426.89
```

Die Instanzvariable `stimmung` des eingelesenen Objektes besitzt den **int**-Initialwert 0, während die übrigen Instanzvariablen über beide Serialisierungsschritte hinweg ihre Werte behalten haben.

14.6.6 Mehr Kontrolle und Eigenverantwortung

Obwohl die (De)serialisierungs-Automatik in Java ihre komplexe Aufgabe gut erledigt, gibt es doch einige Anlässe für ergänzende Arbeiten, z. B.

- Initialisierung von transienten Variablen nach der Deserialisierung (siehe Abschnitt 14.6.5)
- Unterstützung von verschiedenen Versionen einer Klasse (mit unterschiedlichen **serialVersionUID**s) (siehe Abschnitt 14.6.3)
- Versorgung von Erbstücken aus einer nicht-serialisierbaren Oberklasse (siehe Abschnitt 14.6.7)

Um bei der (De)serialisierung ein Wörtchen mitreden zu können, muss man in der serialisierbaren Klasse die Methoden **writeObject()** und **readObject()** mit den folgenden Definitionsköpfen implementieren:

- **private void writeObject(ObjectOutputStream** *oos***) throws IOException**
- **private void readObject(ObjectInputStream** *ois***) throws IOException, ClassNotFoundException**

Diese Methoden dürfen nicht mit den gleichnamigen Methoden der Klassen **ObjectOutputStream** bzw. **ObjectInputStream** verwechselt werden (vgl. Abschnitt 14.6.4).

Sind die Methoden **writeObject()** und **readObject()** in einer serialisierbaren Klasse implementiert, werden sie bei der Serialisierung automatisch aufgerufen. Diese Methoden sind dann für alle in der Klasse definierten Instanzvariablen zuständig, während eventuell vorhandene geerbte Instanzvariablen weiterhin von der Serialisierungsautomatik versorgt werden. Zu Beginn von **writeObject()** bzw. **readObject()** sollte auf jeden Fall über die Methoden **defaultWriteObject()** bzw. **defaultReadObject()** die Standard(de)serialisierung der klasseneigenen Instanzvariablen angefordert werden. In **writeObject()** werden dann nach Bedarf durch Überladungen der **ObjectOutputStream**-Methode **writeObject()** weitere Daten in den Ausgabestrom geschrieben. In **readObject()** werden nach dem obligatorischen **defaultReadObject()** - Aufruf nach Bedarf durch

[1] In einer Klasse können „redundante“ Variablen aus Performanzgründen existieren, um wiederholte Berechnungen zu vermeiden.

Überladungen der **ObjectInputStream**-Methode **readObject()** weitere Daten aus dem Eingabestrom gelesen. Wenn die Serialisierung zum Einsatz kommt, obwohl die im Abschnitt 14.6.1 beschriebenen Sicherheitsrisiken nicht auszuschließen sind, dann sollte die Methode **readObject()** analog zu einem öffentlichen Konstruktor die Gültigkeit des deserialisierten Objekts überprüfen.

Das Implementieren von **writeObject()** und **readObject()** ist insbesondere dann zu empfehlen, wenn sich die physikalische Repräsentation der Daten (z. B. doppelt verkettete Liste mit Zeichenfolgen) von der logischen Struktur (z. B. Liste mit Zeichenfolgen) unterscheidet (Bloch 2018, S. 348ff). Im Beispiel sollten die Zeichenfolgen seriell in den Strom geschrieben werden, statt den komplexen Objektgraphen von der Automatik behandeln zu lassen. Nach den Beobachtungen von Bloch benötigt die Automatik im Beispiel doppelt so viel Zeit und Speichervolumen, wobei es ab einer bestimmten Größe der verketteten Liste sogar zu einem Stackoverflow-Laufzeitfehler kommen kann.

Ein Beispiel für die Verwendung von **writeObject()** und **readObject()** folgt im Abschnitt 14.6.7 im Zusammenhang mit Instanzvariablen, die von einer nicht-serialisierbaren Oberklasse geerbt wurden.

14.6.7 Serialisierung und Vererbung

Eine (direkte oder indirekte) Ableitung einer serialisierbaren Klasse ist ebenfalls serialisierbar. Um ein Serialisieren doch zu verhindern, kann man in der betroffenen Klasse die Methoden **writeObject()** und **readObject()** realisieren (siehe Abschnitt 14.6.6) und dabei durch das Werfen einer Ausnahme den Dienst verweigern.

Trotz der Existenz einer nicht-serialisierbaren Klasse in der Ahnenreihe darf eine Klasse das Interface **Serializable** implementieren. In diesem Fall werden allerdings die von der nicht-serialisierbaren Klasse geerbten Instanzvariablen *nicht* abgespeichert. Nach der Deserialisierung wird automatisch der parameterfreie Konstruktor der ersten nicht-serialisierbaren Oberklasse in der Ahnenreihe aufgerufen, sodass die betroffenen Instanzvariablen initialisiert werden.

Um zusätzliche Maßnahmen zum Kompensieren der fehlenden Serialisierbarkeit von einigen Instanzobjekten zu ergreifen, kann man in der serialisierbaren Klasse die Methoden **writeObject()** und **readObject()** implementieren (siehe Abschnitt 14.6.6). Enthält die serialisierbare Klasse z. B. ein Instanzobjekt der (nicht-serialisierbaren) Klasse **Socket** aus dem Paket **java.net** zur Kommunikation mit einem Server via Internet, dann kann statt des referenzierten Objekts die Serveradresse und die Portnummer abspeichern und beim Deserialisieren mit Hilfe dieser Daten ein neues **Socket**-Objekt erstellen.

Zur Demonstration der Methoden **writeObject()** und **readObject()** durch ein einfaches Beispiel verlagern wir aus der Klasse `Kunde` (siehe Abschnitt 14.6.2) die Instanzvariablen `vorname` und `name`

```
public class Kunde extends Person implements Serializable {
   private static final long serialVersionUID = 1L;
   private transient int stimmung;
   private int nkaeufe;
   private BigDecimal aussen;
  . . .
}
```

in die nicht-serialisierbare Basisklasse `Person`:

```
public class Person {
   protected String vorname;
   protected String name;
}
```

Das (De)serialisations-Testprogramm zeigt, dass beim aktuellen Entwicklungsstand der Klasse `Kunde` die Werte der `Person`-Instanzvariablen verloren gehen:

```
Zu sichern:

Name: Fritz Orth
Stimmung: 1
Anz. Einkäufe: 13, Außenstände: 426.89

Nummer des deserialisierten Falles: 1

Name: null null
Stimmung: 0
Anz. Einkäufe: 13, Außenstände: 426.89
```

Um dies zu verhindern, werden in der serialisierbaren Klasse `Kunde` die Methoden **writeObject()** und **readObject()** implementiert, die zunächst durch einen Aufruf der Methode **defaultWriteObject()** bzw. **defaultReadObject()** den API-Serialisierungs-Service so weit als möglich nutzen und dann für das Schreiben bzw. Lesen der `Person`-Felder sorgen:

```
private void writeObject(ObjectOutputStream oos) throws IOException {
   oos.defaultWriteObject();
   oos.writeObject(vorname);
   oos.writeObject(name);
}

private void readObject(ObjectInputStream ois) throws IOException,
                                                       ClassNotFoundException {
   ois.defaultReadObject();
   vorname = (String) ois.readObject();
   name = (String) ois.readObject();
}
```

Nun klappt die (De-)serialisierung wie erwünscht:

```
Zu sichern:

Name: Fritz Orth
Stimmung: 1
Anz. Einkäufe: 13, Außenstände: 426.89

Nummer des deserialisierten Falles: 1

Name: Fritz Orth
Stimmung: 0
Anz. Einkäufe: 13, Außenstände: 426.89
```

14.6.8 Bewertung der Objektserialisierung und mögliche Alternativen

Als Vorteile der Serialisierungslösung im Java-SE - API sind zu nennen:

- Einfache Handhabung
- keine über Java-SE hinausgehenden Bibliotheken erforderlich
- Gute Unterstützung von Objektgraphen
  Referenzierte Memberobjekte werden automatisch mit übertragen, wobei Objektidentitäten erhalten bleiben.

Damit ist die (De)serialisierung zum Speichern von Objekten und für den Netzwerktransfer zwischen Java-Programmen in vielen Fällen eine praxistaugliche Lösung. Hinzu kommt die Eignung der Speicher-internen (De-)Serialisierung zum Erstellen einer tiefen Objektkopie.

Allerdings muss die Sicherheitswarnung vor dem Deserialisieren von Objekten aus unsicherer Quelle beachtet werden, die in fetten Lettern am Anfang der API-Dokumentation zur Schnittstelle **Serializable** zu finden ist:[1]

> **Warning: Deserialization of untrusted data is inherently dangerous and should be avoided.**

Bei einigen speziellen Klassen rät Bloch (2018, S. 243ff) davon ab, das Interface **Serializable** zu implementierten:

- Potentielle Basisklassen sollten in der Regel nicht serialisierbar sein, weil anderenfalls abgeleitete Klassen mit den im Abschnitt 14.6.3 beschriebenen Kompatibilitätsproblemen belastet werden. Aus demselben Grund sollte die Schnittstelle **Serializable** in der Regel nicht von anderen Schnittstellen erweitert werden. Wenn allerdings eine abgeleitete Klasse serialisierbar sein muss, verursacht eine nicht-serialisierbare Basisklasse einen Zusatzaufwand (siehe Abschnitt 14.6.7).
- Objekte von nicht-statischen inneren Klassen, lokalen Klassen und anonymen Klassen (traditionell oder per Lambda-Ausdruck realisiert) eignen sich nicht zum Serialisieren, weil sie von Compiler generierte Felder mit Referenzen auf umgebende Instanzen oder auf lokale Variablen im umgebenden Gültigkeitsbereich enthalten.

Nicht zuletzt wegen der weitgehend automatischen Erledigung komplexer Aufgaben kann bei der (De-)Serialisierung eines umfangreichen Objektgraphen eine große Datenmeng auftreten. Gelegentlich besteht eine Möglichkeit zur Reduktion der Datenmenge darin, Member-Objekte durch Daten mit primitivem Typ zu ersetzen. Dabei steigt allerdings der Entwicklungsaufwand. Das gilt erst recht für eine Lösung basierend auf dem von **Serializable** abgeleiteten Interface **Externalizable**, wobei das Datenformat vom Entwickler frei definiert werden kann (siehe Inden 2018, Abschnitt 10.3.4).

Wenn

- andere Programmiersprachen beteiligt sind,
- oder eine sehr hohe Effizienz bei der Übertragung bzw. Speicherung gefragt ist,

dann kommen Alternativen zur Java - Serialisierung in Betracht, wobei vor allem zu nennen sind:

- **JSON** (*JavaScript Object Notation*) ist ein textorientiertes Format, das häufig zur Netzwerkübertragung von Objekten zwischen Server und Browser eingesetzt wird, aber auch zum Speichern von Objekten in Dateien verwendet werden kann.
- Als besonders effizient gelten die für mehrere Programmiersprachen (z. B. Java, C#, Python, C++, Ruby) verfügbaren binären **Protocol Buffers** der Firma Google.[2]

Durch die Nutzung dieser Techniken werden zudem die Sicherheitsprobleme der Java-Serialisierung überwunden.

14.7 Daten lesen und schreiben über die NIO.2 - Klasse Files

Die dem NIO.2 -API zugerechnete Klasse **Files** im Paket **java.nio** beherrscht nicht nur Dateisystemzugriffe (siehe Abschnitt 14.2.1) und die Fabrikation von Datenstromobjekten mit Channel-Technik (siehe Abschnitte 14.3.1.3, 14.3.2.3, 14.4.1.6 und 14.4.2.4) sondern auch das Lesen und Schreiben von Daten. Wird beim Schreiben und Lesen von Zeichen über eine **Files**-Methode oder ein per **Files** erstelltes Datenstromobjekt keine Kodierung angegeben, dann kommt UTF-8 zum Einsatz. Außerdem kann die Klasse **Files** den MIME-Typ von Dateien ermitteln und zu einer Textdatei einen **Stream<String>** (im Sinn von Abschnitt 12.2) erstellen.

---

[1] https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/io/Serializable.html

[2] https://developers.google.com/protocol-buffers/

14.7.1 Öffnungsoptionen

In der **Files**-Methode **write()** zum Schreiben in eine Datei sowie in den **Files**-Methoden zum Erstellen eines Datenstromobjekts zu einer Datei (**newOutputStream()**, **newInputStream()**, **newBufferedWriter()**) können über einen Serienparameter vom Interface-Typ **OpenOption** Optionen für das Öffnen der Datei festgelegt werden. Das Interface wird u.a. von der Enumeration **StandardOpenOption** im Paket **java.nio.file** implementiert, die folgende Konstanten (vordefinierte Objekte) für besonders häufig benötigte Öffnungsoptionen enthält:

- **READ**
  Mit dieser Option wird eine Datei zum Lesen geöffnet.
- **WRITE**
  Mit dieser Option wird eine Datei zum Schreiben geöffnet.
- **APPEND**
  Bei einer bereits existenten, zum Schreiben geöffneten Datei sorgt diese Option dafür, dass neue Ausgaben am Ende angehängt werden, statt vorhandene Ausgaben zu überschreiben.
- **TRUNCATE_EXISTING**
  Durch diese nur beim Schreiben erlaubte Option wird bei einer vorhandenen Datei der bisherige Inhalt komplett gelöscht. Lässt man beim Schreiben ab Dateianfang diese Option weg, bleiben eventuell am Dateiende vorhandene Bytes stehen.
- **CREATE**
  Diese Option sorgt dafür, dass eine zum Schreiben zu öffnende Datei nötigenfalls angelegt wird.
- **CREATE_NEW**
  Es wird eine neue Datei angelegt oder bei vorhandener Datei eine Ausnahme vom Typ **FileAlreadyExistsException** geworfen.
- **DELETE_ON_CLOSE**
  Aufgrund dieser Option wird eine Datei beim Schließen nach Möglichkeit automatisch gelöscht, was bei temporären Dateien sinnvoll ist.
- **SPARSE**
  Einige Dateisysteme (z. B. NTFS unter Windows) profitieren von dem Hinweis, dass eine Datei spärlich besetzt ist und größtenteils aus Nullbytes besteht.

Beispiel:

```
InputStream instr = Files.newInputStream(file, StandardOpenOption.READ);
```

14.7.2 Lesen und Schreiben von kleinen Dateien

In diesem Abschnitt werden statische Ein-/Ausgabemethoden der Klasse **Files** vorgestellt, die sich laut Java-Tutorial (Oracle 2019a) im Unterschied zur Verwendung eines **Stream**-Objekts (siehe Abschnitte 14.3 und 14.4) nur für *kleine* Dateien eignen.[1] Der wesentliche Grund für diese Einschränkung besteht darin, dass alle Daten beim Lesen „in einem Rutsch“ in den Hauptspeicher befördert bzw. beim Schreiben von dort abgeholt werden. Weil sich alle Daten simultan im Hauptspeicher befinden, wird dort entsprechend viel Platz benötigt.

Ein Vorteil der anschließend vorgestellten Methoden besteht darin, dass die beteiligten Dateien nach dem (gelungenen oder gescheiterten) Lesen bzw. Schreiben automatisch geschlossen werden.

Soll eine Datei komplett in einen **byte**-Array eingelesen werden, bietet die statische **Files**-Methode **readAllBytes()** eine bequeme Lösung:

[1] http://docs.oracle.com/javase/tutorial/essential/io/file.html#common

public static byte[] readAllBytes(Path *path***) throws IOException**

Im folgenden Beispiel wird ein Foto aus einer Datei im JPEG-Format in einen **byte**-Array eingelesen:

```
byte[] imb = Files.readAllBytes(Paths.get("Emma.jpg"));
```

Mit der Methode **readAllLines()** befördert man *alle* Zeilen einer Textdatei in eine Kollektion vom Typ **List<String>**:

public static List<String> readAllLines(Path *path***, Charset** *cs***) throws IOException**

Die gewünschte Zeichenkodierung wird über ein **Charset**-Objekt gewählt (siehe Abschnitt 14.4.1.2).

Für eine Textdatei mit UTF-8 - Kodierung eignet sich die folgende Überladung ohne **Charset**-Parameter:

public static List<String> readAllLines(Path *path***) throws IOException**

Mit der statischen **Files**-Methode **write()** befördert man einen **byte**-Array in eine Datei:

public static Path write(Path *path***, byte[]** *buffer***, OpenOption...** *options***)**
throws IOException

Mit der folgenden **write()** - Überladung schreibt man die in einem iterierbaren Container befindlichen Zeichenfolgen (Objekte vom Typ **CharSequence**) in eine Datei:

public static Path write(Path *path***, Iterable<? extends CharSequence >** *lines***,**
Charset *cs***, OpenOption...** *options***) throws IOException**

Dabei sind die Zeichenkodierung und der Dateiöffnungsmodus einstellbar.

Für eine Textdatei mit UTF-8 - Kodierung eignet sich die folgende Überladung ohne **Charset**-Parameter:

public static Path write(Path *path***, Iterable<? extends CharSequence >** *lines***,**
OpenOption... *options***) throws IOException**

14.7.3 Datenstrom zu einem Path-Objekt erstellen

Zu einem **Path**-Objekt, das eine Datei repräsentiert, kann man über statische Fabrikmethoden der Klasse **Files** Datenstromobjekte für Byte- bzw. Zeichenströme erstellen (siehe Abschnitt 14.3 bzw. 14.4):

- **public static OutputStream newOutputStream(Path** *path***, OpenOption...** *options***)**
  throws IOException
  Man erhält einen Byte-orientieren Ausgabestrom, der mit einer Datei verbunden ist (siehe Abschnitt 14.3.1.3).
- **public static InputStream newInputStream(Path** *path***, OpenOption...** *options***)**
  throws IOException
  Man erhält einen Byte-orientieren Eingabestrom, der mit einer Datei verbunden ist (siehe Abschnitt 14.3.2.3).
- **public static BufferedWriter newBufferedWriter(Path** *path***, Charset** *cs***,**
  OpenOption... *options***)**
  throws IOException
  Man erhält einen gepufferten, zeichenorientieren Ausgabestrom (siehe Abschnitt 14.4.1.6), der einen durch die eben vorgestellte **Files**-Methode **newOutputStream()** erstellten **OutputStream** für die Verbindung zur Ausgabedatei verwendet.

- **public static BufferedReader newBufferedReader(Path** *path*, **Charset** *cs*) **throws IOException**
  Man erhält einen gepufferten zeichenorientierten Eingabestrom (siehe Abschnitt 14.4.2.4) der einen durch die eben vorgestellte **Files**-Methode **newInputStream()** erstellten **InputStream** für die Verbindung zur Eingabedatei verwendet.

Zu **newBufferedWriter()** und **newBufferedReader()** existieren Überladungen ohne **Charset**-Parameter, wobei die UTF-8 - Kodierung zum Einsatz kommt.

Im Vergleich zu traditionellen, per Konstruktor erstellten Datenstromobjekten (z. B. aus den Klassen **FileOutputStream**, **FileInputStream**, **BufferedWriter**, **BufferedReader**) haben die Produkte der **Files**-Fabrikmethoden folgende Vorteile:

- Bei den Dateizugriffen kommt die Channel-Technik zum Einsatz, wobei ein Leistungsvorteil möglich, aber nicht garantiert ist.
- Für die Ausgabedateien können Öffnungsoptionen gesetzt werden (vgl. Abschnitt 14.7.1).
- Für die die Produkte der **Files**-Fabrikmethoden **newOutputStream()** und **newInputStream()** ist die simultane Nutzung durch mehrere Threads erlaubt.

14.7.4 MIME-Type einer Datei ermitteln

Von der statischen **Files**-Methode **probeContentType()** erhält man eine Information über den MIME-Typ des Dateiinhalts:

public static String probeContentType(Path *path*) **throws IOException**

Ursprünglich zur Beschreibung von Mail-Erweiterungen gedacht (*Multipurpose Internet Mail Extensions*), wird das MIME-Schema mittlerweile recht universell zur Deklaration von digitalen Inhalten verwendet.

Im folgenden Programm

```
import java.io.IOException;
import java.nio.file.*;

class ProbeContentType {
   public static void main(String[] args) throws IOException {
      Path ordner = Paths.get("U:", "Eigene Dateien", "Java", "Test");
      System.out.println("Inhaltstyp der Dateien im Verzeichnis " + ordner + ":\n");
      try (DirectoryStream<Path> stream = Files.newDirectoryStream(ordner)) {
          for (Path path: stream)
              System.out.printf("%-13s %s\n", path.getFileName(),
                                  Files.probeContentType(path));
      }
   }
}
```

wird der MIME-Type für alle Dateien in einem Verzeichnis aufgelistet:

```
Ausgabe.txt   text/plain
Begriffe.pdf  application/pdf
Java13.docx   application/vnd.openxmlformats-officedocument.wordprocessingml.document
JellyFish.jpg image/jpeg
misc.xml      text/xml
```

Wie man durch Umbenennen einer Datei verifizieren kann, orientiert sich **probeContentType()** unter Windows nicht am Dateiinhalt, sondern an der Namenserweiterung.

14.7.5 Stream<String> mit den Zeilen einer Textdatei erstellen

Die statische Methode **lines()** der Klasse **Files** liefert ein Objekt der Klasse **Stream<String>**, das die Verarbeitung der Zeilen einer Textdatei erleichtert und beim Lesen die UTF-8 - Kodierung unterstellt. Im folgenden Code-Segment werden mit der Stromoperation **count()** die Zeilen in einer Datei gezählt (vgl. Abschnitt 12.2.5.4.3):

```
Stream<String> sol = Files.lines(Paths.get("U:/Eigene Dateien/ausgabe.txt"));
System.out.println("Anzahl der Zeilen: " + sol.count());
```

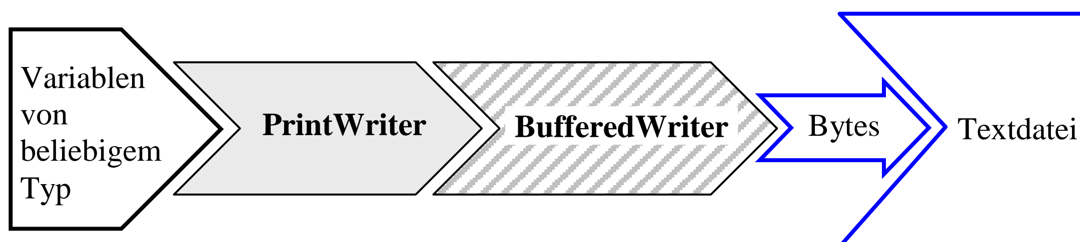
Es ist zu beachten, dass die Klasse **Stream<String>** zu keiner Datenstrom-Hierarchie im Sinne des aktuellen Kapitels 14 gehört, sondern einen Strom im Sinne der mit Java 8 eingeführten Techniken der funktionalen Programmierung repräsentiert (siehe Abschnitt 12.2).

14.8 Empfehlungen zur Ein- und Ausgabe

Weil die Ein-/Ausgabe - Behandlung in Java durch die Vielzahl der beteiligten Klassen und durch die Koexistenz von Lösungen aus verschiedenen Java-Entwicklungsstadien etwas unübersichtlich ist, folgt in diesem Abschnitt eine rezeptartige Beschreibung wichtiger Spezialfälle beim Schreiben in Dateien bzw. beim Lesen aus Dateien.

14.8.1 Ausgabe in eine Textdatei

Um Textdaten (Datentypen **String**, **char**) oder die Zeichenfolgen-Repräsentationen beliebiger andere Datentypen in eine Datei zu schreiben, eignet sich ein durch die statische Methode **newBufferedWriter()** der Klasse **Files** erstellter **BufferedWriter** (siehe Abschnitt 14.4.1.6) in Kombination mit einem **PrintWriter** (siehe Abschnitt 14.4.1.5), der bequeme Ausgabemethoden bietet (z. B. **println()**, **printf()**, **format()**).



Beispiel:

```
import java.io.*;
import java.nio.file.*;
import java.nio.charset.Charset;

class DataToText {
   public static void main(String[] args) throws IOException {
      try (PrintWriter pw =  new PrintWriter(Files.newBufferedWriter(
                             Paths.get("Ausgabe.txt"), Charset.forName("UTF-8")))) {
         pw.println(4711);
         pw.printf("%4.2f", Math.PI);
         String ls = System.getProperty("line.separator");
         pw.println(ls+"Nicht übel!");
      }
   }
}
```

Über Parameter der Methode **newBufferedWriter()**

public static BufferedWriter newBufferedWriter(Path *path*,
Charset *cs*,
OpenOption... *options*)
throws IOException

wählt man:

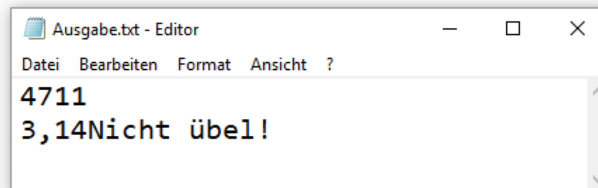
- die Ausgabedatei per **Path**-Objekt (NIO.2 - API)
- eine Kodierung über den Parameter vom Typ **Charset** (vgl. Abschnitt 14.4.1.2)
  Wählt man eine Überladung ohne diesen Parameter, wird die Kodierung UTF-8 verwendet.
- Öffnungsoptionen über den Serienparameter vom Typ **OpenOption** (siehe Abschnitt 14.7.1)
  Per Voreinstellung sind aus der Enumeration **StandardOpenOption** die folgenden Werte in Kraft: **CREATE**, **TRUNCATE_EXISTING** und **WRITE**. Folglich wird eine fehlende Datei erstellt und eine vorhandene Datei zunächst entleert.

Die **PrintWriter**-Methode **printf()** (alias **format()**) ermöglicht eine flexible Formatierung der Ausgabe.

Die bei der Konsolenausgabe häufig verwendete Escape-Sequenz **\n** eignet sich nicht dazu, um einen Zeilenwechsel in eine Textdatei einzufügen. Stattdessen sollte die Plattform-spezifische Zeilenschaltung verwendet werden, die mit dem folgenden Aufruf der statischen **System**-Methode **getProperty()** zu ermitteln ist:

```
System.getProperty("line.separator")
```

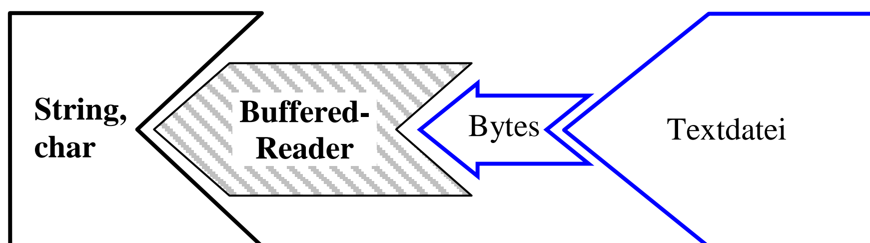
Anderenfalls erkennt z. B. unter Windows der Standardeditor **Notepad** den Zeilenwechsel nicht:



Die **PrintWriter**-Methode **println()** schließt ihre Ausgabe korrekt mit der Plattform-spezifischen Zeilenschaltung ab.

14.8.2 Textzeilen einlesen

Um Texte aus einer Datei zu lesen, eignet sich ein durch die statische Methode **newBufferedReader()** der Klasse **Files** erstellter **BufferedReader** (siehe Abschnitt 14.4.2.4), der die Anzahl der Dateizugriffe reduziert und die bei Dateien mit Zeilenstruktur sehr nützliche Methode **readLine()** bietet.



Beispiel:

```
import java.io.*;
import java.nio.file.*;
import java.nio.charset.Charset;
import java.util.*;
```

```
class ReadText {
   public static void main(String[] args) throws IOException {
      List<String> ls = new ArrayList<String>();
      try (BufferedReader br = Files.newBufferedReader(Paths.get("Quelle.txt"),
                                                     Charset.forName("Windows-1252"))) {
         String s;
         while ((s=br.readLine()) != null)
            ls.add(s);
      }
      for(String s : ls)
         System.out.println(s);
   }
}
```

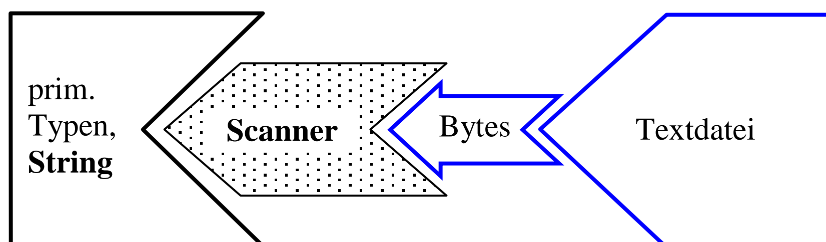
Über Parameter der Methode **newBufferedReader**() wählt man:

- die Eingabedatei per **Path**-Objekt (NIO.2 - API)
- die Kodierung (siehe Abschnitt 14.4.1.2, Voreinstellung: UTF-8)

Zum Lesen von Zeichenfolgen kommt auch die Klasse **Scanner** in Frage (siehe Abschnitte 14.5 und 14.8.3), die den Eingabestrom aufgrund wählbarer Trennzeichen in Bestandteile (Tokens) zerlegen kann und ebenfalls die Wahl eines Kodierungsschemas erlaubt.

14.8.3 Zahlen und andere Tokens aus einer Textdatei lesen

Um Werte primitiver Datentypen und andere separierte Zeichenfolgen (Tokens) aus einer Textdatei zu lesen, kann man ein Objekt aus der Klasse **Scanner** im Paket **java.util** verwenden (siehe Abschnitt 14.5):



Beispiel:

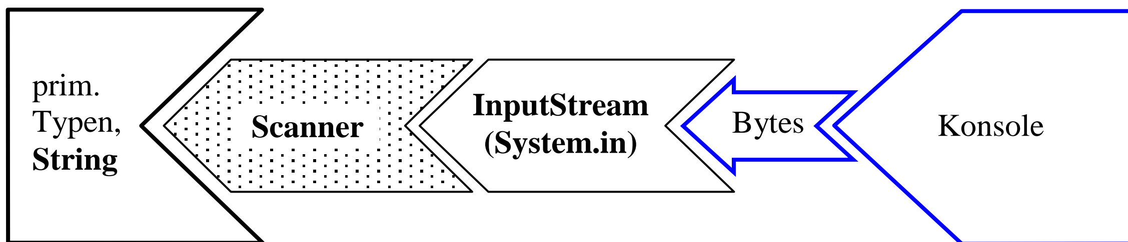
```
import java.io.IOException;
import java.nio.file.Paths;
import java.util.*;

class TokensScannen {
   public static void main(String[] args) throws IOException {
      try (Scanner input = new Scanner(Paths.get("Eingabe.txt"))) {
         while (input.hasNext())
            if (input.hasNextInt())
               System.out.println("int-Wert:    " + input.nextInt());
            else
               if (input.hasNextDouble())
                  System.out.println("double-Wert: " + input.nextDouble());
               else
                  System.out.println("Text:        " + input.next());
      }
   }
}
```

Über Parameter des **Scanner**-Konstruktors wählt man:

- die Eingabedatei per **Path**-Objekt (NIO.2 - API)
- die Kodierung (siehe Abschnitt 14.4.1.2, Voreinstellung: UTF-8)

14.8.4 Eingabe von der Konsole

Im Abschnitt 14.5 wird beschrieben, wie man Tastatureingaben mit Hilfe der Klasse **Scanner** entgegennimmt:

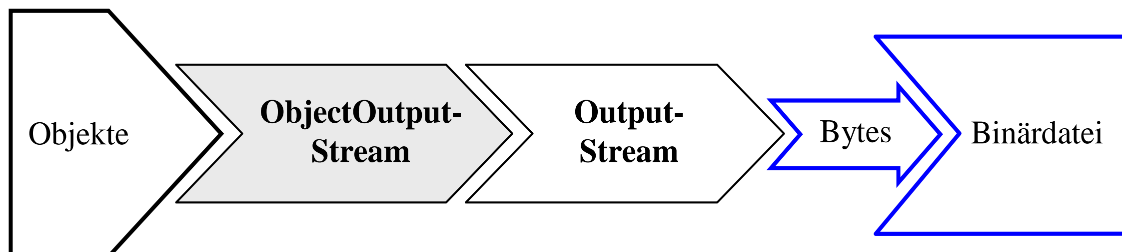


Beispiel:

```
Scanner input = new Scanner(System.in);
System.out.print("Ihr Alter: ");
int alter = input.nextInt();
```

14.8.5 Objekte (de)serialisieren

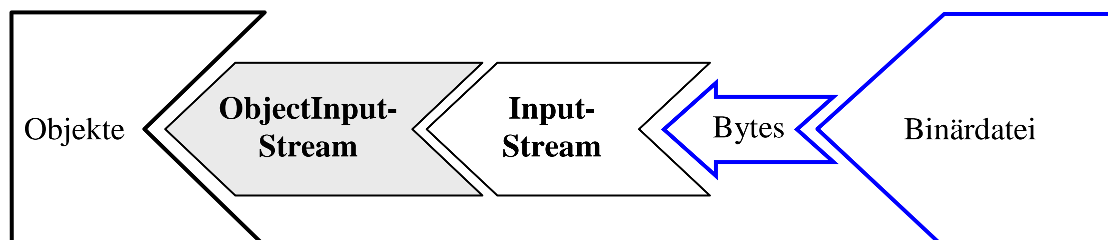
Um Objekte in eine Datei zu schreiben oder aus einer zu Datei lesen, verwendet man die im Abschnitt 14.6 vorgestellten Klassen **ObjectOutputStream** und **ObjectInputStream**. Hier ist die Ausgabe zu sehen:



Beispiel:

```
try (ObjectOutputStream oos = new ObjectOutputStream(
                              new FileOutputStream("test.ser"))) {
   oos.writeObject(kunde);
}
```

Mit der folgenden Datenstromkonstruktion holt man Objekte zurück:



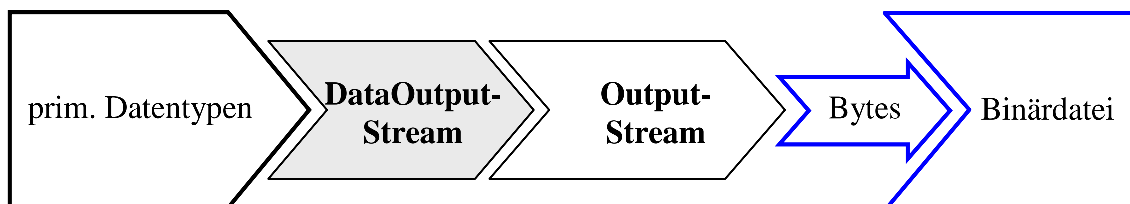
Beispiel:

```
try (ObjectInputStream ois =  new ObjectInputStream(
                              new FileInputStream("test.ser"))) {
   Kunde unbekannt = (Kunde) ois.readObject();
}
```

Wer die Verbindung zur Aus- bzw. Eingabedatei über das NIO.2 - API (vgl. Abschnitt 14.2.1) herstellen möchte, lässt sich von der statischen **Files**-Methode **newOutputStream()** einen **OutputStream** bzw. von der Methode **newInputStream()** einen **InputStream** liefern.

14.8.6 Werte mit primitiven Datentypen in eine Binärdatei schreiben

Um primitive Datentypen (z. B. **int**, **double**) binär in eine Datei zu schreiben, verwendet man ein Filterobjekt aus der Klasse **DataOutputStream** in Kombination mit einem Ausgabeobjekt aus der **OutputStream**-Hierarchie (siehe Abschnitt 14.3.1.4):

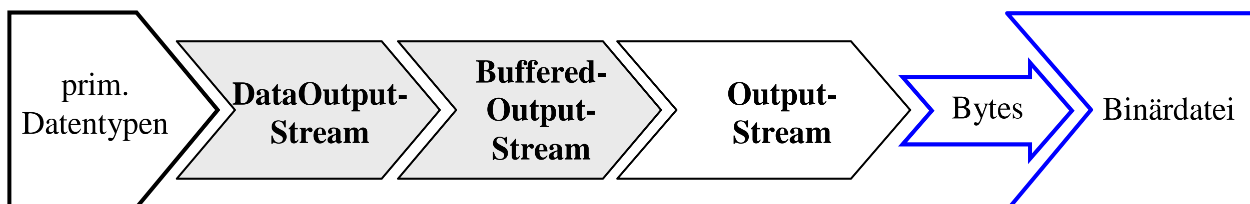


Um die Verbindung zur Ausgabedatei über das NIO.2 - API (vgl. Abschnitt 14.2.1) herzustellen, lässt man sich von der statischen **Files**-Methode **newOutputStream()** einen **OutputStream** liefern.

Beispiel:

```
try (DataOutputStream dos = new DataOutputStream(Files.newOutputStream(file))) {
   dos.writeInt(4711);
   dos.writeDouble(Math.PI);
}
```

Soll die Ausgabe gepuffert erfolgen, um die Anzahl der Dateizugriffe gering zu halten, dann muss ein Filterobjekt aus der Klasse **BufferedOutputStream** eingesetzt werden:



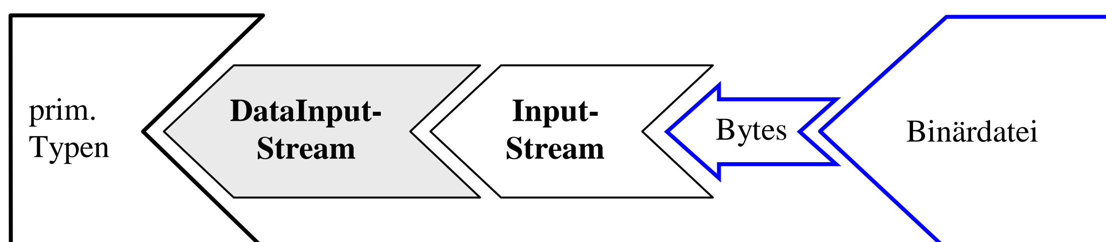
Im folgenden Beispiel wird ein Puffer mit 16384 Bytes Kapazität verwendet:

```
try (DataOutputStream dos = new DataOutputStream(
                                new BufferedOutputStream(
                                new FileOutputStream("demo.dat"), 16384))) {
     . . .
}
```

Ein Puffer muss auf jeden Fall vor dem Programmende entleert werden, was am einfachsten durch die **try** - Anweisung die mit automatischer Ressourcen-Freigabe zu realisieren ist.

14.8.7 Werte mit primitiven Datentypen aus einer Binärdatei lesen

Um primitive Datentypen (z. B. **int**, **double**) aus einer Binärdatei zu lesen, verwendet man ein Filterobjekt aus der Klasse **DataInputStream** in Kombination mit einem Eingabeobjekt aus der **InputStream** - Hierarchie:

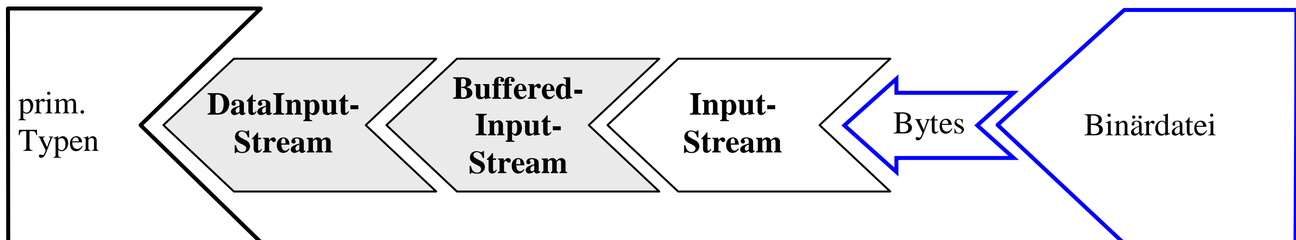


Um die Verbindung zur Eingabedatei über das NIO.2 - API (vgl. Abschnitt 14.2.1) herzustellen, lässt man sich von der statischen **Files**-Methode **newOnputStream()** einen **InputStream** liefern.

Beispiel:

```
Path file = Paths.get("demo.dat");
try (DataInputStream dis = new DataInputStream(Files.newInputStream(file))) {
    int i = dis.readInt();
    double d = dis.readDouble();
}
```

Soll die Eingabe gepuffert erfolgen, um die Anzahl der Dateizugriffe gering zu halten, dann muss ein Filterobjekt aus der Klasse **BufferedInputStream** eingesetzt werden:



Im folgenden Beispiel wird ein Puffer mit Kapazität von 16384 Bytes verwendet:

```
try (DataInputStream dis = new DataInputStream(
                            new BufferedInputStream(
                            new FileInputStream("demo.dat"), 16384))) {
    . . .
}
```

14.9 Übungsaufgaben zum Kapitel 14

1) Welche der folgenden Aussagen sind richtig bzw. falsch?

1. Die Klasse **PrintStream** spielt in aktuellen Java-Programmen keine Rolle mehr.
2. Ein geschlossener Datenstrom kann anschließend nicht mehr zur Ein- bzw. Ausgabe verwendet werden.
3. Die **PrintWriter**-Methoden werfen *keine* **IOException**, sondern setzen ein Fehlersignal, das mit der Methode **checkError**() abgefragt werden kann.
4. Bei der Textausgabe verwendet ein durch die **Files**-Fabrikmethode **newBufferedWriter()** erstelltes Datenstromobjekt per Voreinstellung die UTF-8 - Kodierung.

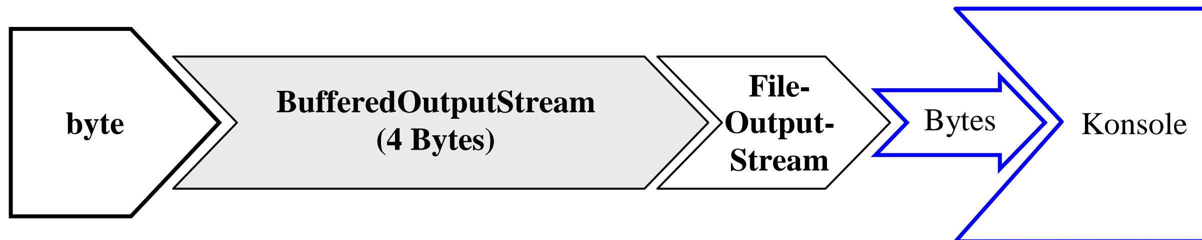
2) Die **FileInputStream**-Methode **read()** versucht, ein Byte aus der angeschlossenen Datei zu lesen. Warum verwendet sie den Rückgabetyp **int**?

3) Erstellen Sie ein Programm zur Demonstration der Ausgabepufferung. Um mitverfolgen zu können, wie bei erschöpfter Pufferkapazität Daten weitergeleitet werden, sollten Sie als Senke die Konsole verwenden.

Wie Sie aus dem Abschnitt 14.3.1.6 wissen, ist der per **System.out** ansprechbare **PrintStream** mit aktivierter **autoFlush**-Option hinter einen **BufferedOutputStream** mit 128 Bytes Puffergröße geschaltet, was insgesamt keine guten Beobachtungsmöglichkeiten bietet. Als Alternative mit besseren Forschungsmöglichkeiten wird daher folgende Ausgabestromkonstruktion vorgeschlagen:

```
FileOutputStream fos =
   new FileOutputStream(FileDescriptor.out);
BufferedOutputStream bos =
   new BufferedOutputStream(fos, 4);
```

Über die statische Variable **out** der Klasse **FileDescriptor** wird der Bezug zur Konsole hergestellt. Dorthin schreibt der **FileOutputStream** `fos`, an den der **BufferedOutputStream** `bos` mit der untypisch kleinen Puffergröße von 4 Bytes gekoppelt ist:



Wir kommen mit der **BufferedOutputStream**-Methode **write()** aus, wenn die auszugebenden Bytes so gewählt werden, dass eine interpretierbare Bildschirmausgabe entsteht. Dies ist z. B. bei folgendem Aufruf der Fall:

```
bos.write(i+47);
```

Bei `i` = 1 wird das niederwertigste Byte der **int**-Zahl 48 (= 0x30) in den Ausgabestrom geschoben. Dieses ist in jedem 8-Bit-Zeichensatz die Kodierung der Null, sodass diese Ziffer auf der Konsole erscheint. Bei `i` = 2 erscheint dementsprechend eine Eins usw.

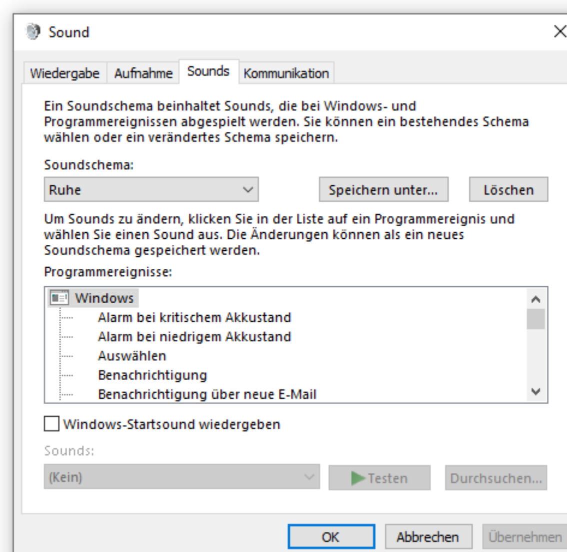
Jetzt müssen Sie nur noch per „Zeitlupe“ dafür sorgen, dass man das Füllen und Entleeren des Puffers mitverfolgen kann, z. B.:

```
long time, start = System.currentTimeMillis();
for (byte i = 1; i <= 10; i++) {
   time = start + i*1000;
   while (System.currentTimeMillis() < time);
   bos.write(i+47);
   System.out.print('\u0007');
}
```

Im Lösungsvorschlag wird das Steuerzeichen \u0007 an die Konsole gesendet,

```
System.out.print('\u0007');
```

um per Ton die Ankunft eines Bytes im Puffer zu melden. Damit in einem Konsolenprogramm unter Windows von dieser Anweisung tatsächlich etwas zu hören ist, muss (über **Systemsteuerung > Sound**) ein geeignetes Soundschema aktiviert sein, z. B.:



Wird ein Konsolenprogramm in IntelliJ ausgeführt, produziert die **print**() - Ausgabe des Steuerzeichens \u0007 allerdings keinen Ton. Stattdessen erscheint in der Konsole ein Rechteck, was zur Demonstration der Ausgabepufferung sogar recht nützlich ist, z. B.:

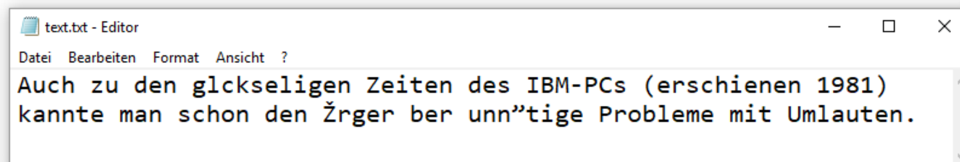
```
□□□□0123□□□□4567□□

Rest aus dem Puffer:
89
```

Sollten sich bei Programmende noch Bytes im Puffer befinden, müssen diese per **flush**() oder **close**() vor dem Untergang bewahrt werden.

4) Wie kann man beim folgenden Programm den Quellcode vereinfachen und dabei auch noch die Laufzeit erheblich reduzieren?

```
import java.io.*;

class AutoFlasche {
  public static void main(String[] egal) throws IOException {
    try (PrintWriter pw = new PrintWriter(new FileOutputStream("pw.txt"), true)) {
      long time = System.currentTimeMillis();
      for (int i = 1; i < 50_000; i++) {
        pw.println(i);
      }
      System.out.println("Zeit: " + (System.currentTimeMillis()-time));
    }
  }
}
```

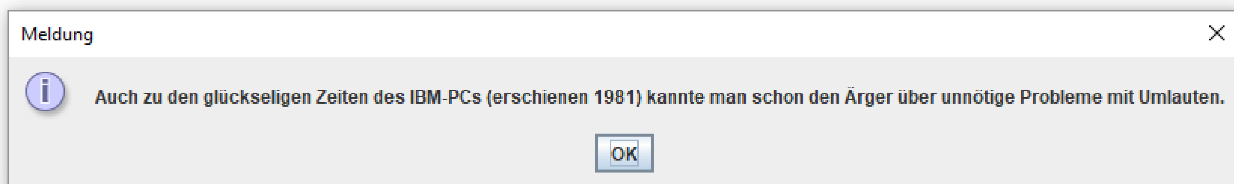
5) Schreiben Sie ein Programm, das den Text (hier unter Windows 10 von Notepad bei fehlerhafter Annahme einer ANSI-Kodierung angezeigt)



in der Datei

...\BspUeb\IO\MS-DOS\text.txt

einlesen und korrekt in einem **JOptionPane**-Meldungsfenster darstellen kann:



6) Erstellen Sie eine Klasse zur Verwaltung einer Datenmatrix bestehend aus den Messwerten von *k* Merkmalen bei *n* Fällen. Verwenden Sie zur Aufbewahrung der Messwerte einen zweidimensionalen **double**-Array. Zu jedem Merkmal soll außerdem ein Name gespeichert werden. Objekte der Klasse sollten Daten aus einer Textdatei nach folgendem Muster aufnehmen können:

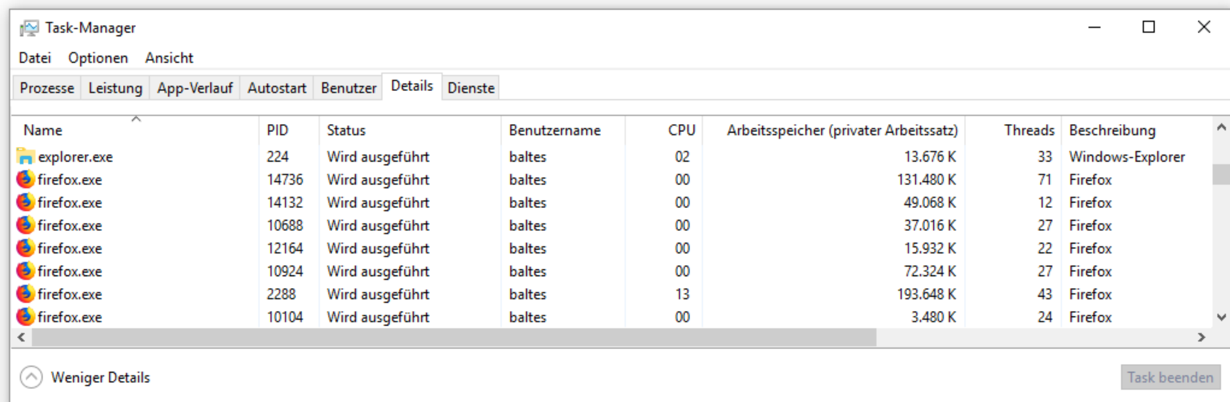
```
nr temp alter gewicht
1 12,3 74,5 123,9
2 11,2 34,4 156,7
3  7,2 83,5 142,1
4 45,2 17,2 129,8
5  1,2 44,4 216,7
6 17,2 23,5 132,1
7 12,2 42,1 182,2
```

In der ersten Zeile stehen die Namen der Merkmale.

15 Multithreading

Wir sind längst daran gewöhnt, dass moderne Betriebssysteme mehrere Programme (Prozesse) parallel ausführen können, sodass z. B. ein längerer Ausdruck keine Zwangspause zur Folge hat. Während der Druckertreiber die Ausgabeseiten aufbaut, kann z. B. ein Java-Programm entwickelt oder im Internet recherchiert werden. Weil in der Regel weniger Prozessoren bzw. virtuelle Prozessorkerne vorhanden sind als arbeitswillige Programme, muss das Betriebssystem die verfügbare CPU-Leistung nach einem Zeitscheibenverfahren auf die rechenwilligen Programme verteilen. Dadurch reduziert sich zwar die Ausführungsgeschwindigkeit jedes Programms, doch ist in den meisten Anwendungen trotzdem ein flüssiges Arbeiten möglich.

Als Ergänzung zum gerade beschriebenen **Multitasking**, das ohne Zutun der Anwendungsprogrammierer vom Betriebssystem bewerkstelligt wird, ist es oft sinnvoll oder gar unumgänglich, auch *innerhalb* einer Anwendung nebenläufige *Ausführungsfäden* zu realisieren, wobei man hier vom **Multithreading** spricht. Bei einem Internet-Browser muss man z. B. nach dem Anstoßen eines längeren Downloads nicht untätig den Fortschrittsbalken im Download-Fenster anstarren, sondern kann parallel mit anderen Fenstern arbeiten. Wie unter Windows die Detailsanzeige im Task-Manager zeigt, sind z. B. bei einer typischen Verwendung des Internet-Browsers Firefox zahlreiche Threads aktiv, wobei die Anzahl ständig schwankt:[1]



Bei einer GUI-Anwendung sorgt die Multithreading-Technik dafür, dass die Bedienoberfläche auch dann noch auf Benutzereingaben reagiert, wenn im Hintergrund ein zeitaufwändiger Auftrag erledigt wird. Eine Server-Anwendung kann dank Multithreading mehrere Klienten simultan versorgen.

Die Multithreading-Technik kommt aber nicht nur dann in Frage, wenn eine Anwendung mehrere Aufgaben gleichzeitig erledigen soll, damit der Prozess nicht durch eine langsame Aufgabe blockiert wird. Weil auf einem Rechner meist mehrere Prozessoren oder Prozessorkerne verfügbar, sollten aufwändige Einzelaufgaben (z. B. das Rendern einer 3D-Ansicht, Virenanalyse einer kompletten Festplatte) in Teilaufgaben zerlegt und auf mehrere CPU-Kerne verteilt werden. Weil die CPU-Hersteller bei der Taktbeschleunigung an physikalische Grenzen gestoßen sind, konzentrieren sie sich seit vielen Jahren darauf, durch eine höhere Anzahl von CPU-Kernen eine Leistungssteigerung zu erzielen. Mittlerweile (2020) sind 4 reale Kerne zum Standard geworden, und viele CPUs der Hersteller AMD und Intel besitzen dank der SMT-Technik (*Simultaneous Multi-Threading*, bei Intel als *Hyper-Threading* bezeichnet) doppelt so viele logische CPU-Kerne. Multi-Core - CPUs erhöhen den Druck auf die Software-Entwickler, per Multithreading für gut skalierende Anwendungen zu sorgen, die auf einem Multi-Core - Rechner deutlich schneller laufen als auf einem Single-Core - Rechner.

[1] Mittlerweile verwenden manche Anwendungen wie z. B. der Firefox-Browser (aus der Quantum-Generation) auch mehrere *Prozesse*, um die Stabilität zu steigern.

Beim Multithreading ist allerdings eine sorgfältige Einsatzplanung erforderlich, denn:

- Das Erstellen, Terminieren, Blockieren und Reaktivieren von Threads ist zeitaufwändig, sodass der Zeitaufwand für eine eher kleine Aufgabe durch das Multithreading sogar steigen kann.
- Threads belegen Speicher für ihren individuellen Stack (z. B. per Voreinstellung 1 MB bei einer 64-Bit - JVM unter Windows), sodass ihre Zahl nicht zu groß werden sollte.[1]
- Die Multi-Thread - Programmierung ist erheblich anspruchsvoller als die Single-Thread - Programmierung. Das gilt vor allem dann, wenn mehrere Threads auf gemeinsame, variable Datenbestände zugreifen, sodass eine Synchronisation der Threads erforderlich ist (siehe Abschnitt 15.2). Hier kommt es oft zu Fehlern, wobei ein Programm ...
    - entweder hängt (Endlosschleife oder Deadlock, siehe Abschnitt 15.3.4)
    - oder fehlerhafte Ergebnisse produziert, was noch weit gravierender ist.

  Eine fehlerhafte Thread-Synchronisation ist zudem aufgrund variabler Folgen schwer zu analysieren.

Während jeder *Prozess* einen eigenen Adressraum besitzt, laufen die *Threads* eines Programms im selben Adressraum ab, sodass sie gelegentlich auch als *leichtgewichtige Prozesse* bezeichnet werden. Sie verwenden einen gemeinsamen Heap-Speicher, wobei aber jeder Thread als selbständiger Kontrollfluss bzw. Ausführungsfaden einen eigenen Stack-Speicher benötigt.

In Java ist das Multithreading in Sprache, Standardbibliothek und Laufzeitumgebung integriert, also ohne großen Aufwand zu nutzen. Um diese Technik erfolgreich einzusetzen, muss man sich allerdings mit neuen Konzepten und Aufgaben vertraut machen.

Übrigens sind bei *jeder* Java - Anwendung mehrere Threads aktiv; so läuft z. B. der Garbage Collector stets in einem eigenen Thread.

15.1 Start und Ende eines Threads

Das direkte Erzeugen von Threads über Objekte aus der Klasse **Thread** wird zunehmend abgelöst durch die Nutzung von Frameworks, die im Hintergrund mit Multithreading-Techniken arbeiten. Allerdings erleichtert es der traditionelle Direktkontakt mit Threads, grundlegende Eigenschaften der Technik kennenzulernen. Wir verwenden ein Beispiel mit Produzenten-Konsumenten - Struktur, um (im Abschnitt 15.1) den Start und das Ende eines Threads sowie (im Abschnitt 15.2) die Koordination von zwei Threads veranschaulichen. Dabei ist von den potentiellen Vorteilen einer Multithreading - Lösung noch nicht viel zu sehen sein wird. Wie bei vielen ambitionierten Programmiertechniken kann man mit kleinen Beispielen zwar das Grundprinzip gut veranschaulichen, aber den Nutzen nicht nachweisen.

15.1.1 Die Klasse Thread

Ein Thread wird in Java durch ein Objekt aus der Klasse **Thread** oder aus einer abgeleiteten Klasse realisiert. Im ersten Beispiel werden die Klassen `ProThread` und `KonThread` aus der Klasse **Thread** abgeleitet. Sie sollen einen Produzenten bzw. einen Konsumenten modellieren, die alternierend auf einen gemeinsamen Lagerbestand einwirken, der von einem Objekt der Klasse `Lager` gehütet wird. Wir betrachten zunächst die (*nicht* von Thread abstammende) Klasse `Lager`:

[1] https://www.oracle.com/technetwork/java/hotspotfaq-138619.html

```
class Lager {
   private static final int MANZ = 20;
   private int bilanz;
   private int anz;

   Lager(int start) {
      bilanz = start;
      System.out.println("Der Laden ist offen (Bestand = " + bilanz + ")\n");
   }

   boolean istOffen() {
      if (anz < MANZ)
         return true;
      else {
         System.out.println("\nLieber " + Thread.currentThread().getName()+
            ", es ist Feierabend!");
         return false;
      }
   }

   private String formZeit() {
      return java.text.DateFormat.getTimeInstance().format(new java.util.Date());
   }

   void ergaenze(int add) {
      bilanz += add;
      anz++;
      System.out.println("Nr. " + anz + ":\t" + Thread.currentThread().getName()+
         " ergänzt\t" + add + "\tum " + formZeit() + " Uhr. Stand: " + bilanz);
   }

   void liefere(int sub) {
      bilanz -= sub;
      anz++;
      System.out.println("Nr. " + anz + ":\t" + Thread.currentThread().getName()+
         " entnimmt\t" + sub + "\tum " + formZeit() + " Uhr. Stand: " + bilanz);
   }
}
```

Die für Klassen im selben Paket verfügbaren `Lager`-Methoden werden vom Produzenten und/oder vom Konsumenten verwendet und im entsprechenden Kontext näher erläutert:

- `istOffen()`
  Der Aufrufer erfährt, ob das Lager noch geöffnet ist.
- `ergaenze()`
  Der Produzent erhöht mit dieser Methode den Lagerbestand.
- `liefere()`
  Der Konsument reduziert mit dieser Methode den Lagerbestand.

Das folgende Hauptprogramm erzeugt ein `Lager`-Objekt mit initialem Bestand

```
class ProKonDemo {
   public static void main(String[] args) {
      Lager lager = new Lager(100);
      ProThread pt = new ProThread(lager);
      KonThread kt = new KonThread(lager);
      pt.start();
      kt.start();
   }
}
```

und generiert dann ein `ProThread`- sowie ein `KonThread`-Objekt. Weil beide Threads mit dem `Lager`-Objekt kooperieren sollen, erhalten sie als Konstruktor-Parameter eine entsprechende Referenz.

Anschließend werden die beiden Threads vom Zustand **new** durch Aufruf ihrer **start()** - Methode in den Zustand **ready** gebracht:

```
pt.start();
kt.start();
```

Von der **start()** - Methode eines Threads wird seine **run()** - Methode aufgerufen, welche die im Thread auszuführenden Anweisungen enthält. Eine aus **Thread** abgeleitete Klasse muss also die **run()** - Methode überschreiben. Es folgt endlich der Quellcode der Klasse `ProThread`:

```
class ProThread extends Thread {
   private Lager lager;

   ProThread(Lager lager) {
      super("Produzent");
      this.lager = lager;
   }

   @Override
   public void run() {
      while (lager.istOffen()) {
         lager.ergaenze((int) (5 + Math.random()*100));
         try {
            Thread.sleep((int) (1000 + Math.random()*3000));
         } catch(InterruptedException ie) {interrupt();}
      }
   }
}
```

In der **run()** - Methode der Klasse `ProThread` läuft eine **while**-Schleife so lange, bis die `Lager`-Methode `istOffen()` den Rückgabewert **false** liefert.

Ein Thread im Zustand **ready** wartet auf die Zuteilung der (bzw. einer) CPU und erreicht dann den Zustand **running**. Die JVM verwaltet die Threads in Zusammenarbeit mit dem Wirtsbetriebssystem, wobei ein Thread mehrfach zwischen den Zuständen **ready** und **running** wechseln kann (siehe Abschnitt 15.3.3.1).

Wenn seine **run()** - Methode beendet ist, befindet sich ein Thread im Zustand **terminated** und kann *nicht* erneut gestartet werden.

Es ist möglich, aber nicht empfehlenswert, einen Thread von außen mit der (mittlerweile abgewerteten) Methode **stop()** abzuwürgen (siehe Abschnitt 15.3.2.2).

Im Beispiel ergänzt der `ProThread` innerhalb einer **while**-Schleife das Lager um eine zufallsbestimmte Menge. Er spricht über die per Konstruktor-Parameter erhaltene Referenz das `Lager`-Objekt an und ruft dessen `ergaenze()` - Methode auf:

```
lager.ergaenze((int) (5 + Math.random()*100));
```

Anschließend legt er sich durch Aufruf der statischen **Thread**-Methode **sleep()** ein (wiederum zufallsabhängiges) Weilchen zur Ruhe:

```
Thread.sleep((int) (1000 + Math.random()*3000));
```

Durch Ausführen dieser Methode wechselt der Thread vom Zustand **running** zum Zustand **sleeping** und konkurriert vorübergehend *nicht* mehr um Prozessorzeit. Schlafphasen eignen sich wegen der unzuverlässigen, vom Wirtsbetriebssystem abhängigen Einhaltung der Zeiten übrigens nicht für eine präzise Programmablaufsteuerung.

Weil von der Methode **sleep()** potentiell eine überwachte **InterruptedException** zu erwarten ist, wird sie in einem **try**-Block ausgeführt. Zur Begründung eines plausiblen **catch**-Blocks, müssen wir etwas ausholen bzw. vorgreifen:

- Einem Thread kann durch einen Aufruf seiner **interrupt()** - Methode ein Unterbrechungssignal zugestellt werden (siehe Abschnitt 15.3.2). Ein kooperativer Thread prüft regelmäßig, ob das Unterbrechungssignal gesetzt ist, und beendet ggf. seine **run()** - Methode.
- Die Methode **sleep()** reagiert folgendermaßen auf das Unterbrechungssignal:
    - Sie hebt das Unterbrechungssignal auf!
    - Sie wirft eine **InterruptedException**.

Im **catch**-Block zur **InterruptedException** sollte in der Regel das Unterbrechungssignal restauriert werden, damit auf einer höheren Ebene darauf reagiert werden kann. Im aktuellen Zustand des Beispielprogramms hat diese Maßnahme zwar noch keine Bedeutung, doch sollten wir uns schon jetzt eine akzeptable Behandlung der **InterruptedException** angewöhnen.

Zum `ProThread`-Konstruktor ist noch anzumerken, dass durch einen Aufruf des Superklassen-Konstruktors ein Thread-Name festgelegt wird.

Der Konsumenten-Thread des Beispielprogramms ist weitgehend analog definiert:

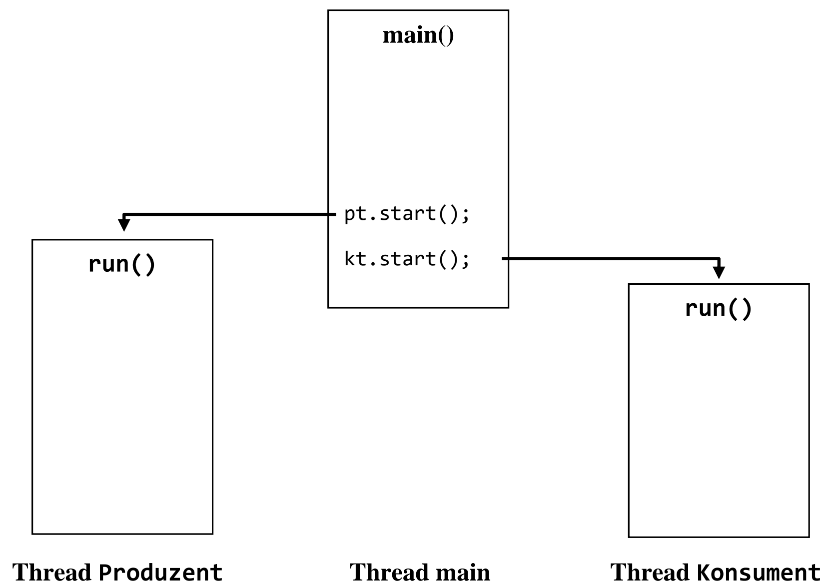
```
class KonThread extends Thread {
   private Lager lager;

   KonThread(Lager lager) {
      super("Konsument");
      this.lager = lager;
   }

   @Override
   public void run() {
      while (lager. istOffen()) {
         lager.liefere((int) (5 + Math.random()*100));
         try {
            Thread.sleep((int) (1000 + Math.random()*3000));
         } catch(InterruptedException ie) {interrupt();}
      }
   }
}
```

Statt den Lagerbestand zu ergänzen, bezieht der Konsument in seiner **run()** - Methode Lieferungen.

In beiden **run()** - Methoden wird vor jedem Schleifendurchgang geprüft, ob das Lager noch offen ist. Nach Dienstschluss des Lagers (im Beispiel: nach 20 Ein- oder Auslieferungen) enden beide **run()** - Methoden und damit auch die zugehörigen Threads.

Auch der automatisch zur Ausführung der Startmethode des Programms kreierte Thread **main** ist zu diesem Zeitpunkt bereits Geschichte. Die Aufrufe der **Thread**-Methode **start()** kehren praktisch unmittelbar zurück, und anschließend endet mit der der **main()** - Methode auch der **main** - Thread:[1]

[1] Nachdem Sie unter Windows ein Java-Programm in einem Konsolenfenster gestartet haben, können Sie mit der Tastenkombination **Strg+Pause** eine Liste seiner aktiven Threads anfordern.



Wenn die drei Benutzer-Threads abgeschlossen sind, endet auch das Programm.[1]

In den beiden Ausführungsfäden `Produzent` bzw. `Konsument` führt ein `ProThread`- bzw. ein `KonThread`-Objekt seine **run**() - Methode aus, wobei das `Lager`-Objekt wesentlich zum Einsatz kommt:

- In seiner Methode `istOffen()`, die in beiden Threads aufgerufen wird, entscheidet es auf Anfrage, ob weitere Veränderungen des Lagers möglich sind.
- Die Methoden `ergaenze()` und `liefere()` erhöhen oder reduzieren den Lagerbestand, aktualisieren die Anzahl der Lagerveränderungen und protokollieren jede Maßnahme.
  Zur Formulierung des Protokolleintrags besorgen sich die Methoden mit der statischen **Thread**-Methode **currentThread**() eine Referenz auf den Thread, in dem sie ausgeführt werden, und stellen per **getName()** dessen Namen fest.
- Mit Hilfe der privaten `Lager`-Methode `formZeit()` erhält das Ereignisprotokoll formatierte Zeitangaben.

In einem typischen Ablaufprotokoll des Programms zeigen sich einige Ungereimtheiten, verursacht durch das unkoordinierte Agieren des Produzenten- und des Konsumenten-Threads:

[1] Neben den bisher behandelten Benutzer-Threads sind in einem Java-Programm noch sogenannte *Daemon-Threads* aktiv, die meist von der JVM gestartet werden und ein Programm *nicht* am Leben erhalten können (siehe Abschnitt 15.8.1).

```
Der Laden ist offen (Bestand = 100)

Nr. 1:     Produzent ergänzt          72      um 12:43:33 Uhr. Stand: 74
Nr. 2:     Konsument entnimmt         98      um 12:43:33 Uhr. Stand: 74
Nr. 3:     Konsument entnimmt         31      um 12:43:35 Uhr. Stand: 43
Nr. 4:     Produzent ergänzt          32      um 12:43:37 Uhr. Stand: 75
Nr. 5:     Konsument entnimmt         42      um 12:43:38 Uhr. Stand: 33
Nr. 6:     Produzent ergänzt          44      um 12:43:39 Uhr. Stand: 77
Nr. 7:     Konsument entnimmt         63      um 12:43:41 Uhr. Stand: 14
Nr. 8:     Produzent ergänzt          42      um 12:43:42 Uhr. Stand: 56
Nr. 9:     Konsument entnimmt         99      um 12:43:43 Uhr. Stand: -43
Nr. 10:    Produzent ergänzt          77      um 12:43:44 Uhr. Stand: 34
Nr. 11:    Konsument entnimmt         13      um 12:43:44 Uhr. Stand: 21
Nr. 12:    Konsument entnimmt         83      um 12:43:47 Uhr. Stand: -62
Nr. 13:    Produzent ergänzt          90      um 12:43:47 Uhr. Stand: 28
Nr. 14:    Produzent ergänzt          47      um 12:43:48 Uhr. Stand: 75
Nr. 15:    Konsument entnimmt         101     um 12:43:51 Uhr. Stand: -26
Nr. 16:    Produzent ergänzt          42      um 12:43:51 Uhr. Stand: 16
Nr. 17:    Konsument entnimmt         79      um 12:43:52 Uhr. Stand: -63
Nr. 18:    Produzent ergänzt          22      um 12:43:53 Uhr. Stand: -41
Nr. 19:    Konsument entnimmt         90      um 12:43:56 Uhr. Stand: -131
Nr. 20:    Produzent ergänzt          54      um 12:43:57 Uhr. Stand: -77

Lieber Konsument, es ist Feierabend!

Lieber Produzent, es ist Feierabend!
```

U.a. fällt negativ auf:

- Im ersten Protokolleintrag wird berichtet, dass vom Startwert 100 ausgehend eine Ergänzung von 72 Einheiten zu einem Bestand von 74 Einheiten geführt habe.
- Der zweite Eintrag behauptet, dass die Entnahme von 98 Einheiten ohne Effekt auf den Lagerbestand geblieben sei.
- Zwischenzeitlich wird der Bestand mehrmals negativ, was in einem realen Lager nicht passieren kann.

Ansonsten zeigt die Verzahnung der beiden Threads keine ausgeprägte Regelmäßigkeit, sondern demonstriert den Indeterminismus bei einem Multithreading - Programmablauf.

Im Abschnitt 15.2 werden Techniken zur Koordination bzw. Synchronisation von Threads vorgestellt, mit denen man fehlerhafte Anzeigen und Schlimmeres verhindern kann.

15.1.2 Das Interface Runnable

Als Basis für einen eigenständigen Kontrollfluss haben wir bisher eine **Thread**-Ableitung definiert und die geerbte **run()** - Methode überschrieben. In Java sind aber auch andere Klassen Thread-fähig, sofern sie das Interface **Runnable** implementieren. Dieses Interface verlangt von implementierenden Klassen eine Instanzmethode ...

- namens **run()**
- ohne Parameter
- mit Rückgabetyp **void**,
- die keine deklarationspflichtige Ausnahmen wirft.

Durch das Interface **Runnable** können Ableitungen beliebiger Basisklassen die Thread-Fähigkeit erwerben, was für die Flexibilität der Programmierung sehr wichtig ist.

Wir verwenden weiterhin das Produzenten-Konsumenten - Beispiel aus Abschnitt 15.1.1, ersetzen aber die **Thread**-Ableitung `ProThread`

```
class ProThread extends Thread {
   . . .
}
```

durch die Klasse `Produzent`, die das Interface **Runnable** implementiert:

```
class Produzent implements Runnable {
   private Lager lager;

   Produzent(Lager lager) {
      this.lager = lager;
   }

   public void run() {
      while (lager. istOffen()) {
         lager.ergaenze((int) (5 + Math.random()*100));
         try {
            Thread.sleep((int) (1000 + Math.random()*3000));
         } catch(InterruptedException ie) {Thread.currentThread().interrupt();}
      }
   }
}
```

Solange die Klasse `Produzent` keine spezielle Basisklasse erweitert, bleibt der potentielle Vorteil der **Runnable**-Konstruktion im Beispiel allerdings ungenutzt.

Im Rumpf der `Produzent`-Definition sind im Vergleich zur `ProThread`-Lösung nur zwei Änderungen erforderlich:

- Im Konstruktor der Klasse `Produzent` kann der Produzenten-Thread keinen Namen erhalten. Eine alternative Möglichkeit zur Benennung wird gleich vorgestellt.
- In der Methode **run()** muss im **catch**-Block zur Behandlung der **InterruptedException** der aktive Thread, an den der **interrupt()** - Appell zu richten ist (zur Begründung siehe Seite 727), mit Hilfe der statischen **Thread**-Methode **currentThread()** ermittelt werden, weil die Klasse des handelnden Objekts keine **Thread**-Ableitung ist, also die **interrupt()** - Methode nicht in ihrem Handlungsrepertoire hat.

Auch wenn die in einem neuen Thread auszuführende **run()** - Methode zu einer beliebigen, das Interface **Runnable** implementierenden Klasse gehört, wird zum *Erzeugen* des Ausführungsfadens doch ein **Thread**-Objekt benötigt, wobei man der passenden Konstruktor-Überladung einen Aktualparameter vom Typ **Runnable** übergibt:

- **public Thread(Runnable** *target***)**
- **public Thread(Runnable** *target***, String** *name***)**

Optional kann man zusätzlich den Namen des neuen Threads festlegen. Dies geschieht in der Startklasse des aktualisierten Beispiels, wo im Vergleich zur vorherigen Lösung nur eine einzige Zeile zu ändern ist:

```
class ProKonDemo {
   public static void main(String[] args) {
      Lager lager = new Lager(100);
      Thread pt = new Thread(new Produzent(lager), "Produzent");
      KonThread kt = new KonThread(lager);
      pt.start();
      kt.start();
   }
}
```

Nun machen wir uns daran, im Produzenten-Konsumenten - Beispiel die beiden Threads so zu synchronisieren, dass keine wirren Protokolleinträge und keine negativen Lagerbestände mehr auftreten.

15.2 Threads koordinieren

In diesem Abschnitt werden Techniken zur Koordination von einzelnen, explizit erstellten Threads vorgestellt. Obwohl Java moderne Frameworks bietet, um die aufwändige und fehleranfällige manuelle Erstellung und Verwaltung von Threads zu reduzieren und gleichzeitig das Potential der Multithreading-Technik besser auszuschöpfen (siehe Abschnitte 15.4 und **Fehler! Verweisquelle konnte nicht gefunden werden.**), sind die im aktuellen Abschnitt vorgestellten Begriffe und Verfahren doch weiter relevant:

- Es gibt individuelle Problemstellungen, welche die Anwendungsvoraussetzungen der Frameworks nicht erfüllen.
- Für die erfolgreiche Entwicklung von Multithreading-Anwendungen ist ein Verständnis der involvierten Begriffe und Verfahren sehr nützlich.

Die Koordination von Threads ist vor allem dann anspruchsvoll und fehleranfällig, wenn die Threads auf gemeinsame, variable Datenbestände zugreifen. Aufgrund einer fehlerhaften Thread-Koordination kann ein Programm ...

- außer Kontrolle geraten (z. B. in eine Endlosschleife oder einen Deadlock, siehe Abschnitt 15.3.4)
- oder fehlerhafte Ergebnisse produzieren, was in der Regel noch gravierender ist.

Eine fehlerhafte Thread-Koordination manifestiert sich oft unregelmäßig (z. B. bei einer Race Condition, siehe Abschnitt 15.2.1.1), was die Analyse erschwert.

Von Problemen beim Multithreading bleibt man weitgehend verschont, wenn die Threads entweder gar keine oder nur unveränderliche Daten teilen.

15.2.1 Fehlerhafte oder veraltete Daten

Wenn mehrere Threads schreibend und/oder lesend auf gemeinsame Daten zugreifen (z. B. auf Instanzvariablen von Objekten auf dem Heap), dann können Daten beschädigt werden, oder ein Thread kann durch die Verwendung von veralteten Daten ein fehlerhaftes Verhalten zeigen.

15.2.1.1 Fehlerhafte Daten wegen nicht-atomarer Operationen

Am Anfang des im Abschnitt 15.1.1 wiedergegebenen Ablaufprotokolls zum Produzenten-Konsumenten - Beispiel stehen zwei „wirre" Einträge, die folgendermaßen durch eine so genannte *Race Condition* zu erklären sind:

- Der (zuerst gestartete) Produzenten-Thread nimmt nach einer erfolgreichen `istOffen()` - Anfrage die Methode `ergaenze()` in Angriff und führt die Anweisung
  ```
  bilanz += add;
  ```
  aus, was zur Zwischenbilanz von 172 führt.
- Dann muss der Produzent seine Arbeit unterbrechen, weil der Konsumenten-Thread aktiviert, d.h. vom Zustand **ready** in den Zustand **running** befördert wird.
- Mit seiner Anforderung von 98 Einheiten bringt der Konsument in der Methode `liefere()` die Lagerbilanz von 172 auf 74.
- Nach dem nächsten Thread-Wechsel macht der Produzent mit seiner Protokollausgabe weiter, wobei aber der *aktuelle* `bilanz`-Wert (unter Berücksichtigung der zwischenzeitlichen Konsumenten-Aktivität) erscheint.
- Schließlich vervollständigt der Konsumenten-Thread seine Meldung.

Es kann nicht nur zu wirren Protokolleinträgen kommen, sondern auch zu einem fehlerhaften `bilanz`-Wert. Scheinbar einschrittige Operationen wie die folgende Anweisung in der vom Produzenten-Thread aufgerufenen Methode `ergaenze()`

```
bilanz += add;
```

haben in einen Rechner mehrere Teilschritte zur Folge, sind also nicht **atomar** (also nicht geschützt vor Unterbrechungen durch andere Threads), z. B.:

- aktuellen `bilanz`-Wert aus dem Hauptspeicher in ein CPU-Register einlesen
- Wert (der lokalen Kopie!) erhöhen
- Neuen Wert in den Hauptspeicher schreiben

In der vom Konsumenten-Thread aufgerufenen Methode `liefere()` führt die Anweisung

```
bilanz -= sub;
```

analog zu folgenden Teilschritten:

- aktuellen `bilanz`-Wert aus dem Hauptspeicher in ein CPU-Register einlesen
- Wert (der lokalen Kopie!) reduzieren
- Neuen Wert in den Hauptspeicher schreiben

Durch unglückliche Thread-Wechsel kann es z. B. zu folgender Sequenz kommen:

- Der Produzent liest den Wert 100.
- Der Konsument liest den Wert 100.
- Der Produzent erhöht seine `bilanz`-Kopie um 10 auf 110 und schreibt das Ergebnis in den Hauptspeicher.
- Der Konsument reduziert seine `bilanz`-Kopie um 10 auf 90 und schreibt das Ergebnis in den Hauptspeicher. Damit ist der Beitrag des Produzenten verloren gegangen.

Es kann sogar passieren, dass ein Thread beim Schreiben oder Schreiben eines **long**- oder **double**-Werts (64 Bit groß) unterbrochen wird, und dass schlussendlich die 64 Bits einer Variablen von zwei verschiedenen Threads stammen (Gosling et al. 2019, Abschnitt 17.7; siehe auch Abschnitt 15.2.6.1). Beim Schreiben einer Variablen mit maximal vier Bytes Speicherbedarf kann es nicht passieren, dass ein Thread unterbrochenen wird und somit ein irregulärer Wert entsteht. Wie der nächste Abschnitt zeigt, ist aber auch bei solchen Variablen eine Synchronisation erforderlich, wenn sie von mehreren Threads verwendet werden.

15.2.1.2 Veraltete Daten im lokalen Cache eines Threads

Bei einer korrekten Multithreading-Programmierung geht es nicht nur darum, parallele Schreibzugriffe mehrerer Threads auf gemeinsame Daten zu koordinieren. Wenn nur ein einziger Thread auf einen Datenbestand schreibend zugreift, kann trotzdem eine fehlerhafte Multithreading-Programmierung vorliegen, wenn andere Threads die Daten lesen und auf aktuelle (korrekte) Werte angewiesen sind. Diese von manchen Programmierern unerwartete Problematik resultiert aus den in modernen Computer-Architekturen realisierten Hauptspeichertechniken mit (mehrstufigen) Cache-Strategien. Über den verschiedenen Hardware-Architekturen liegt das **Java-Speichermodell** (engl.: *Java-Memory-Model*) das folgendermaßen charakterisiert werden kann (Gosling et al. 2019, Abschnitt 17.4; Kreft & Langer 2008a):

- Zwar teilen sich grundsätzlich alle Threads denselben Heap-Speicher, doch verwendet jeder Thread zur Beschleunigung seiner Speicherzugriffe einen lokalen Cache, der sich z. B. in der Nähe des verwendeten CPU-Kerns befindet.
- Beim Start eines Threads werden alle für ihn relevanten Daten in seinen lokalen Cache kopiert (Memory Refresh).
- Es ist garantiert, dass beim Beenden eines Threads der Inhalt seines lokalen Cache-Speichers in den Hauptspeicher zurückgeschrieben wird (Memory Flush). Es ist einer Implementation keinesfalls verboten, auch vor dem Thread-Ende einen Memory Flush durchzuführen, aber darauf darf man sich nicht verlassen.

- Ein Java-Compiler darf in einem Thread A die Anweisungen aus Performanz-Gründen umordnen, solange dies für die Single-Thread - Ausführung ohne Belang ist. Für einen Thread B kann es aber von Bedeutung sein, wenn er die Speichereffekte von Anweisungen im Thread A in unerwarteter Reihenfolge sieht.

Wenn mehrere Threads einen inkonsistenten Blick auf dieselben Daten haben, spricht man von einem *Speicherkonsistenzfehler* (engl.: *memory consistency error*). Zum Glück verhindern die im weiteren Verlauf von Abschnitt 15.2 behandelten Techniken zur Thread-Koordination nicht nur, dass ein Thread in einem ungünstigen Moment unterbrochen wird, sondern sie sorgen auch für Speicherkonsistenz. So wird z. B. ein Memory Flush durchgeführt, sobald ein Thread einen geschützten Code-Bereich verlassen oder geschützte Daten geändert hat. Somit ist garantiert, dass kritische Wertänderungen für andere Threads sofort sichtbar werden.

Neben den Techniken zur Thread-Koordination ist auch die **final**-Deklaration von Instanzvariablen relevant für die Sichtbarkeit von Daten für andere Threads (siehe Kreft & Langer 2008a):

- Nach der Ausführung eines Konstruktors kommt es zu einem partiellen Memory Flush, wobei die als **final** deklarierten Variablen und alle über **final** deklarierte Referenzen erreichbaren Objekte in den allgemeinen Hauptspeicher zurückgeschrieben werden.
- Beim ersten Lesezugriff auf eine finalisierte Instanzvariable kommt es zu einem partiellen Memory Refresh, wobei der Wert der Variablen und im Fall einer Referenz auch erreichbare Objekte aus dem allgemeinen Hauptspeicher in den lokalen Cache des lesenden Threads übertragen werden.

15.2.2 Per Monitor synchronisierte Code-Bereiche

Offenbar muss im Beispiel ...

- verhindert werden, dass zwei Threads simultan auf das Lager zugreifen,
- dafür gesorgt werden, dass jeder Thread bei seinen Operationen mit geteilten Daten die aktuellen, eventuell durch den jeweils anderen Thread veränderten Werte kennt.

Java bietet mehrere Techniken, um eine derartige Thread-Koordination zu realisieren, von denen im weiteren Verlauf von Abschnitt 15.2 ohne Anspruch auf Vollständigkeit beschrieben werden:

- Die traditionelle, einfach und sicher anwendbare und somit für viele Aufgabenstellungen nach wie vor empfehlenswerte Technik der **synchronisierten Code-Bereiche** wird im aktuellen Abschnitt 15.2.2 beschrieben.
- Wenn die synchronisierten Code-Bereiche nicht flexibel genug sind, kommt die im Abschnitt 15.2.3 beschrieben Technik der **expliziten Lock-Objekte** zum Einsatz.
- Im Abschnitt 15.2.4 werden Verfahren zur automatisierten Thread-Koordination für Produzenten-Konsumenten - Konstellationen beschrieben.
- Im Abschnitt 15.2.5 werden Klassen aus dem Paket **java.util.concurrent** zur Unterstützung von generellen Thread-Kooperations-Szenarien vorgestellt.

15.2.2.1 Synchronisierte Methoden und Blöcke

Bei vielen Aufgabenstellungen ist eine angemessene (z. B. hinreichend performante) Thread-Koordination mit dem von Java von Beginn an unterstützten **Monitor**-Konzept leicht zu realisieren. Zu einem Monitor kann jedes Objekt werden, wenn mindestens eine seiner Methoden mit dem Modifikator **synchronized** dekoriert ist.

Sobald ein Thread eine als **synchronized** deklarierte Methode eines noch freien Monitors aufruft, wird er zum Besitzer dieses Monitors. Man kann sich vorstellen, dass er den (einzigen) Schlüssel zum Überwachungsbereich des Monitors (= Menge der synchronisierten Methoden) an sich nimmt. In der englischen Literatur wird der Vorgang als *obtaining the lock* beschrieben. Versucht ein an-

derer Thread, eine der synchronisierten Methoden desselben Monitors aufzurufen, wird er in den Wartezustand versetzt (**waiting for monitor**, vgl. Abschnitt 15.3.3.2). Sobald der Monitor-Besitzer die **synchronized**-Methode beendet, kann ein wartender Thread den Monitor übernehmen und seine Arbeit fortsetzen. Die Freigabe erfolgt auch dann zuverlässig, wenn die **synchronized**-Methode mit einer unbehandelten Ausnahme endet.

Die Synchronisation per Monitor klappt auch bei statischen Methoden, wobei dasjenige Objekt die Monitorrolle übernimmt, das die Klasse in der JVM repräsentiert (siehe Abschnitt 15.3.4).

Bei einem Konstruktor ist der Modifikator **synchronized** *nicht* erlaubt. Daher sollte man im Konstruktor keine Referenz zum entstehenden Objekt veröffentlichen (z. B. über eine statische Variable), wenn interferierende Zugriffe aus anderen Threads zu befürchten sind. Ansonsten könnte zeitgleich zum noch aktiven Konstruktor ein anderer Thread auf die Instanzvariablen des entstehenden Objekts zugreifen und einen inkonsistenten Zustand bewirken.

In unserem Produzent-Lager-Konsument - Beispiel müssen die `Lager`-Methoden `istOffen()`, `ergaenze()` und `liefere()` als **synchronized** deklariert werden, weil sie auf mindestens eine von den beiden kritischen Variablen `bilanz` und `anz` lesend und/oder schreibend zugreifen, z. B.:

```
synchronized void ergaenze(int add) {
   bilanz += add;
   anz++;
   System.out.println("Nr. " + anz + ":\t" + Thread.currentThread().getName() +
      " ergänzt\t" + add + "\tum " + formZeit() + " Uhr. Stand: " + bilanz);
}
```

Nun unterbleiben die wirren Protokolleinträge, doch Ausflüge in negative Lagerzustände sind weiter möglich, z. B.:

```
Der Laden ist offen (Bestand = 100)

Nr. 1:    Produzent ergänzt       54      um 14:54:31 Uhr. Stand: 154
Nr. 2:    Konsument entnimmt      68      um 14:54:31 Uhr. Stand: 86
Nr. 3:    Konsument entnimmt      26      um 14:54:33 Uhr. Stand: 60
Nr. 4:    Produzent ergänzt       58      um 14:54:34 Uhr. Stand: 118
Nr. 5:    Konsument entnimmt      70      um 14:54:35 Uhr. Stand: 48
Nr. 6:    Produzent ergänzt       13      um 14:54:35 Uhr. Stand: 61
Nr. 7:    Konsument entnimmt      74      um 14:54:38 Uhr. Stand: -13
Nr. 8:    Produzent ergänzt       11      um 14:54:38 Uhr. Stand: -2
Nr. 9:    Konsument entnimmt      65      um 14:54:40 Uhr. Stand: -67
Nr. 10:   Produzent ergänzt       26      um 14:54:41 Uhr. Stand: -41
Nr. 11:   Konsument entnimmt      71      um 14:54:42 Uhr. Stand: -112
Nr. 12:   Produzent ergänzt       9       um 14:54:43 Uhr. Stand: -103
Nr. 13:   Konsument entnimmt      8       um 14:54:45 Uhr. Stand: -111
Nr. 14:   Produzent ergänzt       100     um 14:54:46 Uhr. Stand: -11
Nr. 15:   Konsument entnimmt      5       um 14:54:47 Uhr. Stand: -16
Nr. 16:   Produzent ergänzt       43      um 14:54:48 Uhr. Stand: 27
Nr. 17:   Konsument entnimmt      44      um 14:54:51 Uhr. Stand: -17
Nr. 18:   Produzent ergänzt       68      um 14:54:51 Uhr. Stand: 51
Nr. 19:   Konsument entnimmt      97      um 14:54:53 Uhr. Stand: -46
Nr. 20:   Konsument entnimmt      73      um 14:54:54 Uhr. Stand: -119

Lieber Produzent, es ist Feierabend!

Lieber Konsument, es ist Feierabend!
```

Befindet sich ein Thread in einem synchronisierten Bereich, darf er andere, vom *selben* Monitor geschützte Bereiche betreten, was bei verschachtelten oder rekursiven Methodenaufrufen relevant ist.

Neben dem **synchronized**-Modifikator für Methoden bietet Java auch den **synchronisierten Block**, wobei statt einer kompletten Methode nur eine einzelne Blockanweisung in den synchronisierten Bereich aufgenommen und ein beliebiges Objekt als Monitor angegeben wird. Um andere Threads

möglichst wenig zu behindern, muss ein Monitor so schnell wie möglich wieder frei gegeben werden. Daher kann ein möglichst klein gewählter synchronisierter Block günstiger sein als das Synchronisieren einer kompletten Methode.

Obwohl in der `Lager`-Klassendefinition des Produzenten-Konsumenten - Beispiels der **synchronized**-Modifikator gut geeignet ist, ersetzen wir ihn zu Demonstrationszwecken bei der Methode `ergaenze()` durch einen synchronisierten Block:

```
void ergaenze(int add) {
   synchronized (this) {
      bilanz += add;
      anz++;
      System.out.println("Nr. " + anz + ":\t" + Thread.currentThread().getName() +
         " ergänzt\t" + add + "\tum " + formZeit() + " Uhr. Stand: " + bilanz);
   }
}
```

Nach dem Schlüsselwort **synchronized** ist zwischen runden Klammern ein Objekt als Monitor explizit anzugeben, während bei Verwendung des **synchronized**-Modifikators zu einer Instanzmethode das ausführende Objekt diese Rolle automatisch übernimmt. Im Beispiel belassen wir über das Schlüsselwort **this** die Monitor-Rolle beim Lageristen. Der zu einem Monitor gehörige synchronisierte Bereich kann beliebig über synchronisierte Methoden und/oder Blöcke zusammengestellt werden.

Während einer per **sleep()** - Methode ausgelösten Ruhephase werden im Besitz eines Threads befindliche Monitore *nicht* zurückgegeben. Daher ist die **sleep()** - Methode in synchronisierten Bereichen zu vermeiden.

Es ist zu beachten, dass durch die Synchronisierung ein geschützter *Code*-Bereich entsteht, nicht aber ein geschützter *Speicher*bereich. Damit ein geschützter Speichbereich resultiert, müssen *alle* Code-Passagen mit Zugriff auf diesen Speicherbereich in die Zone mit exklusivem Zugriff einbezogen werden (Morrison 2005).

Im Hinblick auf die Sichtbarkeit von gemeinsamen Daten für andere Threads ist von erheblicher Relevanz, dass ...

- beim Betreten des geschützten Bereichs ein Memory Refresh stattfindet, sodass alle Daten im lokalen Cache des Threads aktuell sind,
- beim Verlassen des geschützten Bereichs ein Memory Flush stattfindet, sodass alle Inhalte aus dem lokalen Cache des ehemals berechtigten Threads in den allgemeinen Hauptspeicher übertragen werden (vgl. Abschnitt 15.2.1.2 zum Java-Speichermodell).

Per Synchronisation wird also ...

- einerseits verhindert, dass zwei Threads eine synchronisierte Methode bzw. einen synchronisierten Block gleichzeitig ausführen und dabei simultan auf gemeinsame Daten zugreifen,
- andererseits dafür gesorgt, dass ein Thread beim Betreten eines geschützten Bereichs alle Daten im aktuellen Zustand sieht.

15.2.2.2 Koordination per wait(), notify() und notifyAll()

Mit Hilfe der **Object**-Methoden **wait()** und **notify()** können in unserem Produzent-Lager-Konsument - Beispiel negative Lagerbestände verhindert werden: Trifft eine Konsumenten-Anfrage auf einen unzureichenden Lagerbestand, dann wird der Thread mit der Methode **wait()** in den Zustand **waiting** versetzt (vgl. Abschnitt 15.3.3.2). Die Methode **wait()** kann nur in einem synchronisierten Bereich, aufgerufen werden, z. B.:

```
synchronized void liefere(int sub) {
   while (bilanz < sub)
      try {
         System.out.println(Thread.currentThread().getName() +
            " muss warten: Keine " + sub + " Einheiten vorhanden.");
         wait();
      } catch (InterruptedException ie) {
         System.err.println(ie);
      }

   bilanz -= sub;
   anz++;
   System.out.println("Nr. " + anz + ":\t" + Thread.currentThread().getName() +
      " entnimmt\t" + sub + "\tum " + formZeit() + " Uhr. Stand: " + bilanz);
}
```

Dem wartenden Konsumenten-Thread wird der Monitor entzogen, sodass der Produzenten-Thread freie Bahn hat, den synchronisierten Block zu betreten und das Lager aufzufüllen.

Die **Object**-Methoden **notify()** bzw. **notifyAll()** befördern einen Thread bzw. alle Threads vom Zustand **waiting** in den Zustand **ready**:

- **public final void notify()**
  Ein auf den betroffenen Monitor wartender Thread wird in den Zustand **ready** versetzt, sobald der Aufrufer den synchronisierten Bereich verlassen hat. Die Entscheidung zwischen mehreren Kandidaten ist der JVM überlassen.
- **public final void notifyAll()**
  Alle auf den betroffenen Monitor wartenden Threads werden in den Zustand **ready** versetzt, sobald der Aufrufer den synchronisierten Bereich verlassen hat. Den Monitor können diese Threads natürlich nicht gleichzeitig erwerben, sondern nur nacheinander.

Wie **wait()** können auch **notify()** und **notifyAll()** nur in einem synchronisierten Bereich aufgerufen werden, z. B.:

```
synchronized void ergaenze(int add) {
   bilanz += add;
   anz++;
   System.out.println("Nr. " + anz + ":\t"+Thread.currentThread().getName() +
      " ergänzt\t" + add + "\tum " + formZeit() + " Uhr. Stand: " + bilanz);
   notify();
}
```

Nun produziert das Beispielprogramm nur noch realistische Lagerprotokolle, z. B.:

```
Der Laden ist offen (Bestand = 100)

Nr. 1:    Produzent ergänzt        29      um 15:21:21 Uhr. Stand: 129
Nr. 2:    Konsument entnimmt       78      um 15:21:21 Uhr. Stand: 51
Konsument muss warten: Keine 92 Einheiten vorhanden.
Nr. 3:    Produzent ergänzt        36      um 15:21:25 Uhr. Stand: 87
Konsument muss warten: Keine 92 Einheiten vorhanden.
Nr. 4:    Produzent ergänzt        10      um 15:21:27 Uhr. Stand: 97
Nr. 5:    Konsument entnimmt       92      um 15:21:27 Uhr. Stand: 5
Nr. 6:    Produzent ergänzt        39      um 15:21:29 Uhr. Stand: 44
Nr. 7:    Konsument entnimmt       24      um 15:21:29 Uhr. Stand: 20
Nr. 8:    Produzent ergänzt        6       um 15:21:31 Uhr. Stand: 26
Nr. 9:    Produzent ergänzt        30      um 15:21:32 Uhr. Stand: 56
Konsument muss warten: Keine 62 Einheiten vorhanden.
Nr. 10:   Produzent ergänzt        66      um 15:21:35 Uhr. Stand: 122
Nr. 11:   Konsument entnimmt       62      um 15:21:35 Uhr. Stand: 60
Nr. 12:   Produzent ergänzt        35      um 15:21:36 Uhr. Stand: 95
Konsument muss warten: Keine 97 Einheiten vorhanden.
```

```
Nr. 13:   Produzent ergänzt        98     um 15:21:39 Uhr. Stand: 193
Nr. 14:   Konsument entnimmt       97     um 15:21:39 Uhr. Stand: 96
Konsument muss warten: Keine 99 Einheiten vorhanden.
Nr. 15:   Produzent ergänzt        38     um 15:21:43 Uhr. Stand: 134
Nr. 16:   Konsument entnimmt       99     um 15:21:43 Uhr. Stand: 35
Nr. 17:   Konsument entnimmt       23     um 15:21:45 Uhr. Stand: 12
Nr. 18:   Produzent ergänzt        17     um 15:21:46 Uhr. Stand: 29
Konsument muss warten: Keine 74 Einheiten vorhanden.
Nr. 19:   Produzent ergänzt        87     um 15:21:49 Uhr. Stand: 116
Nr. 20:   Konsument entnimmt       74     um 15:21:49 Uhr. Stand: 42

Lieber Produzent, es ist Feierabend!

Lieber Konsument, es ist Feierabend!
```

Mit **notify**() bzw. **notifyAll**() wird mitgeteilt, dass eine neue Lage eingetreten sei. Ob ein reaktivierter Thread nun die benötigten Voraussetzungen für seine Tätigkeit vorfindet, muss er selbst entscheiden. Endet die Prüfung negativ, muss er erneut **wait()** aufrufen. Daher sollte **wait()** stets in einer Schleife aufgerufen werden, deren Bedingungsteil die kritische Prüfung enthält (siehe oben).

Ein Thread kann per **InterruptedException** aus dem Wartezustand gerissen werden, weil ihm per **interrupt()** aus einem anderen Thread ein Unterbrechungssignal zugestellt wurde (vgl. Abschnitt 15.3.2). Im Beispiel macht die Methode `liefere()` im **catch**-Block zur **InterruptedException** lediglich eine Protokollausgabe, restauriert aber *nicht* das von **wait()** vor dem Werfen der Ausnahme abgeschaltete Unterbrechungssignal (vgl. Empfehlung zur **InterruptedException**-Behandlung auf Seite 727). In anderen Fällen ist es sinnvoll, im **catch**-Block zur **InterruptedException** das Unterbrechungssignal zu restaurieren und anschließend auf höherer Ebene eine Beendigung des Threads in Erwägung zu ziehen.

Man könnte das Beispiel noch um eine Absicherung gegen Lagerüberlauf absichern. Wir gehen jedoch der Einfachheit halber von einem unendlich großen Lager aus.

15.2.3 Explizite Lock-Objekte

Das Interface **Lock** aus dem Namensraum **java.util.concurrent.locks** und die implementierende Klasse **ReentrantLock** kommen bei speziellen Aufgabenstellungen als Alternative zu den synchronisierten Methoden und Blöcken in Frage, die im Abschnitt 15.2.2 behandelt wurden. Man gewinnt an Flexibilität, muss aber auch mehr Verantwortung übernehmen. Wie Goetz et al. (2006, S. 277) betonen, sind die expliziten Lock-Objekte kein Ersatz für die synchronisierten Code-Bereiche, sondern eine Ergänzung.

15.2.3.1 Interface Lock und Klasse ReentrantLock

Beim Betreten eines synchronisierten Bereichs bestehend aus synchronisierten Methoden und Blöcken wird vom aktiven Thread ein Monitor (*impliziter* Lock) erworben und beim Verlassen des Bereichs automatisch zurückgegeben, wobei die Rückgabe selbst im Ausnahmefall sichergestellt ist. Demgegenüber ist bei expliziten **Lock**-Objekten der Gültigkeitsbereich *nicht* an einen Code-Bereich gebunden. Ein (z. B. mit der Methode **lock()** erworbener) expliziter Lock kann in Abhängigkeit von einer Bedingung per **unlock()** - Aufruf freigegeben werden, was nicht unbedingt in derselben Methode geschehen muss, die den Lock erworben hat.

Neben der Flexibilität des Gültigkeitsbereichs bietet das **Lock**-Interface wichtige Optionen beim Erwerb eines Locks. Versucht ein Thread, einen synchronisierten Bereich zu betreten, gelangt er in

einen potentiell endlosen Wartezustand und ist nicht unterbrechbar.[1] Dasselbe passiert bei Verwendung der Methode **lock()** aus dem Interface **Lock**. Allerdings bietet dieses Interface über weitere Methoden wichtige Alternativen zum endlos wartenden, nicht unterbrechbaren Lock-Erwerb:

- **public void lockInterruptibly() throws InterruptedException**
  Der aufrufende Thread bewirbt sich ohne Begrenzung der maximalen Wartezeit um den expliziten Lock, ist aber (teilweise implementationsabhängig) während der Wartezeit unterbrechbar.
- **public boolean tryLock()**
  Der aufrufende Thread erwirbt den expliziten Lock, wenn er aktuell verfügbar ist. Die Methode kehrt sofort zurück und informiert durch ihren Rückgabewert darüber, ob der Lock erworben wurde (**true**) oder nicht (**false**). Im negativen Fall kann der Lock-Interessent sich anderweitig beschäftigen und eventuell später sein Glück erneut versuchen.
- **public boolean tryLock(long** *time*, **TimeUnit** *unit*) **throws InterruptedException**
  Der aufrufende Thread bewirbt sich mit einer Grenze für die maximale Wartezeit um den expliziten Lock und ist (teilweise implementationsabhängig) während der Wartezeit unterbrechbar. Über den Rückgabewert ist zu erfahren, ob der Lock erworben wurde (**true**) oder nicht (**false**).

Mit dem Unterbrechen von Threads werden wir uns im Abschnitt 15.3.2 beschäftigen.

Außerdem sind noch weitere Optionen mit dem **Lock**-Interface verbunden, z. B.:

- Mit einem **Lock**-Objekt lassen sich **Condition**-Objekte verbinden, die es einem Thread ermöglichen, auf das Eintreten einer bestimmten Bedingung zu warten (z. B. neue Daten in einer Warteschlange eingetroffen) oder wartende Threads über das Eintreten einer bestimmten Bedingung zu informieren (siehe Abschnitt 15.2.3.2).
- Über die das Interface **Lock** implementierende Klasse **ReentrantReadWriteLock** kann man *einen* Writer-Thread, aber beliebig viele Reader-Threads zulassen und auf diese Weise unnötige Blockaden vermeiden.

Im folgenden Produzenten-Konsumenten - Programm erhöht die **run()** - Methode des Produzenten-Threads bei Bedarf einen Warenbestand, den ein parallel laufender Kunden-Thread in seiner **run()** - Methode reduziert. Über ein Objekt Klasse **ReentrantLock**, die das Interface **Lock** implementiert, wird der simultane Zugriff auf den Warenbestand durch beide Threads verhindert:

```
import java.util.concurrent.locks.ReentrantLock;

class Lager {
   static ReentrantLock rl = new ReentrantLock();
   static int bestand = 200;

   public static void main(String[] args) {
      Produzent prod = new Produzent();
      Kunde kunde = new Kunde();
      prod.start();
      kunde.start();
   }
}
```

[1] https://javarevisited.blogspot.com/2013/03/reentrantlock-example-in-java-synchronized-difference-vs-lock.html

```
class Produzent extends Thread {
   Produzent () {
      super("Produzent");
   }

   @Override
   public void run() {
      while (true) {
         System.out.println("Produzent beantragt den Lock.");
         Lager.rl.lock();
         try {
            if (Lager.bestand < 500)
               while (Lager.bestand < 1000) {
                  int add = (int) (100 + Math.random()*100);
                  Lager.bestand += add;
                  System.out.println(Thread.currentThread().getName() +
                                     " ergaenzt\t"+add+" Stand: "+Lager.bestand);
                  try {
                    Thread.sleep(100);
                  } catch(InterruptedException ie) {interrupt();}
               }
         } finally {
            System.out.println("Produzent gibt den Lock frei.");
            Lager.rl.unlock();
         }
         try {
           Thread.sleep(2000);
         } catch(InterruptedException ie) {interrupt();}
      }
   }
}


class Kunde extends Thread {
   Kunde () {
      super("Kunde");
   }

   @Override
   public void run() {
      while (true) {
         System.out.println("Kunde beantragt den Lock.");
         Lager.rl.lock();
         try {
            int sub = (int) (100 + Math.random()*100);
            Lager.bestand -= sub;
            System.out.println(Thread.currentThread().getName()+
                               " entnimmt\t"+sub+" Stand: "+Lager.bestand);
         } finally {
            System.out.println("Kunde gibt den Lock frei");
            Lager.rl.unlock();
         }
         try {Thread.sleep(2000);} catch(InterruptedException ie) {interrupt();}
      }
   }
}
```

Die Freigabe eines **Lock**-Objekts sollte unbedingt in der **finally**-Klausel einer **try**-Anweisung geschehen, damit sie unter allen Umständen ausgeführt wird (vgl. Abschnitt 11.2.1.2). Im Beispiel wird diese nachdrückliche Empfehlung (vgl. Goetz 2006, S. 278) umgesetzt, obwohl die Anweisungen im **try**-Block kein nennenswertes Risiko für einen Ausnahmefehler enthalten.

Im Hinblick auf die Sichtbarkeit von gemeinsamen Daten für andere Threads ist von erheblicher Relevanz, dass ...

- beim Erwerb eines expliziten Locks ein Memory Refresh stattfindet, sodass alle Daten im lokalen Cache des erwerbenden Threads aktuell sind,
- bei Rückgabe eines expliziten Locks ein Memory Flush stattfindet, sodass alle Inhalte aus dem lokalen Cache des ehemaligen Lock-Inhabers in den allgemeinen Hauptspeicher übertragen werden (vgl. Abschnitt 15.2.1.2 zum Java-Speichermodell).

Das Beispiel demonstriert die Verwendung und vor allem die Freigabe eines expliziten Lock-Objekts, nutzt aber nicht die höhere Flexibilität der **Lock**-Technik (z. B. bei einem nicht verfügbaren Lock-Objekt) im Vergleich zur **synchronized**-Technik.

15.2.3.2 Koordination per await(), signal() und signalAll()

Mit einem expliziten **Lock**-Objekt lassen sich über die Methode **newCondition()** beliebig viele **Condition**-Objekte verbinden, und deren Methoden **await()**, **signal()** und **signalAll()** erlauben eine Verfeinerung der im Abschnitt 15.2.2.2 beschriebenen Thread-Koordinierung per **wait()**, **notify()** und **notifyAll()**.

In einem Beispiel mit zwei Threads, die einem kontinuierlichen Verbraucher bzw. einen bei Bedarf tätigen Nachfüller simulieren, steht ein **Condition**-Objekt namens `toLow` für einen zu niedrigen Warenbestand und ein **Condition**-Objekt namens `filled` für einen aufgefüllten Bestand:

```
class Lager {
   static ReentrantLock rl = new ReentrantLock();
   static Condition toLow = rl.newCondition();
   static Condition filled = rl.newCondition();
   static int bestand = 200;

   public static void main(String[] args) {
      Produzent prod = new Produzent();
      Kunde kunde = new Kunde();
      prod.start();
      kunde.start();
   }
}
```

Der Nachfüller könnte nach getaner Arbeit ...

- das Lock-Objekt zurückgeben
- und eine Schlafpause einlegen, wobei deren Dauer schlecht an den Arbeitsbedarf anzupassen wäre.

Viel geschickter ist es aber, auf das Eintreten der Bedingung `toLow` zu warten, die neuen Arbeitsbedarf signalisiert. Um diesen speziellen Weckauftrag zu formulieren, ruft der Nachfüller die **await()** - Methode des **Condition**-Objekts `toLow` auf und gibt dabei den Lock automatisch frei:[1]

```
class Produzent extends Thread {
   Produzent () {
      super("Produzent");
   }
```

[1] Im Beispiel sind der Einfachheit halber das **Lock**-Objekt und die **Condition**-Objekte statisch und im gesamten Standardpaket sichtbar definiert, sodass von der wünschenswerten Datenkapselung keine Rede sein kann.

```
    @Override
    public void run() {
       System.out.println("Produzent beantragt den Lock.");
       Lager.rl.lock();
       try {
          while (true) {
             while (Lager.bestand < 1000) {
                int add = (int) (100 + Math.random()*100);
                Lager.bestand += add;
                System.out.println(Thread.currentThread().getName() +
                                   " ergaenzt\t"+add+" Stand: "+Lager.bestand);
                try {
                   Thread.sleep(100);
                } catch(InterruptedException ie) {interrupt();}
             }
             Lager.filled.signal();
             try {
                Lager.toLow.await();
             } catch(InterruptedException ie) {interrupt();}
          }
       } finally {
          System.out.println("Produzent gibt den Lock frei.");
          Lager.rl.unlock();
       }
    }
 }
```

Im zweiten **Condition**-Objekt namens `filled` hinterlässt der Verbraucher-Thread einen Weckauftrag, wenn er sich wegen eines unzureichenden Warenangebots in den Wartezustand begibt:

```
if (Lager.bestand < 100)
   Lager.filled.await();
```

Sobald der Nachfüller seine Arbeit erledigt hat, signalisiert er dies an *einen* Thread, der auf die `filled`-Bedingung wartet:

```
Lager.filled.signal();
```

Mit der Methode **signalAll()** spricht man *alle* Threads an, die auf eine Bedingung warten.

Während die **Object**-Methode Methode **notify()** einen Thread anspricht, der unspezifisch auf den Monitor wartet, erreicht man mit der **Condition**-Methode **signal()** einen Thread, der von der neuen Lage profitieren kann. Wenn im Beispiel der wiedererwachte Verbraucher den Warenbestand stark reduziert hat, signalisiert er dies an einen Thread, der auf die Bedingung `toLow` wartet:

```
if (Lager.bestand < 200)
   Lager.toLow.signal();
```

Bekanntlich dürfen die **Object**-Methoden **wait()**, **notify()** und **notifyAll()** nur innerhalb eines synchronisierten Code-Bereichs aufgerufen werden. Analog zu dieser Bedingung dürfen die **Condition**-Methoden **await()**, **signal()** und **signalAll()** nur dann aufgerufen werden, wenn der aktuelle Thread den zum **Condition**-Objekt gehörigen Lock besitzt.

Wer sich für eine erfolgreiche Anwendung der Thread-Koordination über Objekte vom Typ **ReentrantLock** und **Condition** interessiert, findet sie z. B. in der API-Klasse **ArrayBlockingQueue<E>**, die im Abschnitt 15.2.4.1 vorgestellt wird. Diese Klasse zur Verwaltung einer Warteschlange mit fixierter Kapazität bietet die Methoden **put()** zum Ergänzen eines neuen Eintrags sowie **take()** zur Entnahme eines Eintrags. Wenn **put()** auf eine besetzte Warteschlange oder **take()** auf eine leere Warteschlange trifft, warten die Methoden mit `notFull.await()` bzw. `notEmpty.await()` auf die Voraussetzung für eine Fortsetzung ihrer Tätigkeit.

Das IntelliJ-Projekt mit dem Beispielprogramm befindet sich im Ordner

...\BspUeb\Multithreading\Threads koordinieren\Explizite Lock-Objekte\Condition

15.2.4 Automatisierte Thread-Koordination für Produzenten-Konsumenten - Konstellationen

Die im bisherigen Verlauf von Abschnitt 15.2 beschriebenen Techniken zur Thread-Koordination sind flexibel, aber auch fehleranfällig. Für häufig auftretende Aufgaben bietet Java daher Standardlösungen zur Vermeidung von Aufwand und Fehlern. In diesem Abschnitt werden zwei Lösungen vorgestellt, die sich bei einer Produzenten-Konsumenten - Konstellation bewähren:

- Das Interface **BlockingQueue<E>** mit implementierenden Klassen wie z. B. **ArrayBlockingQueue<E>** und **LinkedBlockingQueue<E>**
- Verbindung von zwei Threads per Pipe über die Datenstromklassen **PipedOutputStream** und **PipedInputStream**

15.2.4.1 BlockingQueue<E>

Die das Interface **BlockingQueue<E>** implementierenden Kollektionsklassen wie **ArrayBlockingQueue<E>** und **LinkedBlockingQueue<E>** (alle Typen aus dem Paket **java.util.concurrent**) funktionieren als **Warteschlangen** nach dem **FIFO-Prinzip** (First-In-First-Out) und sind Thread-sicher, dürfen also von mehreren Threads simultan genutzt werden. Außerdem sind sie in der Lage, einen Produzenten- und einen Konsumenten-Thread automatisch zu koordinieren:

- Der Produzenten-Thread befördert mit der **put()** - Methode Daten in den Container. Hat ein Container (wie z. B. **ArrayBlockingQueue<E>**) eine Maximalkapazität, und ist diese erreicht, dann wird der Produzenten-Thread in den Wartezustand versetzt und bei Bedarf für neue Daten reaktiviert.
- Der Konsumenten-Thread holt mit der **take()** - Methode Daten ab. Bei fehlenden Daten wird er in den Wartezustand versetzt und bei Verfügbarkeit von neuen Daten reaktiviert.

Im Unterschied zu einer **ArrayBlockingQueue<E>** hat eine **LinkedBlockingQueue<E>** *keine* Kapazitätsbeschränkung, sodass praktisch unbegrenzt viele Daten eingeliefert werden können.

Während sich die bisherige Darstellung auf *einen* Produzenten und *einen* Konsumenten beschränkt hat, können bei Bedarf auch mehrere Threads als Datenlieferanten bzw. -konsumenten unter Vermittlung einer **BlockingQueue<E>** tätig werden.

Zur Demonstration verwenden wir ein Beispiel mit der folgenden Startklasse:

```
import java.util.concurrent.*;
class BlockingQueueDemo {
   public static void main(String[] args) {
      BlockingQueue<Integer> depot = new LinkedBlockingQueue<>(3);
      KonThread kt = new KonThread(depot);
      ProThread pt = new ProThread(depot, kt);
      pt.start();
      kt.start();
   }
}
```

Ein Produzent und ein Konsument agieren jeweils in einem eigenen Thread mit Zugriff auf ein Depot, das durch ein Objekt der Klasse **LinkedBlockingQueue<Integer>** nach dem Warteschlangenprinzip verwaltet wird.

Der Produzenten-Thread

```
import java.util.concurrent.BlockingQueue;

class ProThread extends Thread {
   private BlockingQueue<Integer> depot;
   private int[] produkte = {1, 2, 3};
   Thread konsument;

   ProThread(BlockingQueue<Integer> dep, Thread kon) {
      depot = dep;
      konsument = kon;
   }

   @Override
   public void run() {
      for(int i = 0; i < produkte.length; i++) {
         System.out.println("\nDer Produzent liefert gleich.");
         depot.add(produkte[i]);
         try {sleep(10);} catch(InterruptedException ie) {interrupt();}
         System.out.println("Kurz nach der Lieferung ist der Konsument " +
                            konsument.getState());
         try {sleep(5000);} catch(InterruptedException ie) {interrupt();}
         System.out.println("Produzent wacht auf, Konsument ist " +
                            konsument.getState());
      }
   }
}
```

wiederholt in seiner **run()** - Methode per **for**-Schleife die folgenden Aktionen bis zur Erschöpfung des Vorrats an **Integer**-Objekten:

- Unmittelbar nach einer ankündigenden Konsolenausgabe fügt er ein **Integer**-Objekt in die Warteschlange ein. Statt der oben beschriebenen, bei gefülltem Container blockierenden Methode **put()** wird die Methode **add()** verwendet, von der keine deklarationspflichtigen Ausnahmen zu erwarten sind. Weil der Container vom Typ **BlockingQueue<Integer>** keine Kapazitätsgrenze hat, sind die Methoden äquivalent.
- Nach einer kurzen Wartezeit von 10 Millisekunden, die dem Konsumenten-Thread genügen sollte, um das eingetroffene Datenobjekt zu bemerken, protokolliert der Produzent mit Hilfe der Methode **getState()** den Status des Konsumenten-Threads.
- Dann legt sich der Produzent 5 Sekunden schlafen, was zu einem Datenmangel für den Konsumenten führt. Nach dem Aufwachen protokolliert der Produzent erneut den Status des Konsumenten-Threads.

Der Konsumenten-Thread

```
import java.util.concurrent.BlockingQueue;

class KonThread extends Thread {
   private BlockingQueue<Integer> depot;

   KonThread(BlockingQueue<Integer> dep) {
      depot = dep;
   }
```

```
    @Override
    public void run() {
       for (int i = 0; i < 3; i++) {
          try {
             System.out.println("Der Konsument hat bezogen: " + depot.take());
          } catch (InterruptedException ie) {Thread.currentThread().interrupt();}
          for (int j = 0; j < 3_000_000; j++)
             Math.random();
       }
    }
}
```

wiederholt in seiner **run()** - Methode per **for**-Schleife dreimal die folgenden Aktionen:

- Er holt per **take()** ein Element aus der Warteschlange und protokolliert sein Verhalten. Weil von **take()** (wie von **wait()**, vgl. Abschnitt 15.2.2.2) eine **InterruptedException** zu erwarten ist, erfolgt der Aufruf in einer **try**-Anweisung.
- Dann simuliert der Konsument ein geschäftiges Treiben, indem er 3 Millionen Zufallszahlen zieht.

Die Ausgaben des Programms

```
Der Produzent liefert gleich.
Der Konsument hat bezogen: 1
Kurz nach der Lieferung ist der Konsument RUNNABLE
Produzent wacht auf, Konsument ist WAITING

Der Produzent liefert gleich.
Der Konsument hat bezogen: 2
Kurz nach der Lieferung ist der Konsument RUNNABLE
Produzent wacht auf, Konsument ist WAITING

Der Produzent liefert gleich.
Der Konsument hat bezogen: 3
Kurz nach der Lieferung ist der Konsument RUNNABLE
Produzent wacht auf, Konsument ist TERMINATED
```

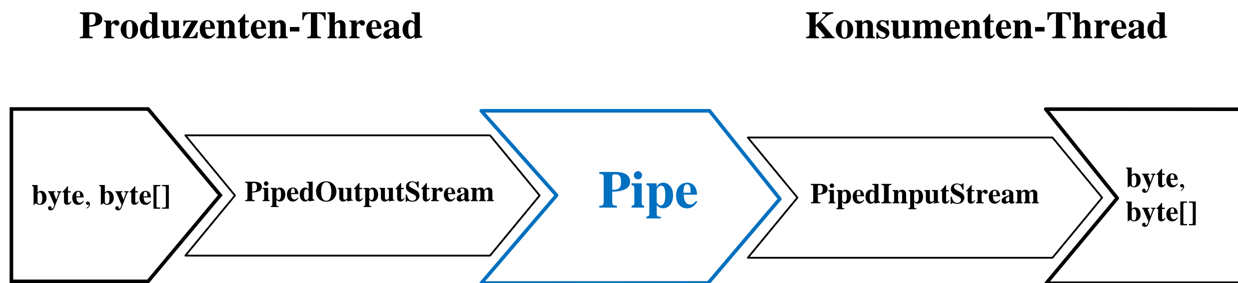
zeigen, ...

- dass der Konsumenten-Thread im Zustand **RUNNABLE** ist, bis die Methode **take()** auf Datenmangel trifft, was zum Zustand **WAITING** führt,
- dass sich ein Thread nach dem Ende seine **run()** - Methode im Zustand **TERMINATED** befindet.

Auf einem Rechner mit abweichender CPU-Leistung müssen eventuell Einstellungen angepasst werden, um dasselbe Muster beobachten zu können.

15.2.4.2 PipedOutputStream und PipedInputStream

Durch ein Tandem aus einem **PipedOutputStream** (Basisklasse: **OutputStream**, vgl. Abschnitt 14.3.1) und einem verbundenen **PipedInputStream** (Basisklasse: **InputStream**, vgl. Abschnitt 14.3.2) lässt sich ein unidirektionaler Datentransfer zwischen zwei Threads einrichten. Ein Thread schreibt Bytes in den **PipedOutputStream** und ein anderer Thread kann aus dem verbundenen **PipedInputStream** lesen, sodass sich eine Produzenten-Konsumenten - Kooperation realisieren lässt. Man kann sich vorstellen, dass die beiden Threads mit einer Röhre (engl. *pipe*) verbunden sind.



Neben der Byte-orientierten Pipe-Verbindung existiert auch eine zeichenorientierte Variante mit Objekten aus den Klassen **PipedWriter** und **PipedReader** am Ein- bzw. Ausgang.

Im nun vorzustellenden Beispielprogramm zur Produzenten-Konsumenten-Kooperation mit Pipe-Lösung ist die Startklasse ähnlich aufgebaut wie bei der **BlockingQueue<E>** - Lösung (vgl. Abschnitt 15.2.4.1):

```
import java.io.*;

class PipedStreamDemo {
   public static void main(String[] args) throws IOException {
      PipedOutputStream pipedOutStream = new PipedOutputStream();
      PipedInputStream pipedInStream = new PipedInputStream(pipedOutStream);

      KonThread kt = new KonThread(pipedInStream);
      ProThread pt = new ProThread(pipedOutStream, kt);
      pt.start();
      kt.start();
   }
}
```

Um die beiden Pipe-Ströme zu verbinden, erhält von beiden Konstruktoren ein beliebig gewählter per Parameter eine Referenz auf das Partnerobjekt. Alternativ könnte die **connect()** - Methode eines Stroms mit dem Partnerobjekt als Parameter aufgerufen werden.

Der Lieferant ruft nach jedem Schreibzugriff auf den Ausgabestrom die Methode **flush()** auf:

```
pipedOutStream.write(produkte[i]);
pipedOutStream.flush();
```

Wenn der Konsumenten-Thread beim Leseversuch

```
pipedInStream.read();
```

keine Daten vorfindet, begibt er sich per **wait()** - Aufruf für eine Sekunde in Wartestellung.[1] Der **flush()** - Aufruf des Produzenten informiert über die Ankunft neuer Daten und beendet so die Wartezeit.

Wichtige Regeln zur Verwendung der Pipe-Kommunikation zwischen Threads:[1]

[1] Das zeigt ein Blick in den Quellcode der Methode **PipedInputStream.read**() im Java 13 - API:

```
while (in < 0) {
    . . .
   /* might be a writer waiting */
   notifyAll();
   try {
      wait(1000);
   } catch (InterruptedException ex) {
      throw new java.io.InterruptedIOException();
   }
}
```

- Es darf nur *ein* Thread in die Pipe schreiben und nur *ein* (anderer) Thread aus der Pipe lesen.
- Auf gar keinen Fall darf derselbe Thread schreiben und lesen, weil der Thread dabei blockiert werden kann (Deadlock).
- Bevor der schreibende Thread endet, muss er den **PipedOutputStream** schließen, z. B. über eine try-with-resources - Anweisung. Der lesende Thread kann weiterhin auf die bereits geschriebenen Daten zugreifen.

15.2.5 Klassen zur Thread-Synchronisation

Das mit Java 5 (alias 1.5) eingeführte Paket **java.util.concurrent** enthält einige Klassen zur Realisation von speziellen Mustern der Thread-Koordination.

15.2.5.1 Semaphore

Ein Objekt der Klasse **Semaphore** verwaltet eine Menge von *k* „Passierscheinen", die an einem Kontrollpunkt (oder an mehreren Kontrollpunkten) benötigt werden. Ein Thread bewirbt sich mit der **Semaphore**-Methode **acquire()** um einen Passierschein und gibt seine Berechtigung durch einen Aufruf der **Semaphore**-Methode **release()** wieder zurück. In der folgenden **Semaphore**-Konstruktorüberladung

public Semaphore(int *permits*, **boolean** *fair*)

legt man ...

- durch den ersten Parameter fest, wie viele Threads gleichzeitig einen Passierschein besitzen können,
- durch den zweiten Parameter fest, ob das faire FIFO-Prinzip (First-In-First-Out) garantiert sein soll.

Im Produzenten-Konsumenten - Programm kann man z. B. mit einem Semaphore-Objekt dafür sorgen, dass maximal 2 Produzenten gleichzeitig anliefern dürfen:

```
private Semaphore sem = new Semaphore(2, true);
```

Alternativ oder gleichzeitig könnte man die Anzahl der simultan tätigen Konsumenten beschränken.

In der folgenden Startklassen-Variante zum Produzenten-Konsumenten - Beispiel werden 5 Produzenten-Threads und ein Konsumenten-Thread gestartet:

```
class SemaphoreDemo {
 public static void main(String[] args) {
  Lager lager = new Lager(100);
  for(int i = 1; i <= 5; i++)
   new ProThread(i, lager).start();
  new KonThread(lager).start();
 }
}
```

In der `Lager`-Methode `ergaenze()`

[1] Einige Regeln stammen von der folgenden Webseite von Daniel Ferber (besucht am 10.03.2018): https://techtavern.wordpress.com/2008/07/16/whats-this-ioexception-write-end-dead/

```
void ergaenze(int nr, int add) {
 try {
  sem.acquire();
  anwProd.add(nr);
  System.out.println("  " + Thread.currentThread().getName() + " kommt");
  for (int i = 0; i < 3; i++) {
   synchronized(this) {
    bilanz += add;
    anz++;
    System.out.println("Nr. " + anz + ":\t" + Thread.currentThread().getName() +
      " ergänzt\t" + add + "\tum " + formZeit() + " Uhr. Stand: " + bilanz +
      " (Prod: " + anwProd + ")");
   }
   Thread.sleep(500);
  }
 } catch(InterruptedException ie) {
  Thread.currentThread().interrupt();
 } finally {
  System.out.println("  " + Thread.currentThread().getName() + " geht");
  anwProd.remove(nr);
  sem.release();
 }
}
```

bewirbt sich ein Produzenten-Thread um einen Passierschein und trägt sich dann in das als synchronisiertes **HashSet<Integer>** - Objekt (vgl. Abschnitt 10.5.1)

```
Set<Integer> anwProd = Collections.synchronizedSet(new HashSet<Integer>());
```

realisierte Verzeichnis der aktiven Produzenten ein:

```
sem.acquire();
anwProd.add(nr);
```

Um die simultane Tätigkeit von zwei Produzenten-Threads beobachten zu können, absolviert ein Produzent bei jedem Lagerbesuch drei Einzellieferungen. Die Protokolleinträge erfolgen weiterhin in einem synchronisierten Block und enthalten am Ende eine Liste mit den aktuell im Lager befindlichen Produzenten. Nach erledigter Arbeit entfernt sich ein Produzent aus dem Verzeichnis der aktiven Lieferanten und gibt seine Lagerberechtigung zurück:

```
anwProd.remove(nr);
sem.release();
```

Die zugehörigen Anweisungen befinden sich in einem **finally**-Block, damit sie unter allen Umständen ausgeführt werden.

Es folgt ein typisches Lagerprotokoll:

```
Der Laden ist offen (Bestand = 100)

  Produzent 1 kommt
Nr. 1:  Produzent 1 ergänzt 77  um 14:24:22 Uhr. Stand: 177 (Prod: [1])
  Produzent 5 kommt
Nr. 2:  Konsument entnimmt  177 um 14:24:22 Uhr. Stand: 0
Nr. 3:  Produzent 5 ergänzt 28  um 14:24:22 Uhr. Stand: 28 (Prod: [1, 5])
Nr. 4:  Produzent 1 ergänzt 77  um 14:24:22 Uhr. Stand: 105 (Prod: [1, 5])
Nr. 5:  Produzent 5 ergänzt 28  um 14:24:22 Uhr. Stand: 133 (Prod: [1, 5])
Nr. 6:  Produzent 1 ergänzt 77  um 14:24:23 Uhr. Stand: 210 (Prod: [1, 5])
Nr. 7:  Produzent 5 ergänzt 28  um 14:24:23 Uhr. Stand: 238 (Prod: [1, 5])
  Produzent 1 geht
  Produzent 2 kommt
Nr. 8:  Produzent 2 ergänzt 94  um 14:24:23 Uhr. Stand: 332 (Prod: [2, 5])
  Produzent 5 geht
  Produzent 4 kommt
Nr. 9:  Produzent 4 ergänzt 27  um 14:24:23 Uhr. Stand: 359 (Prod: [2, 4])
Nr. 10: Produzent 2 ergänzt 94  um 14:24:24 Uhr. Stand: 453 (Prod: [2, 4])
Nr. 11: Produzent 4 ergänzt 27  um 14:24:24 Uhr. Stand: 480 (Prod: [2, 4])
Nr. 12: Konsument entnimmt  7 um 14:24:24 Uhr. Stand: 473
```

```
Nr. 13: Produzent 2 ergänzt 94 um 14:24:24 Uhr. Stand: 567 (Prod: [2, 4])
Nr. 14: Produzent 4 ergänzt 27 um 14:24:24 Uhr. Stand: 594 (Prod: [2, 4])
  Produzent 2 geht
  Produzent 3 kommt
Nr. 15: Produzent 3 ergänzt 34 um 14:24:25 Uhr. Stand: 628 (Prod: [3, 4])
  Produzent 4 geht
Nr. 16: Produzent 3 ergänzt 34 um 14:24:25 Uhr. Stand: 662 (Prod: [3])
  Produzent 1 kommt
Nr. 17: Produzent 1 ergänzt 23 um 14:24:26 Uhr. Stand: 685 (Prod: [1, 3])
Nr. 18: Produzent 3 ergänzt 34 um 14:24:26 Uhr. Stand: 719 (Prod: [1, 3])
Nr. 19: Produzent 1 ergänzt 23 um 14:24:26 Uhr. Stand: 742 (Prod: [1, 3])
  Produzent 3 geht
  Produzent 5 kommt
Nr. 20: Produzent 5 ergänzt 88 um 14:24:27 Uhr. Stand: 830 (Prod: [1, 5])
Nr. 21: Produzent 1 ergänzt 23 um 14:24:27 Uhr. Stand: 853 (Prod: [1, 5])
Nr. 22: Produzent 5 ergänzt 88 um 14:24:27 Uhr. Stand: 941 (Prod: [1, 5])
  Produzent 1 geht

Lieber Produzent 3, es ist Feierabend!

Lieber Konsument, es ist Feierabend!

Lieber Produzent 2, es ist Feierabend!
Nr. 23: Produzent 5 ergänzt 88 um 14:24:28 Uhr. Stand: 1029 (Prod: [5])

Lieber Produzent 4, es ist Feierabend!
  Produzent 5 geht

Lieber Produzent 1, es ist Feierabend!

Lieber Produzent 5, es ist Feierabend!
```

15.2.5.2 CountDownLatch

Objekte der Signalisierungsklasse **CountdownLatch** enthalten einen Zähler und starten mit einem positiven Wert, der sich bei jedem Aufruf der Instanzmethode **countDown()** um 1 verringert. Hat sich ein Thread per **await()** - Aufruf an das Signalisierungsobjekt in Wartestellung begeben, wird er beim Zählerstand 0 reaktiviert.

Wir erlauben uns ein verspieltes Beispielprogramm mit einem Thread, der auf ein **CountdownLatch**-Objekt wartet und nach seiner Reaktivierung einen Raketenstart simuliert:

```
Raketen-Thread wartet auf CountDownLatch

Countdown läuft:
10 9 8 7 6 5 4 3 2 1

Rakete startet:
.. . .  .  .   .   .    .    .     .     .      .      .       .       .
```

Während der Raketen-Thread wartet, setzt der Haupt-Thread durch wiederholte **countDown()** - Aufrufe das Signalisierungsobjekt auf null:

```
import java.util.concurrent.CountDownLatch;
class CountDownLatchDemo {
   static void waitAndLiftOff(CountDownLatch cdl) {
      System.out.println("Raketen-Thread wartet auf CountDownLatch");
      try {cdl.await();
      } catch (InterruptedException ie) {Thread.currentThread().interrupt();}
      System.out.println("\n\nRakete startet:");
      for (int i = 1; i <= 16; i++) {
         System.out.print(".");
         for (int j = 0; j < i/2; j++) System.out.print(" ");
         try { Thread.sleep(300);
         }  catch(InterruptedException ie) {Thread.currentThread().interrupt();}
      }
   }
```

```
    public static void main(String[] args) throws InterruptedException {
       int cdn = 10;
       CountDownLatch cdl = new CountDownLatch(cdn);
       new Thread(() -> waitAndLiftOff(cdl)).start();
       Thread.sleep(300);
       System.out.println("\nCountdown läuft:");
       for (int i = cdn; i > 0; i--) {
          System.out.print(i + " ");
          cdl.countDown();
          Thread.sleep(500);
       }
    }
 }
```

Der Raketen-Thread wird über ein per Lambda-Ausdruck realisiertes **Runnable**-Objekt gestartet. So kann das **CountdownLatch**-Objekt an die ausführende Methode des Threads übergeben werden, obwohl die **run()** - Methode im Interface **Runnable** keinen Parameter besitzt.

Im ernsthaften Anwendungen dient ein **CountdownLatch**-Objekt dazu, einen oder mehrere Threads warten zu lassen, bis andere Threads bestimmte Operationen ausgeführt haben.

15.2.5.3 CyclicBarrier

Ein Objekt der Klasse **CyclicBarrier** realisiert eine Barriere, an der sich mehrere Threads versammeln, nachdem sie jeweils eine Vorleistung erbracht haben. Die Threads warten also aufeinander. Sind alle angekommen, werden optional die Vorleistungen zu einem Endergebnis verarbeitet. Nachdem alle Threads angekommen sind und ggf. auch noch das Endergebnisproduktion erstellt worden ist, dürfen die wartenden Threads ihre Tätigkeit fortsetzten.

In einem halbwegs realistischen Beispielprogramm lassen wir `N_SAMPLES` Threads jeweils den Mittelwert aus `SAMPLE_SIZE` (abgekürzt durch *N*) Pseudozufallszahlen (gleichverteilt im Intervall [0, 1)) berechnen. Wenn alle Threads fertig sind, und dementsprechend `N_SAMPLES` Stichprobenmittelwerte vorliegen, wird als Endergebnis die Standardabweichung der Stichprobenmittelwerte berechnet. Dies ist eine Schätzung für den sogenannten *Standardfehler des Mittelwertes*, dessen exakten Wert man nach der folgenden Formel berechnen kann:

$$\sigma_{\bar{X}}^2 = \sqrt{\frac{1/12}{N}}$$

Dabei geht die Varianz $1/12$ einer Variablen mit Gleichverteilung auf dem Intervall [0, 1) ein.[1]

Das Experiment zur empirischen Schätzung des Standardfehlers des Mittelwerts aus `SAMPLE_SIZE` Fällen wird `N_EXPERIMENTS` mal wiederholt, wobei die Variable `ex` die Nummer der gerade ausgeführten Wiederholung enthält.

Die `N_SAMPLES` Threads

```
 public static void main(String[] args) throws InterruptedException {
    for (int i = 0; i < N_SAMPLES; i++)
       new Thread(new Meaner(i)).start();
 }
```

führen jeweils die **run()** - Methode der folgenden Klasse aus, die das Interface **Runnable** implementiert:

[1] https://www.statistik-nachhilfe.de/ratgeber/statistik/wahrscheinlichkeitsrechnung-stochastik/wahrscheinlichkeitsverteilungen/stetige-verteilungen/stetige-gleichverteilung-rechteckverteilung

```
static class Meaner implements Runnable {
   private int col;
   private Random zzg = new Random();
   Meaner(int col) {
      this.col = col;
   }
   public void run() {
      while (ex < N_EXPERIMENTS) {
         double mean = 0.0;
         for (int i = 0; i < SAMPLE_SIZE; i++)
            mean = mean + zzg.nextDouble();
         data[col] = mean / SAMPLE_SIZE;
         try {
            barrier.await();
         } catch (InterruptedException | BrokenBarrierException ex) {
            return;
         }
      }
   }
}
```

Nachdem ein Thread seinen Mittelwert berechnet und in ein Array-Element geschrieben hat, wartet er an der Barriere auf die anderen Threads:

```
barrier.await();
```

Das verwendete Objekt vom Typ **CyclicBarrier**

```
static final CyclicBarrier barrier = new CyclicBarrier(N_SAMPLES, barrierAction);
```

wird von der folgenden Konstruktor-Überladung erstellt:

public CyclicBarrier(int *parties*, **Runnable** *barrierAction*)

Durch den ersten Konstruktorparameter wird die Anzahl der teilnehmenden Threads deklariert, und durch den zweiten Konstruktorparameter wird ein **Runnable**-Objekt zur Endergebnisberechnung benannt.

Im Beispiel wird von der *barrierAction* - Methode die erwartungstreue Schätzung der Standardabweichung der N_SAMPLES Stichprobenmittelwerte unter Verwendung der Verschiebungsformel (siehe Übungsaufgabe im Abschnitt 5.5) berechnet und dann ausgegeben:

```
static Runnable barrierAction = new Runnable()  {
   @Override
   public void run() {
      double sum = 0.0;
      double qs = 0.0;
      for(int i = 0; i < N_SAMPLES; i++) {
         sum += data[i];
         qs += data[i] * data[i];
      }
      double mean = sum / N_SAMPLES;
      double variance = (qs - N_SAMPLES *mean*mean) / (N_SAMPLES - 1);
      double se = Math.sqrt(variance);
      System.out.println("SE(" + ex + ") = " + se);
      ex++;
   }
};
```

Schließlich wird noch die Variable ex inkrementiert. Nach der Beendigung der Methode **run()** dürfen die wartenden Threads ihre Arbeit fortsetzen und jeweils den nächsten Mittelwert berechnen.

Aus der Stichprobengröße

```
static final int SAMPLE_SIZE = 10_000;
```

ergibt sich nach der oben angegebenen Formel ein wahrer Standardfehler von 0,002886751. In die fünfmal durchgeführte empirische Schätzung des Standardfehlers wurden jeweils 10 Stichproben einbezogen:

```
static final int N_SAMPLES = 10;
static final int N_EXPERIMENTS = 5;
```

Wie das folgende Ablaufprotokoll zeigt, kann der Standardfehler aus 10 Stichproben noch nicht sehr genau geschätzt werden:

```
SE(0) = 0.0025556975714659674
SE(1) = 0.0023379809583384367
SE(2) = 0.0019856120048262144
SE(3) = 0.004137407692079622
SE(4) = 0.0031958711560183084
```

Es sollte noch eine Begründung für den Bestandteil **Cyclic** im Namen der aktuell behandelten Synchronisierungsklasse geliefert werden. Während ein **CountdownLatch**-Objekt ausgebraucht ist, nachdem das Schloss (engl.: *latch*) aufgesprungen ist, kann das Treffen an einer Barriere beliebig oft wiederholt werden.

15.2.5.4 Phaser

Die mit Java 7 eingeführte Synchronisierungsklasse **Phaser** kann bei höherer Flexibilität die Aufgaben der älteren Klassen **CountDownLatch** und **CyclicBarrier** übernehmen (siehe Abschnitte 15.2.5.2 und 15.2.5.3):

- Während bei einem **CyclicBarrier**-Objekt die Anzahl der beteiligten Threads schon im Konstruktor festgelegt werden muss, kann sie sich bei einem **Phaser**-Objekt ändern. Diese Flexibilität ist z. B. nützlich, wenn der Verzeichnisbaum eines Dateiensystems analysiert werden soll, wobei die Anzahl der beteiligten Threads variiert. Die Anzahl der bei einem **Phaser**-Objekt registrierten Threads kann durch eine von den folgenden Methoden erhöht werden:

  public int register()
  public int bulkRegister(int *parties***)**

  Ein Thread kann seine Ankuft an der Barriere melden und gleichzeitig mitteilen, an den weiteren Phasen *nicht* mehr teilnehmen zu wollen:

  public int arriveAndDeregister()
- Ein **Phaser**-Objekt ist ebenso wiederverwendbar wie ein **CyclicBarrier**-Objekt, wobei man von *Phasen* statt von *Zyklen* spricht.
- Ein Thread kann seine Ankunft an der Barriere melden, um auf den nächsten Phasenwechsel (Advance) zu warten:

  public int arriveAndAwaitAdvance()

  Diese Methode entspricht der **CyclicBarrier**-Methode **await()**. Allerdings wirft **wait()** eine Ausnahme vom Typ **InterruptedException**, wenn der wartende Thread eine Terminierungsaufforderung erhält (siehe Abschnitt 15.3.2.2), während **arriveAndAwaitAdvance()** von dieser Aufforderung *nicht* tangiert wird. Um die **CyclicBarrier.await()** - Reaktion auf eine Terminierungsaufforderung zu erhalten, verwendet man die **Phaser**-Methode **awaitAdvanceInterruptibly()**.
- Statt **arriveAndAwaitAdvance()** aufzurufen, kann ein Thread mit der Methode **arrive()** seine Ankunft an der Barriere melden, *ohne* auf die anderen Threads und damit auf den nächsten Phasenwechsel zu warten:

  public int arrive()

Diese nicht-blockierende Methode liefert als Rückgabe entweder die Nummer der aktuellen Phase oder eine negative Zahl, wenn der **Phaser** bereits terminiert wurde. Während bei der Klasse **CyclicBarrier** *alle* an der Barriere ankommenden Threads in den Wartezustand wechseln, erlaubt die Klasse **Phaser** auch eine Ankunft ohne Warten.

- Zum Phasenwechsel (Advance) kommt es, wenn alle aktuell zu erwartenden Threads eingetroffen sind. Dann ...
  - werden die wartenden Threads reaktiviert,
  - wird der Phasenzähler inkrementiert,
  - werden die Zählerstände für die zu erwartenden bzw. für die bereits eingetroffenen Threads zurückgesetzt.
- Soll am Ende einer Phase eine (mit der *barrierAction* der Klasse **CyclicBarrier** vergleichbare) Aktion stattfinden, dann muss man eine **Phaser**-Ableitung definieren und die Methode **onAdvance()** überschreiben:

  protected boolean onAdvance(int *phase*, **int** *registeredParties*)
- Mit der **onAdvance()** - Rückgabe **true** lässt sich der **Phaser** terminieren. Diese Option wird oft dazu benutzt, die per **Phaser** kontrollierten Aktivitäten unter einer Bedingung (z. B. nach einer bestimmten Anzahl von Iterationen) zu beenden. Die Basisklassenvariante der Methode **onAdvance()** terminiert den **Phaser**, wenn die Anzahl der registrierten Threads durch einen Aufruf der Methode **arriveAndDeregister()** auf den Wert 0 gebracht wird. Versuche zur Registrierung bei einem terminierten **Phaser** bleiben ohne Effekt.
- Die Phasen sind null-basiert nummeriert, und ein Thread kann gezielt auf das Ende einer bestimmte Phase warten:

  public int awaitAdvance(int *phase*)

  Die Methode meldet die aktuelle Phase zurück (also z. B. 1, wenn auf das Ende der Phase 0 gewartet wurde) oder signalisiert mit einer negativen Rückgabe, dass der **Phaser** bereits terminiert worden ist.
- Bei der **Phaser**-Konstruktion kann optional ein elterliches **Phaser**-Objekt angegeben werden, sodass sich mehrere **Phaser**- Objekte in eine Baumstruktur bringen lassen. Ein untergeordnetes **Phaser**-Objekt wird automatisch (de-)registriert.
- Über die wesentlichen Betriebszustände eines **Phaser**-Objekts informieren die folgenden Methoden:
  - **public boolean isTerminated()**
  - **public int getRegisteredParties()**
    Wie viele Threads sind in der aktuellen Phase registriert?
  - **public int getArrivedParties()**, **public int getUnarrivedParties()**
    Wie viele von den registrierten Threads sind bereits angekommen bzw. fehlen noch.

Weitere Informationen zur Klasse **Phaser** finden sich u.a. bei Kreft & Langer (2012) und in der API-Dokumentation.[1]

In der folgenden Variante des Beispielprogramms zur Klasse **CyclicBarrier** (siehe Abschnitt 15.2.5.3) wird eine anonyme **Phaser**-Ableitung mit Überschreibung der Methode **onAdvance()** definiert, die durch den Rückgabewert **true** für eine Terminierung nach einer festgelegten Anzahl von Phasen sorgt:

[1] https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/Phaser.html

```
import java.util.*;
import java.util.concurrent.*;

class PhaserAsCyclicBarrier {
   static final int SAMPLE_SIZE = 10_000;
   static final int N_SAMPLES = 10;
   static final int N_EXPERIMENTS = 5;
   static double[] data = new double[N_SAMPLES];

   static final Phaser phaser = new Phaser(N_SAMPLES) {
      @Override
      protected boolean onAdvance(int phase, int registeredParties) {
         double sum = 0.0;
         double qs = 0.0;
         for(int i = 0; i < N_SAMPLES; i++) {
            sum += data[i];
            qs += data[i] * data[i];
         }
         double mean = sum / N_SAMPLES;
         double variance = (qs - N_SAMPLES *mean*mean) / (N_SAMPLES - 1);
         double se = Math.sqrt(variance);
         System.out.println("SE(" + phase + ") = " + se);
         return phase == N_EXPERIMENTS - 1;
      }
   };

   static class Meaner implements Runnable {
      private int col;
      private Random zzg = new Random();
      Meaner(int col) {
         this.col = col;
      }
      public void run() {
         while (!phaser.isTerminated()) {
            double mean = 0.0;
            for (int i = 0; i < SAMPLE_SIZE; i++)
               mean = mean + zzg.nextDouble();
            data[col] = mean / SAMPLE_SIZE;
            phaser.arriveAndAwaitAdvance();
         }
      }
   }

   public static void main(String[] args) {
      for (int i = 0; i < N_SAMPLES; i++)
         new Thread(new Meaner(i)).start();
   }
}
```

Im Beispielprogramm stellen die Arbeits-Threads ihre Tätigkeit ein, wenn das **Phaser**-Objekt terminiert ist:

```
while (!phaser.isTerminated()) {...}
```

15.2.6 Nicht-blockierende Koordination

Wenn implizite oder explizite Lock-Objekte häufig und nur für kurze Zeit erworben werden, kann der anteilige Aufwand für die Thread-Koordination unverhältnismäßig hoch ausfallen. Weitere Nachteile der blockierenden Thread-Koordination sind:

- Threads mit hoher Priorität können durch einen Thread mit niedriger Priorität aufgehalten werden, der im Besitz eines Locks ist (zu den Prioritäten von Threads siehe Abschnitt 15.3.3.1).
- Wenn die Thread-Koordination nicht nur ineffizient ist, sondern sogar fehlerhaft, dann kann es zum Programmstillstand aufgrund eines Deadlocks kommen (siehe Abschnitt 15.3.4).

Anschließend werden zwei Verfahren zur nicht-blockierenden Thread-Koordination beschrieben. Streng genommen sind dabei ebenfalls Sperren im Spiel, die aber durch die Ausnutzung von Hardware-Unterstützung nur geringe Kosten verursachen. Allerdings lassen sich durch die beiden kostengünstigen und einfachen Verfahren nur spezielle Koordinierungsprobleme lösen.

15.2.6.1 Modifikator volatile

Greifen zwei Threads schreibend auf ein Feld vom Typ **double** oder **long** zu, kann es durch einen unglücklichen Thread-Wechsel passieren, dass die beiden Threads jeweils 32 Bit zu einem sinnlosen Speicherwert mit insgesamt 64 Bit beisteuern. Dies wird durch den Modifikator **volatile** verhindert, z. B.:

```
private volatile long counter;
```

Im Hinblick auf die Sichtbarkeit von gemeinsamen Daten für andere Threads ist von erheblicher Relevanz, dass ...

- beim Lesen einer als **volatile** deklarierten Variablen ein Memory Refresh stattfindet, sodass alle Daten im lokalen Cache des lesenden Threads aktualisiert werden,
- beim Schreiben in eine **volatile** deklarierten Variable ein Memory Flush stattfindet, sodass alle Inhalte aus dem lokalen Cache des schreibenden Threads in den allgemeinen Hauptspeicher übertragen werden (vgl. Abschnitt 15.2.1.2 zum Java-Speichermodell).

Weil beim Zugriff auf eine als **volatile** deklarierte Variable der gesamte Thread-lokale Cache-Speicher mit dem Hauptspeicher abgeglichen wird, ist der Zugriff auf **volatile**-Variablen relativ langsam (Kreft & Langer 2008a; Ziesche & Arinir 2010, S. 257).

Für eine als **volatile** deklarierte Variable ist also garantiert, dass bei jedem Lesezugriff der aktuelle, von irgendeinem Thread veränderte Wert ermittelt wird. Dieser Effekt des Modifikators ist potentiell bei Feldern von beliebigem Typ relevant, z. B.:

```
private volatile boolean stopLoop;
```

Damit eignet sich eine Variable mit **volatile**-Modifikator dazu, einen Status oder eine Nachricht (z. B. Aufgabe erledigt, Übertragung stoppen) über Thread-Grenzen hinweg zu signalisieren. Weitere Details zum Modifikator **volatile** und seiner Rolle bei der Thread-Koordination finden sich bei Kreft & Langer (2008b).

15.2.6.2 Atomare Variablen

Die seit Java 5 (alias 1.5) verfügbaren **Atomic**-Klassen (z. B. **AtomicBoolean**, **AtomicInteger**, **AtomicLong**, **AtomicReference**) im API-Paket **java.util.concurrent.atomic** ermöglichen eine *Lock-freie* Thread-Koordination beim Zugriff auf *eine einzelne Variable*. So lässt sich z. B. ein durch mehrere Threads genutzter Zähler *ohne* implizites oder explizites Lock-Objekt realisieren.

Viele **Atomic**-Klassen arbeiten als Wrapper, indem sie einen primitiven Wert in ein Objekt verpacken (z. B. **AtomicBoolean**, **AtomicInteger**, **AtomicLong**). Abweichend von den im Abschnitt 5.3 beschriebenen Wrapper-Klassen (z. B. **Boolean**, **Integer**, **Long**) sind die **Atomic**-Objekte allerdings veränderlich.

Oft werden die **Atomic**-Objekte als bessere Alternative zu **volatile**-Variablen betrachtet (siehe z. B. Goetz 2006, S. 325). Der **volatile**-Modifikator (siehe Abschnitt 15.2.6.1) sorgt für:

- Atomarität von Wertveränderungen bei 64-Bit - Typen (**long**, **double**)
- sowie für die Sichtbarkeit einer Wertveränderung bei beliebigen Typen für alle Threads

Die wichtigen Prä- und Post-Inkrement- sowie Dekrementoperatoren arbeiten jedoch trotz **volatile**-Dekoration *nicht* atomar, sodass die simultane Verwendung durch mehrere Threads zu Fehlern führen kann. Benutzt z. B. eine Klasse die folgende Konstruktion für eine Zählvariable, dann können

zwei simultane Aufrufe der Methode `incrementCounter()` durch verschiedene Threads zu einem lediglich um eins (statt zwei) erhöhten Wert führen:

```
private volatile int counter;

public void incrementCounter() {
    counter++;
}
```

Demgegenüber bieten die **Atomic**-Klassen für wichtige zusammengesetzte Operationen eine atomare Behandlung, z. B. bei **AtomicInteger**:

- **public final int incrementAndGet()**, **public final int getAndIncrement()**
  Diese Methoden leisten eine atomare Prä- bzw. Postinkrementoperation.
- **public final int decrementAndGet()**, **public final int getAndDecrement()**
  Diese Methoden leisten eine atomare Prä- bzw. Postdekrementoperation.
- **public final int getAndAdd(int** *delta***)**, **public final int addAndGet(int** *delta***)**
  Man erhält den Wert der verpackten Variablen, bevor bzw. nachdem der Parameterwert addiert worden ist.

Hier ist die Thread-sichere Alternative für den problematischen Code aus dem letzten Beispiel:

```
private AtomicInteger counter;

public void incrementCounter() {
    counter.incrementAndGet();
}
```

Wenn es bei einer Thread-Koordination lediglich um das Zählen oder um die Vergabe von eindeutigen Indizes geht, bieten die **Atomic**-Klassen eine sichere und einfache Lösung.

Dabei verwenden die **Atomic**-Klassen keine von den oben beschriebenen (schwergewichtigen) Lock-Techniken, sondern arbeiten mit der in modernen CPUs vorhandenen Compare-and-Set - Instruktion (CAS) zur Unterstützung von konkurrierenden Variablenzugriffen. Diese Thread-sicher realisierte Instruktion prüft, ob eine Variable den erwarteten Wert hat, und setzt sie genau dann auf den gewünschten Wert. In der Klasse **AtomicInteger** steht die folgende CAS-basierte Methode zur Verfügung, auf der andere Methoden (siehe oben) aufbauen, um atomare Operationen zu realisieren:[1]

public final boolean compareAndSet(int *expectedValue*, **int** *newValue***)**

Weil die CAS-Basismethode im Wesentlichen mit einem einzigen Maschinenbefehl auskommt, bieten die darauf aufbauenden **Atomic**-Klassen Thread-sichere Operationen bei minimalen Kosten.

Nach Oechsle (2018, S. 165f) darf man sich die Arbeitsweise der **AtomicInteger**-Methode **incrementAddGet()**, die eine Thread-sichere Präinkrement-Operation realisiert, so vorstellen:

[1] In der API-Klasse **AtomicInteger** aus Java 13 ist sinngemäß der beschriebene Aufbau realisiert, was aber im Quellcode durch Verwendung der Klasse **jdk.internal.misc.Unsafe** nicht unmittelbar erkennbar ist.

```
public class AtomicInteger {
   private volatile int value;

   public AtomicInteger(int initialValue) {
      value = initialValue;
   }
   public AtomicInteger() {}

   public final boolean compareAndSet(int expectedValue, int newValue) {
    . . .
   }

   public final int incrementAndGet() {
      int currentValue, newValue;
      do {
         currentValue = value;
         newValue = currentValue + 1;
      } while(!compareAndSet(currentValue, newValue));
      return newValue;
   }

 . . .
}
```

Die Methode **incrementAndGet()** versucht in einer **do-while** - Schleife, ihr Ziel zu erreichen und verwendet dabei die Methode **compareAndSet()**, welche dank Hardware-Unterstützung Thread-sicher und effizient den inkrementierten Wert einträgt, wenn der bei seiner Berechnung zugrunde gelegte alte Wert noch aktuell ist. In diesem Fall liefert **compareAndSet()** die Rückgabe **true**, und die **do-while** - Schleife endet. Mit dem Modifikator **volatile** wird dafür gesorgt, dass jede Änderung des im **AtomicInteger**-Objekt gekapselten Werts sofort in allen Threads sichtbar ist.

15.3 Direkt gestartete Threads verwalten

Die in diesem Abschnitt beschriebenen Aufgaben und Probleme sind vor allem beim traditionellen Multithreading (mit direkt gestarteten Threads) relevant. Einige Themen (z. B. Thread-Prioritäten, Deadlocks) sind aber von generellem Interesse für die Multithreading-Programmierung.

15.3.1 Weck mich, wenn Du fertig bist (join)

Wenn ein Thread erst dann weiterarbeiten möchte, wenn ein anderer Thread seine Tätigkeit beendet hat, kann er diesen mit der Methode **join()** um entsprechende Benachrichtigung bitten und dann auf die Reaktivierung warten.

Ein aus der folgenden Klasse resultierender Thread schreibt fünf Zeilen auf die Konsole:

```
class Thread1 extends Thread {
   @Override
   public void run() {
      for (int i = 0; i < 5; i++)
         System.out.println("Thread 1, i = " + i);
   }
}
```

Bevor ein Thread aus der folgenden Klasse ebenfalls fünf Zeilen schreibt, wartet er auf das Arbeits-ende eines Kollegen, den er per Konstruktor kennenlernt:

```
class Thread2 extends Thread {
   private Thread1 t1;

   Thread2(Thread1 t1) {
      this.t1 = t1;
   }

   @Override
   public void run() {
      try {
         t1.join(5000);
      } catch (InterruptedException  ie) {
         return;
      }
      for (int i = 0; i < 5; i++)
         System.out.println("Thread 2, i = " + i);
   }
}
```

Wie **sleep()**, **wait()** und **await()** reagiert auch **join()** bei einer **interrupt()** - Aufforderung an den passiven Thread mit einer **InterruptedException**.

Ist der per **join()** angesprochene Thread bereits terminiert, hat der Aufruf keinen Effekt.

Nach dem Start der beiden Threads

```
class JoinDemo {
   public static void main(String[] args) {
      Thread1 t1 = new Thread1();
      Thread2 t2 = new Thread2(t1);
      t1.start();
      t2.start();
   }
}
```

arbeiten sie nacheinander:

```
Thread 1, i = 0
Thread 1, i = 1
Thread 1, i = 2
Thread 1, i = 3
Thread 1, i = 4
Thread 2, i = 0
Thread 2, i = 1
Thread 2, i = 2
Thread 2, i = 3
Thread 2, i = 4
```

Ohne Koordination per **join()** resultiert eine schlecht vorhersehbare Sequenz:

```
Thread 1, i = 0
Thread 1, i = 1
Thread 2, i = 0
Thread 2, i = 1
Thread 2, i = 2
Thread 2, i = 3
Thread 2, i = 4
Thread 1, i = 2
Thread 1, i = 3
Thread 1, i = 4
```

In einer alternativen **join()** - Überladung kann man die maximale Wartezeit in Millisekunden angeben, z. B.

```
t1.join(5000);
```

15.3.2 Andere Threads unterbrechen, fortsetzen oder abbrechen

In diesem Abschnitt beschäftigen wir uns damit, wie ein anderer Thread unterbrochen und fortgesetzt oder abgebrochen werden kann. Im Unterschied zu der in den Abschnitten 15.2.2.2 und 15.2.3.2 beschriebenen Thread-Koordination geht die Initiative diesmal nicht vom betroffenen Thread aus, sondern von einem anderen Thread.

15.3.2.1 Unterbrechen und fortsetzen

Zum Unterbrechen und Reaktivieren eines anderen Threads sollten die früher vorgesehenen **Thread**-Methoden **suspend()** und **resume()** *nicht* mehr verwendet werden. Ein **suspend()** - Aufruf kann leicht zu einem Deadlock (siehe Abschnitt 15.3.4) führen, weil ein suspendierter Thread die in seinem Besitz befindlichen (impliziten oder expliziten) Lock-Objekte behält.

Stattdessen wird eine Inter-Thread - Kommunikation über die **Object**-Methoden **wait()** und **notify()** (bei Verwendung von synchronisierten Code-Bereichen) bzw. über die **Condition** -Methoden **await()** und **signal()** (bei Verwendung von expliziten Lock-Objekten) empfohlen (siehe Abschnitt 15.2.2.2 bzw. 15.2.3.2). Dabei wird dem Ziel-Thread z. B. über eine von ihm regelmäßig zu beobachtende Variable signalisiert, dass er sich in den Zustand **waiting** begeben soll. In seinem Besitz befindliche Monitore und Lock-Objekte werden dabei automatisch zurückgegeben, sodass kein Deadlock droht.[1] Im Produzenten-Lager-Konsumenten - Beispiel spielt der Lagerbestand die Rolle des vom Konsumenten zu beobachtenden Signals.

15.3.2.2 Abbrechen

Normalerweise endet ein Thread, wenn seine **run()** - Methode abgeschlossen ist. Es ist möglich, aber in der Regel *nicht empfehlenswert*, einen Thread von außen mit der seit Java 1.2 als veraltet bzw. herabgestuft (engl.: *deprecated*) markierten **Thread**-Methode **stop()** abzuwürgen. Anders als bei der ebenfalls herabgestuften **Thread**-Methode **suspend()** (siehe Abschnitt 15.3.2.1) besteht *keine* Deadlock-Gefahr, weil die vom gestoppten Thread belegten Monitore bzw. Lock-Objekte freigegeben werden. Aber die abrupt unterbrochene Tätigkeit kann bearbeitete Objekte in einem inkonsistenten Zustand hinterlassen, sodass bei der anschließenden Verwendung dieser Objekte durch andere Threads ein fehlerhaftes Verhalten zu befürchten ist.

Um einen Thread von außen zur Beendigung seiner Tätigkeit auffordern zu können, sollte ein entsprechendes Kommunikationsverfahren in seine **run()** - Methode integriert werden. Man kann (analog zu dem im Abschnitt 15.3.2.1 beschrieben Verfahren für das Unterbrechen eines Threads) eine Variable als Terminierungssignal verwenden. Das anschließend beschriebene Verfahren basiert auf derselben Idee, verwendet aber eine in die Klasse **Thread** integrierte Signaltechnik:

- Mit der Methode **interrupt()** wird einem Thread signalisiert, dass er seine Tätigkeit einstellen soll. Der betroffene Thread wird nicht abgebrochen, sondern sein Interrupt-Signal wird auf den Wert **true** gesetzt, falls der Java-Security-Manger keine Einwände hat.
- Ein gut erzogener Thread, der mit einem Interrupt-Signal rechnen muss, prüft in seiner **run()** - Methode regelmäßig durch Aufruf der **Thread**-Methode **isInterrupted()**, ob er sein Wirken einstellen soll. Falls ja, verlässt er die **run()** - Methode und erreicht damit den Zustand **terminated**.

[1] Das beschriebene Verfahren wird von Oracle auf der folgenden Webseite empfohlen:
http://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html

- Bei einem durch die Methoden **sleep()**, **wait()** oder **join()** aus der Klasse **Object** oder durch die Methode **await()** aus der Klasse **Condition** in den Wartezustand versetzen Thread führt der **interrupt()** - Aufruf zu einer **InterruptedException**, und bei der Ausnahmebehandlung wird über das weitere Vorgehen entschieden (siehe unten).

Wir greifen auf das Produzenten-Lager-Konsumenten - Beispiel in der Variante mit synchronisierten Methoden (siehe Abschnitt 15.2.2.1) zurück, verschaffen dem `Lager`-Objekt eine Referenz auf den Konsumenten-Thread und erweitern die vom Produzenten-Thread ausgeführte `Lager`-Methode `ergaenze()` so, dass dem Konsumenten-Thread ein Interrupt-Signal zugestellt wird, wenn nach der aktuellen Einlieferung der Lagerbestand kleiner als 50 ist:

```
synchronized void ergaenze(int add) {
   bilanz += add;
   anz++;
   System.out.println("Nr. " + anz + ":\t"+Thread.currentThread().getName() +
      " ergänzt\t" + add + "\tum " + formZeit() + " Uhr. Stand: " + bilanz);
   if (bilanz < 50) {
      System.out.println("\n" + konsument.getName() +
           " wegen Lager < 50 beendet\tum " + formZeit() + " Uhr. Stand: " + bilanz);
      konsument.interrupt();
   }
}
```

In der **while**-Schleife seiner **run()** - Methode prüft der Konsumenten-Thread, ob sein Interrupt-Signal gesetzt ist, und beendet ggf. (wenn auch murrend) seine Tätigkeit per **return**:

```
public void run() {
   while (pl. istOffen()) {
      if (isInterrupted()) {
         System.out.println("Nicht sehr kundenorientiert!\n");
         return;
      }
      pl.liefere((int) (5 + Math.random()*100));
      try {
         Thread.sleep((int) (1000 + Math.random()*3000));
      } catch(InterruptedException ie){interrupt();}
   }
}
```

Im folgenden Lagerverlauf ereilt den Konsumenten-Thread das Schicksal, von außen (durch den Produzenten-Thread) das Interrupt-Signal zu erhalten:

```
Der Laden ist offen (Bestand = 100)

Nr. 1:  Konsument entnimmt   81   um 13:57:07 Uhr. Stand: 19
Nr. 2:  Produzent ergänzt    27   um 13:57:07 Uhr. Stand: 46

Konsument wegen Lager < 50 beendet um 13:57:07 Uhr. Stand: 46
Nicht sehr kundenorientiert!

Nr. 3:  Produzent ergänzt    56   um 13:57:08 Uhr. Stand: 102
Nr. 4:  Produzent ergänzt    39   um 13:57:11 Uhr. Stand: 141
Nr. 5:  Produzent ergänzt    93   um 13:57:13 Uhr. Stand: 234
   .  .  .
Nr. 20: Produzent ergänzt    78   um 13:57:47 Uhr. Stand: 1071

Lieber Produzent, es ist Feierabend!
```

Natürlich kann das Interrupt-Signal auch durch eine im selben Thread ausgeführte Methode gesetzt werden.

Wie bereits im Abschnitt 15.1.1 erläutert, hat ein **interrupt()** - Aufruf (wie z. B. in der obigen `Lager`-Methode `ergaenze()`) an einen Thread, der sich durch einen so genannten *blockierenden Methodenaufruf*

- Methode **sleep()** oder **join()** aus der Klasse **Thread**
- Methode **wait()** aus der Klasse **Object**
- Methode **await()** aus der Klasse **Condition**

in den Wartezustand begeben hat, die folgenden Effekte:

- Der Thread wird sofort in den Zustand **ready** versetzt.
- Die für den Wartezustand verantwortliche Methode endet mit einer **InterruptedException**, und das Interrupt-Signal wird *aufgehoben* (auf **false** gesetzt).

Es ist oft sinnvoll, **interrupt()** in der **catch**-Klausel der **InterruptedException**-Behandlung erneut aufzurufen, um das Interrupt-Signal wieder auf **true** zu setzen, damit an anderer Stelle (in derselben Methode oder in einer aufrufenden Methode) passend reagiert werden kann.

Falls beim angesprochenen Thread kein Wartezustand aus den eben genannten Gründen vorliegt, wird das Interrupt-Signal gesetzt. Das kann auch einem Thread passieren, der gerade auf einen Monitor oder Lock wartet, wobei das Warten *nicht* unterbrochen wird. Weitere Informationen zur **InterruptedException** sind bei Goetz (2006) zu finden.

15.3.3 Thread-Lebensläufe

In diesem Abschnitt wird zunächst die Vergabe von Arbeitsberechtigungen für konkurrierende Threads behandelt. Dann fassen wir unsere Kenntnisse über die verschiedenen Zustände eines Threads und über Anlässe für Zustandswechsel zusammen.

15.3.3.1 Scheduling und Prioritäten

Den Bestandteil der virtuellen Maschine, der die verfügbare Rechenzeit auf die arbeitswilligen Threads verteilt, bezeichnet man als **Scheduler**. Er orientiert sich u.a. an den **Prioritäten** der Threads, die in Java Werte von 1 bis 10 annehmen können:

int-Konstante in der Klasse **Thread**	Wert
Thread.MAX_PRIORITY	10
Thread.NORM_PRIORITY	5
Thread.MIN_PRIORITY	1

Es hängt allerdings zum Teil von der Wirtsplattform ab, wie viele Prioritätsstufen wirklich unterschieden werden.

Der in einer Java-Anwendung automatisch gestartete Thread **main** hat die Priorität 5, was man unter Windows in einer Java-Konsolenanwendung über die Tastenkombination **Strg+Pause** in Erfahrung bringen kann, z. B.:

```
"main" prio=5 tid=0x00035b28 nid=0xd48 runnable [0x0007f000..0x0007fc3c]
```

Ein Thread überträgt seine aktuelle Priorität auf die bei seiner Ausführung gestarteten Threads, z. B.:

```
"Konsument" prio=5 tid=0x00ab5a40 nid=0xa74 waiting on condition [0x0ad0f000..0x0ad0fd68]
   .  .  .
"Produzent" prio=5 tid=0x00ab58c0 nid=0xfb0 waiting on condition [0x0accf000..0x0accf9e8]
```

Mit den **Thread**-Methoden **getPriority()** bzw. **setPriority()** lässt sich die Priorität eines Threads feststellen bzw. ändern.

In der Spezifikation für die virtuelle Java-Maschine wird das Verhalten des Schedulers bei der Rechenzeitvergabe an die Threads nicht sehr präzise beschrieben. Er muss lediglich sicherstellen, dass die einem Thread zugeteilte Rechenzeit mit der Priorität ansteigt.

In der Regel kommt von den arbeitswilligen Threads derjenige mit der höchsten Priorität zum Zug, jedoch kann der Scheduler Ausnahmen von dieser Regel machen, z. B. um das *Verhungern* (engl. *starvation*) eines anderen Threads zu verhindern, der permanent auf Konkurrenten mit höherer Priorität trifft. Daher darf der korrekte Ablauf eines Programms nicht davon abhängig sein, dass sich die Rechenzeitvergabe an Threads in einem strengen Sinn an den Prioritäten orientiert.

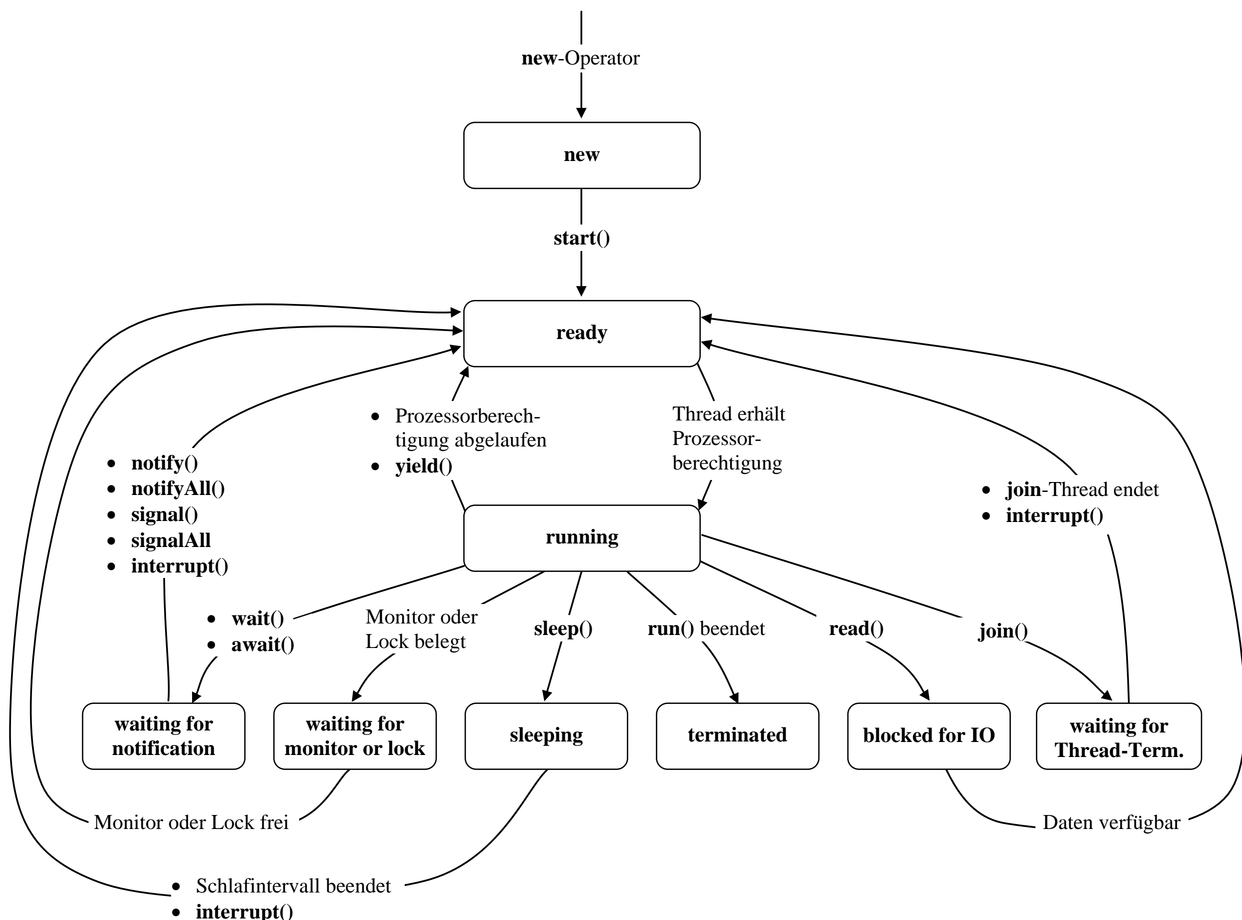
Weil der JVM-Scheduler eng mit dem Wirtsbetriebssystem zusammenarbeiten muss, besteht bei der Verteilung von Rechenzeit auf mehrere Threads mit *gleicher* Priorität *keine vollständige* Plattformunabhängigkeit. Auf einigen Plattformen (z. B. Windows) kommt das **preemtive Zeitscheibenverfahren** zum Einsatz:

- Threads gleicher Priorität werden reihum (*Round-Robin*) jeweils für eine festgelegte Zeitspanne ausgeführt.
- Ist die Zeitscheibe eines Threads verbraucht, wird er vom Scheduler in den Zustand **ready** versetzt, und der Nachfolger erhält Zugang zu einem Prozessor.

Über die Methode **yield()** kann ein **Thread** seine Zeitscheibe freiwillig abgeben und sich wieder in die Warteschlange der rechenwilligen Threads einreihen.

15.3.3.2 Zustände von Threads

In der folgenden Abbildung werden für einen explizit kreierten Thread, der nicht von einem Thread-Pool (siehe Abschnitt 15.4) verwaltet wird, wichtige Zustände und Anlässe für Zustandsübergänge dargestellt:



Die **Thread**-Methode **getState()** meldet den Zustand **RUNNABLE**, wenn ein Thread aus der Sicht der JVM ausgeführt werden kann. Diesem Status entsprechen in der Abbildung die Zustände **ready** und **running**, die von der Zuteilung der Prozessorberechtigung durch das Wirtsbetriebssystem abhängen.

15.3.4 Deadlock

Wer sich beim Einsatz von Monitoren oder expliziten Lock-Objekten zur Thread-Synchronisation ungeschickt anstellt, kann einen so genannten *Deadlock* (deutsch: eine *Systemverklemmung*) produzieren, wobei sich Threads gegenseitig blockieren. Im folgenden Beispiel sind die beiden **ReentrantLock**-Objekte `lock1` und `lock2` im Spiel:

```
import java.util.concurrent.locks.*;

class Deadlock {
   static Lock lock1 = new ReentrantLock();
   static Lock lock2 = new ReentrantLock();

   public static void main(String[] args) {
      (new T1()).start();
      (new T2()).start();
   }
}

class T1 extends Thread {
   @Override
   public void run() {
      Deadlock.lock1.lock();
      System.out.println("Thread 1 besitzt Lock 1.");
      try {Thread.sleep(100);} catch (Exception e) {interrupt();}
      System.out.println("Thread 1 moechte Lock 2 erwerben.");
      Deadlock.lock2.lock();
      System.out.println("Thread 1 besitzt Lock 2.");
      Deadlock.lock2.unlock();
      Deadlock.lock1.unlock();
   }
}

class T2 extends Thread {
   @Override
   public void run() {
      Deadlock.lock2.lock();
      System.out.println("Thread 2 besitzt Lock 2.");
      try {Thread.sleep(100);} catch (Exception e) {interrupt();}
      System.out.println("Thread 2 moechte Lock 1 erwerben.");
      Deadlock.lock1.lock();
      System.out.println("Thread 2 besitzt Lock 1.");
      Deadlock.lock1.unlock();
      Deadlock.lock2.unlock();
   }
}
```

Zwei Threads versuchen jeweils, beide Sperrobjekte zu erwerben:

- Der erste Thread erwirbt `lock1`, beschäftigt sich ein Weilchen (simuliert per **sleep()** - Aufruf) und versucht dann, zusätzlich auch noch `lock2` zu erwerben.
- Der zweite Thread erwirbt `lock2`, beschäftigt sich ein Weilchen (simuliert per **sleep()** - Aufruf) und versucht dann, zusätzlich auch noch `lock1` zu erwerben.

Die beiden Threads sind im ersten Schritt erfolgreich und blockieren sich dann gegenseitig:

```
Thread 1 besitzt Lock 1.
Thread 2 besitzt Lock 2.
Thread 1 moechte Lock 2 erwerben.
Thread 2 moechte Lock 1 erwerben.
```

Wenn beide Threads in *derselben Reihenfolge* vorgehen, also z. B. beide zunächst `lock1` anstreben und danach `lock2`, kommen sie zum Erfolg, wobei sich ein Thread etwas gedulden muss, z. B.:

```
Thread 1 besitzt Lock 1.
Thread 1 moechte Lock 2 erwerben.
Thread 1 besitzt Lock 2.
Thread 2 besitzt Lock 1.
Thread 2 moechte Lock 2 erwerben.
Thread 2 besitzt Lock 2.
```

15.4 Aufgaben per Threadpool erledigen

Um zahlreiche Einzelaufgaben im Parallelbetrieb zu erledigen, muss ein Programm nicht für jeden Auftrag einen neuen Thread erzeugen und nach Erledigung wieder abschreiben. Stattdessen kann ein sogenannter *Threadpool* beauftragt werden. Neue Aufträge werden auf die verfügbaren Pool-Threads verteilt. Nach Erledigung eines Auftrags wird ein Thread nicht beendet, sondern er steht für weitere Aufgaben bereit.

15.4.1 ExecutorService

Ein bequemer und (z. B. von Bloch 2018, S. 324) empfohlener Weg zum erfolgreichen Threadpool-Einsatz führt über die statische Methode **newCachedThreadPool**() der Klasse **Executors**, z. B.:

```
ExecutorService es = Executors.newCachedThreadPool();
```

Man erhält ein Objekt aus einer Klasse, die das Interface **ExecutorService** implementiert und folglich u.a. die Methode **execute()** beherrscht:

public void execute(Runnable *runnable***)**

Die zum Parameterobjekt gehörige **run()** - Methode wird in einem eigenen Thread ausgeführt, nach Möglichkeit durch Wiederverwendung eines vorhandenen Pool-Threads, z. B.:

```
es.execute(new KonThread(lager, i));
```

Findet sich für einen unbeschäftigten Thread binnen 60 Sekunden keine neue Verwendung, wird er terminiert und aus dem Pool entfernt, sodass sich der Ressourcenverbrauch stets am Bedarf orientiert.[1]

Wer mehr Kontrolle über die Eigenschaften eines Threadpools benötigt (z. B. maximale Idle-Zeit eines ruhenden Threads, maximale Anzahl der Pool-Threads), kann ein Objekt der Klasse **ThreadPoolExecutor** verwenden. Diese Klasse implementiert ebenfalls das Interface **ExecutorService** und bietet Steuerungsoptionen über Konstruktorparameter und Methoden (z. B. **setCorePoolSize**()).

In einer weiteren Variante unseres Produzenten-Konsumenten - Beispiels wird ein Threadpool verwendet, um eine Anzahl von Konsumenten zu bedienen:

[1] Die für einen Threadpool eingestellte Minimalbesetzung (**corePoolSize**) wird allerdings bei der Thread-Terminierung wegen Untätigkeit nicht unterschritten. Ein per **newCachedThreadPool**() erstellter Threadpool hat vermutlich die Minimalbesetzung 1.

```
import java.util.concurrent.*;

class ProKonDemo {
   public static void main(String[] args) throws InterruptedException {
      Lager lager = new Lager(1000);
      ProThread pt = new ProThread(lager);
      pt.start();

      ExecutorService es = Executors.newCachedThreadPool();
      for (int i = 1; i <= 5; i++) {
         es.execute(new KonThread(lager, i));
         Thread.sleep(3000);
      }
   }
}
```

Ein Objekt der leicht modifizierten Konsumentenklasse beendet seine Einkaufstour nach `MANZ` Zugriffen:

```
class KonThread extends Thread {
   private Lager lager;
   private int custID;
   final static private int MANZ = 2;
   private int nr;

   KonThread(Lager lager, int custID) {
      super ("Kunde Nr " + custID);
      this.custID = custID;
      this.lager = lager;
   }

   @Override
   public void run() {
      while (nr++ < MANZ) {
         lager.liefere((int) (5 + Math.random()*100), custID);
         try {
            Thread.sleep((int) (1000 + Math.random()*3000));
         } catch(InterruptedException ie) {interrupt();}
      }
      System.out.println("\nDer Kunde " + custID + " hat keine Wünsche mehr.\n");
   }
}
```

Wie das folgende Ablaufprotokoll zeigt, versorgt z. B. der Thread **pool-1-thread-1** nacheinander die Kunden 1, 3 und 5:

```
Der Laden ist offen (Bestand = 1000)

Nr. 1:   Produzent ergänzt                        593   um 17:03:11 Uhr. Stand: 1593
Nr. 2:   pool-1-thread-1 (Kunde 1) entnimmt       14    um 17:03:11 Uhr. Stand: 1579
Nr. 3:   pool-1-thread-1 (Kunde 1) entnimmt       86    um 17:03:13 Uhr. Stand: 1493
Nr. 4:   Produzent ergänzt                        526   um 17:03:13 Uhr. Stand: 2019
Nr. 5:   pool-1-thread-2 (Kunde 2) entnimmt       49    um 17:03:14 Uhr. Stand: 1970
Nr. 6:   Produzent ergänzt                        590   um 17:03:15 Uhr. Stand: 2560

Der Kunde 1 hat keine Wünsche mehr.

Nr. 7:   pool-1-thread-2 (Kunde 2) entnimmt       79    um 17:03:17 Uhr. Stand: 2481
Nr. 8:   pool-1-thread-1 (Kunde 3) entnimmt       32    um 17:03:17 Uhr. Stand: 2449

Der Kunde 2 hat keine Wünsche mehr.
```

```
Nr. 9:   Produzent ergänzt                          528   um 17:03:18 Uhr. Stand: 2977
Nr. 10:  pool-1-thread-1 (Kunde 3) entnimmt         30    um 17:03:19 Uhr. Stand: 2947
Nr. 11:  pool-1-thread-2 (Kunde 4) entnimmt         57    um 17:03:20 Uhr. Stand: 2890
Nr. 12:  Produzent ergänzt                          593   um 17:03:20 Uhr. Stand: 3483

Der Kunde 3 hat keine Wünsche mehr.

Nr. 13:  Produzent ergänzt                          562   um 17:03:23 Uhr. Stand: 4045
Nr. 14:  pool-1-thread-1 (Kunde 5) entnimmt         33    um 17:03:23 Uhr. Stand: 4012
Nr. 15:  pool-1-thread-2 (Kunde 4) entnimmt         55    um 17:03:24 Uhr. Stand: 3957
Nr. 16:  pool-1-thread-1 (Kunde 5) entnimmt         27    um 17:03:25 Uhr. Stand: 3930
Nr. 17:  Produzent ergänzt                          555   um 17:03:26 Uhr. Stand: 4485

Der Kunde 4 hat keine Wünsche mehr.

Der Kunde 5 hat keine Wünsche mehr.

Nr. 18:  Produzent ergänzt                          515   um 17:03:30 Uhr. Stand: 5000
Nr. 19:  Produzent ergänzt                          597   um 17:03:32 Uhr. Stand: 5597
Nr. 20:  Produzent ergänzt                          585   um 17:03:36 Uhr. Stand: 6182

Der Produzent macht Feierabend.
```

Das IntelliJ-Projekt zum Beispiel finden Sie im Ordner

...\BspUeb\Multithreading\Threadpool\ExecutorService

Ein **ExecutorService** bietet noch weitere Optionen zur flexiblen Verwaltung von Aufgaben, z. B.:

- Man kann den **ExecutorService** mit **shutdown()** beauftragen, keine neuen Aufgaben anzunehmen, die anstehenden Aufgaben zu erledigen und dann seinen Dienst zu quittieren. Die blockierende Methode **awaitTermination()** kehrt zurück, wenn ein **ExecutorService** nach der **shutdown()** - Aufforderung seinen Dienst beendet hat.
- Über die blockierenden Methoden **invokeAny()** bzw. **invokeAll()** übergibt man eine Kollektion mit Aufgaben an den **ExecutorService** und wartet dann, bis irgendeine Aufgabe beendet ist bzw. bis alle Aufgaben beendet sind. Man erhält ein Ergebnis zurück, und die übergebenen Aufgaben müssen das Interface **Callable<T>** erfüllen, das im nächsten Abschnitt beschrieben wird.

15.4.2 Interface Callable<V>

Um die Kommunikation zwischen Threads bzw. Aufgaben zu verbessern wurde in Java 1.5 (bzw. 5.0) das generische Interface **Callable<V>** eingeführt, das ausschließlich die Methode **call()** vorschreibt:

```
public interface Callable<V> {
   V call() throws Exception;
}
```

Implementiert eine Klasse dieses Interface, kann die **call()** - Methode eines Objekts in einem eigenen Thread ausgeführt werden. Das gelingt zwar auch mit der **run()** - Methode einer Klasse, die das Interface **Runnable** implementiert (vgl. Abschnitt 15.1.2), doch bietet das Interface **Callable<V>** einige Vorteile, die nun zu erläutern sind.

Im Unterschied zur **Runnable**-Methode **run()**, liefert die **Callable**-Methode **call()** einen Rückgabewert. Natürlich lässt sich eine Ergebnisübergabe auch per **Runnable**-Klasse realisieren, wobei aber ein höherer Aufwand erforderlich ist, z. B.:

- Ergebnis in einer Instanzvariablen speichern
- Abfragemethode anbieten

Außerdem darf **call()** eine vorbereitungspflichtige Ausnahme (vgl. Abschnitt 11.4.2) werfen, was der Methode **run()** aufgrund der **Runnable**-Definition verboten ist. Allerdings ist die Methode **call()** nicht dazu verpflichtet, eine Ausnahme anzukündigen und zu werfen.

Es ist *nicht* möglich, ein **Callable**-Objekt als Argument an einen **Thread**-Konstruktor zu übergeben. Zur Ausführung in einem eigenen Thread wird stattdessen ein Objekt aus einer Klasse benötigt, die das Interface **ExecutorService** implementiert und daher u.a. die Methode **submit()** beherrscht:

```
public interface ExecutorService extends Executor {
   <T> Future<T> submit(Callable<T> task);
    . . .
}
```

Die generische Methode **submit()** erwartet einen Parameter vom Typ **Callable<V>** und liefert eine Rückgabe vom Typ **Future<V>**. Durch die generische Schnittstelle **Future<V>** wird das Ergebnis einer asynchron (in einem anderen Thread) Operation repräsentiert. Es sind u.a. Methoden vorgeschrieben, ...

- um den Status der Operation abzufragen (**isDone**()),
- um das Ergebnis anzufordern (**get**()).

Ein **ExecutorService**-Objekt verschafft man sich in der Regel mit einer statischen Methode der Klasse **Executors**, die wir im Abschnitt 15.4.1 kennengelernt haben, z. B.:

```
ExecutorService es = Executors.newSingleThreadExecutor();
```

Von der Methode **newSingleThreadExecutor()** erhält man einen Exekutor, der einen *einzelnen* Thread für verschiedene Aufgaben verwenden kann.

Nun wird es Zeit, eine das Interface **Callable<V>** implementierende Beispielklasse vorzustellen:

```
import java.util.concurrent.Callable;
import java.util.concurrent.TimeoutException;

class RandomNumberCruncher implements Callable<Double> {
    public Double call() throws TimeoutException {
        final int mxNum = 100_000_000;
        final int mxTime = 5_000;
        double d = 0.0;
        long start = System.currentTimeMillis();
        for (int i = 0; i < mxNum; i++) {
            d += Math.random();
            if (System.currentTimeMillis() - start > mxTime)
                throw new TimeoutException();
        }
        return d/mxNum;
    }
}
```

Objekte dieser Klasse berechnen in ihrer **call()** - Methode das arithmetische Mittel von reichlich vielen **double**-Zufallszahlen aus dem Intervall [0, 1), realisieren also eine Zufallsgröße mit dem Erwartungswert 0,5.

Um eine Berechnung zu starten und an das Ergebnis heran zu kommen, wird die **ExecutorService**-Methode **submit()** mit einem `RandomNumberCruncher`-Objekt als Aktualparameter aufgerufen:

```
import java.text.DateFormat;
import java.util.Date;
import java.util.concurrent.*;

class CallableDemo {
  public static void main(String[] args) throws InterruptedException {
    DateFormat df = DateFormat.getDateTimeInstance();
    ExecutorService es = Executors.newSingleThreadExecutor();
    Future<Double> fd = es.submit(new RandomNumberCruncher());
    while (!fd.isDone()) {
      System.out.println("Warten auf call() - Ende (" + df.format(new Date()) + ")");
      Thread.sleep(1_000);
    }
    try {
      System.out.println("\nMittelwert der Zufallszahlen: " + fd.get());
    } catch (Exception e) {
      System.out.println("\nException beim get() - Aufruf:\n  "+e);
    }
    es.shutdown();
  }
}
```

Der **submit()** - Aufruf endet sofort und liefert im Beispiel als Rückgabe ein Objekt aus Klasse, die das Interface **Future<Double>** implementiert. Über die Methode **isDone()** kann man sich bei diesem Objekt über die Fertigstellung des Auftrags informieren. Seine Methode **get()** liefert schließlich die **call()** - Rückgabe oder leitet ggf. ein von **call()** geworfenes Ausnahmeobjekt weiter.

Bei einem gelungenen Aufruf (ohne **TimeoutException**) lieferte das Beispielprogramm die Ausgabe:

```
Warten auf call() - Ende (22.03.2020 07:41:49)
Warten auf call() - Ende (22.03.2020 07:41:50)
Warten auf call() - Ende (22.03.2020 07:41:51)

Mittelwert der Zufallszahlen: 0.5000057115593642
```

Der vom **ExecutorService** verwaltete Thread (mit dem Namen **pool-1-thread-1**) verbleibt nach Abschluss des **call()** - Aufrufs in Parkstellung und verhindert das Programmende:

```
"pool-1-thread-1" #10 prio=5 os_prio=0 tid=0x0000000019844000 nid=0x3638 waiting on
condition [0x0000000019e3f000]
   java.lang.Thread.State: WAITING (parking)
```

Im Beispiel wird der überflüssig gewordene **ExecutorService** über seine **shutdown**()-Methode gestoppt:

```
es.shutdown();
```

Beim folgenden Programmablauf hat sich die **call()** - Methode mit dem Werfen einer **Timeout-Exception** verabschiedet:

```
Warten auf call() - Ende (22.03.2020 07:45:59)

Exception beim get() - Aufruf:
  java.util.concurrent.ExecutionException: java.util.concurrent.TimeoutException
```

Das IntelliJ-Projekt mit dem Beispielprogramm finden Sie im Ordner

...\BspUeb\Multithreading\Threadpool\Callable

15.4.3 Threadpools mit Timer-Funktionalität

Mit der im Manuskript nicht behandelten Klasse **Timer** (im Paket **java.util**) ist es möglich, Aufgaben einmalig oder regelmäßig zu einer vorbestimmten Zeit in einem Hintergrund-Thread ausführen zu lassen. Bei einer periodisch auszuführenden Aufgabe werden zeitliche Überschneidungen von aufeinanderfolgenden Episoden verhindert. Bevor die nächste Wiederholung gestartet wird, muss also die vorherige beendet sein. Als Einschränkungen der Klasse **Timer** sind zu nennen:

- Es wird nur *ein* Thread verwendet, der alle anstehenden Aufgaben nacheinander ausführt.
- Als Aufgaben sind nur Objekte aus einer von **TimerTask** abstammenden Klasse zugelassen.

Die Klasse **ScheduledThreadPoolExecutor** im Paket **java.util.concurrent** bietet mehr Flexibilität, denn:

- Es wird ein Threadpool mit festem Umfang verwendet.
  Die Pool-Threads können flexibel für unterschiedliche Aufgaben eingesetzt und damit effizient genutzt werden.
- Als Aufgaben sind Objekte aus einer das Interface **Runnable** implementierenden Klasse zugelassen.

Im folgenden Beispiel kommt ein **ScheduledThreadPoolExecutor**-Objekt zum Einsatz, das maximal zwei Threads einsetzen darf:

```
import java.util.concurrent.*;
class ScheduledThreadPoolExecutorDemo {
  public static void main(String[] args) throws InterruptedException {
    ScheduledThreadPoolExecutor stpe = new ScheduledThreadPoolExecutor(2);
    stpe.scheduleAtFixedRate(new ScheduledRunner(1, stpe), 0, 1000, TimeUnit.MILLISECONDS);
    stpe.scheduleAtFixedRate(new ScheduledRunner(2, stpe), 2000, 2000, TimeUnit.MILLISECONDS);
    stpe.scheduleAtFixedRate(new ScheduledRunner(3, stpe), 3000, 3000, TimeUnit.MILLISECONDS);
    Thread.sleep(5000);
    stpe.shutdown();
  }
}
```

Dem **ScheduledThreadPoolExecutor** werden über seine Methode **scheduleAtFixedRate()** drei Aufträge erteilt, wobei jeweils mit individuellem Zeitplan (Initialverzögerung und Startabstand) die **run()** - Methode eines Objekts der folgenden Klasse auszuführen ist:

```
import java.util.concurrent.ScheduledThreadPoolExecutor;
class ScheduledRunner implements Runnable {
    private int nr;
    private ScheduledThreadPoolExecutor stpe;
    public ScheduledRunner(int nr, ScheduledThreadPoolExecutor stpe) {
        this.nr = nr;
        this.stpe = stpe;
    }
    private String formZeit() {
        return java.text.DateFormat.getTimeInstance().format(new java.util.Date());
    }
    public void run() {
        System.out.println("ScheduledRunner " + nr + ", Zeit: " + formZeit() +
                ", Aktive Pool-Threads: " + stpe.getActiveCount());
        try {Thread.sleep(500*nr);
        } catch (InterruptedException ignored) {Thread.currentThread().interrupt();}
    }
}
```

Wie das folgende Ablaufprotokoll zeigt, variiert die Anzahl der aktiven Threads, wobei das Maximum 2 nicht überschritten wird:

```
ScheduledRunner 1, Zeit: 21:12:15, Aktive Pool-Threads: 1
ScheduledRunner 1, Zeit: 21:12:16, Aktive Pool-Threads: 1
ScheduledRunner 1, Zeit: 21:12:17, Aktive Pool-Threads: 1
ScheduledRunner 2, Zeit: 21:12:17, Aktive Pool-Threads: 2
ScheduledRunner 1, Zeit: 21:12:18, Aktive Pool-Threads: 2
ScheduledRunner 3, Zeit: 21:12:18, Aktive Pool-Threads: 2
ScheduledRunner 1, Zeit: 21:12:19, Aktive Pool-Threads: 2
ScheduledRunner 2, Zeit: 21:12:20, Aktive Pool-Threads: 2
ScheduledRunner 1, Zeit: 21:12:20, Aktive Pool-Threads: 2
```

Die **main()** - Methode der Startklasse fordert den **ScheduledThreadPoolExecutor** nach fünf Sekunden per **shutdown()** - Methode auf, seine Tätigkeit einzustellen. Daraufhin werden keine neuen Ausführungen mehr begonnen, sodass die Pool-Threads nach einiger Zeit enden. Laufende Ausführungen werden aber noch zu Ende geführt.

Weitere Regeln für die Umsetzung der Zeitpläne:

- Die nächste Ausführung eines Auftrags wird erst dann gestartet, wenn die vorherige abgeschlossen ist.
- Endet eine Ausführung eines Auftrags mit einer Ausnahme, finden keine weiteren Ausführungen dieses Auftrags mehr statt.
- Sind alle Pool-Threads im Einsatz, kann sich der Start einer Ausführung verzögern.

15.5 Fork-Join - Framework

Durch moderne Multithreading-Frameworks wird das Ziel verfolgt, die Programmierer von den Details und Risiken der Multithreading-Programmierung zu entlasten, um die Nutzung der parallelen Programmierung zu fördern. Beim Fork-Join - Framework geht es nicht um die quasi-gleichzeitige Ausführung mehrerer Aufgaben, sondern um die beschleunigte Erledigung *einer* Aufgabe durch die Zerlegung in simultan ausführbare Teilaufgaben. Wir werden das Framework im Abschnitt 15.5.1 direkt verwenden. Im Abschnitt 15.5.2 über die parallelen Aggregatoperationen mit Strömen ist das Framework als Basistechnik ebenfalls involviert.

15.5.1 Direkte Verwendung des Fork-Join - Frameworks

Das mit Java 7 eingeführte Fork-Join - Framework unterstützt die Aufteilung einer Aufgabe zur parallelen Bearbeitung durch mehrere CPU-Kerne. Durch Verzweigung (*forking*) entstehen separate „Produktionsstraßen“, die später wieder zusammengeführt werden (*joining*). Voraussetzung ist eine Gesamtaufgabe, die sich in unabhängig ausführbare Teilaufgaben zerlegen lässt, sodass mehrere Threads *ohne nennenswerten Koordinierungsbedarf* jeweils eine Teilaufgabe erledigen können. Trotz der speziellen Anforderungen finden sich zahlreiche Beispiele für eine erfolgreiche Anwendung des Fork-Join - Frameworks:

- In der Statistik ist zum Schätzen von Mittelwert bzw. Varianz für einen Array mit Zahlen die Summe der einfachen bzw. quadrierten Werte zu bestimmen. Man kann segmentweise Zwischensummen berechnen, die später zusammengeführt werden.
- Ein weiteres Beispiel aus dem Bereich der Statistik sind Bootstrap-Schätzmethoden, wobei aus einer Primärstichprobe zahlreiche (z. B. 1000) Sekundärstichproben gezogen werden, um daraus jeweils dieselben Parameterschätzungen zu berechnen.
- Beim Filtern einer Bitmap-Grafik lässt sich die Gesamtaufgabe in einzelne Kacheln zerlegen.

Zur Lösung einer Aufgabe verwendet man im Fork-Join - Framework ein rekursives Verfahren, das folgendermaßen beschrieben werden kann:

Liegt der Umfang der Aufgabe unterhalb einer Schwelle?	
Ja	**Nein**
Erledige die Aufgabe (z.B. mit einer sequentiellen Technik).	**Zerlege die Aufgabe in zwei Teilaufgaben.** **Lasse die Teilaufgaben vom Framework in eigenen Threads erledigen.** **Warte auf die Fertigstellung der Teilaufgaben.** **Führe die Ergebnisse zusammen.**

Das Verfahren wird vom Fork-Join - Framework so weit wie möglich automatisiert, wobei ein Threadpool zum Einsatz kommt. Anwendungsprogrammierer haben Einfluss auf zwei Stellgrößen des Verfahrens:

- Wie viele Threads sollen beteiligt sein?
  Als in der Regel geeignete Voreinstellung verwendet das Framework die Anzahl der verfügbaren CPU-Kerne.
- Wie viele Teilaufgaben sollen gebildet werden?
  Über die Anzahl der Teilaufgaben entscheidet man durch die Wahl des Aufteilungskriteriums. In der Regel wählt man die Zahl der Teilaufgaben *höher* als die Zahl der verfügbaren CPU-Kerne, um dem Framework Flexibilität bei der Auftragsplanung zu lassen. Diese kommt z. B. dann zum Tragen, wenn die Teilaufgaben unterschiedlich lange Bearbeitungszeiten haben (Grossmann 2012). In der Praxis zeigt sich, dass die Anzahl der Teilaufgaben abgesehen von extremen Fällen (überhaupt keine Aufteilung, maximale Zersplitterung) wenig Einfluss auf die gesamte Bearbeitungsdauer hat.

Für eine einfache Anwendung des Fork-Join - Frameworks werden die folgenden Klassen (aus dem Paket **java.util.concurrent.forkjoin**) benötigt:

- Zur **Modellierung einer Teilaufgabe** verwendet man eine Ableitung der generischen Klasse **ForkJoinTask<T>** als Basisklasse für eine eigene, aufgabenspezifische Klassendefinition. Müssen die Teilaufgaben *kein* Ergebnis melden (z. B. beim Zerlegen einer Bitmap-Filterung auf einzelne Kacheln), dann verwendet man die Klasse **RecursiveAction** als Basisklasse, anderenfalls kommt die Klasse **RecursiveTask<T>** zum Einsatz. In der eigenen Aufgabenklasse sind folgende Kompetenzen zu implementieren:
  - Direktlösung einer hinreichend kleinen Aufgabe
  - Rekursives Abspalten von „halbierten“ Teilaufgaben
- Für die **Verwaltung der Teilaufgaben und der Pool-Threads** verwendet man ein Objekt der Klasse **ForkJoinPool**, die das Interface **ExecutorService** implementiert. Jeder Pool-Thread besitzt eine Warteschlange von Teilaufgaben, die er zu erledigen hat. Wenn ein Thread die eigene Aufgabenwarteschlange abgearbeitet hat und warten muss, dann übernimmt er Teilaufgaben aus den Warteschlangen anderer Pool-Threads (*work stealing*). Zum Starten der Auftragsabwicklung erteilt man der **ForkJoinPool**-Instanz den Auftrag **invoke()** und übergibt als Parameter ein Objekt der aufgabenspezifischen eigenen Klasse, das den kompletten Arbeitsumfang repräsentiert.

Als Beispiel bietet sich die Berechnung der Fakultät an, weil Sie im Rahmen einer Übungsaufgabe zum Kapitel 12 (über das funktionale Programmieren) bereits eine Multithreading-Lösung zu dieser

Aufgabe durch Reduktion eines parallelen Stroms erstellt haben (siehe auch Abschnitt 12.2.5.4.2). Zur Parallelverarbeitung von Strömen wurde im Kapitel 12 berichtet, dass im Hintergrund das Fork-Join - Framework zum Einsatz kommt. Nun machen wir uns daran, eine explizite Lösung mit der Fork-Join - Technik zu erstellen.

In der von **RecursiveTask<BigDecimal>** abstammenden Klasse `FacTask`

```
private static class FacTask extends RecursiveTask<BigDecimal> {
   private int start, ende;
   private int schwelle;

   public FacTask(int start, int ende, int schwelle) {
      this.start = start;
      this.ende = ende;
      this.schwelle = schwelle;
   }

   private BigDecimal product(int start, int ende) {
      BigDecimal fac = new BigDecimal(start);
      for (int i = start+1; i <= ende; i++)
         fac = fac.multiply(new BigDecimal(i));
      return fac;
   }

   @Override
   protected BigDecimal compute() {
      BigDecimal fac;
      int umfang = ende - start;
      if (umfang <= schwelle)
         fac = product(start, ende);
      else {
         int haelfte = umfang/2;
         FacTask task1 = new FacTask(start, start+haelfte, schwelle);
         FacTask task2 = new FacTask(start+haelfte+1, ende, schwelle);
         task2.fork();
         BigDecimal erg1 = task1.compute();
         BigDecimal erg2 = task2.join();
         fac = erg1.multiply(erg2);
      }
      return fac;
   }
}
```

ist eine (Teil)Aufgabe definiert durch (siehe Konstruktor):

- Start- und Endindex der zu bearbeitenden Teilfolge
- Schwellenwert für eine hinreichend kleine, direkt zu bearbeitende Teilfolge

Die Methode `product()` ist für den simplen Job zuständig, eine hinreichend kleine Aufgabe direkt zu lösen, also das Produkt $a \cdot (a+1) \cdot (a+2) \cdot \ldots \cdot (b-1) \cdot b$ für die Teilfolge von a bis b zu berechnen. Weitaus interessanter ist die Methode **compute()**, die sich nach einer Umfangsbeurteilung zwischen der Direktlösung und der Aufgabenzerlegung entscheidet. Bei einer Aufgabenzerlegung ...

- werden zwei neue FacTask-Objekte mit einem ungefähr halbiertem Umfang gebildet.
- Dann wird dem Framework eine Teilaufgabe (task2) durch einen Aufruf der Methode **fork()** zur parallelen Bearbeitung übergeben, wobei in der Regel ein anderer Pool-Thread zum Einsatz kommt. Dieser Methodenaufruf kehrt sofort zurück.[1]
- Anschließend wird das erste Teilaufgabenobjekt (task1) durch einen Aufruf seiner Methode **compute()** aufgefordert, im aktiven Thread seine Aufgabe zu erledigen. Der **compute()** - Aufruf kehrt erst dann zurück, wenn die Teilaufgabe erledigt ist.
- Nach Rückkehr des **compute()** - Aufrufs wird das zweite Teilaufgabenobjekt (task2) durch die Methode **join()** aufgefordert, sein Ergebnis abzuliefern. Liegt das Ergebnis noch nicht vor, kümmert sich der aktive Thread um andere Teilaufgaben in seiner eigenen Warteschlange. Ist diese leer, übernimmt er Teilaufgaben aus den Warteschlangen anderer Pool-Threads (Goetz 2007). Somit verhält sich die **ForkJoinTask<T>** - Methode **join()** deutlich anders als die gleichnamige Methode der Klasse **Thread** (vgl. Abschnitt 15.3.1).
- Sind beide Teilaufgaben abgeschlossen, werden die Ergebnisse zusammengefasst. Im Beispiel sind dazu die Ergebnisse aus den beiden Teilfolgen miteinander zu multiplizieren.

Um die Berechnung der Fakultät über eine statische Methode namens factorial() bequem nutzbar zu machen, wird im Beispiel die Klasse FacForkJoin definiert und die Aufgabenklasse FacTask als statische Mitgliedsklasse implementiert:

```
import java.math.BigDecimal;
import java.util.concurrent.*;

class FacForkJoin {
   static public BigDecimal factorial(int argument, int schwelle) {
      FacTask task = new FacTask(1, argument, schwelle);
      ForkJoinPool pool = new ForkJoinPool();
      pool.invoke(task);
      pool.shutdown();
      return task.join();
   }

   static public BigDecimal factorial(int argument) {
      return factorial(argument, Runtime.getRuntime().availableProcessors()*10);
   }

   private static class FacTask extends RecursiveTask<BigDecimal> {
    . . .
   }
}
```

In der statischen FacForkJoin-Methode factorial() wird ...

- ein Objekt der Aufgabenklasse FacTask mit der kompletten Aufgabe erstellt,
- ein Objekt der Aufgabenverwaltungsklasse **ForkJoinPool** erstellt,
- die Bearbeitung durch einen Aufruf der **ForkJoinPool**-Methode **invoke()** gestartet, die als Parameter das FacTask-Objekt erhält,
- der **ForkJoinPool** mit der Methode **shutdown()** beauftragt, sich nach Erledigung des Auftrags zu beenden,
- durch einen Aufruf der **ForkJoinTask<BigDecimal>** - Methode **join()** das Ergebnis ermittelt und an den Aufrufer gemeldet.

[1] Die **ForkJoinPool**-Methode **invoke()**, mit der das gesamte Fork-Join - Verfahren gestartet wird (siehe unten), sollte keinesfalls aus der **compute()** - Methode eines **RecursiveTask<T>** - oder **RecursiveAction**-Objekts gestartet werden (Grossmann 2012).

Damit ist die Fork-Join - Technik zur Bestimmung der Fakultät leicht anzuwenden, z. B.:

```
import java.math.BigDecimal;
import java.util.stream.IntStream;

class ForkJoinTest {
  public static void main(String[] args) {
    int argument = 50_000;
    long zeit;
    BigDecimal ergebnis;
    zeit = System.currentTimeMillis();
    ergebnis = FacForkJoin.factorial(argument);
    System.out.printf("\nLaufzeit mit Fork-Join in Millisekunden:%7d (Ergebnis: %e)",
                     (System.currentTimeMillis()-zeit), ergebnis);
  }
}
```

Mit einer etwas erweiterten Variante des obigen Testprogramms wurden Laufzeitvergleiche zwischen einer Single-Thread - Lösung und der Fork-Join - Lösung vorgenommen, z. B. (bei 5 Wiederholungen wegen variabler Ergebnisse, gemessen auf einem PC mit der Intel-CPU Core i3 mit 3,2 GHz, 2 Kerne plus Hyperthreading):

```
Laufzeit mit Fork-Join in Millisekunden:    263 (Ergebnis: 3,347321e+213236)
Laufzeit mit Fork-Join in Millisekunden:    355 (Ergebnis: 3,347321e+213236)
Laufzeit mit Fork-Join in Millisekunden:    205 (Ergebnis: 3,347321e+213236)
Laufzeit mit Fork-Join in Millisekunden:     89 (Ergebnis: 3,347321e+213236)
Laufzeit mit Fork-Join in Millisekunden:    217 (Ergebnis: 3,347321e+213236)

Laufzeit mit Fork-Join in Millisekunden:   2148 (Ergebnis: 3,347321e+213236)
Laufzeit mit Fork-Join in Millisekunden:   1317 (Ergebnis: 3,347321e+213236)
Laufzeit mit Fork-Join in Millisekunden:   1444 (Ergebnis: 3,347321e+213236)
Laufzeit mit Fork-Join in Millisekunden:   1328 (Ergebnis: 3,347321e+213236)
Laufzeit mit Fork-Join in Millisekunden:   1318 (Ergebnis: 3,347321e+213236)
```

Die Überlegenheit der Multithreading-Lösung ist erheblich größer, als es bei 2 realen CPU-Kernen plus Hyperthreading zu erwarten war.

Ein IntelliJ-Projekt mit dem Beispielprogramm befindet sich im Ordner

...\BspUeb\Multithreading\Fork-Join\Direkte Verwendung

15.5.2 Parallele Aggregatoperationen mit Strömen

Im Abschnitt 12.2.5.4.2 wurde bereits ein Verfahren zur Fakultätsberechnung unter Verwendung der in Java 8 eingeführten Ströme mit seriellen bzw. parallelen Aggregatoperationen vorgestellt. Wir verwenden das Beispiel erneut und beschränken uns auf das automatisierte Multithreading bei parallelen Strömen, das auf der Fork-Join - Technik basiert. Das folgende Programm

```
import java.math.BigDecimal;
import java.util.stream.IntStream;

class FactorialByParallelStream {
  public static void main(String[] args) {
    int argument = 50_000;
    long zeit;
    BigDecimal ergebnis;
    System.out.println("\n");
    for (int i = 0; i < 5; i++) {
      zeit = System.currentTimeMillis();
      ergebnis = IntStream
               .rangeClosed(1, argument)
               .parallel()
               .mapToObj(BigDecimal::new)
               .reduce(BigDecimal.ONE, BigDecimal::multiply);
      System.out.printf("\nLaufzeit mit parallelem Strom in Millisekunden:" +
            "%7d (Ergebnis: %e)", System.currentTimeMillis()-zeit, ergebnis);
    }
  }
}
```

zeigt im Vergleich zu der im letzten Abschnitt vorgestellten expliziten Fork-Join-Lösung eine enorme Vereinfachung.

Wie bei der manuellen Fork-Join - Lösung (vgl. Abschnitt 15.5.1) erhält man für wiederholte Ausführungen desselben Auftrags deutlich variierende Laufzeiten:

```
Laufzeit mit parallelem Strom in Millisekunden:    414 (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden:    286 (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden:    198 (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden:    166 (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden:    233 (Ergebnis: 3,347321e+213236)
```

Die Parallelstrom-Lösung kann bei der Laufzeit mit der händischen Fork-Join-Lösung durchaus mithalten.

15.6 Thread-sichere Kollektionen im Paket java.util.concurrent

Die Klasse **Collections** im Paket **java.util** besitzt etliche statische Fabrikmethoden (z. B. **synchronizedList** (), **synchronizedSet()**, **synchronizedMap()**), die zu Klassen aus dem Java Collections Framework (JCF), die z. B. das Interface **List<E>**, **Set<E>** oder **Map<K,V>** implementieren, eine synchronisierte Verpackungsklasse liefern. Die so erstellten synchronisierten Kollektionen sind leicht zu erstellen, haben aber Schwächen:

- Behinderung von Threads durch das einzige Sperrobjekt (schlechte Skalierbarkeit)
  Es wird nur *eine* Sperre für das gesamte Kollektionsobjekt verwendet. Während ein Thread im Besitz dieser Sperre ist, kann kein anderer Thread auf die Kollektion zugreifen. Auch ein lesend zugreifender Thread blockiert alle anderen Threads, die entweder lesend oder schreibend zugreifen wollen. Mit der Anzahl der am Zugriff interessierten Threads steigt die Wahrscheinlichkeit dafür, dass Threads in der Bewerber-Warteschlage für das *eine* Sperrobjekt aufgehalten werden, und die Performanz der Anwendung leidet. Das synchronisierte Kollektionsobjekt sorgt als Flaschenhals für eine schlechte Skalierbarkeit der Anwendung und eignet sich daher z. B. nicht für einen geteilten Cache in einer Server-Anwendung.

- Nur *bedingte* Thread-Sicherheit
  Wird über die Elemente einer von **Collections** produzierten Verpackungsklasse iteriert, dann muss eine manuelle (externe) Synchronisierung vorgenommen werden. Darüber lässt die API-Dokumentation zur Klasse **Collections** keinen Zweifel zu:[1]

  > It is imperative that the user manually synchronize on the returned collection when traversing it via Iterator, Spliterator or Stream.

  Die von Klassen im JCF implementierten Iteratoren sind fail-fast (siehe Abschnitt 10.4): Ist die Kollektion nach der Erstellung des Iterators auf andere Weise als durch die **remove()** - Methode des Iterators strukturell geändert worden, dann werfen die kritischen Methoden des Iterators eine **ConcurrentModificationException**, um erratisches Verhalten zu verhindern. Allerdings gelingt die Detektion schon im Single-Thread-Betrieb nicht zuverlässig, und im Multi-Thread-Betrieb besteht ein großes Risiko, das durch die automatische Synchronisation nicht beseitigt wird. Weil die Thread-Sicherheit der von **Collections** produzierten Verpackungsklassen folglich nicht für *alle* Methoden gilt, spricht man hier von einer *bedingten Thread-Sicherheit*.[2]

Das in Java 5 (alias 1.5) ergänzte Paket **java.util.concurrent** enthält u.a. Kollektionstypen, die für Programme mit intensiver Parallelität besser geeignet sind als die automatisch synchronisierten JCF-Kollektionen. Besonders populär ist die zur Verwaltung von (Schlüssel-Wert) - Paaren geeignete Klasse **ConcurrentHashMap<K,V>** mit folgenden Lösungen für die beiden eben beschriebenen Probleme (Goetz 2003; Goetz et al. 2006, S. 85f):

- Statt *ein* Kollektions-globales Sperrobjekt zu verwenden, kommen bei der Klasse **ConcurrentHashMap<K,V>** *mehrere* Sperrobjekte zum Einsatz, die jeweils mehrere Buckets schützen. Als Ergebnis der insgesamt feiner granulierten Sperrtechnik ...
  - sind simultane Lesezugriffe durch beliebig viele Threads erlaubt,
  - sind gleichzeitige Schreibzugriffe durch eine Anzahl von Threads möglich,
  - können lesende und schreibende Threads simultan agieren.
- Über die als **keySet()** - Rückgabe erhältliche Schlüsselmenge kann man iterieren, ohne die Kollektion blockieren zu müssen, um eine **ConcurrentModificationException** zu verhindern. Allerdings sollten nicht mehrere Threads gleichzeitig über die Kollektion iterieren. Der Iterator zeigt den Zustand der Kollektion zu einem Zeitpunkt während oder nach seiner Erstellung. Es wird also nicht unbedingt über die Kollektion im aktuellen Zustand iteriert. Man spricht hier von einem *schwach konsistenten Iterator* (engl.: *weekly consistent iterator*).

Das Design der Klasse **ConcurrentHashMap<K,V>** hat auch Nachteile, die aber in der Regel keine große Bedeutung haben:

- Methoden, die den Zustand der gesamten Kollektion beurteilen (z. B. **isEmpty()**, **size()**) liefern kein perfekt verlässliches Ergebnis, wenn gleichzeitig Schreibzugriffe durch andere Threads stattfinden.
- Die Zugangsexklusivität für einen einzigen Thread, die bei einer **synchronizedMap()** - Rückgabe als meist unerwünschter Effekt des solitären Sperrobjekts besteht, ist bei der Klasse **ConcurrentHashMap<K,V>** nicht realisierbar.

Insgesamt ist die Klasse **ConcurrentHashMap<K,V>** in der Regel gegenüber einer synchronisierten **HashMap<K,V>** - Kollektion zu bevorzugen, wenn viele Threads auf eine Abbildung zugreifen.

---

[1] https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Collections.html

[2] Während die Thread-Sicherheit der von **Collections** produzierten Verpackungsklassen beim Verhalten ihrer Iteratoren nach wie vor eingeschränkt ist, wurde sie in Java 8 deutlich verbessert durch die Erweiterung der JCF-Schnittstellen um **default**-Methoden zur Unterstützung zusammengesetzter Operationen (z. B. **putIfAbsent()** in **Map<K,V>**).

Zwei andere populäre Klassen aus dem Paket **java.util.concurrent** wurden schon im Abschnitt 15.2.4.1 vorgestellt: Die Kollektionsklassen **ArrayBlockingQueue<E>** und **LinkedBlockingQueue<E>** realisieren Thread-sichere Warteschlangen nach dem FIFO-Prinzip (First-In-First-Out).

Weitere erwähnenswerte Klassen aus dem Paket **java.util.concurrrent** sind:

- **ConcurrentSkipListMap<K,V>**
  Diese Klasse realisiert eine Thread-sichere Kollektion zur Verwaltung von (Schlüssel-Wert) - Paaren mit einem *geordneten* Schlüsseltyp. **ConcurrentSkipListMap<K,V>** steht zu **ConcurrentHashMap<K,V>** im selben Verhältnis wie **TreeMap<K,V>** zu **HashMap<K,V>**. **ConcurrentSkipListMap<K,V>** erfüllt (wie **TreeMap<K,V>**) die Schnittstelle **NavigableMap<K,V>** und ist außerdem performanter als die von **Collections.synchronizedSortedMap()** gelieferte synchronisierte Variante von **TreeMap<K,V>**. Damit eignet sich **ConcurrentSkipListMap<K,V>**, wenn Thread-Sicherheit für eine Abbildung mit geordneten Elementen gefragt ist, weil die Elemente in Sortierordnung aufgesucht werden müssen, oder eine Methode aus der Schnittstelle **NavigableMap<K,V>** benötigt wird. Wenn viele Threads auf eine Abbildung mit geordnetem Schlüsseltyp zugreifen, ist **ConcurrentSkipListMap<K,V>** in der Regel gegenüber einer synchronisierten **TreeMap<K,V>** - Kollektion zu bevorzugen.
- **ConcurrentSkipListSet<E>**
  Diese (auf **ConcurrentSkipListMap<K,V>** basierende Klasse realisiert eine Thread-sichere Menge mit geordneten Elementen. Sie erfüllt (wie **TreeSet<E>**) die Schnittstelle **NavigableMap<E>** und ist außerdem performanter als die von **Collections.synchronizedSortedSet()** gelieferte synchronisierte Variante von **TreeSet<E>**. Damit eignet sich **ConcurrentSkipListSet<E>**, wenn Thread-Sicherheit für eine Menge mit geordneten Elementen gefragt ist, weil die Elemente in Sortierordnung aufgesucht werden müssen, oder eine Methode aus der Schnittstelle **NavigableSet<E>** benötigt wird.
- Mengenverwaltungskollektion auf **ConcurrentHashMap<K,V>** - Basis
  Eine Klasse namens **ConcurrentHashSet<E>** sucht man im Paket **java.util.concurrent** vergeblich. Von der **ConcurrentHashMap<K,V>** - Methode **newKeySet()** kann man aber ein Kollektionsobjekt mit der gewünschten Funktionalität und Thread-Sicherheit erstellen lassen. Man erhält eine Verpackung der Klasse **ConcurrentHashMap<K,Boolean>**, die das Interface **Set<K>** implementiert. Alle Elemente haben **Boolean.TRUE** als Wert, z. B.:
  ```
  Set<Integer> anwesend = new ConcurrentHashMap<Integer,Boolean>().newKeySet();
  ```
- **CopyOnWriteArrayList<E> CopyOnWriteArraySet<E>**
  Diese Thread-sicheren Kollektionen zur Verwaltung einer Liste bzw. einer Menge taugen nicht generell als Alternative zur Klasse **ArrayList<E>** bzw. zu einer **Set<E>** - Implementation, können aber gegenüber einer synchronisierten Kollektion (z. B. **ArrayList<E>**) einen Performanzvorteil bieten, wenn auf eine von mehreren Thread benutzte Kollektion meist nur lesend und nur in seltenen Fällen auch schreibend zugegriffen wird. Ihr Vorteil besteht darin, ohne Synchronisierungsaufwand für Thread-Sicherheit sorgen zu können, was folgendermaßen gelingt (Horstmann 2015, S. 328): Jeder Iterator erhält eine Referenz auf den aktuellen Array. Vor einer späteren Änderung wird zunächst eine Kopie angelegt, die den alten Array ersetzt. Iteratoren im Besitz einer Referenz auf den mittlerweile veralteten Zustand erfahren also nichts von den Veränderungen, können aber ohne Konsistenzprobleme weiterarbeiten.

Joshua Bloch, der sich als JCF-Designer ein Urteil erlauben darf, kommt beim Vergleich der synchronisierten Kollektionen mit der Konkurrenz aus dem Paket **java.util.concurrent** zum Ergebnis (2018, S. 326):

> Concurrent collections make synchronized collections largely obsolete. For example, use ConcurrentHashMap in preference to Collections.synchronizedMap. Simply replacing synchronized maps with concurrent maps can dramatically increase the performance of concurrent applications.

Weitere Informationen zur Verwendung von Kollektionen in Programm mit Multithreading finden sich z. B. bei Goetz (2006, Abschnitt 5.2).

15.7 Threads und JavaFX

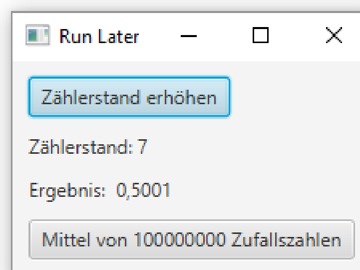
Wie die meisten modernen GUI-Frameworks ist auch JavaFX aus Performanzgründen nach dem Single-Thread - Prinzip konzipiert. Daher muss der Zugriff auf die GUI-Komponenten von JavaFX dem **JavaFX Application Thread** (alias: *UI-Thread*) vorbehalten bleiben.

Bei allen im JavaFX Application Thread ausgeführten Methoden muss sich der Zeitaufwand in Grenzen halten (maximal 100 Millisekunden[1]), weil sonst die Bedienoberfläche zäh reagiert. Solange eine Methode läuft (z. B. gestartet als Reaktion auf ein Ereignis), kann die Anwendung nicht auf andere Ereignisse (z. B. Mausklicks auf Steuerelemente) reagieren. Auch ist keine Aktualisierung der Anzeige möglich, zu der z. B. das Laufzeitsystem auffordert, weil ein bisher verdeckter Fensterbereich sichtbar geworden ist.

Zeitaufwändige Arbeiten (z. B. Netzwerk- oder Datenbankzugriffe, aufwändige Berechnungen) gehören in einen separaten Thread. In der Regel müssen aber irgendwann Ergebnisse der Hintergrundtätigkeit auf der Oberfläche sichtbar werden, wobei wegen der eingangs genannten Regel aus dem separaten Thread keine direkten Zugriffe auf Steuerelemente möglich sind.

15.7.1 JavaFX-Komponenten aus einem Hintergrund-Thread modifizieren

Wir konstruieren eine Variante des Mehrzweckzählprogramms aus Abschnitt 13.7.1, um zu demonstrieren, dass die Berechnung des Mittelwerts aus 100 Millionen pseudozufälligen **double**-Zahlen in einem separaten Thread die Bedienbarkeit eines JavaFX-Programms nicht beeinträchtigt:



Um die Mittelwertberechnung aus dem UI-Thread herauszuhalten, definieren wir eine Klasse, die das Interface **Runnable** erfüllt und die Rechnung in ihrer **run()** - Methode erledigt:

[1] Diese Empfehlung stammt von der Webseite:
http://www.oracle.com/technetwork/articles/javase/swingworker-137249.html

```
class RandomNumberCruncher implements Runnable {
   private double d = 0.0;
   @Override
   public void run() {
      for (long i = 0; i < anz; i++) {
         d += Math.random();
      }
      Platform.runLater(() ->
        lblMessage.setText("Ergebnis: " + String.format("%7.4f", d/anz)));
   }
}
```

Nach einem Mausklick auf den unteren Befehlsschalter des Beispielprogramms wird ein Objekt der Klasse `RandomNumberCruncher` erzeugt und dessen **run()** - Methode in einem eigenen Thread ausgeführt:

```
Button btnTask = new Button("Mittel von " + anz + " Zufallszahlen");
btnTask.setOnAction(event -> {
   task = new RandomNumberCruncher();
   Thread thread = new Thread(task);
   thread.setDaemon(true);
   thread.start();
});
```

Mit der **Thread**-Methode **setDaemon()** wird dafür gesorgt, dass die Hintergrundaktivität in einem Daemon-Thread (vgl. Abschnitt 15.8.1) stattfindet, der beim Schließen des Programmfensters automatisch beendet wird. Ohne diese Maßnahme würde ein aktiver Daemon-Thread seine Arbeit auch nach dem Schließen des Programmfensters fortsetzen.

Nach Vollendung seiner Arbeiten möchte der Daemon-Thread die **text**-Eigenschaft der **Label**-Komponente ändern, um das Ergebnis zu präsentieren. Ein direkter Zugriff

```
lblMessage.setText("Ergebnis: " + String.format("%7.4f", d/anz));
```

aus dem Daemon-Thread führt allerdings zu einem Laufzeitfehler.

Mit Hilfe der statischen Methode **runLater()** aus der Klasse **Platform** kann der Daemon-Thread ein **Runnable**-Objekt erstellen, das baldigst im UI-Thread ausgeführt werden soll:

```
Platform.runLater(() ->
  lblMessage.setText("Ergebnis: " + String.format("%7.4f", d/anz)));
```

So gelingt die Ergebnispräsentation unter Beachtung der Single-Thread - Regel.

Das vollständige IntelliJ-Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

...\BspUeb\Multithreading\JavaFX\RunLater

15.7.2 Das JavaFX-Multithreading - API

Mit den Typen im Paket **javafx.concurrent** kann man Aufgaben in separate Threads verlagern und dabei den Status der Auftragsbearbeitung, den aktuellen Bearbeitungsfortschritt sowie die Ergebnisse im JavaFX Application Thread verwenden. Die wesentlichen Typen sind:

- **Worker<V>**
  Diese Schnittstelle schreibt Eigenschaften (im Sinn von Abschnitt 13.5.1) für einen Hintergrundauftrag vor (z. B. **state** mit dem aktuellen Status, **progress** mit dem anteiligen Bearbeitungsstand, **value** mit dem Ergebnis). Die möglichen Statusangaben sind durch die innerhalb von **Worker<V>** definierte Enumeration **State** festgelegt (z. B. **READY**, **SUCCEEDED**).
- **Task<V>**
  Diese Klasse implementiert das Interface **Worker<V>** und eignet sich für eine einmalig auszuführende Aufgabe, weil ein **Task<V>** - Objekt mit einem terminalen Status (**CANCELLED**, **SUCCEEDED** oder **FAILED**) nicht wiederverwendet werden kann.
- **Service<V>**
  Ein Objekt der Klasse **Service<V>** enthält ein **Task<V>** - Objekt und kann nach dem Erreichen eines terminalen Zustands reaktiviert werden.
- **ScheduledService<V>**
  Die von **Service<V>** abgeleitete Klasse **ScheduledService<V>** unterstützt die automatisierte Wiederverwendung gemäß Einsatzplan.

Alle in der Schnittstelle **Worker<V>** vorgeschriebenen JavaFX-Properties sind vom ReadOnly-Typ (vgl. Abschnitt 13.5.1.2).

Name	Klasse	Beschreibung
title	**ReadOnlyStringProperty**	Beschreibt die Aufgabe
state	**ReadOnlyObjectProperty <Worker.State>**	Siehe obige Auflistung
running	**ReadOnlyBooleanProperty**	Diese Eigenschaft hat genau dann den Wert **true**, wann der Status **SCHEDULED** oder **RUNNING** ist.
message	**ReadOnlyStringProperty**	Damit kann eine Aufgabe darüber informieren, was sie gerade tut.
totalWork	**ReadOnlyDoubleProperty**	Werte von 0 bis **Double.MAX_VALUE** stehen für das gesamte Arbeitsvolumen, -1 steht für einen undefinierten Wert.
workDone	**ReadOnlyDoubleProperty**	Werte von 0 bis **totalWork** stehen für das bereits bewältigte Arbeitsvolumen, -1 steht für einen undefinierten Wert.
progress	**ReadOnlyDoubleProperty**	Werte von 0 bis 1 stehen für den bereits bewältigten Anteil der Arbeit, -1 steht für einen undefinierten Wert.
value	**ReadOnlyObjectProperty<V>**	Diese Eigenschaft enthält das Ergebnis, wenn der Auftragsstatus **SUCCEEDED** erreicht ist.
exception	**ReadOnlyObjectProperty <Throwable>**	Kommt es während der Auftragsbearbeitung zu einem Ausnahmefehler (und damit zum Status **FAILED**), dann ist das Ausnahmeobjekt über die Eigenschaft **exception** abrufbar.

Anschließend sind die Werte des Enumerationstyps **Worker.State** und damit die möglichen Zustände eines Hintergrundauftrags aufgelistet:

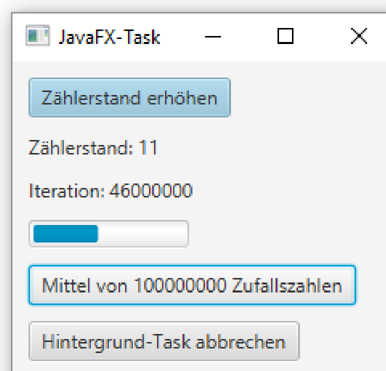
- **READY**
  Der Worker ist (re)initialisiert.
- **SCHEDULED**
  Der Worker ist zur Bearbeitung eingeplant und wartet z. B. auf einen freien Pool-Thread.
- **RUNNING**
  Unmittelbar vor Arbeitsbeginn erhält ein Worker den Zustand RUNNING.

- **SUCCEEDED**
  Der Worker hat seine Arbeit erfolgreich beendet, und seine Eigenschaft **value** enthält ein gültiges Ergebnis.
- **CANCELLED**
  Der Worker wurde durch die zur Schnittstelle **Worker<V>** gehörige Methode **cancel()** abgebrochen.
- **FAILED**
  Bei der Auftragsbearbeitung ist ein Fehler aufgetreten. In der Regel ist eine Ausnahme aufgetreten, deren Adresse in der Eigenschaft **exception** hinterlegt ist.

Aus Zeitgründen kann im Manuskript nur die Klasse **Task<V>** behandelt werden. Ein gründliche Darstellung des gesamten JavaFX-Multithreading-APIs bietet Sharan (2015, Kapitel 27).

15.7.3 Die Klasse Task<V>

Das empfehlenswerte Verfahren zur Realisation einer einmalig auszuführenden Hintergrundaktivität in einem JavaFX-Programm besteht darin, von **Task<V>** eine eigene Klasse abzuleiten und minimal die Methode **call()** zu überschreiben. Wir modifizieren das Mehrzweckzählprogramm aus Abschnitt 15.7.1, um zu demonstrieren, dass die Einbindung einer Hintergrundaktivität (Berechnung des Mittelwerts aus 100 Millionen pseudozufälligen **double**-Zahlen) in ein JavaFX-Programm mit Hilfe der Klasse **Task<V>** erheblich flexibler und zuverlässiger gelingt als mit der im Abschnitt 15.7.1 vorgestellten Technik:



U.a. wird es möglich, ...

- die im Abschnitt 15.7.2 beschriebenen JavaFX-Properties der Hintergrundaktivität im UI-Thread zu nutzen (z. B. für eine Fortschrittsanzeige), weil die Klasse **Task<V>** das Interface **Worker<V>** implementiert,
- bei der Hintergrundbearbeitung aufgetretene Ausnahmefehler an den JavaFX Application Thread zu berichten,
- die Hintergrundaktivität geordnet zu stoppen.

Weil unsere Hintergrundaktivität als Ergebnis eine **Double-**Zahl liefert, verwenden wir als Basisklasse **Task<Double>**:

```
class RandomNumberCruncher extends Task<Double> {
   @Override
   protected Double call() {
      double d = 0.0;
      for (long i = 0; i < anz; i++) {
         if (this.isCancelled()) {
            this.updateMessage("Die Aufgabe wurde abgebrochen.");
            break;
         }
         d += Math.random();
         if ((i+1)%1_000_000 == 0)
            this.updateMessage("Iteration: " + (i+1));
         this.updateProgress(i, anz);
      }
      return d/anz;
   }
   @Override
   protected void succeeded() {
      super.succeeded();
      this.updateMessage("Ergebnis: " + this.getValue());
   }
}
```

In **call()** werden zwei **Task<V>** - Methoden benutzt, um JavaFX-Properties des Hintergrundauftrags (siehe Liste im Abschnitt 15.7.2) im JavaFX Application Thread zu aktualisieren:

- **protected void updateMessage(String** *message***)**
  Diese Methode aktualisiert die Eigenschaft **message**.
- **protected void updateProgress(long** *workDone***, long** *max***)**
  Diese Methode aktualisiert die Eigenschaften **workDone**, **totalWork** und **progress**, was die Voraussetzungen für eine Fortschrittsanzeige im UI-Thread schafft. Kommt in der **call()** - Methode der Task eine Schleife zum Einsatz, sind geeignete Aktualparameter für **updateProgress()** schnell gefunden.

In der Bedienoberfläche des Beispielprogramms werden die Auftrags-Properties über die Binding-Technik von JavaFX (vgl. Abschnitt 13.5.3) mit der **text**-Property eines Labels bzw. mit der **progress**-Property eines **ProgressBar**-Steuerelements verbunden und so sichtbar gemacht:

```
Label lblMessage = new Label();
ProgressBar progBar = new ProgressBar();

Button btnTask = new Button("Mittel von " + anz + " Zufallszahlen");
btnTask.setOnAction(event -> {
   task = new RandomNumberCruncher();
   Thread thread = new Thread(task);
   thread.setDaemon(true);
   thread.start();
   lblMessage.textProperty().bind(task.messageProperty());
   progBar.progressProperty().bind(task.progressProperty());
});
```

Damit die Bindungen korrekt arbeiten, dürfen die Task-Properties nur im UI-Thread geändert werden. Es ist generell zu unterlassen, aus einem anderen Thread Schreibzugriffe auf Objekte vorzunehmen, an die GUI-Eigenschaften gebunden sind, weil damit gegen die zu Beginn von Abschnitt 15.7 formulierte Single-Thread-Regel verstoßen wird.

Das vorige Quellcodesegment zeigt, wie im Beispielprogramm ein Objekt der **Task<Double>** - Ableitung `RandomNumberCruncher` ins Spiel kommt: Bei jedem Aufruf der Klickbehandlungsmethode zum oberen Befehlsschalter wird ein frisches Objekt erzeugt, das nach Erledigung seiner Arbeit nicht mehr zu gebrauchen ist.

Der Einfachheit halber wird auf den Einsatz eines Thread-Pools verzichtet und stattdessen Ressourcen verschwendend jeweils ein neues **Thread**-Objekt erzeugt. Weil die Klasse **Task<V>** das Interface **Runnable** implementiert, können Objekte dieses Typs als Aktualparameter an den **Thread**-Konstruktor übergeben werden.

Ein **Task<V>** - Objekt kann mit der Methode **cancel()** aus einem beliebigem Thread aufgefordert werden, seine Tätigkeit zu beenden, z. B. in der Klickbehandlungsmethode zum Abbrechen-Schalter des Beispielprogramms:

```
Button btnCancel = new Button("Hintergrund-Task abbrechen");
btnCancel.setOnAction(event -> {if (task!= null) task.cancel();});
```

Weil die Inter-Thread-Kooperation in Java im Wesentlichen auf Kooperation basiert (siehe Abschnitt 15.3.2), muss ein **Task<V>** - Objekt im Rahmen seiner **call()** - Methode regelmäßig prüfen, ob das Abbruchsignal gesetzt worden ist. Dazu steht die **Task<V>** - Methode **isCancelled()** bereit.

Befindet sich ein Thread in einem blockierenden Methodenaufruf (z. B. **sleep()**), dann hat **cancel()** eine **InterruptedException** zur Folge, und der Exception-Handler muss unbedingt per **isCancelled()** auf eine beantragte Terminierung prüfen.

Mit der Reaktion auf Statusveränderungen bei einem Auftrag kann man das **Task<V>** - Objekt oder einen **EventHandler<WorkerStateEvent>** beauftragen, was am Beispiel des Übergangs zum Status **SUCCEEDED** demonstriert werden soll:

- **protected void succeeded()**
  Diese **Task<V>** - Methode wird aufgerufen, wenn der Auftrag den Status **SUCCEEDED** erreicht hat. Das Auftragsobjekt agiert und hat seine Umgebung (z. B. seine Instanzvariablen) zur Verfügung. Im Beispielprogramm wird diese Technik gewählt.
- **protected void setOnSucceeded(EventHandler<WorkerStateEvent>** *value***)**
  Dieser **Task<V>** - Methode ist ein Objekt der Klasse **EventHandler<WorkerStateEvent>** zu übergeben, und dessen Methode **handle()** wird aufgerufen, wenn der Status **SUCCEEDED** erreicht ist. Der Event-Handler kann z. B. per Lambda-Ausdruck realisiert werden und hat dann Zugriff auf die umgebende Methode und Klasse, z. B.:

  ```
  task = new RandomNumberCruncher();
  task.setOnSucceeded(e -> { . . . });
  ```

Bei beiden Techniken läuft die Methode zur Behandlung des Zustandswechsels im JavaFX Application Thread ab.

Weitere Informationen zur Klasse **Task<V>** liefert u.a. die OpenJDK-Dokumentation.[1]

Das vollständige IntelliJ-Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

...\BspUeb\Multithreading\JavaFX\Task

15.8 Weitere Thread-Themen

15.8.1 Daemon-Threads

Neben den bisher behandelten Benutzer-Threads kennt Java noch sogenannte *Daemon-Threads*, die eine niedrige Priorität besitzen, also nur aktiv werden, wenn ungenutzte Rechenzeit vorhanden ist. Sie werden meist von der JVM gestartet und dienen oft zu Unterstützung von Benutzer-Threads, was z. B. für den Thread gilt, der obsolet gewordene (nicht mehr referenzierte) Objekte abräumt (Garbage Collecting).

[1] https://openjfx.io/javadoc/13/javafx.graphics/javafx/concurrent/Task.html

Um das Terminieren von Daemon-Threads braucht man sich in der Regel nicht zu kümmern, denn ein Java-Programm endet, sobald alle Benutzer-Threads ihre Tätigkeit eingestellt haben, und folglich nur noch Daemon-Threads vorhanden sind. Ein Daemon-Thread muss also auf ein abruptes Ende gefasst sein, wobei selbst ein **finally**-Block nicht mehr ausgeführt wird.

Mit der **Thread**-Methode **setDaemon()** lässt sich ein Benutzer-Thread dämonisieren, was vor dem Aufruf seiner **start()** - Methode geschehen muss. Auf diese Weise lässt sich bei einer JavaFX-Anwendung verhindern, dass die virtuelle Maschine nach dem Schließen der letzten Bühne (des letzten Fensters) wegen eines Benutzer-Threads noch weiterläuft.

Ein Thread erbt den Daemon-Status des Threads, in dem er erstellt wurde. Mit der **Thread**-Methode **isDaemon()** lässt sich feststellen, ob ein Daemon-Thread vorliegt.

15.8.2 Thread-Gruppen

Bei den in manchen Lehrbüchern behandelten Thread-Gruppen (siehe z. B. Krüger & Hansen 2014, S. 851f) beschränken wir uns darauf, Joshua Bloch (2008, S. 288) zu zitieren:

> Thread groups are obsolete.

15.9 Übungsaufgaben zum Kapitel 15

1) Welche der folgenden Aussagen sind richtig bzw. falsch?

1. Ein Java-Programm endet zusammen mit seinem Thread **main**.
2. Ereignisbehandlungsmethoden laufen im UI-Thread (JavaFX) und sollten nach spätestens 100 Millisekunden beendet sein.
3. Ein terminierter Thread kann nicht mehr neu gestartet werden.
4. Ein Thread im Zustand **sleeping** gibt alle Monitore und **Lock**-Objekte zurück.

2) Das folgende Programm startet einen Thread aus der Klasse `Schnarcher`, lässt ihn 3 Sekunden lang gewähren und versucht dann, den Thread zu beenden:

```
class Prog {
   public static void main(String[] args) throws InterruptedException {
      Schnarcher st = new Schnarcher();
      st.start();
      System.out.println("Thread gestartet");
      Thread.sleep(3000);
      while(st.isAlive()) {
         st.interrupt();
         System.out.println("\nThread beendet!?");
         Thread.sleep(1000);
      }
   }
}
```

Der `Schnarcher`-Thread führt in seiner **run()** - Methode eine **while**-Schleife aus, prüft bei jedem Umlauf zunächst, ob das Interrupt-Signal gesetzt ist, und beendet sich ggf. per **return**. Falls keine Einwände gegen seine weitere Tätigkeit bestehen, schreibt der Thread nach einer kurzen Wartezeit ein Sternchen auf die Konsole:

```
class Schnarcher extends Thread {
   @Override
   public void run() {
      while (true) {
         if(isInterrupted())
            return;
         try {sleep(100);} catch (InterruptedException ie) {}
         System.out.print("*");
      }
   }
}
```

Wie die Ausgabe eines Programmlaufs zeigt, bleiben die **interrupt()**-Aufrufe wirkungslos:

```
Thread gestartet
******************************
Thread beendet!?
*********
Thread beendet!?
**********
Thread beendet!?
***********
Thread beendet!?
*********
  .  .  .
```

Wie ist das Verhalten zu erklären, und wie sorgt man für ein zuverlässiges Beenden des Threads?

3) Warum ist der Modifikator **volatile** für lokale Variablen überflüssig (und verboten)?

Anhang

A. Operatorentabelle

In der folgenden Tabelle sind alle im Manuskript behandelten Operatoren in absteigender Bindungskraft (von oben nach unten) aufgelistet. Gruppen von Operatoren mit gleicher Bindungskraft sind durch fette horizontale Linien begrenzt.

<table>
<tr><th>Operator</th><th>Bedeutung</th></tr>
<tr><td>[]
.</td><td>Array-Index
Komponentenzugriff</td></tr>
<tr><td>!
++, --
-
(Typ)
new</td><td>Negation
Prä- oder Postinkrement bzw. -dekrement
Vorzeichenumkehr
Typumwandlung
Objekterzeugung</td></tr>
<tr><td>*, /
%</td><td>Punktrechnung
Modulo</td></tr>
<tr><td>+, -
+</td><td>Strichrechnung
String-Verkettung</td></tr>
<tr><td><<, >></td><td>Links- bzw. Rechts-Shift</td></tr>
<tr><td>>, <,
>=, <=
instanceof</td><td>Vergleichsoperatoren
Typprüfung</td></tr>
<tr><td>==, !=</td><td>Gleichheit, Ungleichheit</td></tr>
<tr><td>&
&</td><td>Bitweises UND
Logisches UND (mit unbedingter Auswertung)</td></tr>
<tr><td>^</td><td>Exklusives logisches ODER</td></tr>
<tr><td>|
|</td><td>Bitweises ODER
Logisches ODER (mit unbedingter Auswertung)</td></tr>
</table>

<table>
<tr><th>Operator</th><th>Bedeutung</th></tr>
<tr><td>&&</td><td>Logisches UND (mit bedingter Auswertung)</td></tr>
<tr><td>||</td><td>Logisches ODER (mit bedingter Auswertung)</td></tr>
<tr><td>? :</td><td>Konditionaloperator</td></tr>
<tr><td>=

+=, -=,
*=, /=,
%=</td><td>Wertzuweisung

Wertzuweisung mit Aktualisierung</td></tr>
</table>

Mit Ausnahme der Zuweisungsoperatoren sind alle binären Operatoren *links*-assoziativ. Die Zuweisungsoperatoren und der Konditionaloperator sind *rechts*-assoziativ.

B. Lösungsvorschläge zu den Übungsaufgaben

Kapitel 1 (Einleitung)

Aufgabe 1

Das Prinzip der Datenkapselung reduziert die Fehlerquote und damit den Aufwand zur Fehlersuche und -bereinigung. Die perfektionierte Modularisierung durch die Kopplung von Eigenschaften und zugehörigen Handlungskompetenzen in einer Klassendefinition erleichtert die …

- Kooperation von mehreren Programmierern bei großen Projekten,
- die Wiederverwendung von Software.

Aufgabe 2

1. **Richtig**
2. **Falsch**
   Jedes Java-Programm muss *eine* Startklasse enthalten, und nur eine Startklasse benötigt eine Methode namens **main()**.
3. **Falsch**
   Der vom Java-Compiler erstellte Bytecode muss vom Java-Interpreter in den Maschinencode der aktuellen CPU übersetzt werden.
4. **Richtig**

Kapitel 2 (

Werkzeuge zum Entwickeln von Java-Programmen)

Aufgabe 2

Das Programm enthält folgende Fehler:

- Die schließende Klammer zum Rumpf der Klassendefinition fehlt.
- Die Zeichenfolge im **println()** - Aufruf muss mit dem " - Zeichen abgeschlossen werden.
- Der Methodenname „mein" ist falsch geschrieben.
- Die Methode **main()** muss als **public** definiert werden.

Aufgabe 3

1. **Richtig**
2. **Falsch**
3. **Richtig**
4. **Falsch**
   Während der Datentyp **String[]** des **main()** - Parameters in der Tat zwingend vorgeschrieben ist, kann man seinen Namen frei wählen.

Kapitel 3 (Elementare Sprachelemente)

Abschnitt 3.1 (Einstieg)

Aufgabe 1

Dieser Aufruf klappt:	`public static void main(String[] irrelevant) { … }` Der *Name* des **main()** - Parameters ist beliebig.
Dieser Aufruf scheitert:	`public void main(String[] args) { … }` Der Modifikator **static** fehlt.
Dieser Aufruf scheitert:	`public static void main() { … }` Falsche Parameterliste
Dieser Aufruf klappt:	`static public void main(String[] args) { … }` Die Modifikatoren **static** und **public** müssen vor dem Rückgabetyp stehen, wobei ihre Reihenfolge irrelevant ist (siehe Syntaxdiagramm im Abschnitt 3.1.3.2).
Dieser Aufruf klappt:	`public static void main(String[] Args) { … }` Der Parametername ist frei wählbar, und `Args` verstößt gegen keine Compiler-Regel. Per Konvention verwendet man allerdings bei Parameternamen wie bei Variablennamen das Camel Casing (siehe Abschnitt 3.3.2), startet also mit einem Kleinbuchstaben
Dieser Aufruf scheitert:	`static public void Main(String[] args) { … }` Der Anfangsbuchstabe im Methodennamen **main()** muss klein geschrieben werden.
Dieser Aufruf scheitert:	`static public int main(String[] args) { … }` Die Programmiersprachen C, C++ und C# besitzen ebenfalls eine Funktion bzw. Methode namens **main()** bzw. **Main()**, und dort wird (optional) der Rückgabetype **int** verwendet, der beim Verlassen des Programms die Übergabe eines Returncodes an das Betriebssystem erlaubt. In Java hat die **main()** - Methode obligatorisch den Rückgabetyp **void**, doch kann z. B. mit der statischen Methode **exit()** der Klasse **System** ein Returncode an das Betriebssystem übergeben werden (siehe Abschnitt 11.1).

Aufgabe 2
Unzulässig sind:

- `4you`
  Bezeichner müssen mit einem Buchstaben beginnen.
- `else`
  Schlüsselwörter wie **else** sind als Bezeichner verboten.

Obwohl **main** kein Schlüsselwort ist, wird man mit einem derart irritierenden Bezeichner (z. B. für eine Variable) wenig Ruhm und Sympathie gewinnen.

Aufgabe 3
Das Paket **java.lang** der Standardbibliothek wird automatisch in jede Quellcodedatei importiert (vgl. Abschnitt 3.1.7).

Abschnitt 3.2 (Ausgabe bei Konsolenanwendungen)

Aufgabe 1
Das folgende Programm erzeugt die erwünschte Ausgabe:

```
class Prog {
  public static void main(String[] args) {
    System.out.printf("%1$-10.1f %1$-10.2f %1$-10.3f%n", Math.PI);
    System.out.printf("%1$-10.4f %1$-10.5f %1$-10.6f", Math.PI);
  }
}
```

Wenn die längliche Formatierungszeichenfolge nicht stört, kommt man auch mit einem **printf()** - Aufruf aus:

```
class Prog {
  public static void main(String[] args) {
    System.out.printf("%1$-10.1f %1$-10.2f %1$-10.3f%n%1$-10.4f %1$-10.5f %1$-10.6f", Math.PI);
  }
}
```

Wie aus Abschnitt 3.2.1 bekannt, kann man die Formatierungszeichenfolge per Plusoperator zusammensetzen und so auf zwei Zeilen verteilen:

```
class Prog {
  public static void main(String[] args) {
    System.out.printf("%1$-10.1f %1$-10.2f %1$-10.3f%n" +
                      "%1$-10.4f %1$-10.5f %1$-10.6f", Math.PI);
  }
}
```

Um einen Zeilenwechsel zu erreichen, kann man statt der Formatspezifikation **%n** auch die Escape-Sequenz **\n** verwenden:

```
class Prog {
  public static void main(String[] args) {
    System.out.printf("%1$-10.1f %1$-10.2f %1$-10.3f\n%1$-10.4f %1$-10.5f %1$-10.6f",Math.PI);
  }
}
```

Aufgabe 2

Der im **println()** - Parameter unmittelbar auf die Zeichenkette folgende Plusoperator wird *zuerst* ausgeführt. Weil sein linkes Argument eine Zeichenfolge ist, wird auch sein rechtes Argument in eine Zeichenfolge gewandelt, um eine sinnvolle Operation zu ermöglichen, nämlich die Verkettung von zwei Zeichenfolgen. Der Ausdruck

```
"3.3 + 2 = " + 3.3
```

wird also behandelt wie

```
"3.3 + 2 = " + "3.3"
```

und man erhält:

```
"3.3 + 2 = 3.3"
```

Anschließend arbeitet der zweite Plus-Operator analog, sodass insgesamt die Zeichenfolgen „3.3“ und „2“ nacheinander an die Zeichenfolge „3.3 + 2 = “ angehängt werden.

Durch Klammerung muss dafür gesorgt werden, dass der *rechte* Plusoperator *zuerst* ausgeführt wird:

Quellcode	Ausgabe

`class Prog {`
`   public static void main(String[] args) {`
`      System.out.println("3.3 + 2 = " + (3.3 + 2));`
`   }`
`}`
	`3.3 + 2 = 5.3`

Er trifft folglich auf zwei *numerische* Argumente und addiert diese. Der Ausdruck

```
3.3 + 2
```

ergibt

```
5.3
```

Anschließend bewirkt der linke Plus-Operator eine Zeichenfolgenverkettung. Der Ausdruck

```
"3.3 + 2 = " + 5.3
```

ergibt

```
"3.3 + 2 = 5.3"
```

und dieses **println()** - Argument landet auf der Konsole.

Abschnitt 3.3 (Variablen und Datentypen)

Aufgabe 1

1. **Falsch**
2. **Richtig**
3. **Falsch**
   Referenzvariablen haben einen bestimmten *Inhalt* (eine Objektadresse). Sie werden als lokale Variablen von Methoden (abgelegt auf dem Stack), als Instanzvariablen von Objekten (abgelegt auf dem Heap) und als Klassenvariablen (abgelegt in der Method Area) benötigt.
4. **Falsch**
   Dieser Satz ist kompletter Unfug.

Aufgabe 2

char gehört zu den integralen (ganzzahligen) Datentypen. Zeichen werden über ihre Nummer im Unicode-Zeichensatz gespeichert, das Zeichen ‚c' offenbar durch die Nummer 99 (im Dezimalsystem).

In der folgenden Anweisung wird der **char**-Variablen `z` die Unicode-Escape-Sequenz für das Zeichen ‚c' zugewiesen:

```
char z = '\u0063';
```

Der dezimalen Zahl 99 entspricht die hexadezimale Zahl 0x63 (= 6 · 16 + 3).

Aufgabe 3

Die Variable `i` ist nur im innersten Block gültig.

Aufgabe 4

Lösungsvorschlag:

```
class Prog {
   public static void main(String[] args) {
      System.out.println("Dies ist ein Java-Zeichenkettenliteral:\n   \"Hallo\"");
   }
}
```

Aufgabe 5

Die behobenen Fehler sind durch einen großen Schriftgrad gekennzeichnet:

```
class Prog {
   public static void main(String[] args) {

      float PI = 3.141593F;

      double radius = 2.0;

      System.out.println("Der Flächeninhalt beträgt: " + PI*radius*radius);
   }
}
```

Abschnitt 3.4 (Eingabe bei Konsolen)

Aufgabe 1

Im folgenden Programm werden die `Simput`-Methoden `gchar()` und `gdouble()` verwendet:

```
class Prog {
    public static void main(String[] args) {
        System.out.print("Setzen Sie bitte ein Zeichen: ");
        char c = Simput.gchar();
        System.out.println("Ihr Zeichen: " + c);
        System.out.print("\nNun bitte eine gebrochene Zahl (mit Dezimalkomma!): ");
        double d = Simput.gdouble();
        System.out.printf("Ihre Zahl: " + d);
    }
}
```

Abschnitt 3.5 (Operatoren und Ausdrücke)

Aufgabe 1

Ausdruck	Typ	Wert	Anmerkungen

`6/4*2.0`	**double**	`2.0`	Abarbeitung mit Zwischenergebnissen:
			`6/4 = 1`
			`1*2.0 = 2.0`

`(int)6/4.0*3`	**double**	`4.5`	Der Typumwandlungsoperator hat die höchste Bindungskraft und bezieht sich daher (ohne Wirkung) auf die 6.
			Abarbeitung mit Zwischenergebnissen:
			`(int)6 = 6`
			`6/4.0 = 1.5`
			`1.5*3 = 4.5`

`(int)(6/4.0*3)`	**int**	`4`	Abarbeitung mit Zwischenergebnissen:
			`6/4.0 = 1.5`
			`1.5*3 = 4.5`
			`(int)4.5 = 4`

`3*5+8/3%4*5`	**int**	`25`	Abarbeitung mit Zwischenergebnissen:
			`3*5 = 15`
			`8/3 = 2`
			`2%4 = 2`
			`2*5 = 10`
			`15+10 = 25`

Aufgabe 2

Nach der Tabelle mit den Ergebnistypen der Ganzzahlarithmetik im Abschnitt 3.5.1 resultiert der Datentyp **int**.

Aufgabe 3

`erg1` erhält den Wert 2, denn:

- `(i++ == j ? 7 : 8)` hat den Wert 8, weil 2 ≠ 3 ist.
- 8 **%** 3 ergibt 2.

`erg2` erhält den Wert 0, denn:

- Der Präinkrementoperator trifft auf die bereits vom Postinkrementoperator in der vorangehenden Zeile auf den Wert 3 erhöhte Variable `i` und setzt sie auf den Wert 4.
- Dies ist auch der Wert des Ausdrucks `++i`, sodass die Bedingung im Konditionaloperator erneut den Wert **false** hat.
- `(++i == j ? 7 : 8)` hat also den Wert 8, und 8 **%** 2 ergibt 0.

Aufgabe 4

Die Vergleichsoperatoren (>, ==) haben eine höhere Bindungskraft als die logischen Operatoren und der Zuweisungsoperator, sodass z. B. in der folgenden Anweisung

```
la1 = 2 > 3 && 2 == 2 ^ 1 == 1;
```

auf runde Klammern verzichtet werden konnte. Besser lesbar ist aber die äquivalente Variante:

```
la1 = (2 > 3) && (2 == 2) ^ (1 == 1);
```

`la1` erhält den Wert **false**, denn der Operator **^** wird aufgrund seiner höheren Bindungskraft vor dem Operator **&&** ausgeführt.

`la2` erhält den Wert **true**, weil die runden Klammern dafür sorgen, dass der Operator `^` zuletzt ausgeführt wird.

`la3` erhält den Wert **false**.

Aufgabe 5
Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\Exp

Aufgabe 6
Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DM2Euro

Aufgabe 7
Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\UnGerade

Abschnitt 3.6 (Über- und Unterlauf bei numerischen Variablen)

Aufgabe 1

1. **Falsch**
2. **Richtig**
3. **Falsch**
4. **Falsch**
   Bei Objekten aus den Klassen **BigDecimal** und **BigInteger** tritt im Vergleich zu den primitiven Datentypen ein erheblich höherer Speicher- und Zeitaufwand auf. Sie sollten daher nur verwendet werden, wenn der größere Wertebereich bzw. die höhere Genauigkeit tatsächlich erforderlich sind.

Abschnitt 3.7 (Anweisungen (zur Ablaufsteuerung))

Aufgabe 1
Im logischen Ausdruck der **if**-Anweisung findet an Stelle eines Vergleichs eine *Zuweisung* statt.

Aufgabe 2
In der **switch**-Anweisung wird es versäumt, per **break** den „Durchfall" zu verhindern.

Aufgabe 3
Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\PrimitivOB

Aufgabe 4
Das Semikolon am Ende der Zeile

```
while (i < 100);
```

wird vom Compiler als die zur **while**-Schleife gehörige (leere) Anweisung interpretiert, sodass mangels `i`-Inkrementierung eine *Endlosschleife* vorliegt. Hinter der **while**-Schleife steht eine Blockanweisung, die nie ausgeführt wird.

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DM2EuroS

Aufgabe 6

Lösungsvorschläge mit den beiden Algorithmus-Varianten befinden sich in den Ordnern:

...\BspUeb\Elementare Sprachelemente\GGT.Diff
...\BspUeb\Elementare Sprachelemente\GGT.Mod

Aufgabe 7

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\FloP

Bei $i = 52$ erhält man letztmals das mathematisch korrekte Vergleichsergebnis. Dies ist gerade die Anzahl der Bits in der **double**-Mantisse (vgl. Abschnitt 3.3.6).

Die Zahl 1,0 hat nach der Norm IEE 754 die folgende normalisierte **double**-Darstellung (vgl. Abschnitt 3.3.7.1)

$$(-1)^0 \cdot 2^{(1023-1023)} \cdot (1+0{,}0)$$

mit den Bits:

Vorz.	Exponent	Mantisse
0	01111111111	0000000000000000000000000000000000000000000000000000

Das Bitmuster zum Vergleichswert:

$$1{,}0 + 2^{-i} = (-1)^0 \cdot 2^{(1023-1023)} \cdot (1 + 2^{-i}),\ i = 1, \ldots, 52$$

hat eine zusätzliche Eins beim i-ten Mantissen-Bit. Bei $i = 52$ ist diese Eins am rechten Rand angekommen und letztmals vorhanden:

Vorz.	Exponent	Mantisse
0	01111111111	0000000000000000000000000000000000000000000000000001

Während bei binärer Gleitkommatechnik mit **double**-Präzision die Zahl

$$1{,}0 + 2^{-53} \approx 1{,}0 + 1{,}1102230246251565 \cdot 10^{-16} \approx 1{,}00000000000000011102230246251 57$$

nicht mehr von der Zahl 1,0 zu unterscheiden ist, kann die Zahl

$$2^{-53} \approx 1{,}1102230246251565 \cdot 10^{-16}$$

mit dem Bitmuster

Vorz.	Exponent	Mantisse
0	01111001010	0000000000000000000000000000000000000000000000000000

von der 0,0 mit dem Bitmuster

Vorz.	Exponent	Mantisse
0	00000000000	0000000000000000000000000000000000000000000000000000

unterschieden werden.

Erst ab 2^{-1023} ($\approx 1{,}1125369292536007 \cdot 10^{-308}$) wird erstmals die denormalisierte Darstellung benötigt (mit dem festen Exponentialfaktor 2^{-1022})

Vorz.	Exponent	Mantisse
0	00000000000	1000000000000000000000000000000000000000000000000000

und bei $2^{-1074} = 2^{-1022} \cdot 2^{-52}$ ($\approx 4{,}9406564584124654 \cdot 10^{-324}$)

Vorz.	Exponent	Mantisse
0	00000000000	0000000000000000000000000000000000000000000000000001

ist letztmals eine Unterscheidung von der Zahl 0,0 möglich.

Kapitel 4 (Klassen und Objekte)

Aufgabe 1

1. **Richtig**
2. **Falsch**
   Mit der Datenkapselung wird verhindert, dass die *Methoden fremder Klassen* auf Instanzvariablen zugreifen. Es geht *nicht* darum, Objekte einer Klasse voreinander zu schützen. Die von einem Objekt ausgeführten Methoden haben stets vollen Zugriff auf die Instanzvariablen eines anderen Objekts derselben Klasse, sofern eine entsprechende Referenz vorhanden ist. Der Klassendesigner ist für das sinnvolle Verhalten der Methoden verantwortlich.
3. **Falsch**
   Ohne Schutzstufendeklaration haben alle Klassen *im selben Paket* vollen Zugriff.
4. **Falsch**
   Lokale Variablen werden grundsätzlich *nicht* initialisiert, auch die lokalen Referenzvariablen nicht.

Aufgabe 2

Eine Instanzvariable mit Vollzugriff für die Methoden der eigenen Klasse und Schreibschutz gegenüber Methoden fremder Klassen erhält man folgendermaßen:

- Deklaration als **private** (Datenkapselung)
- Definition einer **public**-Methode zum Lesen des Wertes
- *Verzicht* auf eine **public**-Methode zum Verändern des Wertes

Aufgabe 3

1. **Falsch**
   Es kann durchaus sinnvoll sein, private Methoden für den ausschließlich klasseninternen Gebrauch zu definieren.
2. **Falsch**
   Der Rückgabetyp spielt bei der Signatur keine Rolle (vgl. Abschnitt 4.3.4).
3. **Falsch**
   Typkompatibilität genügt (siehe Abschnitt 4.3.2).
4. **Richtig**

Aufgabe 4

Bei einer Methode mit Rückgabewert muss jeder mögliche Ausführungspfad mit einer **return**-Anweisung enden, die einen Rückgabewert mit kompatiblem Typ liefert. Die vorgeschlagene Methode verstößt beim Aufruf mit einem von 0 verschiedenen Aktualparameterwert gegen diese Regel.

Aufgabe 5

Die beiden Methoden können *nicht* in einer Klasse koexistieren, weil ihre Signaturen identisch sind:

- gleiche Methodennamen
- gleichlange Parameterlisten mit identische Parametertypen an allen Positionen

Dass an jeder Position die beiden typgleichen Formalparameter verschiedene Namen haben, ist für die Signatur irrelevant.

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\TimeDuration

Aufgabe 7

Begriff	**Pos.**
Definition einer Instanzmethode mit Referenzrückgabe	7
Deklaration einer lokalen Variablen	4
Definition einer Instanzmethode mit Referenzparameter	6
Deklaration einer Instanzvariablen	1
Methodenaufruf	5

Begriff	**Pos.**
Konstruktordefinition	3
Deklaration einer Klassenvariablen	2
Objekterzeugung	9
Definition einer Klassenmethode	8

Aufgabe 8

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\FakulRek

Das Programm eignet sich übrigens dazu, einen Stapelüberlauf (**StackOverflowError**) zu provozieren:

```
Exception in thread "main" java.lang.StackOverflowError
```

Allerdings hat bei der Wahl eines passend großen Arguments (z. B. 15000) die resultierende Fakultät den **double**-Wertebereich längst in Richtung Infinity verlassen.

Aufgabe 9

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\R2Vek

Kapitel 5 (Elementare Klassen)

Abschnitt 5.1 (Arrays)

Aufgabe 1

1. **Falsch**
2. **Richtig**
3. **Richtig**
4. **Richtig**
5. **Falsch**
   Für diesen Zweck ist die finalisierte Instanz*variable* **length** zu verwenden.

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Arrays\Lotto

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Arrays\Eratosthenes

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Arrays\DataMat

Abschnitt 5.2 (Klassen für Zeichen)

Aufgabe 1

1. **Falsch**
   Weder ist ein **String**-Objekt ein Array, noch erfüllt die Klasse **String** das Interface **Iterable** oder das Interface **Iterable<T>** (vgl. Abschnitt 3.7.3.2).
2. **Richtig**
3. **Falsch**
   Stattdessen ist die **String**-Methode **charAt()** zu verwenden.
4. **Richtig**
   Für variable Zeichenketten eignen sich die Klassen **StringBuilder** und **StringBuffer**.

Aufgabe 2

Der interne **String**-Pool wird erweitert durch die Anweisungen mit den Kommentarnummern (1) und (5). Die Anweisungen mit den Kommentarnummern (2) und (3) erzeugen Objekte auf dem allgemeinen Heap. Durch die Anweisung mit der Kommentarnummer (4) findet keine Erweiterung des **String**-Pools statt, weil dort bereits ein inhaltsgleiches **String**-Objekt vorhanden ist.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Zeichenfolgen\PerZuf

Aufgabe 4

Die Klasse **StringBuilder** hat die von **java.lang.Object** geerbte **equals()** - Methode *nicht* überschrieben, sodass *Referenzen* verglichen werden. In der Klasse **String** ist **equals()** jedoch so überschrieben worden, dass die referenzierten *Zeichenfolgen* verglichen werden.

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Zeichenfolgen\StringUtil

Abschnitt 5.3 (Verpackungsklassen für primitive Datentypen)

Aufgabe 1

Lösungsvorschlag:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
    public static void main(String[] args) {
        System.out.println("Min. byte-Zahl:\n " + Byte.MIN_VALUE);
    }
}</pre></td><td><pre>Min. byte-Zahl:
 -128</pre></td></tr>
</table>

Beim Datentyp **byte** ist zu beachten, dass er in Java (wie alle anderen Ganzzahltypen) vorzeichenbehaftet ist, während z. B. die Programmiersprache C# einen vorzeichenfreien 8-Bit - Ganzzahltyp namens **byte** mit Werten von 0 bis 255 besitzt.

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Verpackungsklassen\MaxFakul

Für die Fakultät von 170 ($\approx 7{,}26 \cdot 10^{306}$) wird noch ein regulärer **double**-Wert ermittelt, während die Berechnung der Fakultät von 171 ($\approx 1{,}24 \cdot 10^{309}$) zum Wert **Double.POSITIVE_INFINITY** führt.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Verpackungsklassen\Mint

Abschnitt 5.4 (Aufzählungstypen)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Enumerationen\Wochentage

Kapitel 6 (Pakete und Module)

Aufgabe 1

Beide Klassen gehören zum Standardpaket und haben die voreingestellte Sichtbarkeit *package*, sodass sie sich gegenseitig sehen können. Weil die `Worker`-Methode ohne Zugriffsmodifikator definiert wurde, besitzt sie die voreingestellte Schutzstufe *package* und kann von allen Klassen im selben Paket verwendet werden. Auch der Standardkonstruktor der Klasse `Worker` ist für alle Klassen im selben Paket verfügbar, weil er dieselbe Schutzstufe besitzt wie seine Klasse (nämlich *package*).

Aufgabe 2

1. **Falsch**
   Es wird empfohlen, die Namen von exportierten Paketen mit dem Modulnamen beginnen zu lassen. Das ist aber nicht vorgeschrieben und z. B. beim API-Paket **java.lang** (ab Java 9 im Modul **java.base**) aus Kompatibilitätsgründen *nicht* realisiert.
2. **Richtig**
3. **Richtig**
4. **Falsch**
   Nur die automatischen Module (siehe Abschnitt 6.2.9.1) können das unbenannte Modul (mit den via Klassenpfad erreichbaren **class**-Dateien) sehen.

Kapitel 7 (*Vererbung und Polymorphie*)

Aufgabe 1

In der Klasse `Spezial` kommt der Standardkonstruktor zum Einsatz, welcher den parameterfreien Basisklassenkonstruktor aufruft. Ein solcher fehlt aber in der Klasse `General`.

Aufgabe 2

In der Klasse `Figur` haben `xpos` und `ypos` den voreingestellten Zugriffsschutz (**package**). Weil `Kreis` und `Figur` nicht zum selben Paket gehören, hat die `Kreis`-Klasse keinen direkten Zugriff. Soll dieser Zugriff möglich sein, müssen `xpos` und `ypos` in der `Figur`-Definition die Schutzstufe **protected** (oder **public**) erhalten.

Aufgabe 3

1. **Richtig**
2. **Falsch**
3. **Falsch**
4. **Falsch**
5. **Richtig**

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Vererbung und Polymorphie\Abstand

Aufgabe 5

Es muss nur die Basisklasse neu übersetzt werden. Eine abgeleitete Klasse enthält keinen Bytecode für geerbte Methoden, was man z. B. mit dem JDK-Werkzeug **javap.exe** überprüfen kann. Um den Bytecode einer Klasse in lesbarer Form anzeigen zu lassen, gibt man beim Aufruf die Option **-c** und den Klassennamen an, z. B.:

```
>javap -c Kreis
```

Kapitel 8 (Generische Klassen und Methoden)

Aufgabe 1

Lösungsvorschlag:

```
static <T> void printAll(List<T> list) {
    for (T e : list)
        System.out.println(e);
}
```

Wie das Beispiel demonstriert, können viele Methoden alternativ mit Typformalparametern oder Wildcard-Datentypen realisiert werden (siehe Bloch 2018, S. 144f).

Aufgabe 2

Weil `sl` vom parametrisierten Typ **ArrayList<String>** ist, fügt der Compiler die folgende Casting-Operation ein:

```
System.out.println((String)sl.get(0));
```

Diese scheitert, weil ein **Integer**-Objekt in `sl` eingeschmuggelt worden ist. Das konnte passieren, weil im ersten Parameter der Methode `addElement()` der **ArrayList**-Rohtyp verwendet wird.

Aufgabe 3

Einer Referenzvariablen vom Rohtyp (z. B. **ArrayList**) kann ein Objekt mit einem beliebigen parametrisierten Typ zugewiesen werden (z. B. **ArrayList<String>**). Anschließend kann der Compiler nicht verhindern, dass über die Rohtyp-Referenz ein Element mit abweichendem Typ eingefügt und somit das Kollektionsobjekt beschädigt wird.

Zeigt eine Referenzvariable auf eine Parametrisierung mit dem Elementtyp **Object**, verhindert der Compiler das beschriebene Problem.

Kapitel 9 (*Interfaces*)

Aufgabe 1

1. **Falsch**
2. **Richtig**
3. **Falsch**
   Das im Abschnitt 9.2 vorgeführt API-Interface **Serializable** enthält keine Methoden. Es ist ein sogenanntes *Marker-Interface*.
4. **Richtig**
5. **Richtig**

Aufgabe 2

Sie finden einen Lösungsvorschlag (als IntelliJ-Projekt) im Verzeichnis:

...\BspUeb\Interfaces\Bruch implements Comparable

Im Beispiel implementiert die Klasse `Bruch` das generische Interface **Comparable<E>**, während die zum Sortieren verwendete **Arrays**-Methode **sort()**

public static void sort(Object[] *oar***)**

vom Elementtyp ihres Array-Parameters das *nicht*-generische Interface **Comparable** erwartet. Das klappt „dank" Typlöschung.

Aufgabe 3

Beim Aufruf der folgenden Methode

```
static <T extends Basis & SayOne & SayTwo> void moin(T x) {
   int max = x.getRep();
   for (int i = 0; i < max; i++) {
      x.sayOne();
      x.sayTwo();
      System.out.println();
   }
}
```

muss der Typ des Aktualparameters die Klasse `Basis` in seiner Ahnenreihe haben. Außerdem muss er die Schnittstellen `SayOne` und `SayTo` erfüllen. Den vollständigen Quellcode finden Sie im Ordner

...\BspUeb\Interfaces\MultiBound

Kapitel 10 (Java Collections Framework)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Java Collections Framework\Mengen\BaumLotto

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Java Collections Framework\Abbildungen\StringUtil

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Java Collections Framework\Mengen\Mengenlehre

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Java Collections Framework\Abbildungen\DataMat

Kapitel 11 (Ausnahmebehandlung)

Aufgabe 1

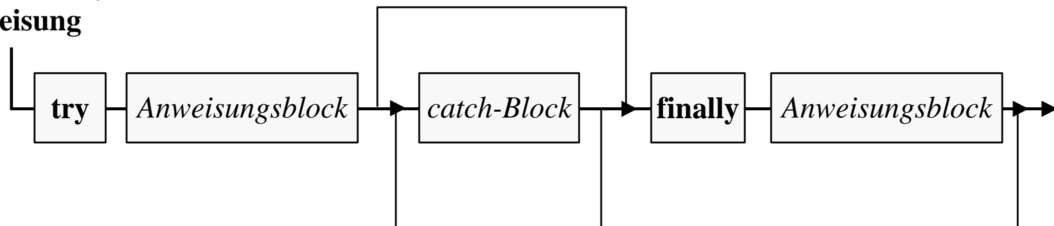
1. **Falsch**
   Bei der Ausnahmeklasse **RuntimeException** ist die *Vorbereitung* (z. B. per **try-catch-finally** - Anweisung) freiwillig. Bleibt jedoch eine geworfene Ausnahme dieser Klasse unbehandelt, wird das Programm (genauer: der betroffene Thread) von der JVM beendet.
2. **Richtig**
3. **Falsch**
   Ist ein **finally**-Block vorhanden, wird dieser auch nach einem störungsfreien **try**-Block ausgeführt, bevor es hinter der **try-catch-finally** - Anweisung weitergeht.
4. **Falsch**
   In einem **catch**- oder **finally**-Block ist selbstverständlich auch eine **try-catch-finally**-Anweisung erlaubt.
5. **Richtig**

Man kann auch bei Verzicht auf einen **catch**-Block per **finally**-Block dafür sorgen, dass im Ausnahmefall vor dem Verlassen der Methode noch bestimmte Anweisungen ausgeführt werden.

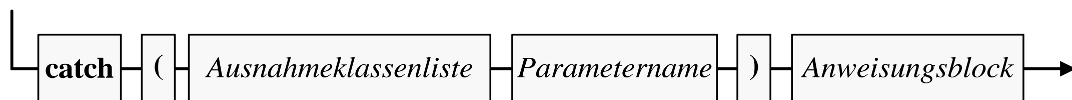
Aufgabe 2

Lösungsvorschlag:

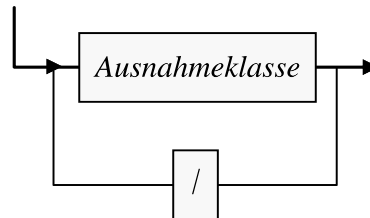
try-catch-finally - Anweisung



catch-Block



Ausnahmeklassenliste



Aufgabe 3

Lösungsvorschlag:

```
class Prog {
    public static void main(String[] args) {
        Object o = null;
        System.out.println(o.toString());
    }
}
```

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Ausnahmebehandlung\DuaLog\IllegalArgumentException

Kapitel 12 (Funktionales Programmieren)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Funktionales Programmieren\Lambda-Ausdrücke\LambdaVsAK

Mit einem Lambda-Ausdruck ist das Problem sehr elegant zu lösen:

```
int sq100 = IntStream.rangeClosed(1, 100)
       .map(n -> n*n)
       .sum();
```

Bei Verwendung einer anonymen Klasse steigt der Aufwand, während sich die Lesbarkeit des Quellcodes verschlechtert:

```
int sq100 = IntStream.rangeClosed(1, 100)
       .map(new IntUnaryOperator() {
           public int applyAsInt(int n) {
              return n*n;
           }
         })
       .sum();
```

Aufgabe 2

Bei der *Erstellung* eines Objekts auf der Basis eines Lambda-Ausdrucks muss der Compiler das zu implementierende funktionale Interface kennen, um per Typinferenz die erforderlichen Prüfungen und Einstellungen vornehmen zu können. Anschließend kann die Adresse des per Lambda-Ausdruck realisierten Objekts durchaus in einer Referenzvariablen vom Typ **Object** abgelegt werden, z. B.:

```
Predicate<String> ps = s -> s.length() >= 5;
Object obj = ps;
```

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Funktionales Programmieren\Ströme\Fakultät

Der Lösungsvorschlag liefert z. B. zum Argument 1.000.000 bei Verwendung eines parallelen Stroms nach wenigen Sekunden eine Lösung und arbeitet dabei parallel mit mehreren CPU-Kernen, was an der Nutzung von mehr als 25% CPU-Zeit auf einem Rechner mit 4 virtuellen Kernen zu erkennen ist:

Task-Manager

Datei Optionen Ansicht

Prozesse | Leistung | App-Verlauf | Autostart | Benutzer | Details | Dienste

Name	Status	99% CPU	84% Arbeitss...	0% Datenträ...	0% Netzwerk	Stromverbrauch	Stromverbrau...
Windows-Befehlsprozessor (3)		95,5%	707,3 MB	0 MB/s	0 MBit/s	Sehr hoch	Sehr hoch
Java(TM) Platform SE binary		95,5%	704,4 MB	0 MB/s	0 MBit/s	Sehr hoch	Mittel

Weniger Details | Task beenden

Kapitel 13 (*GUI-Programmierung mit JavaFX*)

Aufgabe 1

1. **Falsch**
   Meist ist ein Layoutmanager als Wurzelknoten tätig. Zur Realisation komplexer Bedienoberflächen werden Layoutmanager oft auch als eingeschachtelte Kindelemente auf verschiedenen Hierarchieebenen des Szenegraphen eingesetzt.
2. **Richtig**
3. **Falsch**
   Die **main()** - Methode in der Startklasse einer JavaFX-Anwendung sollte ausschließlich einen Aufruf der Methode **launch()** enthalten:

```
Launch(args);
```

Die Methode **main()** darf fehlen, wobei dann die Laufzeitumgebung für den Aufruf der Methode **launch()** sorgt.

4. **Falsch**
   Es ist zumindest bei komplexen Bedienoberflächen sinnvoll, eine Deklaration per FXML vorzunehmen, doch kann auch eine JavaFX-Anwendung komplett durch Anweisungen realisiert werden.
5. **Richtig**

Aufgabe 2
Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\JavaFX\Steuerelemente\SliderLabelSync

Aufgabe 3
Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\JavaFX\Steuerelemente\DM-Euro - Konverter

Kapitel 14 (Ein- und Ausgabe über Datenströme)

Aufgabe 1

1. **Falsch**
   Es stimmt, dass die schon in Java 1.0 vorhandene Klasse **PrintStream** durch die Klasse **PrintWriter** ersetzt worden ist. Der per **System.in** ansprechbare Standardausgabestrom sowie der per **System.err** ansprechbare Standardfehlerausgabestrom werden allerdings nach wie vor durch Objekte der Klasse **PrintStream** realisiert.
2. **Richtig**
3. **Richtig**
4. **Richtig**

Aufgabe 2
Bei der **read()** - Rückgabe stehen Werte von 0 bis 255 am Ende eines erfolgreichen Leseversuchs. Das erreichte Dateiende signalisiert **read()** durch den Rückgabewert -1. Durch die Wahl des Typs **int** kann der Rückgabewert Nutzdaten oder eine Fehlerinformation transportieren (vgl. Abschnitt 11.3). Weil es *kein* außergewöhnliches Ereignis darstellt, beim Lesen einer Datei irgendwann auf deren Ende zu stoßen, informiert die Methode **read()** in diesem Fall über den Kombirückgabewert. Bei unerwarteten Problemen wirft **read()** hingegen eine **IOException**.

Aufgabe 3
Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\IO\OutputStream\BufferedOutputStream\Konsole

Aufgabe 4

Im **PrintWriter**-Konstruktor ist die **autoFlush**-Option eingeschaltet, was bei einer Dateiausgabe in der Regel keinen Nutzen bringt. So hat jeder **println()** - Aufruf einen zeitaufwändigen Dateizugriff zur Folge, und die voreingestellte Pufferung durch den eingebundenen **OutputStreamWriter** (vgl. Abschnitte 14.4.1.2 und 14.4.1.5) wird abgeschaltet. Für das Programm ist der folgende **PrintWriter**-Konstruktoraufruf besser geeignet:

```
try (PrintWriter pw = new PrintWriter(new FileOutputStream("pw.txt"))) {
    . . .
}
```

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\IO\MS-DOS

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\IO\Datenmatrix

Kapitel 15 (Multithreading)

Aufgabe 1

1. **Falsch**
2. **Richtig**
   Werden im UI-Thread länger laufende Methoden ausgeführt, reagiert die Bedienoberfläche zäh.
3. **Richtig**
4. **Falsch**
   Daher sollte sich ein Thread niemals in einem synchronisierten Bereich zur Ruhe begeben.

Aufgabe 2

Der **interrupt()** - Aufruf zum Setzen des Unterbrechungssignals trifft fast immer auf einen schlafenden `Schnarcher`-Thread. In dieser Situation wirft die Methode **sleep()** eine **InterruptedException** und löscht das Unterbrechungssignal wieder. Damit die **run()** - Methode plangemäß reagieren kann, muss in der Regel bei der Ausnahmebehandlung **interrupt()** erneut aufgerufen werden, um des Interrupt-Signal zu restaurieren:

```
class Schnarcher extends Thread {
   @Override
   public void run() {
      while (true) {
         if(isInterrupted())
            return;
         try {
            sleep(100);
         } catch(InterruptedException ie) {interrupt();}
         System.out.print("*");
      }
   }
}
```

Im aktuellen Beispiel wäre auch eine **return**-Anweisung im **catch**-Block angemessen, um die **run()** - Methode und den Thread zu beenden.

Aufgabe 3

Weil jeder Thread seinen eigenen Stapelspeicher für Methodenaufrufe besitzt, sind bei lokalen Variablen konkurrierende Zugriffe durch mehrere Threads unmöglich.

Literatur

Baltes, S. & Diehl, S. (2014). *Sketches and Diagrams in Practice*. Paper presented at the International Symposium on the Foundations of Software Engineering (November 2014, Hong Kong).

Baltes-Götz, B. & Götz, J. (2016). *Einführung in das Programmieren mit Java 8*. Online-Dokument: https://www.uni-trier.de/fileadmin/urt/doku/java/v80/java8.pdf

Baltes-Götz, B. & Götz, J. (2018). *Einführung in das Programmieren mit Java 9*. Online-Dokument: https://www.uni-trier.de/index.php?id=22787

Baltes-Götz, B. (2018). *Einführung in die Entwicklung von Apps für Android 8*. Online-Dokument: https://www.uni-trier.de/index.php?id=60390

Baltes-Götz, B. (2019). *Einführung in das Programmieren mit C# 7.3*. Online-Dokument: https://www.uni-trier.de/index.php?id=22777

Balzert, H. (2011). *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2*. Heidelberg: Spektrum.

Bloch, J. (2008). *Effective Java* (2nd. ed.). Upper Saddle River, NJ: Addison-Wesley.

Bloch, J. (2018). *Effective Java* (3rd. ed.). Upper Saddle River, NJ: Addison-Wesley.

Booch, G. et al. (2007). *Object-Oriented Analysis and Design with Applications* (3rd ed.). Boston, MA: Addison-Wesley.

Bracha, G. (2004). *Generics in the Java Programming Language*. Online-Dokument: http://www.cs.rice.edu/~cork/312/Readings/GenericsTutorial.pdf

Epple, A. (2015). JavaFX 8. Heidelberg: dpunkt-Verlag.

Evans, B.J. & Flanagan, D. (2015). *Java in a Nutshell* (5th ed.). Beijing: O’Reilly.

Flanagan, D. (2005). *Java in a Nutshell* (5th ed.). Sebastopol, CA: O’Reilly.

Goetz, B. (2003). *Concurrent Collection Classes*. Online-Dokument: https://www.ibm.com/developerworks/library/j-jtp07233/index.html

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. & Lea, D. (2006). *Java Concurrency in Practice*. Addison-Wesley. Upper Saddle River, NJ: Addison-Wesley.

Goetz, B. (2007). *Java Theory and Practice. Stick a Fork in it, Part 1*. Online-Dokument: http://public.dhe.ibm.com/software/dw/java/j-jtp11137-pdf.pdf

Goll, J., Weiß, C. & Rothländer, P. (2000). *Java als erste Programmiersprache*. Stuttgart: Teubner.

Goll, J. & Heinisch, C. (2016). *Java als erste Programmiersprache* (8. Aufl.). Wiesbaden: Springer Vieweg

Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A. & Smith, D. (2019). *The Java Language Specification. Java SE 13 Edition* (Ausgabe 2019-08-21). Online-Dokument: https://docs.oracle.com/javase/specs/jls/se13/jls13.pdf

Grammes, R., Lehmann, M. & Schaal, K, (2017). Java 9 bringt das neue Modulsystem Jigsaw. Online-Dokument: https://www.informatik-aktuell.de/entwicklung/programmiersprachen/java-9-das-neue-modulsystem-jigsaw-tutorial.html

Grossmann, D. (2012). Beginner's Introduction to Java's ForkJoin Framework. Online-Dokument: http://www.cs.washington.edu/homes/djg/teachingMaterials/grossmanSPAC_forkJoinFramework.html

Hommel, S. (2014). Oracle JvaFX. Implementing JavaFX Best Practices. Online-Dokument: http://docs.oracle.com/javafx/2/best_practices/jfxpub-best_practices.pdf

Horstmann, C.S. (2014a). *Java SE 8 for the Really Impationt*. Upper Saddle River, NJ: Addison Wesley.

Horstmann, C.S. (2014b). *Lambda Expressions in Java 8*. Online-Dokument: http://www.drdobbs.com/jvm/lambda-expressions-in-java-8/240166764?pgno=1

Horstmann, C.S. (2015). *Core Java for the Impationt*. Upper Saddle River, NJ: Addison Wesley.

Horstmann, C.S. & Cornell, G. (2002). *Core Java. Volume II – Advanved Features*. Palo Alto, CA: Sun Microsystems Press.

Inden, M. (2015). *Java 8. Die Neuerungen* (2. Aufl.). Heidelberg: dpunkt-Verlag.

Inden, M. (2018). *Der Weg zum Java-Profi* (4. Aufl.). Heidelberg: dpunkt-Verlag.

Kegel, H. & Steimann, F. (2008). *Systematically Refactoring Inheritance to Delegation in JAVA*. In: Schäfer, W., Dwyer, M.B. & Gruhn, V. (eds.) 30th International Conference on Software Engineering (ICSE), S. 431-440. Leipzig.

Kreft, K & Langer, A. (2008a). *Java Memory Model: Überblick*. Online-Dokument: http://www.angelikalanger.com/Articles/EffectiveJava/38.JMM-Overview/38.JMM-Overview.html

Kreft, K & Langer, A. (2008b). *Regeln für die Verwendung von volatile*. Online-Dokument: http://www.angelikalanger.com/Articles/EffectiveJava/42.JMM-volatileIdioms/42.JMM-volatileIdioms.html

Kreft, K. & Langer, A. (2012). *Java 7. Thread-Synchronisation mit Hilfe des Phasers*. Online-Dokument: http://www.angelikalanger.com/Articles/EffectiveJava/63.Java7.Phaser/63.Java7.Phaser.html

Kreft, K & Langer, A. (2014). *Java 8. Default-Methoden und statische Methoden in Interfaces*. Online-Dokument: http://www.angelikalanger.com/Articles/EffectiveJava/72.Java8.DefaultMethods/72.Java8.DefaultMethods.html

Krüger, G. & Hansen, H. (2014). *Java-Programmierung - Das Handbuch zu Java 8*. Köln: O'Reilly

Langbridge, J.A. (2014). *Professional Embedded ARM Development*. Indianapolis: Wiley & Sons.

Lau, O. (2009). Faites vos jeux! Zufallszahlen erzeugen, erkennen und anwenden. *c't Magazin für Computertechnik*. 2009, Heft 2, 172-178.

Lahres, B. & Rayman, G. (2009). *Praxisbuch Objektorientierung* (2. Aufl.). *Professionelle Entwurfsverfahren*. Bonn: Galileo

Langer, A. & Kreft, K. (2004). *Java Generics - Type Erasure*. Online-Dokument: http://www.angelikalanger.com/Articles/JavaMagazin/Generics/GenericsPart2.html

Liskov, B. H. & Wing, J. M. (1999). *Behavioral Subtyping Using Invariants and Constraints*. Online-Dokument: http://reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-156.ps

Mak, S. & Bakker, P. (2017). *Java 9 Modularity*. Sebastopol, CA: O'Reilly.

Martin, R.C. (2002). *SRP: The Single Responsibility Principle*. Online-Dokument: http://www.objectmentor.com/resources/articles/srp.pdf

Mednieks, Z., Dornin, L., Meike, G. B. & Nakamura, M. (2013). *Android Programmierung* (2. Aufl.). Köln: O'Reilly.

Mitran, M., Sham, I. & Stepanian, L. (2008). *Decimal floating-point in Java 6*. Online-Dokument: http://www-03.ibm.com/servers/enable/site/education/wp/181ee/181ee.pdf

Mössenböck, H. (2003). *Softwareentwicklung mit C#. Ein kompakter Lehrgang*. Heidelberg: dpunkt.

Morrison, V. (2005). *Concurrency: What Every Dev Must Know About Multithreaded Apps*. MSDN Magazine. August 2005. Online-Dokument: http://msdn.microsoft.com/de-de/magazine/cc337899.aspx.

Mössenböck, H. (2005). *Sprechen Sie Java? Einführung in das systematische Programmieren*. Heidelberg (3. Aufl.). Heidelberg, dpunkt.Verlag.

Naftalin, M. & Wadler, P. (2007). *Java Generics and Collections*. Sebastopol, CA: O'Reilly.

Oaks, S. (2014). *Java Performance: The Definitive Guide*. Sebastopol, CA: O'Reilly.

Oechsle, R. (2018). *Parallele und verteilte Anwendungen in Java*. München: Hanser.

Oracle (2010). *Java Object Serialization Specification*. Online-Dokument: https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html.

Oracle (2019a). *The Java Tutorials*. Online-Dokument: http://download.oracle.com/javase/tutorial/.

Oracle (2019b). *Java Platform, Standard Edition, Oracle JDK Migration Guide*. Online-Dokument: https://docs.oracle.com/en/java/javase/11/migrate/index.html

Petre, M. (2013). UML in practice. In: *35th International Conference on Software Engineering (ICSE 2013)*, 18-26 May 2013, San Francisco, CA, USA (forthcoming), pp. 722–731.

Reinhold, M. (2016). *The State of the Module System*. Online-Dokument: http://openjdk.java.net/projects/jigsaw/spec/sotms/

Sosnoski, D. (2014). Java 8 language changes. Online-Dokument: https://www.ibm.com/developerworks/library/j-java8lambdas/j-java8lambdas-pdf.pdf

Sharan, K. (2015). *Learn JavaFX 8*. New York: Apress.

Simons, R. (2004). *Hardcore Java*. Sebastopol, CA: O'Reilly.

Strey, A. (2005). Computer-Arithmetik. Online-Dokument: http://www.informatik.uni-ulm.de/ni/Lehre/SS05/CompArith/

Ullenboom, C. (2016). *Java ist auch eine Insel* (12. Aufl.). Bonn: Rheinwerk. OpenBook: http://openbook.rheinwerk-verlag.de/javainsel/

Urma, R.-G. (2014). *Processing Data with Java SE 8 Streams, Part 1*. Online-Dokument: http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

Vorontsov, M. (2014). *Java Performance Tuning Guide. String.intern in Java 6, 7 and 8 – string pooling*. Webseite: http://java-performance.info/string-intern-in-java-6-7-8/

Weaver, J. (2014). *Pro JavaFX 8: A Definitive Guide to Building Desktop, Mobile, and Embedded Java Clients*. New York: Apress.

Ziesche, P. & Arinir, D. (2010). *Java: Nebenläufige und verteilte Programmierung* (2. Aufl.). Herdecke: W3L-Verlag.

Index

&

&
- Bei beschränkt. Typformalparametern...422
- Bitweises UND 147

@

@FXML...608

A

Abhängigkeiten 620
Ablaufsteuerung......................................166
Abschluss 555
abstract 396
Abstract Windowing Toolkit 589
AbstractSet<E>..478
Abstrakte
- Klasse..397
- Methode 396

Abstraktion...19
accept()...670
acos()...284
add()
- Collection<E>.......................................465
- ListIterator<E>.....................................478
- Set<E>..479

addAll()
- Collection<E>.......................................465
- List<E> 469
- Set<E>..479

addAndGet()..755
add-exports...374
addListener() 617
Aggregat-Operationen...............................561
Aggregatormodul 351
Aktualisierungsoperatoren 152
Aktualparameter...............................224, 225
Alan Kay 198
Algorithmen 26
alignment...634, 635
allMatch()...580
Amazon 53
AnchorPane...632
Android 44
Annotation..417
Annotationen..453
Annotationselemente..................................454
Anonyme Klassen 241, 262, 546
ANSI 686
Anweisungen..165
Anweisungsblöcke 165
anyMatch()..580
Anzeigeeinstellungen.................................630
API 21, 42, 375
append()
- StringBuilder..308

Application 594
applyAsInt() 565
Äquivalenzrelation.....................................467
Archivdateien...339
Arithmetische Operatoren..........................137
Arithmetischer Ausdruck...........................137
Array 287
- mehrdimensionaler 295

ArrayBlockingQueue<E>.................742, 776
ArrayDeque<E>...501
ArrayIndex¬Out¬Of¬BoundsException ...291
ArrayIndexOutOfBoundsException 509
ArrayList.....................................310, 405
ArrayList<E>..472
Arrays 287, 292, 319, 435
- Klasse..319

ArrayStoreException 290, 411
ART 44
ASCII...685
asList() 551, 563, 581
- Arrays 463

Assembler 40
assert 527
AssertionError..527
Assoziative Funktion 574
Assoziativität 154
Assoziativität von Operatoren 154
Atomar 732
Atomare Variablen......................................754
AtomicInteger..................................323, 755
Aufzählungen...315
Ausdrücke 136
Ausdrucksanweisungen 165
Ausführungskonfiguration...................91, 175
Ausnahmen 507
Auswertungsreihenfolge 153
Autoboxing 310
AutoCloseable..................................539, 655
autoFlush
- PrintStream 679
- PrintWriter 691

Automatische Typanpassung 148
Automatisches Modul.................................373
Autounboxing 311
available() 673
- FileInputStream 681

average() 566, 577

await() 740, 758
AWT 589
B
Balancierter Binärbaum 483
BasicFileAttributes 661
Beans 612
Bedingte Anweisung 166
Bedingte Thread-Sicherheit 775
Befehlsschalter 640
Benutzer-Thread 728
Beobachtbare Listen 624
Beschränkte Typformalparameter 419
Bezeichner 99
Big Ball of Mud 348
BigDecimal 116, 142, 162, 163, 703
Big-Endian 685
BigInteger 159
Binärbaum 483
Binäre Gleitkommadarstellung 114
Binäre Operatoren 137
BinaryOperator<T> 574
binarySearch()
  Arrays 292
  Collections 502
bind() 619, 623
bindBidirectional() 619
Binding-Klassen 620
Bindings 622
Bindungskraft 154
Bitorientierte Operatoren 147
Bitweises UND 147
Block 121
Blockanweisung 121, 165
Blockierender Methodenaufruf 760
BlockingQueue<E> 742
BMP 639
Boilerplate-Code 550
boolean 113
boolean-Literale 127
BorderPane 634
Boxing 310
break-Anweisung 173, 184
breakpoint 228
Brückenklassen 685
Bucket 482, 495
BufferedInputStream 681, 718
BufferedOutputStream 676, 717
BufferedReader 695
BufferedWriter 689
Button 640
byte 112
ByteArrayInputStream 680, 701
ByteArrayOutputStream 672, 701
Bytecode 40
C
C++ 47, 122
Calendar 662
Call Back - Routinen 588
Callable<V> 765
Callback<P, R> 627
Camel Casing 108, 211, 219
cancel() 782
canWrite() 668
CAS 755
case-Marke 173
Casting 148
Casting-Operator 149
catch-Block 511
ceiling()
  NavigableSet<E> 489
ceilingEntry()
  NavigableMap<K,V> 498
ceilingKey()
  NavigableMap<K,V> 498
changed()
  ChangeListener<T> 617
ChangeListener<? super T> 617
Channel 649, 674
ChannelInputStream 683
char 113
Character 315
CharArrayReader 694
CharArrayWriter 684
charAt() 303
char-Literale 127
CharSequence 453
Charset 686, 711
CheckBox 644
checked exceptions 528
checkError() 522, 678, 690
children 606
Class 201, 456
ClassCastException 406, 410, 413
classpath
  -Kommandozeilenargument 59
CLASSPATH 334
  -Umgebungsvariable 58
Cleaner 245
clear()
  Collection<E> 465
  Map<K,E> 491
close() 243, 654, 677
closure 555
collect() 566, 568, 577

Collection<E> 464, 468
Collections 473, 481, 490, 501
Collectors 578
ColumnConstraints.....................................632
columnIndex...631
columnSpan...631
Comparable<T> 435, 438
comparator()
  SortedMap<K,V>......................................497
  SortedSet<E> 487
Comparator<E>.................................486, 497
Comparator<T>..598
Compare-and-Set 755
compareAndSet() 755
compareTo().....................................419, 436
  Path...659
  String 302
Compiler..40
compute() 771
computeValue()...622
ConcurrentHashMap<K,V>.......................775
ConcurrentSkipListMap<K,V> 776
ConcurrentSkipListSet<E> 776
Condition..740
contains()
  Collection<E> 465
  Map<K, V> 491
  Set<E>..479
containsAll()
  Collection<E> 465
containsValue()
  Map<K, V> 491
continue-Anweisung 184
Control 596
Controller 272, 607
controls 587
copy()
  Files 663
copyOf() 292, 418
CopyOnWriteArrayList<E> 776
CopyOnWriteArraySet<E>........................776
CopyOption 664
Corretto 53
cos()..284
count() 565, 568, 577
countDown() 748
CountdownLatch..748
Cp1252 686
Cp850 686
CPU 40
createDirectories 660
createDirectory() 660
createFile() 660
createNewFile()
  File 668
currentThread() 728
currentTimeMillis() 195, 293
Custom Modular Runtime Image 371
CyclicBarrier...749

D

Daemon-Thread 778, 782
Dalvik 44
dangling else 170
DataInputStream 538, 652, 683, 717
DataOutputStream 653, 675, 717
Datei
  explizit erstellen ab Java 7.....................660
  explizit erstellen in Java 6......................668
  löschen ab Java 7 665
  löschen in Java 6....................................670
  umbenennen in Java 6............................670
  umbenennen oder verschieben ab Java 7
   664
Datei-Öffnungsoptionen 710
Dateisystem..665
Datenkapselung.................................198, 215
Datenströme 649
Datentyp...106
Datentypen
  primitive...112
Deadlock 762
Debug...228
decrementAndGet() 755
Default Button 641
default package 328
DeflaterOutputStream................................672
Deklarative Programmierung.....................453
Delegation..400
delete() 665
  File 670
  Files..665
  StringBuilder..308
deleteIfExists()...665
Denormalisierte
  Gleitkommadarstellung..........................116
Dependencies 620
Deprecated 453, 457
Deque<E> 501
descendingIterator()
  NavigableSet<E>.....................................489
design pattern...437
Dezimale Gleitkommadarstellung 116
Dezimaltrennzeichen 699
Differenz von zwei Mengen 479

distinct() 566, 568
Documented 458
Dokumentationskommentar 97, 457
Doppelt verkettete Liste 473
do-Schleife 183
double 112, 142
Double 161, 435
dropWhile() 569
DRY-Prinzip 551
Dualer Logarithmus 541
Duke 639
Durchfall 173
Durchschnitt von zwei Mengen 479
Dynamisches Binden 396

E

Eager Execution
  Binding<T> 621
Eclipse 82
Eingeschachtelte Klassen 257
Eingeschachtelte Schnittstellen 445
Einschränkende Konvertierung 149
Einstellige Operatoren 137
else-Klausel 167
emptyList()
  Collections 502
emptyMap()
  Collections 502
emptySet()
  Collections 502
Encodings 685
Endlosschleifen 184
entry point 344
entrySet()
  Map<K,V> 493
Entwurfsmuster 437
Enumerationen 315
equals() 467, 478
  Integer 312
  String 301
Eratosthenes 320
Error 526
ERRORLEVEL 509
Ersetzbarkeitsregel 398
Erweiternde Typanpassung 138, 148
Erweiterte for-Schleife 180
Escape-Sequenzen 127
Euklidischer Algorithmus 26, 195
Exception-Handler 511
Exceptions 507
execute() 763
Executors 763, 766
ExecutorService 763, 766, 770
exists() 659
  File 667
exit() 509
  JavaFX 595
Exitcode 509
Exklusives logisches ODER 146
Explizit terminierter Kommentar 97
Explizite Lock-Objekte 737
Expliziten Module 373
Explodiertes Modul 356
exports (JPMS) 351
extends
  Typrestriktion 420
  Vererbung 385

F

Fabrikmethode 241
fail-fast 477
Fakultät 323, 575
Faltung 574
Fehlerausgabestrom 660
Fehlerstatus 522
Fehlerstatuskontrolle 132
Felder
  überdecken 393
FIFO 742
File 656, 667
FileAlreadyExistsException 660, 664
FileDescriptor 679, 719
FileInputStream 539, 673, 681, 717
FilenameFilter 670
FileNotFoundException 672
FileOutputStream 672, 717
FileReader 695
Files 674, 682, 692, 696, 709
FileStore 665
FileSystem 665
FileSystems 665
FileTime 661
FileWriter 687, 688
fill()
  Arrays 292
filter() 566, 568
Filterklassen 652
Filter-Map-Reduce 582
Filter-Map-Reduce - Muster 560
FilterReader 694
FilterWriter 684
final 123, 214, 386, 392
Finalisierte
  Instanzvariablen 214
  Klassen 386
  Methoden 392

Finalisierte lokale Variablen 123
finalize() 243, 539, 654, 677
finally 515, 538
finally-Block 511
findAny()..580
findFirst() 580
first()
SortedSet<E> 487
firstEntry()
NavigableMap<K,V>............................498
firstKey()
SortedMap<K,V>.................................497
fixed arity - Methoden...............................223
Fließkommazahl.......................................113
float 112, 142
floating point number................................113
floor()
NavigableSet<E> 489
floorEntry()
NavigableMap<K,V>............................498
floorKey()
NavigableMap<K,V>............................498
Fluent API 560, 621
flush() 677, 720
Flussdiagramm 166
Font 645
font()...645
forEach() 573
fork() 772
Fork-Join - Framework 769
ForkJoinPool 770
ForkJoinTask<T>......................................770
Formalparameter 220
format() 102, 281
PrintWriter 690
Formatierungszeichenfolge 103
Formatter 104
for-Schleife..179
FowPane..636
Framework 437
Freigabe von Ressourcen 538
fromMillis()..662
Function<T,R> 570
FunctionalInterface 545
Funktion höherer Ordnung........................583
Funktionale Programmierung 543
Funktionale Schnittstellen.........................544
Funktionsobjekt..546
Future<T> 766
FXCollections 597, 603, 624
FXML...605
FXMLLoader 594, 610

G

Ganzzahlarithmetik...................................138
Ganzzahlliterale 124
Garbage Collector.......................47, 242, 782
gc() 244
gchar()...147, 183
gdouble() 193
generate() 565
Generische
Methoden 422
Generizität...405
Geprüfte Ausnahmen 528
get()
Callable<V>...767
List<E> 469
Map<K,V>...491
Paths...657
getAbsolutePath() 668
getAndAdd() 755
getAndDecrement() 755
getAndIncrement()...................................755
getAnnotation()...456
getCause() 531, 533
getClass() 201
getDefault()...665
getFileName() 657
getFileStores()...666
getLastModifiedTime().............................661
getMessage()....................................519, 531
getName()
File 669
Path 658
getNameCount() 658
getParent().......................................658, 665
getPriority()..760
getProperty() 657
getRoot() 658
getState()..743, 762
getter 216
getText() 643
getTotalSpace()..666
getUncaughtExceptionHandler() 510, 520
getUsableSpace() 666, 668
GGT 26
GIF 639
gint() 131, 700
GitHub 77
Gitterlinien.......................................598, 630
Gleitkommaarithmetik......................116, 138
Gleitkommadarstellung
binär 114
dezimal..116

Gleitkommaliterale....126
Gleitkommazahl....113
Globale Variablen....111
Gluon....84, 590
GMT....663
GridPane....594, 597, 604, 606, 629
Größter gemeinsamer Teiler....26
groupingBy()....579
GUI....587
Gültigkeitsbereich....208
  lokale Variablen....121
H
halignment....631
Hallo-Beispielprogramm....53
Hashcode....390
hashCode()....482
Hash-Funktion....482
Hash-Kollision....482
HashMap<K,V>....418, 494
Hashtabelle....481
Hashtable<K,V>....490
hasNext()....698
  Iterator<E>....476
hasNextBigDecimal()....698
hasNextDouble()....698
hasNextInt()....698
hasPrevious()
  ListIterator<E>....478
HBox....633
Header-Dateien....47
headMap()
  SortedMap<K,V>....497
headSet()
  SortedSet<E>....487
Heap....110, 208, 236, 724
heigher()
  NavigableSet<E>....489
heigherEntry()
  NavigableMap<K,V>....499
heigherKey()
  NavigableMap<K,V>....498
Heimatverzeichnis....657
Hexadezimalsystem....124
hgap....630
hgrow....631, 633
High-Level - Binding-API....621
High-Level Binding-API....640
Hollywood-Prinzip....588
I
IBM850....686
IEEE-754....114, 164
if-Anweisung....166
IllegalArgumentException....534, 541
Image....639
ImageView....639
immutable....299, 310
implements....447
Implizite Typumwandlung....148
Import
  Alle Typen eines Paketes....336
  Einzelner Typ....336
  Statische Mitglieder....336
import-Deklaration....100
incrementAndGet()....755
indeterminate....644
indexOf()
  List<E>....469
  String....303
InflaterInputStream....681
information hiding....215
Information Hiding....198
Inherited....458
init()
  JavaFX....595
Initialisierer
  statische....251
Initialisierung....118
Initialisierungsliste....311
Initialisierungslisten....294
Initializable....608
initialize()....608
Injektion....607
Innere Ausnahme....531
Innere Klasse....258
InputMismatchException....698
InputStream....528, 680
InputStreamReader....694
insert()
  StringBuilder....308
Insets....630, 632, 633, 635
instanceof....395, 409, 448, 472
Instanzinitialisierer....240, 623
Instanzvariablen....110, 207
int....112
IntBinaryOperator....575
Integer....309
IntegerProperty....614
IntelliJ
  Ausführungskonfiguration....91, 175
  Pakete....330
IntelliJ IDEA....51, 133
  Installieren....61
Interface....435, 463
Intermediäre Operationen....566

intern() 304
Interner String-Pool 299
Interpreter 41
interrupt() 758
InterruptedException 514, 516, 727, 757, 782
Interrupt-Signal 758
IntStream 560, 564, 565, 574, 575
IntUnaryOperator 565
invalidated()
  InvalidationListener 617
InvalidationListener 617
Invarianz 411
invoke() 770
IOException 522, 678, 690
iOS 44
IPS 40
isAbsolute() 658
isBound() 619
isDaemon() 783
isDigit() 315
isDirectory() 661
  File 667
isDone() 767
isEmpty()
  Collection<E> 465
  Map<K,E> 493
isExecutable() 662
isInfinite() 162
isInterrupted() 758
isLetter() 315
isLetterOrDigit() 315
isLowerCase() 315
isNaN() 162
ISO Latin-1 685
ISO8859_1 685
ISO-8859-1 685
isReadable() 662
isRegularFile() 661
isSymbolicLink() 661
isUpperCase() 315
isWhitespace() 315, 698
isWritetable() 662
Iterable<E> 465, 476
Iterable<T> 181, 464
iterate() 564
iterator()
  Iterable<E> 465
Iteratoren 476

J

Jakarta EE 375
jar 361
jar.exe 341
jar-Dateien 339, 359
Java Collections Framework 461
Java Development Kit 40
Java FX Packager 347
Java Platform Module System 326
Java Runtime Environment 41
java.base 351
java.exe 34, 57
java.io 649
java.lang 100, 326
java.lang - Paket 141
JavaBeans 612
javac.exe 40, 55
javadoc 97, 327
JavaFX 84, 129, 587
JavaFX Application Thread 777
JavaFX Script 589
Java-Memory-Model 732
javap 240
javap.exe 800
Java-SE - API 375
Java-Speichermodell 732
javaw.exe 36, 344
Jazelle DBX 40
Jazelle DBX 44
jdeps 374
JDK 40
JEE 43
jlink 348, 371
JME 43
jmod 376
join()
  ForkJoinTask<T> 772
  Thread 756
joining() 579
JOptionPane 186
JPEG 639
JPMS 326
JRE 41
JSON 709
JVM 41

K

KeyIterator 259
keySet()
  Map<K,V> 492
Klasse 19, 197
  abstrakte 397
  Syntaxdiagramm 93
Klassen
  lokale 261
klassenbezogene
  Konstruktoren 251

Methoden 249
klassenbezogene Variablen 246
Klassenliterale 409
Klassenmethode 54
Klassenvariablen 110
Kodierungsschema 698
Kollektionen 461
Kommentar 96
Dokumentationskommentar 97
Explizit terminiert 97
Zeilenblock in IntelliJ auskommentieren 97
Zeilenrest 96
Komponenten 587
Komposition 254, 400
Konditionaloperator 153
Konstanten 247
Konstruktoren 237, 380
Konstruktorreferenzen 558
Kontrollkästchen 644
Kontrollstrukturen 166
Konvertierung
einschränkende 149
Kotlin 44
Kovarianz 290, 411
Kurzschlussauswertung 146, 567

L

Label 638
Lacy Execution
Binding<T> 621
Ladungsfaktor einer Hashtabelle 483
Lambda-Ausdrücke 543
last()
SortedSet<E> 487
lastEntry()
NavigableMap<K,V> 498
lastIndexOf()
List<E> 469
lastKey()
SortedMap<K,V> 497
lastModified() 668
Latin-1 685
launch() 594
layout constraints 630
Layoutmanager 628
lazy 567
Leere Anweisung 165
Leertaste 641
length
Array 287, 291
length()
File 668
String 302
StringBuilder 308
lexical scoping 555
limit() 564, 569
LineNumberReader 694
lines() 565, 713
lineSeparator() 686
LinkedBlocking¬Queue<E> 742, 776
LinkedHashMap<K,V> 496
LinkedHashSet<E> 483
LinkedList<E> 473
Links-Shift-Operator 147
Liskovsches Substitutionsprinzip 398
List<E> 469
ListChangeListener<E> 625
Listen 468
listFiles() 669
listIterator()
List<E> 469
ListIterator<E> 478
ListView<String> 597
Literale 124, 409
Little-Endian 686
Live Templates 72
Lock 737
lock() 737
lockInterruptibly() 738
Logarithmus
dualer 541
Logikfehler 59
Logische Operatoren 144
Logisches ODER 145
Logisches UND 145
Lokale Klassen 261
Lokale Variablen 110
long 112
LongStream 564
lower bound
Wildcard-Typen 427
lower()
NavigableSet<E> 489
lowerEntry()
NavigableMap<K,V> 499
lowerKey()
NavigableMap<K,V> 498
Low-Level - Binding-API 622
LSP 398

M

main() 27, 90
Main-Class 343
Manifest 343
MANIFEST.MF 347
map() 566, 570

Map<K,V> 490
mapToInt() 571
margin 632, 633, 635
Marker Interface 447
Marker-Annotation 455
Marker-Annotationen 454
Maschinencode 40
Math 141, 160, 294
max()
  Collections 501
MAX_VALUE 161
  Double 315
Mehrdimensionale Arrays 295
Mehrfachvererbung 386, 439
Member 23, 197
Memory Flush 732, 735, 740, 754
memory leaks 242, 260
Memory Refresh 732, 735, 740, 754
Mengen
  Differenz 479
  Durchschnitt 479
  Vereinigung 479
Meta-Annotationen 458
Metainformationen 453
Method 456
Method Area 110, 216, 247
Methode
  abstrakte 396
  statische 54
  Syntaxdiagramm 94
Methoden 215
  Aufruf 224
  Definition 216
  Parameter 219
  rekursive 252
  Rückgabewert 218
  statische 249
  Überladen 232
Methodenreferenzen 556
MIME-Type 712
min()
  Collections 502
MIN_VALUE
  Double 315
Mitgliedsklasse 257
mkdir()
  File 667
mkdirs()
  File 667
MKLNK 656
Model-View-Controller - Konzept 600
Modularisierung 198, 199
Moduldeskriptor 349, 356
Module (JPMS) 347
Modulo 139
Modulpfad 357
Monitor 733
move
  Files 664
move() 664
Multi-Catch - Block 512
Multitasking 723
Multithreading 48, 723
Murphy's Law 507
MVC 601

N

Namen 99
  von Klassen 207
  von Methoden 219
Namensparameter 220
NaN 161, 315, 541
NavigableMap<K,V> 498
NavigableSet<E> 486, 488
Nebeneffekt 137, 139, 146
Nebeneffekt-Produzenten 573
Negation 145
NEGATIVE_INFINITY
  Double 315
newBufferedReader() 696, 712
newBufferedWriter() 692, 711, 713
newCachedThreadPool() 763
newCondition() 740
newDirectoryStream() 663
newInputStream() 682, 711
new-Operator 235, 237
newOutputStream() 674, 711
next() 698
  Iterator<E> 477
nextBigDecimal() 698
nextDouble() 698
nextInt() 293, 698
  Random 100
  Scanner 130
NIO.2 - API 656
NoClassDefFoundError 526
Node 596
noneMatch() 580
Normalisierte
  Gleitkommandarstellung 115
normalize()
  Path 659
notExists() 659
notify() 736, 758
notifyAll() 736

now() ... 621
null ... 130, 212, 235
NullPointerException ... 212, 541
Nulltyp ... 130
NumberFormatException ... 313, 510

O

Object
  hashCode() ... 482
ObjectInputStream ... 680, 704
ObjectOutputStream ... 704
Objekte ... 234
Objektgraph ... 701
Objektorientierung ... 45
Objektserialisierung ... 701
Observable ... 617
observableArrayList() ... 624
ObservableList<E> ... 624
ObservableList<String> ... 597
ObservableValue<T> ... 617
of() ... 564
  List<E> ... 470
  Map<K,V> ... 494
  Set ... 487
  Set<E> ... 480
Öffnungsoptionen ... 710
ojdkbuild ... 30, 53
Oktalsystem ... 124
onChanged() ... 625
open (JPMS) ... 353
Open-Closed - Prinzip ... 202, 464
OpenJFX ... 84, 587, 590
OpenOption ... 710
opens (JPMS) ... 352
Operationen
  intermediäre ... 566
  terminale ... 566
  zustandsbehaftete ... 567
  zustandslose ... 566
Operatoren ... 136
  Arithmetische ... 137
  bitorientierte ... 147
  logische ... 144
  vergleichende ... 141
Optional<T> ... 524, 574
OptionalInt ... 552, 574
ordinal() ... 318
Ordner
  anlegen ... 660
  anlegen in Java 6 ... 667
  Inhalt auflisten ab Java 7 ... 663
  Inhalt auflisten in Java 6 ... 669
  löschen ab Java 7 ... 665
  löschen in Java 6 ... 670
  umbenennen ab Java 7 ... 664
  umbenennen in Java 6 ... 670
Orientierung von Operatoren ... 154
out-Objekt in System ... 97
OutOfMemoryError ... 526
OutputStream ... 671
OutputStreamWriter ... 685
Override ... 390, 447, 454, 458

P

Package Splitting ... 358
package-Deklaration ... 327
package-info.java ... 327
packages ... 325
padding ... 606, 630
Pakete ... 325
  java.lang ... 141
Pane ... 596
PAP ... 166
parallel() ... 564, 576
parallelStream()
  Collection<E> ... 465, 563
Parametrisierter Typ ... 408
Parent ... 594
parseDouble() ... 313
parseInt() ... 175
parseLong() ... 187
Pascal ... 204
Pascal Casing ... 207
PasswordField ... 644
Passwörter ... 644
Path ... 657
PATH
  Umgebungsvariable ... 56
Paths ... 657
pathSeparatorChar
  File ... 667
peek() ... 572
Phaser ... 751
PipedInputStream ... 680, 744
PipedOutputStream ... 671, 744
PipedReader ... 694
PipedWriter ... 684
Pipeline ... 561
Platform ... 778
Plusoperator ... 102, 188
PNG ... 639
pollFirst()
  NavigableSet<E> ... 489
pollFirstEntry()
  NavigableMap<K,V> ... 498
pollLast()

NavigableSet<E> 489
pollLastEntry()
NavigableMap<K,V> 498
Polymorphie 202, 395
Portabilität 45
Positionsparameter 220, 225
POSITIVE_INFINITY 541
Double 315
Postinkrement bzw. -dekrement 139
Postinkrementoperator 137
Potenzfunktion 141
pow() 141
PowerShell 357
Präinkrement bzw. -dekrement 139
Präprozessor 47
Predicate<T> 568
Preemtives Zeitscheibenverfahren 761
prefWith 634
Preview 176
previous()
ListIterator<E> 478
previousIndex()
ListIterator<E> 478
Primitive Datentypen 108, 112
Primzahlen 185
print() 102
printf() 102, 224, 678
PrintWriter 690
println() 102
printStackTrace() 508, 510, 520
PrintStream 677
PrintWriter 679, 690
Prinzip einer einzigen Verantwortung 199, 272
Prioritäten 760
private 210
Private Methoden
in Schnittstellen 444
probeContentType() 712
Produktivität 22
Programmablaufplan 166
Programmargumente 174
ProgressBar 781
Properties 399, 611
Property<T> 619
protected 379, 386
Protocol Buffers 709
provides (JPMS) 352
Pseudozufallszahlengenerator 292
public
Klassenmodifikator 53
Puffergröße 676
Pufferung
BufferedOutputStream 676
Punktoperator 212, 224
PushbackInputStream 681
PushbackReader 694
put()
Map<K,V> 491
putAll()
Map<K,V> 491
Q
Qt 46
Quellcode 25
Queue<E> 501
Quick-Fixes 71
R
Race Condition 731
RadioButton 644
Radioschalter 646
Random 251, 293
random() 294
RandomAccess 472
range() 564
rangeClosed() 564
read()
FileInputStream 681
readAllBytes() 710, 711
readAllLines() 711
readAttributes() 661
readObject() 704, 707
ReadOnlyIntegerProperty 614
ReadOnlyIntegerWrapper 614, 615
record 204
RecursiveAction 770
RecursiveTask<T> 770
Red Hat 30, 53
reduce() 566, 574
ReentrantLock 515, 738, 762
ReentrantReadWriteLock 738
Refaktorieren 75
Referenzliteral 130, 235
Referenzparameter 221
Referenztypen 109
Referenzvariablen 234
Reflexion 352, 453, 456
Region 630
Regulärer Ausdruck 304
Reifizierbarer Typ 413
Rekursive Methoden 252
Rekursive Typeinschränkung 420
remove()
Collection<E> 465
Iterator<E> 477
List<E> 470

  - Map<K,V> 491
  - Set<E> 479
- removeAll()
  - Collection<E> 465
  - Set<E> 479
- removeIf()
  - Collection<E> 466
- removeListener() 617
- renameTo()
  - File 670
- replace()
  - String 303
- replace()
  - StringBuilder 308
- replace()
  - Map<K,V> 491
- requireNonNull() 525
- requires (JPMS) 350
- Reservierte Wörter 99
- resolve() 658, 660
- resolveSibling() 658, 664
- Ressourcen freigeben 538
- Restmantisse 115
- resume() 758
- retainAll()
  - Collection<E> 466
  - Set<E> 479
- Retention 455, 459
- RetentionPolicy 455
- return-Anweisung 218
- Returncode 510, 520
- reverse()
  - Collections 502
- Robert C. Martin 199
- Robustheit 47
- Rohtyp 310, 405, 408
- Rot-Schwarz -Architektur 486
- Round-Robin 761
- RowConstraints 632
- rowIndex 631
- rowSpan 631
- Rückgabewert 218, 510, 520
- Run - Fenster 76
- runLater() 778
- Runnable 729, 765
- RuntimeException 528

S

- SafeVarargs 458
- Scanner 130, 471, 515, 697
- Scene 596
- Scene Builder 265, 603
  - installieren 84
- scheduleAtFixedRate() 768
- ScheduledService<V> 779
- ScheduledThreadPoolExecutor 768
- Scheduler 760
- Schleife 178
- Schnittstelle 198, 435, 463
- Schriftauszeichnung 644
- Schwache Konsistenz eines Iterators 775
- scope 121
- Seiteneffekte 582
- selected 644
- Selection Sort 319
- Semantikfehler 59
- Semaphore 746
- SequenceInputStream 680
- Serializable 435, 447, 702
- serialVersionUID 703
- Serienparameter 223
- Service<V> 779
- ServiceLoader 352
- set()
  - List<E> 470
  - ListIterator <E> 478
- *Set*<E> 479
- setAlignment() 643
- setAllowIndeterminate() 644
- setContentDisplay() 639
- setDaemon() 778, 783
- setDefaultButton() 641
- setEditable 648
- setEditable() 644
- setFont() 645
- setGraphic() 639
- setGraphicTextGap() 640
- setHgap() 630
- setImage() 639
- setLastModified() 669
- setLastModifiedTime() 662
- setMnemonicParsing() 641
- setOnAction() 547, 598, 640
- setOut() 679
- setPadding() 630
- setPriority() 760
- setter 216
- setText() 639
- setVgap() 630
- setVgrow() 630
- setWritable() 669
- short 112
- showConfirmDialog() 189
- showInputDialog() 187
- showMessageDialog() 187

shuffle()
  Collections 502
shutdown() 767, 769
Sicht
  Map<K,V> 492
  SortedSet<E> 487
Sichtbarkeitsbereich 208
  lokale Variablen 121
Sieb des Eratosthenes................................320
signal() 740, 758
signalAll() 740
Signatur 232, 389
SimpleIntegerProperty 614, 619
Simput 131, 147, 193, 395, 700
sin() 284
Single-Catch - Block 512
Single-Responsibility - Prinzip 199, 272
Singleton-Pattern.......................................197
size()...661
  Collection<E> 466
  Map<K,V> 493
Skalarprodukt 284
Skalierbarkeit 774
sleep()...726
Smalltalk 198
sort() 435
  Arrays..292
  Collections 502
sorted() 566, 571
SortedList<> 598
SortedMap<K,V>.....................................497
SortedSet<E> 486
Sortieren
  von Strings 302
Sortieren durch Auswahl...........................319
Spätes Binden..396
Speicherkonsistenzfehler 733
Speicherlöcher..................................242, 260
Spring-Framework 43
sqrt() 283
Stabilität 22
Stack.................................110, 208, 224, 724
  Überlauf 254
Stack Frame...228
stack trace...520
Stack<E>...501
StackOverflowError..........................526, 797
StackPane 637
Standard Widget Toolkit...........................590
Standardausgabe...............................102, 520
Standardausgabestrom 678
Standarddialoge...186
Standardfehlerausgabe 520
Standardfehlerausgabestrom......................678
Standardkonstruktor..................................237
Standardpaket 100, 131, 328
Stapel 501
start()...594, 726
Startfähige Klasse 90
Startklasse 27
startsWith()
  String..303
starvation..761
static 246
  bei Mitgliedsklassen 260
Statische
  Felder 246
  Initialisierer..251
  Mitgliedsklasse 260
Statische Methode.......................................54
Statische Methoden
  in Klassen...249
  in Schnittstellen 443
Steuerelemente..587
stop() 758
  JavaFX 595
Strategie-Entwurfsmuster 546
stream()
  Arrays 563
  Collection<E>.......................466, 560, 563
Stream<T> 559, 570
StreamEncoder..687
Streams 649
strictfp 164
StrictMath 164
String...298
StringBuffer 306
StringBuilder...306
String-Pool 299
StringProperty...615
StringReader 694
StringTableSize..305
StringTokenizer 323
StringWriter 684
struct 204
Struktogramm 254
Strukturiertes Programmieren...................203
subList()
  List<E> 470
subMap()
  SortedMap<K,V> 498
submit()...766
subpath()
  Path 658

subSet()
SortedSet<E> 488
Substitutionsprinzip 398
substring()
String 303
subtract() 640
sum() 564, 577
super
Basisklassenkonstruktor 385, 387
überdecktes Feld 393
überschriebene Methode 390
Superklasse 384
SuppressWarnings 417, 458
suspend() 758
switch 317
switch-Anweisung 172
switch-Ausdruck 176
SWT 590
symbolische Links 661
Symbolische Links 656
Synchronisierter Block 734
synchronized 733
synchronizedCollection()
Collections 502
synchronizedList() 473
Collections 502
synchronizedMap() 490
Collections 502
synchronizedSet() 481
Collections 502
Syntaxdiagramm 92
Syntaxfehler 59
System.err 678
System.in 528
System.out 97
Szenengraph 596
T
tailMap()
SortedMap<K,V> 497
tailSet()
SortedSet<E> 487
take() 742
takeWhile() 569
Target 459
Task<V> 779, 780
Terminale Operationen 566
test() 568
Textblöcke 308
TextField 642
TextInputControl 642
this 212, 226, 239, 244, 246
Thread 724
Thread-Gruppen 783
ThreadLocalRandom 294
Threadpool 763
Thread-Sicherheit
bedingte 775
throw 531
Throwable 526
throws 532, 533
ThumbEE 44
Timer 768
toArray() 579
Collection<E> 466
toCharArray() 303
TOCTTOU 660
toDegrees() 284
toFile() 659, 673, 682, 688, 696
ToggleGroup 646
Token 323
Tokens 697
toList() 578
toLower() 175
toLowerCase() 304
Top-Level - Klassen 257
Top-Level - Typen 377
toRadians() 284
toString()
StringBuilder 308
toUpperCase() 304
toUri()
Path 659
Transformationsklassen 652
transient 705
TreeMap<K,V> 499
TreeSet<E> 448, 486
trimToSize() 472
try with resources 539, 655
try-catch-finally 510
tryLock() 738
Typ
parametrisierter 408
Typanpassung
erweiternde 138, 148
Typformalparameter 414, 423
Typgenerizität 405
Typinferenz 107, 119, 408, 423
Typlöschung 408
Typsicherheit 106, 406
Typumwandlung
Automatische 148
Explizite 149
U
Überdecken

von statischen Methoden 391
Überdeckte Felder 393
Überladen
von Methoden 232
von Operatoren 301
Überlauf 158
Überschreiben
von Instanzmethoden 389, 533
UI-Thread 777
UML 23
Umschalter 644
Unäre Operatoren 137
Unbenanntes Moduls 373
unbind() 619
unbindBidirectional() 619
Unboxing 311
unchecked exceptions 528
Undefinierte Werte 161
Unendlich 160
Ungeprüfte Ausnahmen 528
Unicode 648
UnicodeBigUnmarked 685
Unicode-Escape-Sequenzen 128
UnicodeLittleUnmarked 686
Unicode-Zeichensatz 99
Unified Modeling Language 23
Unit Testing 200
unmodifiableCollection()
Collections 502
unmodifiableList()
Collections 502
unmodifiableMap()
Collections 502
unmodifiableSet()
Collections 502
unnamed package 328
Unterbrechungspunkt 228
Unterlauf 162
Unterpakete 329
Unterprogramme 203
upper bound
Wildcard-Typen 427
Upper bound
Typparameter 408
URI 659
US-ASCII 685
useDelimiter() 698
uses (JPMS) 352
UTF-16BE 685
UTF-16LE 686
UTF-8 685

V

valignment 631
value() 455
valueOf() 309
Double 313
String 188, 314
values()
Enumerationen 318
Map<K,V> 492
var 107, 119, 123, 237
Varargs-Parameter 223
variable arity - Methoden 223
Variablen 105
finalisierte 123
globale 111
lokale 110
Variablendeklaration 118, 165
VBox 633
VCS 77
Vector<E> 472
Verbundanweisung 121, 165
Vereinigung von zwei Mengen 479
Vererbung 200
Klassen 383
Schnittstellen 439
Vergleich 141
Vergleichsoperatoren 141
Verketten von Strings 301
Verkettete Liste 473
Verschiebungsformel 321
Version der JVM 30
Verzeichnis
anlegen 660
anlegen in Java 6 667
Inhalt auflisten ab Java 7 663
Inhalt auflisten in Java 6 669
löschen ab Java 7 665
löschen in Java 6 670
umbenennen ab Java 7 664
umbenennen in Java 6 670
vgap 630
vgrow 631, 633
View
Map<K,V> 492
SortedSet<E> 487
Virtuelle Java-Maschine 40
void 218
volatile 754
Vollständige Ordnung 483
Vorschaustatus 176

W

Wahrheitstafeln 144

wait() 735, 758
Warteschlangen 501, 742
wasAdded() 625
wasPermutated() 625
wasRemoved() 625
wasReplaced() 625
wasUpdated() 625
Wertzuweisung 120
while-Schleife 182
WhiteSpace 698
widgets 587
Wiederholungsanweisung 178
Wildcard-Datentypen 425
  Beschränkung nach oben 426
  Beschränkung nach unten 427
Windows Latin-1 686
Work Stealing 770
Worker<V> 779
Wrapper-Klassen 309
write()
  Files 711
  Writer 684
writeObject() 704, 707
Writer 684

X

XML-Verarbeitungsinstruktionen 606

Y

Year 621
yield
  switch 177
yield() 761

Z

Zahlenkreis 158
Zeichenfolgenliterale 128
Zeilennummern 69
Zeilenrestkommentar 96
Zeilenumbruch 322
Zeitscheibenverfahren 761
ZIP-Dateiformat 340
ZipInputStream 681
ZipOutputStream 672
Zufallszahlen 292
Zugriffsmodifikatoren 377, 379
  protected 386
Zugriffsrechte für Dateien 662
Zugriffsschutz 198
Zusammengesetzte Anweisung 166
Zustandsbehaftete Operationen 567
Zustandslose Operationen 566
Zuweisungsoperator 150
Zweierkomplement 158
Zweistellige Operatoren 137