Universität Trier

Bernhard Baltes-Götz & Johannes Götz

Einführung in das Programmieren mit Java 8

2016 (Rev. 160330)

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 und Johannes Götz
E-Mail: baltes@uni-trier.de

Vorwort

Dieses Manuskript entstand als Begleitlektüre zum Java-Einführungskurs, den das Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK) an der Universität Trier im Wintersemester 2015/2016 angeboten hat, ist aber auch für das Selbststudium geeignet. In hoffentlich seltenen Fällen enthält der Text noch Formulierungen, die nur für Kursteilnehmer(innen) perfekt passen.

Inhalte und Lernziele

Die von der Firma **Sun Microsystems** (mittlerweile von der Firma **Oracle** übernommen) entwickelte 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 Softwareentwicklung 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 2011). 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. Groovy, Scala, Clojure, Jython, JRuby). Sie bieten dieselbe Plattformunabhängigkeit wie Java und können teilweise alternative Programmiertechniken wie dynamische Typisierung oder funktionales 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 zeichnet sich die aktuelle Version 8 durch eine Erweiterung der Sprache um funktionale Konzepte aus.

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-Varianten für unternehmensweite oder serverorientierte Lösungen (*Java Enterprise Edition*, JEE) sowie für Kommunikationsgeräte mit beschränkter Leistung (*Java Micro Edition*, JME).

Moderne Smartphones und Tablets zählen nicht mehr zu den Geräten mit beschränkter Leistung. Sofern diese Geräte das Betriebssystem Android benutzen, kommt zur Software-Entwicklung eine angepasste Java-Version zum Einsatz (siehe Baltes-Götz 2016).

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 Softwareentwicklung vorgestellt, wobei die objektorientierte Programmierung einen großen Raum einnimmt.

Voraussetzungen bei den Leser(innen)[1]

- **Programmierkenntnisse**
  Programmierkenntnisse werden *nicht* vorausgesetzt. Leser *mit* Programmiererfahrung werden sich bei den ersten Kapiteln eventuell etwas langweilen.
- **EDV-Plattform**
  Im Manuskript wird ein PC unter Windows 7 verwendet. Weil dabei ausschließlich Java-Software zum Einsatz kommt, ist die Plattformunabhängigkeit jedoch garantiert.

Software zum Üben

Für die unverzichtbaren Übungen sollte ein Rechner zur Verfügung stehen, auf dem das Java SE Development Kit 8.0 der Firma Sun installiert ist. Als integrierte Entwicklungsumgebung zum komfortablen Erstellen von Java-Software wird Eclipse in einer Version ab 4.5 empfohlen. 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 in Abschnitt 2.1.

Manuskript und Dateien zum Kurs

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

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

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

baltes@uni-trier.de

werden dankbar entgegen genommen.

Trier, im März 2016 Bernhard Baltes-Götz & Johannes Götz

[1] Zur Vermeidung von sprachlichen Umständlichkeiten beschränkt sich das Manuskript meist auf die männliche Form.

Inhaltsverzeichnis

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 Softwareentwicklung 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 einer bestimmten Aufgabe. 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) 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 Entfernung, Fahrtzeit, Mautkosten etc.).

Wir wollen präzisere und komplettere Definitionen zum komplexen Begriff eines Computerprogramms den Lehrbüchern überlassen (siehe z.B. Goll et al. 2000) 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 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 eine Detailanalyse. Beispielsweise können wir das eben zur Illustration einer realen Aufgabenstellung verwendete, aber potentiell sehr aufwendige Speditionsverwaltungsprogramm jetzt nicht im Detail 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.

1.1.1 Objektorientierte Analyse und Modellierung

Einer objektorientierten Programmentwicklung 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**. Eine solche Klasse ist gekennzeichnet durch:

- **Eigenschaften bzw. Zustände**
  Viele 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 so genannten *Variablen*. Dies ist ein benannter Speicherplatz, der Werte eines bestimmten Typs (z.B. Zahlen, Zeichen) 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 zugeordnet (z.B. einen Bruch kürzen) oder der Klasse selbst (z.B. über die Anzahl der erzeugten Brüche informieren). Im letztlich entstehenden Programm sind die Handlungskompetenzen durch so genannte **Methoden** repräsentiert. Diese ausführbaren Programmbestandteile enthalten die oben angesprochenen Algorithmen. Die Kommunikation zwischen Klassen bzw. Objekten besteht darin, ein anderes Objekt oder eine andere 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,
- kann 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.

In unserem Bruchrechnungsbeispiel ergibt sich bei der objektorientierten Analyse, dass vorläufig nur eine Klasse zum Modellieren von Brüchen benötigt wird. Beim möglichen Ausbau des Programms zu einem Bruchrechnungstrainer kommen jedoch weitere Klassen hinzu (z.B. Aufgabe, Schüler).

Dass Zähler und Nenner die zentralen **Eigenschaften** eines Bruchs 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.

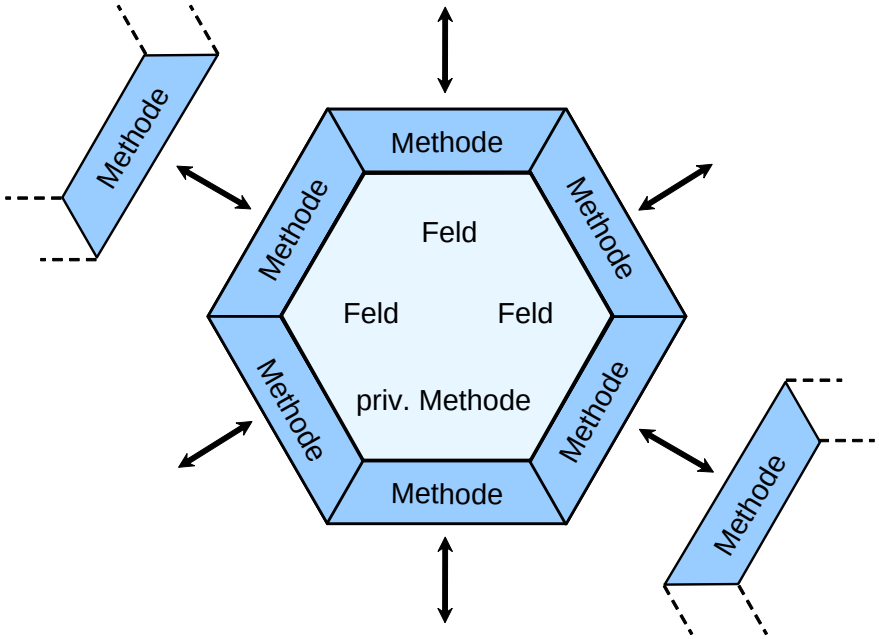
Im objektorientierten Paradigma ist jede Klasse für die Manipulation ihrer Eigenschaften selbst verantwortlich. Diese sollen **eingekapselt** und so vor dem direkten Zugriff durch fremde Klassen geschützt sein. So kann sichergestellt werden, dass nur sinnvolle Änderungen der Eigenschaften mög-

lich sind. Außerdem wird aus später zu erläuternden Gründen die Produktivität der Softwareentwicklung durch die Datenkapselung gefördert.

Demgegenüber sind die **Handlungskompetenzen** (Methoden) einer Klasse in der Regel von anderen Agenten (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.

Die folgende, an Goll et al. (2000) 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 Instanzmethode 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 geschützten Eigenschaft ist auch der direkte *Lesezugriff* ausgeschlossen, so dass im `Bruch`-Beispiel auch noch Methoden zum Ermitteln von Zähler und Nenner erforderlich sind (z.B. mit den Namen `gib-`

`Zaehler()` 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 erhebliche 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, so dass bei der Entwicklung großer Softwaresysteme zahlreiche Programmierer 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 erforderlich, weil ein direkter Zugriff auf die Eigenschaften nicht vorgesehen ist. Aus der Datenkapselung resultiert für ein betroffenes Feld neben dem Schreibschutz stets auch eine Lesesperre.
- `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 durch die Außenwelt erschlossen.

Beim Eigenschaftsbegriff ist eine (ungefährliche) Zweideutigkeit festzustellen, die je nach Anwendungsbeispiel mehr oder 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 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 `Bruch`-Klasse enthält 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!? 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 erstellen zu können.

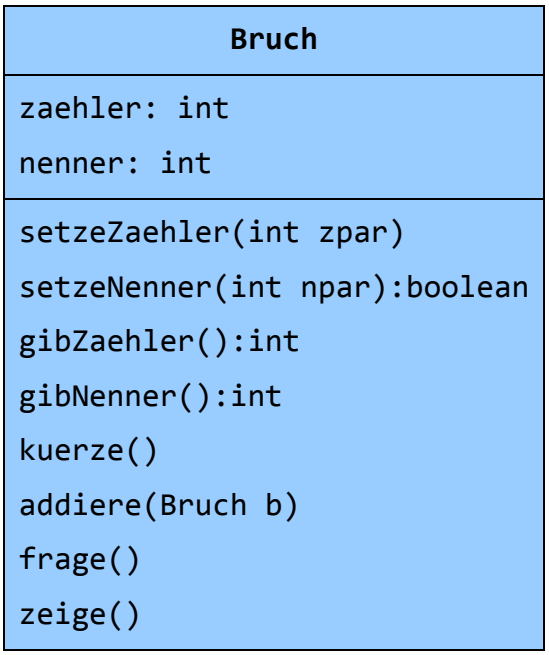
Um die durch objektorientierte Analyse gewonnene Modellierung eines Anwendungsbereichs standardisiert und übersichtlich zu beschreiben, wurde die **Unified Modeling Language** (UML) entwickelt.[1] Hier wird eine Klasse durch ein Rechteck mit drei Abschnitten dargestellt:

- 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).

[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).

- 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 letzten Satz genau gemeint ist, werden Sie bald erfahren.

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



Sind bei einer Anwendung *mehrere* Klassen beteiligt, dann sind auch die *Beziehungen* zwischen den Klassen wesentliche Bestandteile des Modells.

Nach der sorgfältigen Modellierung per UML muss übrigens die Kodierung eines Softwaresystems nicht am Punkt Null beginnen, weil UML-Entwicklerwerkzeuge üblicherweise Teile des Quellcodes automatisch aus dem Modell erzeugen können.[1]

Das relativ einfache Einstiegsbeispiel sollte Sie nicht dazu verleiten, den Begriff *Objekt* auf *Gegenstände* zu beschränken. Auch *Ereignisse* wie z.B. die Fehler eines Schülers in einem entsprechend ausgebauten Bruchrechnungsprogramm kommen als *Objekte* in Frage.

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 so genannte **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. Dem-

[1] Für die im Kurs bevorzugte Java-Entwicklungsumgebung Eclipse (siehe Abschnitt 1) sind etliche, teilweise kostenlose UML-Werkzeuge verfügbar (siehe z.B. http://www.eclipse.org/modeling/mdt/).

gegenüber 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.

```
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 ggt = 0;
         int az = Math.abs(zaehler);
         int an = Math.abs(nenner);
         do {
            if (az == an)
               ggt = az;
            else
               if (az > an)
                  az = az - an;
               else
                  an = an - az;
         } while (ggt == 0);

         zaehler /= ggt;
         nenner  /= ggt;
      } else
         nenner = 1;
   }

   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("Zaehler: ");
          setzeZaehler(Simput.gint());
       } while (Simput.checkError());
       do {
          System.out.print("Nenner : ");
          // Bei irregulärer Eingabe liefert gint() eine 0 und setzt einen Fehlerindikator.
          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.println("   "+zaehler+"\n -----\n   "+nenner);
    }
}
```

Allerdings ist das Programm schon zu umfangreich für die bald anstehenden ersten Gehversuche mit der Softwareentwicklung 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 Softwareentwicklung 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 Voraussetzungen, 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.

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. Beim 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 v und $(u - v)$, dann gibt es natürliche Zahlen t_v und t_d mit

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

Folglich ist t auch ein Teiler von u:

$$v + (u - v) = u = (t_v + t_d) \cdot t$$

Weil die Paare (u, v) und (u - v, v) dieselben Mengen gemeinsamer Teiler besitzen, sind auch die größten gemeinsamen Teiler identisch. Weil die Zahl Eins als trivialer Teiler zugelassen ist, existiert zu zwei natürlichen Zahlen immer ein größter gemeinsamer Teiler, der eventuell gleich Eins ist.

Dieses Ergebnis wird in `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 verkleinerten Problem startet das Verfahren neu.

Man erhält auf jeden Fall in endlich vielen Schritten zwei identische Zahlen und damit den ggT.

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

1.1.4 Startklasse und main() - Methode

Bislang wurde im Anwendungsbeispiel 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 ihren klassenbezogenen Handlungsrepertoire besitzt. Beim Start eines Programms wird die Startklasse ausfindig gemacht und aufgefordert, ihre Methode **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 Startmethode unterzubringen.

Obwohl prinzipiell möglich, erscheint 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 hat. 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 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 so genannte **Konstruktor** (siehe unten):

Quellcode	Ein- und Ausgabe
`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();` `  }` `}`	`1. Bruch` `Zaehler: 20` `Nenner : 84` `  5` `  -----` `  21` `2. Bruch` `Zaehler: 12` `Nenner : 36` `  1` `  -----` `  3` `Summe` `  4` `  -----` `  7`

Wir haben zur Lösung der Aufgabe, ein Programm für die Addition von 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 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 einen 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**, so dass im Beispielsprojekt die Quellcodedateien **Bruch.java** und **Bruchaddition.java** entstehen. Weil die Klasse `Bruch` mit dem Zugriffsmodifikator **public** definiert wurde, *muss* ihr Quellcode unbedingt in einer Datei mit dem Namen **Bruch.java** gespeichert werden (siehe unten). Es ist erlaubt, aber nicht empfehlenswert, 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 in Abschnitt 1.

1.1.5 Zusammenfassung zu Abschnitt 1.1

Im Abschnitt 1.1 sollten Sie einen ersten Eindruck von der Softwareentwicklung mit Java gewinnen. Alle dabei erwähnten Konzepte der objektorientierter 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 elementarer Sprachelemente werden oft mit einer einzigen Klasse auskommen. Praxisgerechte Programme bestehen in der Regel aus zahlreichen 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 wird 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 aber 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 in 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 JRE installieren

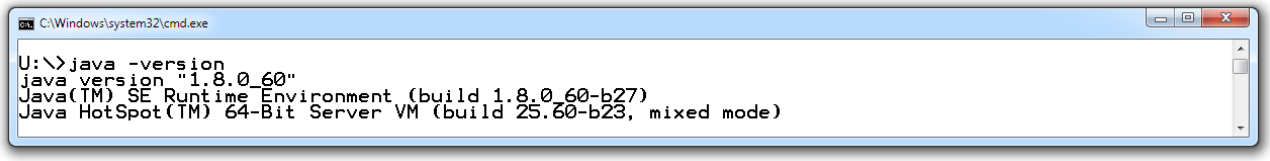
Um das Programm auf einem Rechner ausführen zu können, muss dort eine **Java Runtime Environment** (JRE) mit hinreichend aktueller Version installiert sein. Mit den technischen Grundlagen und Aufgaben dieser Ausführungsumgebung für Java-Programme werden wir uns in Abschnitt 1.3.2 beschäftigen. Bei der in Abschnitt 2.1 beschriebenen und für Kursteilnehmer empfohlenen JDK-Installation (*Java Development Kit*) landet optional auch eine JRE auf der Festplatte. Es ist aber problemlos möglich, bei Bedarf die JRE jetzt schon zu installieren und bei der späteren JDK-Installation auf die entsprechende Option zu verzichten. Für die Kunden Ihrer Java-Programme ist

die Installation bzw. Aktualisierung der JRE auf jeden Fall eine sinnvolle Option, so dass wir uns jetzt damit beschäftigen wollen.

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

```
>java -version
```

ab. Im folgenden Beispiel geschieht dies auf einem Rechner mit der 64-Bit Version von Windows 7:



Die JRE ist in einer 32-Bit und einer 64-Bit-Version verfügbar, und auf einem Rechner mit 64-Bit - Betriebssystem ist es eventuell sinnvoll, *beide* Varianten zu installieren:

- Wer Java-Applets (via Internet bezogene, im Browser-Fenster ablaufende Programme) nutzen möchte und einen Browser mit 32-Bit-Technik benutzt, was heute (2015) noch üblich ist, benötigt die 32-Bit - Variante der JRE.
- Damit ein Java-Programm bei der Speichernutzung die 32-Bit - Grenze (theoretisch 4 GB, real nur 1,5 GB) überwinden kann, muss die 64-Bit - Variante der JRE verwendet werden.

Um die Version der 32-Bit-Ausführung zu ermitteln, können sie z.B. in einem Konsolenfenster den zugehörigen Installationspfad ansteuern und dann die Versionsangabe anfordern, z.B.

Die Firma Oracle stellt die JRE für viele Desktop-Betriebssysteme (Linux, MacOS, Solaris, Windows) kostenlos zur Verfügung, wobei die Installationsprogramme als Offline- und als Online-Variante angeboten werden:

- Das *IFTW*-Installationsprogramm (*Install from the Web*) lädt die erforderlichen Dateien während der Installation aus dem Internet.
- Das Offline-Installationsprogramm beinhaltet alle benötigten Dateien und ist in der Regel zu bevorzugen. Es ist über die folgende Webseite zu beziehen:

  http://www.oracle.com/technetwork/java/javase/downloads/index.html

Wer eine Windows-Version mit 64-Bit - Architektur benutzt und später (vgl. Abschnitt 2.1) bei der empfehlenswerten JDK-Installation die 64-Bit - Variante wählt, wird dabei Gelegenheit haben, eine 64-Bit - JRE ohne großen Zusatzaufwand zu installieren, sodass eine separate JRE-Installation vor allem bei der 32-Bit - Variante lohnt.

Im ersten Dialog des Offline-Installationsprogramms kann man per Kontrollkästchen den Wunsch anmelden, auf den **Zielordner** Einfluss zu nehmen:

Den Zielordner zu ändern, verursacht einigen Aufwand, der bei jedem Update wiederholt werden muss. Daher sollte man besser den voreingestellten Ordner akzeptieren.[1] Die 32-Bit - Variante der Version 8u60 (Update 60 zur Version 8) landet per Voreinstellung im Ordner:[2]

C:\Program Files (x86)\Java\jre1.8.0_60

Wenn das Installationsprogramm veraltete JRE-Versionen ermittelt und deren Deinstallation anbietet, sollte man zustimmen, z.B.:

Während der Installation meldet Oracle stolz eine stattliche Installationsbasis für die Java-Technologie (3 Milliarden Geräte):

---

[1] Durch Referenzierungstricks der Firma Oracle ist unter Windows sichergestellt, dass trotz versionsspezifischer Ordnernamen die aktuelle Version der Java-Runtime gefunden wird:

- In der Umgebungsvariablen PATH befindet sich der Ordner **C:\ProgramData\Oracle\Java\javapath**.
- In diesem Ordner finden sich symbolische Links (z.B. **java.exe**), welche auf die realen Dateien zeigen und bei jedem Update aktualisiert werden.

Weil die symbolischen Links entweder auf die 32-Bit -oder auf die 64-Bit - Dateien zeigen, sollte die 64-Bit - JRE zuletzt installiert werden.

[2] Die JRE - Update-Frequenz ist bei weitem nicht so hoch, wie die Nummer 60 vermuten lässt. Oracle lässt bei der Nummerierung große Lücken.

1.2.2 Updates für die JRE

Wie viele andere Softwaresysteme mit Internetkontakt (z.B. Betriebssysteme, WWW-Browser) benötigt auch die JRE häufig Sicherheits-Updates. Unter Windows kann die JRE automatisch aktualisiert werden, was seit dem Update 20 zur Version 8 endlich auch für die 64-Bit - Variante der Laufzeitumgebung gilt und ggf. *beide* auf einem Rechner installierte Varianten (32 und 64 Bit) einbezieht. Zur Verwaltung der Updates dient ein zur JRE-Installation gehöriges Konfigurationsprogramm, das folgendermaßen zu starten ist:

- Öffnen Sie das Fenster der **Systemsteuerung**. Die passende Option findet sich bei Windows 7 im Startmenü und bei Windows 10 im Kontextmenü zum Windows-Symbol am linken Rand der Taskleiste.
- Wählen Sie als **Anzeige** über das Bedienelement oben rechts die Option **Kleine Symbole**.
- Starten Sie das Systemsteuerungselement **Java**.[1]

Wenn unter Windows JRE-Einstellungsänderungen permanent gespeichert werden sollen, was bei einer 64-Bit-Version von Windows im folgenden Registry-Key

HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\JavaSoft\Java Update\Policy\jucheck

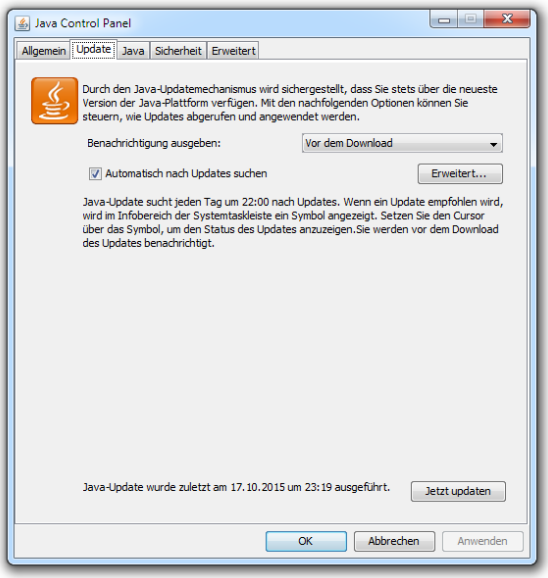
geschieht, dann muss das Java-Konfigurationsprogramm mit Administratorrechten gestartet werden, z.B. so:

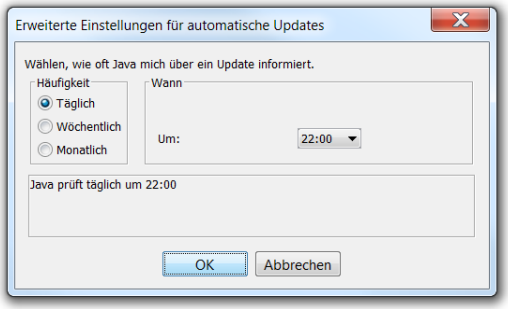
- Explorer-Fenster öffnen und auf den **bin**-Ordner der 32-Bit-JRE-Installation positionieren, unter Windows 64 mit installierter JRE Java 8 Update 77 z.B. auf:
  C:\Program Files (x86)\Java\jre1.8.0_77\bin
- Rechter Mausklick auf die Datei **javacpl.exe** und aus dem Kontextmenü wählen: **Als Administrator ausführen**.

Die Registerkarte **Update** informiert über den aktuellen Update-Plan und erlaubt nach einem Klick auf den Schalter **Erweitert** eine Anpassung. Aus Sicherheitsgründen ist eine *tägliche* Update-Prüfung zu empfehlen:

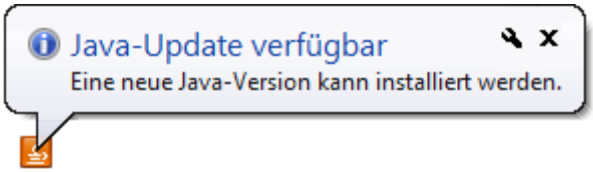
---

[1] Ist unter Windows 64 ausschließlich die 32-Bit-JRE installiert, heißt das Systemsteuerungselement **Java (32 Bit)**.





Auf ein anstehendes Update wird unter Windows über ein Java-Symbol mit temporärer Sprechblase im Infobereich der Taskleiste aufmerksam gemacht:



Per Mausklick auf dieses Symbol wird die Aktualisierung gestartet, z.B.:

Ist in dieser Situation ein Java-Programm (z.B. Eclipse) aktiv, kann die neue Version installiert werden, doch verbleiben unerwünschte Reste der alten Version auf dem Rechner. Wenn das automatische Update scheitert, empfiehlt sich die manuelle Installation der aktuellen Laufzeitumgebung auf dem in Abschnitt 1.2.1 beschriebenen Weg.

1.2.3 Konsolenprogramme ausführen

Nach der Beschäftigung mit der JRE machen wir uns endlich daran, das Bruchadditionsprogramm zu starten. Kopieren Sie von der oben angegebenen Quelle 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 das Manuskript entworfene Klasse `Simput` in eigenen Programmen einsetzen sollen, wird sie näher vorgestellt. In Abschnitt 2.2.4 lernen Sie eine Möglichkeit kennen, die in mehreren Projekten benötigten **class**-Dateien zentral abzulegen und durch eine passende Definition der Umgebungsvariablen CLASSPATH allgemein verfügbar zu machen.

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

- Öffnen Sie ein Konsolenfenster, z.B. mit

  Start > Alle Programme > Zubehör > Eingabeaufforderung

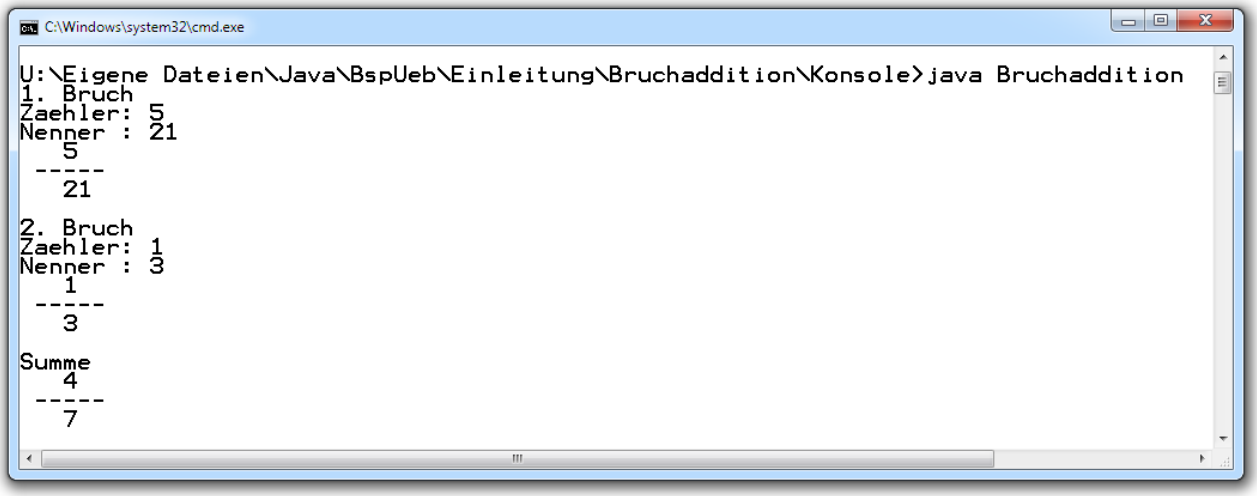
- Wechseln Sie zum Ordner mit den **class**-Dateien, z.B.:

```
>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:

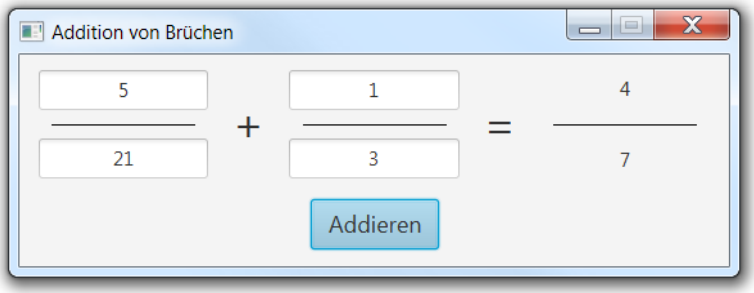
```
>java Bruchaddition
```

Ab jetzt sind Bruchadditionen kein Problem mehr:



1.2.4 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 Sprachelemente kennen gelernt haben, werden wir uns natürlich auch mit der Programmierung von grafischen Bedienoberflächen beschäftigen. In folgendem Programm zur Addition von Brüchen wird die oben definierte Klasse `Bruch` verwendet, wobei an Stelle ihrer Methoden `frage()` und `zeige()` jedoch grafikorientierte Techniken zum Einsatz kommen:



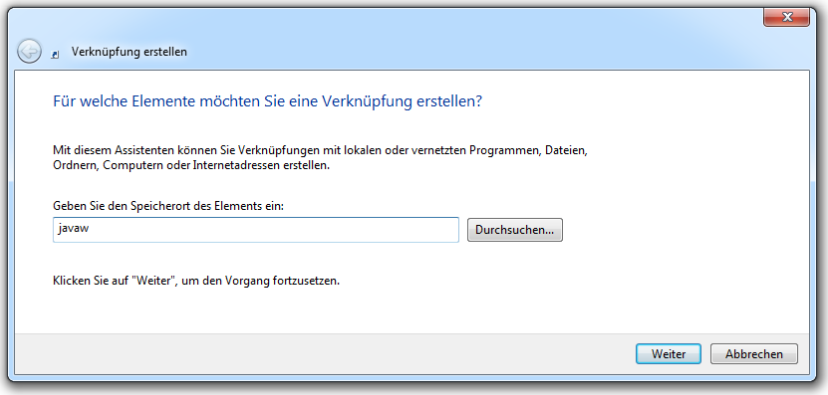
Mit dem Quellcode zur Gestaltung der grafischen Bedienoberfläche könnten Sie im Moment noch nicht allzu viel anfangen. Nach der Lektüre des Manuskripts werden Sie derartige Anwendungen aber mit Leichtigkeit erstellen, zumal die Erstellung grafischer Bedienoberflächen durch die neue GUI-Technologie JavaFX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

Zum Ausprobieren des Programms startet man mit Hilfe der Java Runtime Environment (JRE, vgl. Abschnitt 1.2) aus dem Ordner

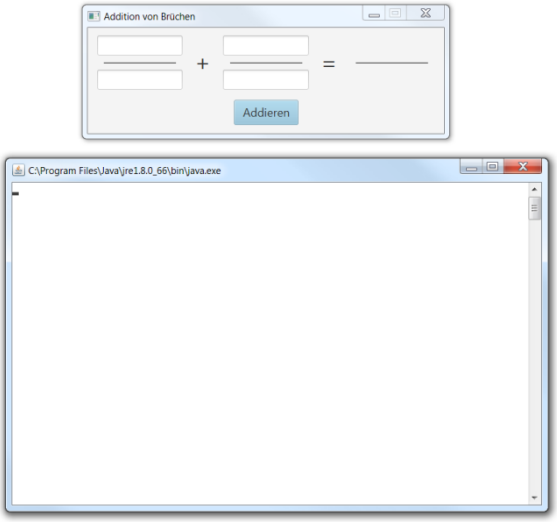
...\BspUeb\Einleitung\Bruchaddition\JavaFX\bin

die Klasse `Bruchaddition`:

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

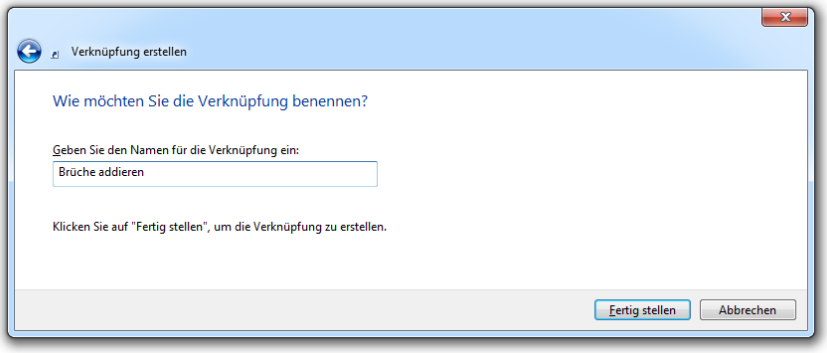


Weil das Programm keine Konsole benötigt, sondern ein Fenster als Bedienoberfläche anbietet, verwendet man bei der Link-Definition als JRE-Startprogramm die Variante **javaw.exe** (mit einem *w* am Ende des Namensstamms). Bei Verwendung von **java.exe** als JRE-Startprogramm würde zusätzlich zum Bruchadditionsprogramm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

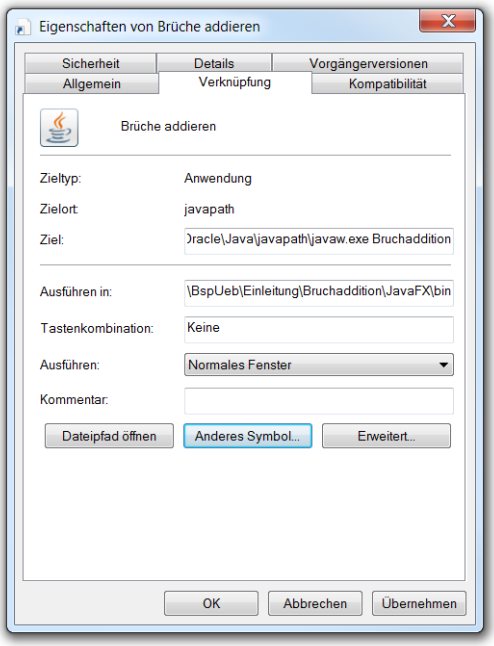


Während das Konsolenfenster beim normalen Programmablauf leer bleibt, erscheinen dort bei einen Laufzeitfehler hilfreiche diagnostische Ausgaben. Daher ist ein Programmstart *mit* Konsolenfenster (per **java.exe**) bei der Fehlersuche durchaus sinnvoll.

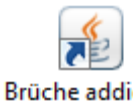
Im nächsten Dialog des Assistenten für neue Verknüpfungen trägt man den gewünschten Namen der Link-Datei ein:



Im Eigenschaftsdialog zur Verknüpfungsdatei ergänzt man in Feld **Ziel** hinter **javaw.exe** den Namen der Startklasse (samt Paketpräfix `application`) und trägt im Feld **Ausführen in** den Ordner ein, in dem der Paketunterordner **application** enthalten ist. Außerdem kann über den Schalter **Anderes Symbol** noch das Java-Icon aus der Datei **javaw.exe** übernommen werden, z.B.:



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



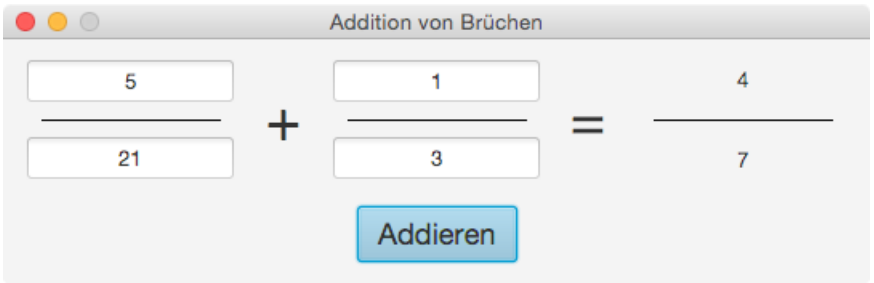
Professionelle Java-Programme werden als Java-Archivdatei (mit der Namenserweiterung **.jar**, siehe Abschnitt 6.4) ausgeliefert und sind unter Windows nach einer korrekten JRE-Installation über einen Doppelklick auf diese Datei zu starten.

1.2.5 Ausführung auf einer beliebigen unterstützten Plattform

Dank der Portabilität (Binärkompatibilität) von Java können wir z.B. das im letzten Abschnitt demonstrierte, unter Windows entwickelte Programm auch unter anderen Betriebssystemen ausführen, z.B. auch unter Mac OS X. Es genügt, die Bytecode-Dateien auf den Mac zu kopieren, wobei dort natürlich eine Java-Laufzeitumgebung installiert sein muss. Zum Starten des Programms aus einem Terminalfenster taugt dasselbe Kommando wie unter Windows:[1]

```
>java Bruchaddition
```

Es erscheint das vertraute Programm mit dem bei Mac OS X üblichen Fensterdekor:



[1] Ist unter Mac OS X 10.10 das JDK 8 installiert, klappt der Start von Java-Programmen per Terminal-Kommando spontan.

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 **Laufzeitumgebung** (JVM, JRE) 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. 2015). 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 aufgrund von unterschiedlichen Quellen und Gewichtungen deutlich unterscheiden:

- TIOBE Programming Community Index (Oktober 2015, Java-Rangplatz: 1)
  http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
- RedMonk Programming Language Rankings (Juni 2015, Java-Rangplatz: 2)
  https://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/
- Most Popular Programming Languages of 2015 (Februar 2015, Java-Rangplatz: 2)
  http://blog.codeeval.com/codeevalblog/2015
- GULP IT-Projektmarktindex (September 2015, Java-Rangplatz: 1)
  https://www.gulp.de/projektmarktindex
- Language Trends on GitHub (August 2015, Java-Rangplatz: 2)
  https://github.com/blog/2047-language-trends-on-GitHub

Mit einem durchschnittlichen Rangplatz von 1,6 (die Statistik-Puristen mögen mir diese Auswertungstechnik verzeihen) steht Java deutlich vor der härtesten Konkurrenz:

Sprache	Mittlerer Rang
Java	1,6
JavaScript	4
C++	4,2
C#, Python	5,2
C, PHP	6

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, so dass sich Umsteiger von diesen sowohl im Windows- als auch im Linux/UNIX - Bereich weit verbreiteten Sprachen schnell in Java einarbeiten können. Wesentliche Ziele bei der Weiterentwicklung waren Einfachheit, 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

In Abschnitt 1.1 haben Sie Java als eine Programmiersprache kennen gelernt, die Ausdrucksmittel zur Modellierung von Anwendungsbereichen 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 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 (2015) die CPU eines handelsüblichen Arbeitsplatzrechners bis zu 300 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:

Kompilieren: 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 Maschi-

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

nencode durchlaufen. Sein Name geht darauf zurück, dass die Instruktionen der virtuellen Maschine jeweils genau ein Byte (= 8 Bit) lang sind. Weil Bytecode kompakter ist als Maschinencode, eignet er sich gut für die Übertragung via Internet.

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. .[1] 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**) der Firma Oracle (siehe Abschnitt 1) enthaltenen Compiler **javac.exe** setzen auch manche Java-Entwicklungsumgebungen im Hintergrund ein (z.B. der JCreator). Demgegenüber verwendet die im Manuskript bevorzugte Entwicklungsumgebung Eclipse einen eigenen Compiler, der inkrementell arbeitet und schon beim Editieren eines Programms tätig wird.

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 einer Bauplans 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 abhängige JVM, um den Java-Bytecode auszuführen. Diese Software wird meist in der Programmiersprache C++ realisiert.

Für viele Desktop-Betriebssysteme (Linux, MacOS, Solaris, Windows) liefert die Firma Oracle kostenlos die zur Ausführung von Java-Programmen erforderliche **Java Runtime Environment** (**JRE**). Deren Beschaffung und Verwendung wurde schon in Abschnitt 1.2 behandelt. Die wichtigsten Komponenten der JRE sind:

- JVM
  Neben der Bytecode-Übersetzung erledigt die JVM bei der Ausführung eines Java-Programms noch weitere Aufgaben, mit denen wir uns später noch im Detail beschäftigen werden, z.B.
  - Speicherverwaltung mit der automatischen Entfernung überflüssig gewordener Objekte (Garbage Collection)
  - Synchronisation von Programmen mit Multi-Thread - Technik

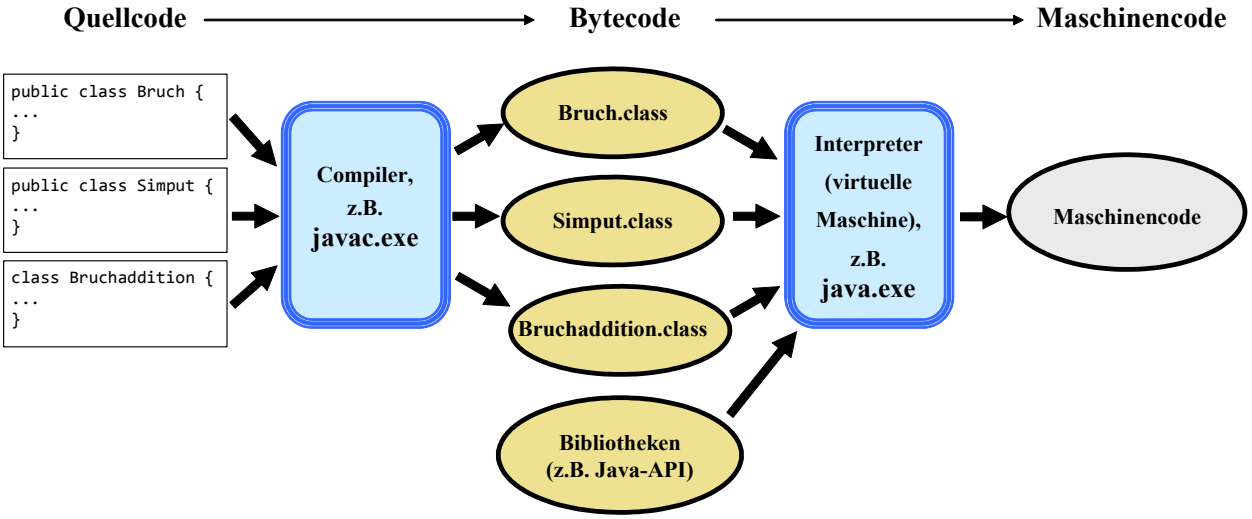
[1] Siehe: http://en.wikipedia.org/wiki/Jazelle

- Klassenlader
  Er lädt die von einem Programm benötigten Klasen in den Speicher und nimmt dabei auch eine Bytecode-Verifikation vor, um potentiell gefährliche Aktionen zu verhindern.
- Java-Standardbibliothek mit Klassen für alle Routineaufgaben (siehe Abschnitt 1.3.3)

Diese JRE oder eine äquivalente Software muss auf einem Rechner installiert werden, damit dort Java-Programme ablaufen können. Wie Sie bereits aus Abschnitt 1.1 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-**i**n-**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 Hardware 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 JRE) 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 kennen lernen, und im Kapitel über Pakete werden die wichtigsten API-Bestandteile grob skizziert. Eine vollständige Behandlung ist wegen des enormen Umfangs unmöglich und auch nicht erforderlich.

Wir halten fest, dass die Java-Technologie einerseits auf einer Programmiersprache mit einer bestimmten Syntax und Semantik 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 bereits formulierbaren Konstrukte in der Standardbibliothek unterzubringen. Es resultierte eine sehr kompakte Sprache (siehe Gosling et al. 2015), 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, so dass hier erhebliche Änderungen stattfinden. Einige Klassen sind mittlerweile schon als *deprecated* (überholt, zurückgestuft, nicht mehr zu benutzen) eingestuft worden. Gelegentlich stehen für eine Aufgabe verschiedene Lösungen aus unterschiedlichen Entwicklungsstadien zur Verfügung (z.B. Multithreading-Lösungen für die nebenläufige Programmausführung).

Mit der 2004 erschienenen Version 1.5 hat auch die Programmiersprache Java substantielle Veränderungen erfahren (z.B. generische Typen, Auto-Boxing), so dass sich die Firma Sun (mittlerweile in der Firma Oracle aufgegangen) entschied, die an vielen signifikanten Stellen (z.B. im Namen der Datei mit dem JDK) präsente Versionsnummer 1.5 durch die „fortschrittliche“ Nummer 5 zu ergänzen. Auch die aktuelle Version 8 alias 1.8.0 der Java Standard Edition (JSE) hat mit der funktionalen Programmierung (den Lambda-Ausdrücken) eine wesentliche Erweiterung der Programmiersprache Java gebracht.

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

In Kurs bzw. Manuskript steht zunächst die Programmiersprache Java im Vordergrund. Mit wachsender Kapitelnummer geht es aber vor allem darum, wichtige Pakete der Standardbibliothek mit Lösungen für Routineaufgaben kennen zu lernen (z.B. GUI-Programmierung, Ein-/Ausgabe, Multithreading, Netzwerkprogrammierung, Datenbankzugriff).

1.3.4 Java-Editionen für verschiede Einsatzszenarien

Weil die Java-Plattform so mächtig und vielgestaltig geworden ist, hat die Firma Oracle 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 in 2015: JSE 8 und JEE 7).
- **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 neu ausgerichtet worden für den Einsatz bei eingebetteten Lösungen (Stichwort: Internet der Dinge).[1]

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

Wir werden uns im Manuskript weder mit der JEE noch mit der JME beschäftigen, doch sind erworbene Java-Programmierkenntnisse natürlich hier uneingeschränkt verwendbar, und elementare Klassen der JSE-Standardbibliothek sind auch für die anderen Editionen relevant.

Weil sich die Standardklassenbibliotheken der Editionen stark unterschieden, muss man z.B. vom *Java SE - API* oder 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.

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, setzt der Konkurrent Google in seinem Smartphone - und Tablet - Betriebssystem **Android** Java als Standardsprache zur Anwendungsentwicklung ein. Man verwendet jedoch eine modifizierte Standardbibliothek, sodass man das Android-Java nicht in eine von den obigen Editionen einordnen kann. Außerdem 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 als Standard für die Entwicklung von Android-Apps gewählt worden ist. Mittlerweile trägt Android erheblich zur Attraktivität von Java bei, denn Android hat auf dem Markt für Smartphone-Betriebssysteme einen Marktanteil von ca. 80% 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).

1.3.5 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.

1.3.5.1 Objektorientierung

Java wurde als objektorientierte Sprache konzipiert und erlaubt im Unterschied zu hybriden Sprachen wie C++ außerhalb von Klassendefinitionen keine Anweisungen. Der objektorientierten Programmierung geht eine objektorientierte *Analyse* voraus, die alle bei einer Problemstellung involvierten Objekte und ihre Beziehungen identifizieren soll. Unter einem Objekt kann man sich grob einen *Akteur* mit *Eigenschaften* (auf internen, meist vor direkten Zugriffen geschützten Feldern basierend) und *Handlungskompetenzen* (Methoden) vorstellen. Auf dem Weg der Abstraktion fasst

---

[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: http://www.idc.com/getdoc.jsp?containerId=prUS25450615

[3] Quelle: http://www.idc.com/getdoc.jsp?containerId=prUS25480015

man identische oder zumindest sehr ähnliche Objekte zu Klassen zusammen. Java ist sehr gut dazu geeignet, das Ergebnis einer objektorientierten Analyse in ein Programm umzusetzen. Dazu definiert man die beteiligten Klassen und erzeugt aus diesen Bauplänen die benötigten Objekte. Dabei können sich verschiedene Objekte derselben Klasse durch unterschiedliche Ausprägungen der gemeinsamen Eigenschaften durchaus unterscheiden.

Im Programmablauf interagieren Objekte durch den gegenseitigen Aufruf von Methoden miteinander, wobei man im objektorientierten Paradigma einen Methodenaufruf als das Zustellen einer Nachricht auffasst. Ist bei dieser freundlichen und kompetenten Kommunikation eine Rolle nur *einfach* zu besetzen, kann eine passend definierte Klasse den Job erledigen, und es wird kein Objekt dieses Typs kreiert. Bei den meisten Programmen für Arbeitsplatz-Computer darf auch der Anwender mitmischen.

In unserem Einleitungsbeispiel wurde einiger Aufwand in Kauf genommen, um einen realistischen Eindruck von objektorientierter Programmierung (OOP) zu vermitteln. 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. Im Abschnitt 1 werden auch wir solche pseudo-objektorientierten (POO-) Programme benutzen, um elementare Sprachelemente in möglichst einfacher Umgebung kennen zu lernen. Aus den letzten Ausführungen ergibt sich u.a., dass Java zwar eine objektorientierte Programmierweise nahe legen und unterstützen, aber nicht erzwingen kann.

Nachdem das objektorientierte Paradigma die Softwareentwicklung über Jahrzehnte dominiert hat, gewinnt das ältere, aber lange Zeit auf akademische Diskurse konzentrierte bzw. beschränkte funktionale Paradigma in den letzten Jahren an Bedeutung. Ein wesentlicher Grund ist seine 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).

1.3.5.2 Portabilität

Die in Abschnitt 1.3.2 beschriebene Übersetzungsprozedur führt zusammen mit der Tatsache, dass sich Bytecode-Interpreter für aktuelle EDV-Plattformen relativ leicht implementieren lassen, zur guten Portabilität von Java. Man mag einwenden, dass sich der Quellcode vieler Programmiersprachen (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 JRE verfügbare Standardbibliothek mit ihren insgesamt ca. 4000 Klassen weitreichende Fähigkeiten für die Gestaltung grafischer Bedienoberflächen, für Datenbank- und Netzwerkzugriffe usw., so dass sich plattformunabhängige Anwendungen mit modernem Funktionsumfang und Design realisieren lassen.

---

[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.

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.

Wie unsere Entwicklungsumgebung Eclipse zeigt, werden gelegentlich bei Java-basierten Software-Projekten Plattform-spezifische Bestandteile in Kauf genommen, um z.B. eine 100-rozentige GUI-Konformität mit dem lokalen Betriebssystem zu erreichen. In diesem Fall ist die Binärportabilität eingeschränkt.

1.3.5.3 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 Runtime Environment speziell im Jahr 2013 immer wieder 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 tot gesagten) Java-*Applets* sind strikt von lokal installierten Java-*Anwendungen* für Desktop-Rechner zu unterscheiden.[1] Noch weniger als Java-Anwendungen für Desktop-Rechner waren die außerordentlich wichtigen Server-Anwendungen der Java Enterprise Edition sowie die Java-Apps für Android-Geräte von der Sicherheitsmisere betroffen.

Häufig waren überzogene Empfehlungen zu lesen, z.B.:[2]

> Wegen einer aktuellen Sicherheitslücke sollten Sie Java derzeit nicht einsetzen.

Sachkundiger hat sich das Bundesamt für Sicherheit in der Informationstechnik geäußert:[3]

> Aus diesem Grund empfiehlt das BSI, das Ausführen von Java-Anwendungen im Browser zu deaktivieren.

Von dieser Empfehlung ist nur ein irrelevanter Bruchteil der Java-Software betroffen. Mittlerweile hat bei Java-Applets das Risiko ein Internet-übliches Niveau erreicht, weil Browser und auch die JRE bei Applets zu Recht sehr vorsichtig geworden sind, so dass z.B. der Firefox-Browser per Voreinstellung ein Applet nur dann ausführt, ...

- wenn der Anwender explizit zustimmt (*click to play*),
- und der Hersteller das Applet signiert hat.

Grundsätzlich sind von Sicherheitsproblemen auch Java-Desktopanwendungen betroffen, so dass auf jeden Fall auf eine aktuelle JRE- bzw. JDK-Installation zu achten ist (siehe Abschnitt 1.2.2). Ebenso müssen das Betriebssystem und der Virenschutz stets aktuell gehalten werden. Außerdem sollte ausschließlich aus sicheren Quellen stammende Software verwendet werden.

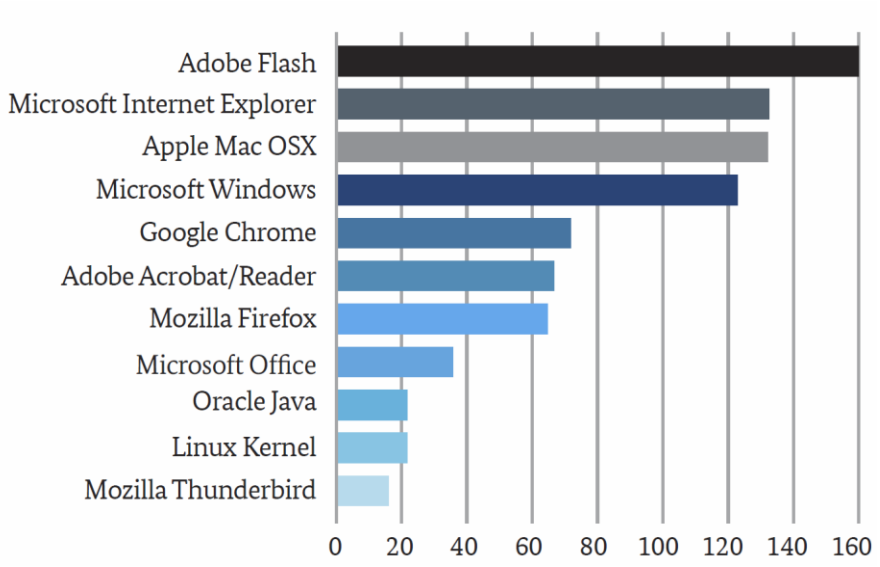
---

[1] Im aktuellen Kurs bzw. Manuskript wird den Applets im Unterschied zu früheren Ausgaben kein eigenes Kapitel gewidmet.

[2] Quelle (abgerufen am 27.10.2014): http://www.heise.de/security/dienste/Java-403125.html

[3] Quelle (abgerufen am 27.10.2014): https://www.bsi-fuer-buerger.de/BSIFB/DE/SicherheitImNetz/WegInsInternet/DerBrowser/Sicherheitsmassnahmen/Java/Java_Sicherheistempfehlungen/java_sicherheitsempfehlungen.html

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 2015 zur IT-Sicherheit in Deutschland bei Java relativ wenige kritische Schwachstellen fest:[1]



Am 20.11.2015 zeigt die CERT-Sicherheitsampel für Java ein günstiges Bild, das als typisch für die Jahre 2014/15 gelten kann:[2]

Oracle					
Produktname	geschlossene Schwachstellen		offene Schwachstellen		BSI Bewertung
	insgesamt	davon kritisch	insgesamt	davon kritisch	
Java SE Development Kit (JDK)	76	32	0	0	
Java Runtime Environment (JRE)	75	31	0	0	
Gesamtbewertung offener Schwachstellen			0	0	

1.3.5.4 Robustheit

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

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

- Pointer-Arithmetik
- Benutzerdefiniertes Überladen von Operatoren
- Mehrfachvererbung

Weil sich bei Java-Software ein automatischer Garbage Collector um obsolet gewordene Objekte kümmert, bleibt den Programmierern viel Aufwand und eine gravierende Fehlerquelle erspart (siehe

---

[1] Quelle (abgerufen am 20.11.2015): https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Lageberichte/Lagebericht2015.pdf

[2] Quelle (abgerufen am 28.10.2015): https://www.cert-bund.de/schwachstellenampel

den nächsten Abschnitt über *Einfachheit*). Außerdem werden die Programmierer zu einer systematischen Behandlung der bei einem Methodenaufruf potentiell zu erwartenden Ausnahmefehler gezwungen. Von den sonstigen Maßnahmen zur Förderung der Stabilität ist noch die generell aktive Indexgrenzenüberwachung bei Arrays (siehe unten) zu erwähnen.

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

1.3.5.5 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 so genannte **Garbage Collector** (*Müllsammler*) implementiert, der den Speicher nicht mehr benötigter Objekte automatisch frei gibt. 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.

Längst gibt es etliche Java-Entwicklungsumgebungen, die bei vielen Routineaufgaben (z.B. Gestaltung von Bedienoberflächen, Datenbankzugriffe, Web-Anwendungen) das Erstellen des Quellcodes erleichtern (z.B. *Eclipse*, *IntelliJ IDEA*, *NetBeans*) und meist kostenlos verfügbar sind.

1.3.5.6 Multithreaded-Architektur

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. Weil mittlerweile Mehrkern- bzw. Mehrprozessor-Systeme üblich sind, wird für Programmierer die Beherrschung der Multithread-Technik immer wichtiger.

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

---

[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.

1.3.5.7 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*), **SOAP** (*Simple Object Access Protocol*) oder **RMI** (*Remote Method Invocation*) laufen.

1.3.5.8 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 Performanz-kritische Anwendungen geht (z.B. Spiele). Mit unserer Entwicklungsumgebung Eclipse werden Sie eine (fast) komplett in Java erstellte, recht komplexe und dabei flott agierende Anwendung kennen lernen.

1.3.5.9 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. Hardwarezugriffs 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 zu Kapitel 1

1) Warum steigt die Produktivität der Softwareentwicklung durch objektorientiertes Programmieren?

2) Welche von den 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, vom 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)** der Firma Oracle. 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 **Eclipse**, die auf vielfältige Weise die Programmentwicklung unterstützt. Eclipse bietet u.a.:

- einen Editor mit ...
  - farblicher Unterscheidung verschiedener Syntaxbestandteile
  - Syntaxvervollständigung
  - Unterschlängeln von Fehlern
  - usw.
- einen inkrementellen Compiler, der Syntaxfehler schon während der Eingabe erkennt
- 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)

Eclipse hat unter den zahlreich vorhandenen Java-Entwicklungsumgebungen den größten Verbreitungsgrad gefunden, obwohl es z.B. mit **IntelliJ IDEA**[1] und **NetBeans**[2] ebenfalls kostenlose und leistungsfähige Alternativen gibt. Eine Umfrage bei 2164 Java-Entwicklern (*Java Tools and Technologies Landscape for 2014*) ergab folgende Marktanteile der Entwicklungsumgebungen:[3]

Eclipse:	48%
IntelliJ IDEA Ultimate	26%
IntelliJ IDEA Community	7%
NetBeans	10%

Anschließend werden die für Kursteilnehmer bzw. Leser empfohlenen Installationen beschrieben. Alle Pakete sind auf den unten genannten Web-Seiten kostenlos für alle relevanten Betriebssysteme verfügbar.

2.1 JDK 8 mit Dokumentation installieren

2.1.1 JDK

Das Java Development Kit (JDK) der Firma Oracle enthält u.a. …

- den Java-Compiler **javac.exe**
- zahlreiche Werkzeuge (z.B. den Dokumentationsgenerator **javadoc.exe** und den Archivgenerator **jar.exe**)
- den Quellcode der meisten Klassen im JSE-API

---

[1] Zu IntelliJ IDEA bietet die Firma JetBrains eine kostenlose Community Edition sowie eine kostenpflichtige Ultimate Edition (Preis am 28.10.2015: 499$) an (https://www.jetbrains.com/idea/download/). Das Android Studio der Firma Google basiert auf der IntelliJ IDEA Community Edition.

[2] Netbeans ist eine von der Firma Oracle gesponserte Open Source - Software (https://netbeans.org/).

[3] http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/

- eine interne Java Runtime Environment, z.B. für die Verwendung durch die Entwicklungswerkzeuge

Das JDK-Installationsprogramm kann auch eine öffentliche, durch beliebige Java-Programme und -Applets zu verwendende Java Runtime Environment einrichten.

Wir werden zusätzlich ein separat von Oracle angebotenes Dokumentationspaket zum JDK auf unseren Rechner befördern.

Zur Entwicklung von Java-Software mit dem später zu installierenden Eclipse 4.5.1 muss sich auf Ihrem Rechner lediglich eine JRE (ab Version 7) befinden. Es ist aber trotzdem sinnvoll, das JDK zu installieren, damit die zusätzlichen Entwicklungswerkzeuge und der Quellcode zu den API-Klassen verfügbar sind.

Das JDK 8 kann unter folgenden Windows-Versionen eingesetzt werden (jeweils 32 Bit und 64 Bit):

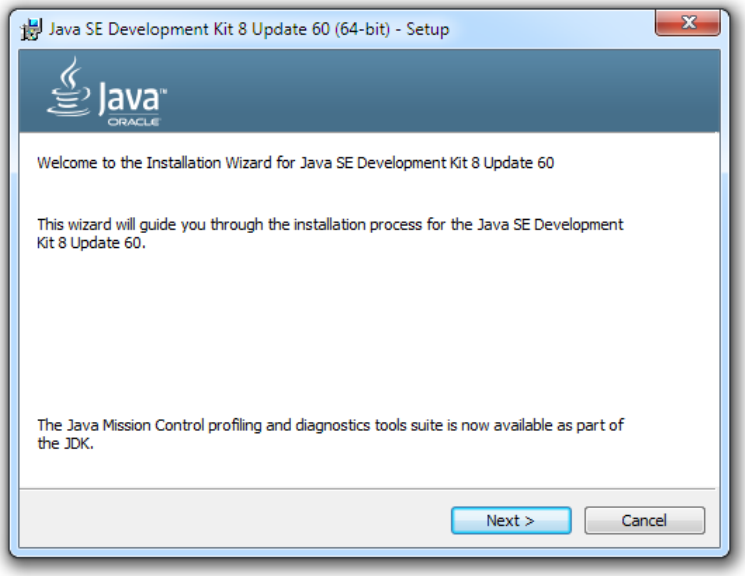
- Vista (SP2)
- 7 (SP1)
- 8.x
- 10

Webseite zum Herunterladen:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Anschließend wird exemplarisch die mit Administratorrechten durchzuführende Installation des Versionsstands Update 60 zum JDK 8 unter Windows 7 (64 Bit) beschrieben:[1]

- Doppelklick auf **jdk-8u60-windows-x64.exe**
- Es geht los mit dem **Welcome**-Dialog,



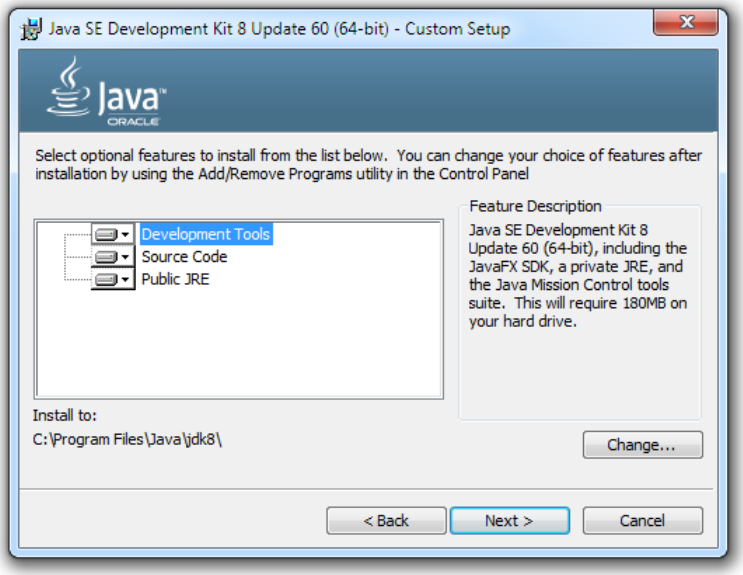
- Im Dialog **Custom Setup** können Sie über den Schalter **Change** einen alternativen Installationsordner wählen (statt **C:\Program Files\Java\jdk1.8.0_60**). Es ist sinnvoll, die Update-Nummer aus dem Ordnernamen heraus zu halten, weil der JDK-Pfad gelegentlich

[1] Wenn Sie diesen Text lesen, ist mit großer Wahrscheinlichkeit ein höherer Update-Stand aktuell.

einzutragen ist (z.B. in der Entwicklungsumgebung Eclipse), so dass ein variabler Name unpraktisch ist. Wählen Sie z.B.:

C:\Program Files\Java\jdk8

Per Voreinstellung



richtet das Installationsprogramm eine öffentliche 64-Bit - JRE (Java Runtime Environment) auf Ihrem Rechner ein (per Voreinstellung im Ordner **C:\Program Files\Java\jre1.8.0_60**). Während die im JDK grundsätzlich vorhandene *interne* JRE normalerweise nur zur Softwareentwicklung dient, ist die separate öffentliche JRE für *alle* Java-Anwendungen und -Applets sichtbar durch eine Registrierung beim Betriebssystem und bei den WWW-Browsern. Wer noch keine öffentliche 64-Bit - JRE mit Version 8 auf seinem Rechner hat, sollte die Installation zulassen (zusätzlicher Festplattenspeicherbedarf ca. 175 MB).[1] Auf die Installation des Source Codes sollten Sie auf keinen Fall verzichten.

Von einem Java-Update (vgl. Abschnitt 1.2.2) ist auch das JDK betroffen, wobei aber *kein* Automatismus aktiv wird. Gehören Alt- und Neuversion zur Java-Generation 8, kann man den bisherigen Installationsordner weiterverwenden. Bei der Installation werden die veralteten Dateien ersetzt, während eigene Zusatzordner (z.B. mit der Dokumentation) unangetastet bleiben.

2.1.2 7-Zip

Bei Bedarf können Sie das Hilfsprogramm 7-Zip zum Erstellen und Auspacken von Archiven installieren und für die anschließend auftauchenden ZIP-Dateien verwenden. 7-Zip ist bedienungsfreundlicher und schneller als die in Windows integrierte ZIP-Funktionalität.

Webseite zum Herunterladen:

[1] Eine 32-Bit - JRE, die von 32-Bit - Browsern zur Ausführung von Java-Applets benötigt wird, gelangt bei dieser Installation *nicht* auf Ihren Rechner. In Abschnitt 1.2.1 wird die separate Installation der JRE für 32 oder 64 Bit beschrieben.

http://www.7-zip.org/download.html

Unter Windows 7 (64 Bit) wird die Installation der Version 9.20 durch einen Doppelklick auf die Datei **7z920-x64.msi** gestartet. 7-Zip steht nach der Installation ohne Neustart samt Integration in den Windows-Explorer zur Verfügung.

2.1.3 Dokumentation

2.1.3.1 Java SE

Die systematische Dokumentation zu allen Klassen im JSE-API und zu den JDK-Werkzeugen ist bei der Erstellung von Java-Software sehr nützlich. Natürlich ist die Dokumentation auch im Internet verfügbar und wird z.B. von Eclipse spontan dort gesucht. Man kann also auf die lokale Installation verzichten, wenn eine schnelle und zuverlässige Internet-Anbindung verfügbar ist.

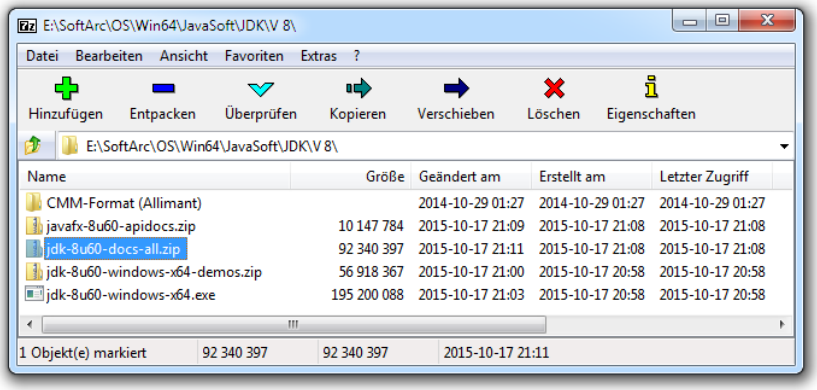
Auf der Festplatte belegt die Dokumentation im ausgepackten Zustand ca. 350 MB.

Webseite zum Herunterladen:

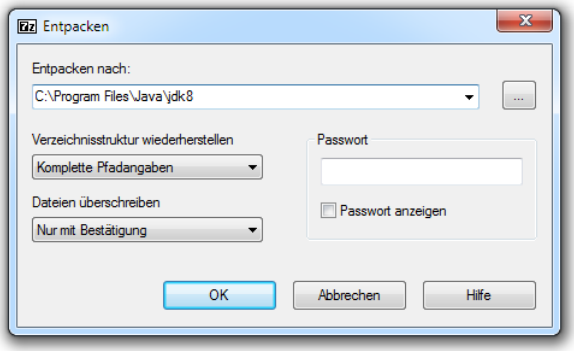
http://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html

Um die Dateien und Ordner der Dokumentation aus dem herunter geladenen Archiv **jdk-8u60-docs-all.zip** zu extrahieren, können Sie z.B. das Programm 7-Zip benutzen (vgl. Abschnitt 2.1.2):

- 7-Zip mit Administratorrechten ausführen (aus dem Kontextmenü zum Startlink wählen: **Als Administrator ausführen**)
- ZIP-Datei mit der JDK-Dokumentation markieren



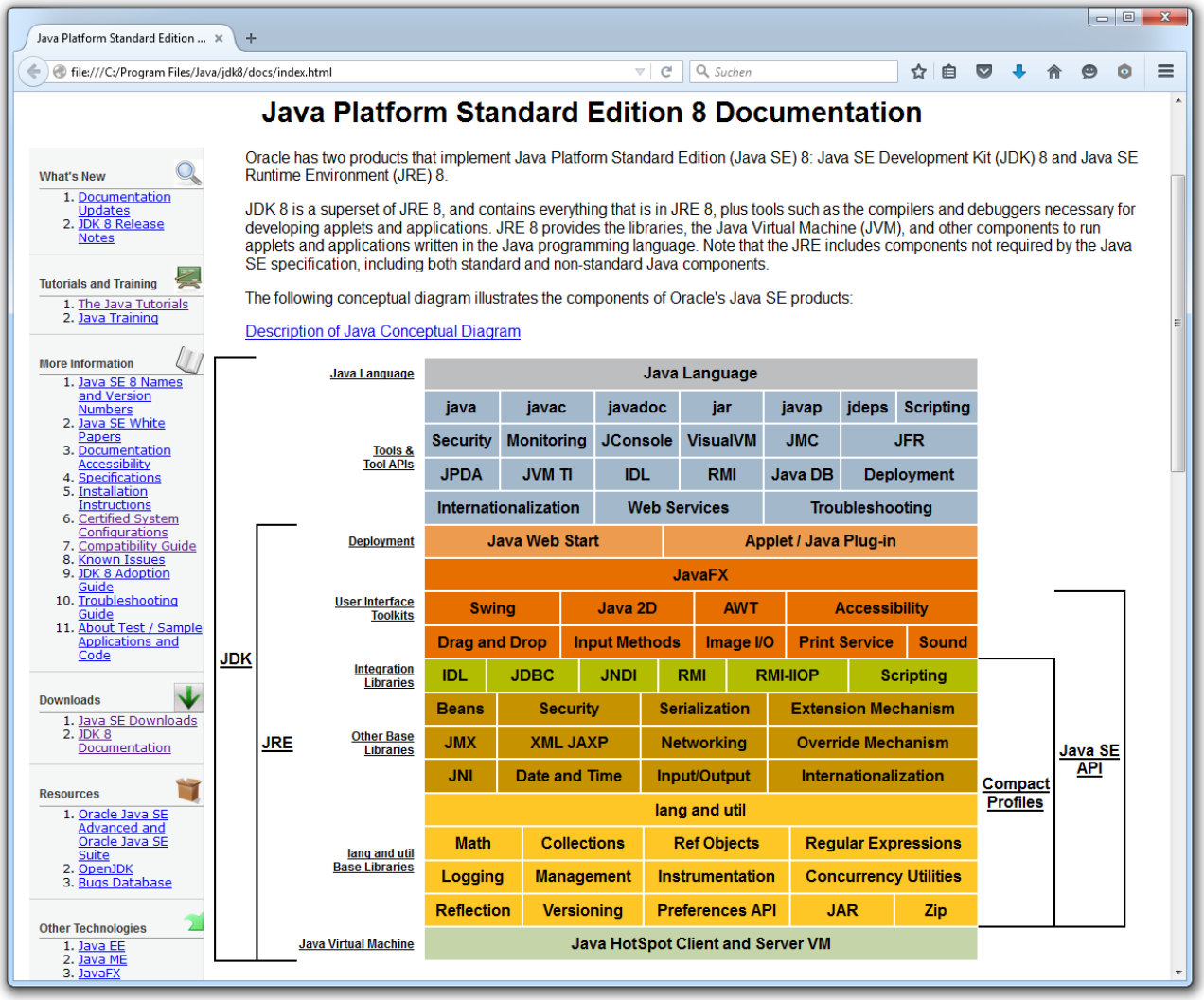
- Klick auf **Entpacken**
- Ein geeignetes Ziel ist der JDK-Installationsordner (z.B. **C:\Program Files\Java\jdk8**),



so dass dort der Unterordner **docs** entsteht.

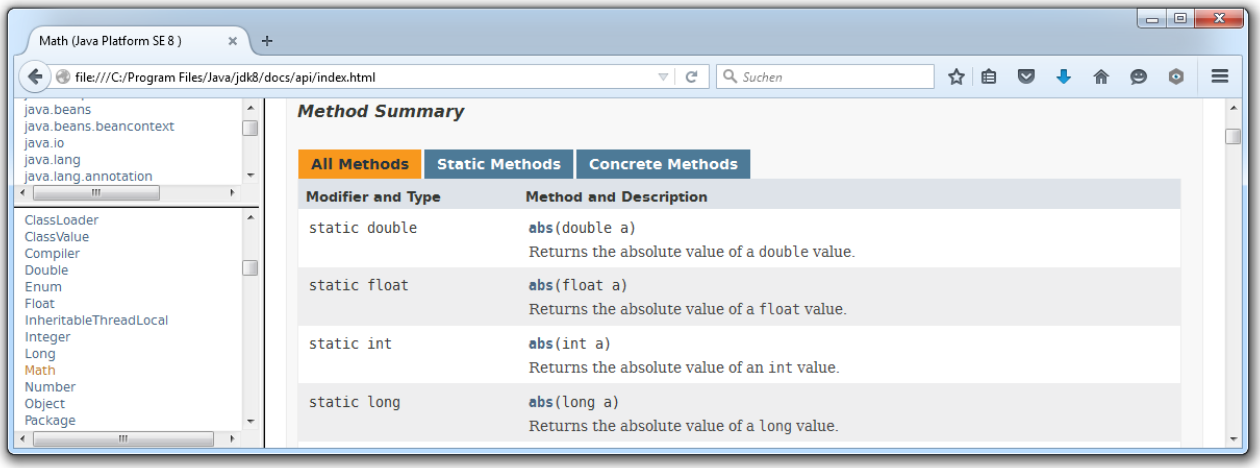
Weil wir gelegentlich einen neugierigen Blick auf den Quellcode einer Klasse aus dem JSE-API (also aus der Standardbibliothek von Java) werfen, sollten Sie die zum JDK gehörige Quellcodedatei **src.zip** analog zur JDK-Dokumentation behandeln und in den JDK-Unterordner **src** auspacken. Wenn Sie bei der JDK-Installation (siehe Abschnitt 2.1.1) die Quellcode-Option nicht deaktiviert haben, landet die Datei **src.zip** im Basisordner der Installation. Übrigens nutzt Eclipse den API-Quellcode zur Unterstützung der Fehlersuche und kommt dabei mit dem ZIP-Archiv zurecht. Wer den Quellcode unabhängig von Eclipse durchstöbern will, packt das ZIP-Archiv aber besser aus.

Nach dem Öffnen der Startdatei zur JDK-Dokumentation (z.B. **C:\Program Files\Java\jdk8\docs\index.html**) zeigt sich, dass es im Java-Land viel zu entdecken gibt:



Von der Startseite aus erreicht man über den Link **Java SE API** (am rechten Rand zu finden) die beim Programmieren besonders oft benötigte **API-Dokumentation** mit einer detaillierten Beschreibung aller Typen (Klassen und Schnittstellen) der Standardbibliothek. Bei geöffneter API-Dokumentation gelangen Sie z.B. auf folgende Weise zur Beschreibung der in unserem Beispielprogramm aus Abschnitt 1.1.2 verwendeten Methode **abs()** der Klasse **Math**, die zu einer Zahl den Betrag ermittelt:

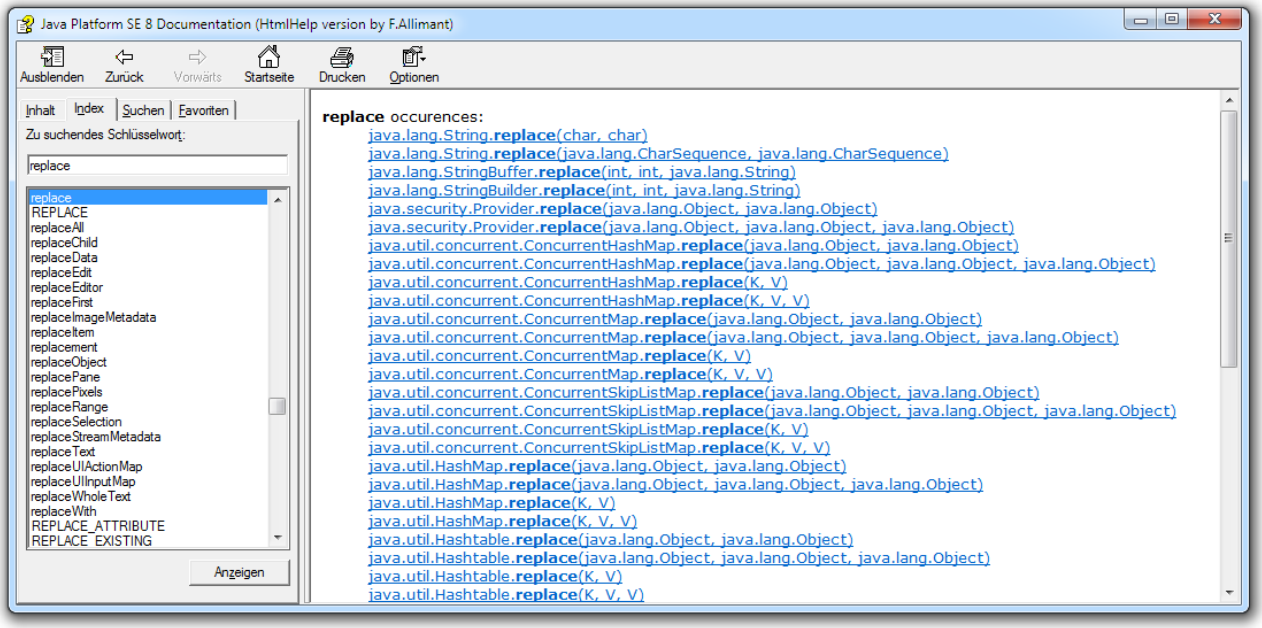
- Klicken Sie im linken oberen Frame auf **All Classes** oder (zur Verkürzung der Liste im linken unteren Frame) auf das Paket **java.lang**, zu dem die Klasse **Math** gehört (siehe Kapitel 6).
- Klicken Sie im linken unteren Frame auf den Klassennamen **Math**. Anschließend erscheinen im rechten Frame detaillierte Informationen über die Klasse **Math**, u.a. über ihre Methoden:



Im Vergleich zu der von Oracle gelieferten API-Dokumentation erleichtert die von Franck Allimant gepflegte und auf der folgenden Webseite

http://www.allimant.org/javadoc/index.php

angebotene Version im CHM-Format der Windows-Hilfedateien das Suchen nach Begriffen, z.B.:



2.1.3.2 JavaFX

Wir werden graphische Bedienoberflächen von Programmen mit der JavaFX-Technologie entwickeln. Obwohl wir uns dabei auf die deklarative Programmierung mit dem XML-Dialekt *FXML* konzentrieren und außerdem durch Verwendung des Werkzeugs *Scene Builder* um den direkten FXML-Kontakt meist herumkommen, kann eine Beschäftigung mit den zugrunde liegenden Ja-

vaFX-Klassen notwendig werden, so dass die zugehörige Dokumentation genauso auf die lokale Festplatte befördert vorbereitet werden sollte wie die JDK-Dokumentation.

Die Datei **javafx-8u60-apidocs.zip** mit der JavaFX-Dokumentation wird auf derselben Webseite angeboten, von der wir schon die JDK-Dokumentation bezogen haben:

http://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html

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):

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

Im Unterschied zu *hybriden* Programmiersprachen wie C++ und Delphi, die neben der objektorientierten auch die rein 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. Es ist typisch für die objektorientierte Programmierung in Java, dass hier ein *konkretes Objekt* mit der Ausgabe beauftragt wird. Anonyme Funktionsaufrufe, die „der Computer“ auszuführen hat, gibt es nicht.

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:

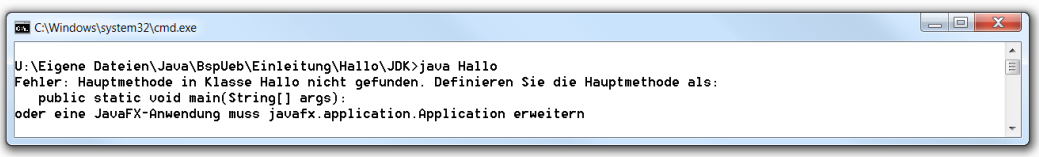
- Nach dem Schlüsselwort **class** folgt der frei wählbare Klassenname. Hier ist wie bei allen Bezeichnern zu beachten, dass Java streng zwischen Groß- und Kleinbuchstaben unterscheidet.

  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. Viele 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 at. (2015) lassen bei Startklassen, die nur von der JRE 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 JRE beim Programmstart ausgeführt und dient bei „echten“ OOP-Programmen (direkt oder indirekt) dazu, Objekte zu erzeugen.
- 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 JRE gesucht und ausgeführt. Sie muss (zumindest ab Java 1.4.x) den Zugriffsmodifikator **public** erhalten. Anderenfalls reklamiert die JRE beim Startversuch:



  - **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. In einem objektorientierten Programm hat sie insbesondere die Aufgabe, die ersten Objekte zu erzeugen (siehe die Klasse `Bruchaddition` auf Seite 10).
  - **void**
    Die Methode **main()** erhält den Typ **void**, weil sie keinen Rückgabewert liefert.

  Die beiden Modifikatoren **public** und **static** stehen in beliebiger Reihenfolge am Anfang der Methodendefinition. Ihnen folgt die Typdeklaration, die 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:
  - *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 JRE 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 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:

- Der Dateinamensstamm (vor dem Punkt) sollte unbedingt mit dem Klassennamen übereinstimmen. Ansonsten resultiert eine Namensabweichung zwischen Quellcode- und Bytecode-Datei, denn die vom Compiler erzeugte Bytecode-Datei übernimmt den Namen der *Klasse*.
- Die Dateinamenserweiterung muss **.java** lauten.
- 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).

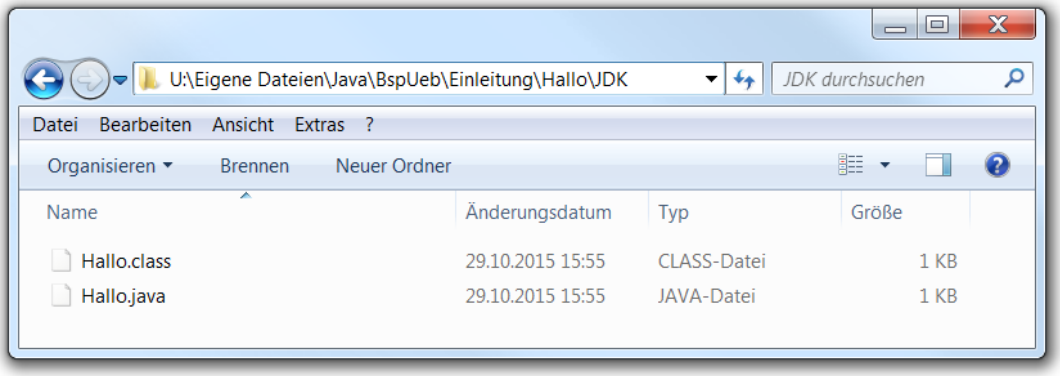
2.2.2 Übersetzen

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 JDK-Compiler **javac** übersetzen, z.B.:

```
>"C:\Program Files\Java\jdk8\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.:

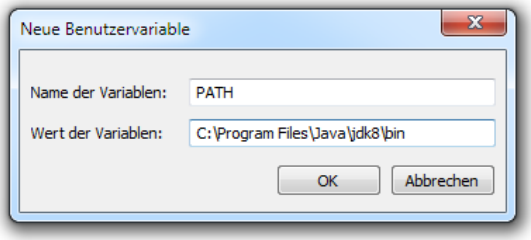


Damit unter Windows der Compiler ohne Pfadangabe von jedem Verzeichnis aus gestartet werden kann,

```
>javac Hallo.java
```

muss das **bin**-Unterverzeichnis der JDK-Installation (mit dem Compiler **javac.exe**) in die Definition der Umgebungsvariablen PATH aufgenommen werden:[1]

- Öffnen Sie das Fenster der **Systemsteuerung**. Die passende Option findet sich bei Windows 7 im Startmenü und bei Windows 10 im Kontextmenü zum Windows-Symbol am linken Rand der Taskleiste.
- Wählen Sie als **Anzeige** über das Bedienelement oben rechts die Option **Kleine Symbole**.
- Wählen Sie im renovierten Fenster per Mausklick die Option **Benutzerkonten**.
- Klicken Sie im linksseitigen Menü des nächsten Fensters auf den Link **Eigene Umgebungsvariablen ändern**.
- Nun können Sie in der Dialogbox **Umgebungsvariablen** die **Benutzervariable** PATH anlegen oder erweitern. Mit Administratorrechten lässt sich auch die Definition der regelmäßig vorhandenen **Systemvariablen** gleichen Namens erweitern.
- Im folgenden Dialog wird eine **neue Benutzervariable** angelegt:



  Beim Erweitern einer PATH-Definition trennt man zwei Einträge durch ein Semikolon.

Beim Kompilieren einer Quellcodedatei wird auch jede darin benutzte fremde Klasse neu übersetzt, falls deren Bytecode-Datei fehlt oder älter als die zugehörige Quellcodedatei ist. Sind etwa im Bruchadditionsbeispiel die Quellcodedateien **Bruch.java** *und* **Bruchaddition.java** geändert worden, dann genügt der folgende Compiler-Aufruf, um *beide* neu zu übersetzen:

```
>javac Bruchaddition.java
```

[1] Bei der Software-Entwicklung mit Eclipse spielt der Compiler **javac.exe** allerdings keine Rolle, und ein Verzicht auf den PATH-Eintrag wirkt sich im Kurs kaum 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.

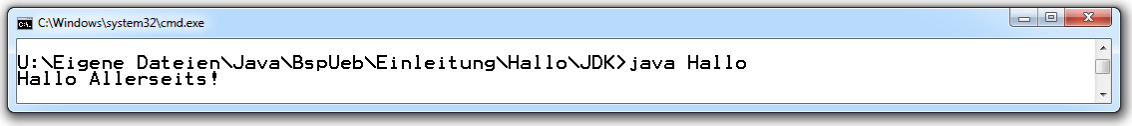
2.2.3 Ausführen

Wie Sie bereits wissen, wird zum *Ausführen* von Java-Programmen *nicht* das JDK (mit Entwicklerwerkzeugen, Dokumentation etc.) benötigt, sondern lediglich die **Java Runtime Environment** (JRE) mit dem Interpreter **java.exe** und der Standardklassenbibliothek. Wie man dieses Produkt beziehen und installieren kann, wurde schon in Abschnitt 1.2.1 beschrieben. Man darf die JRE zusammen mit eigenen Programmen weitergeben, um diese auf beliebigen Rechnern lauffähig zu machen.

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

```
>java Hallo
```

sollte zum folgenden Ergebnis führen:



In der Regel müssen Sie *keinen* PATH-Eintrag vornehmen, um **java.exe** von jedem Verzeichnis aus ohne Pfadangabe starten zu können, weil sich das JRE-Installationsprogramm, darum kümmert.

Beim Programmstart ist zu beachten:

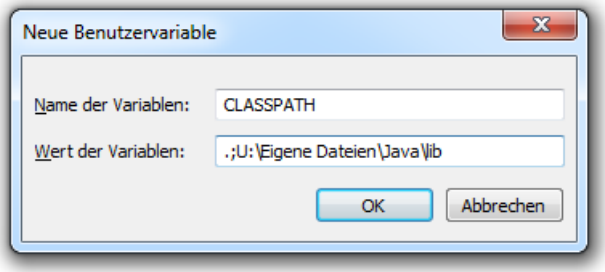
- Beim Aufruf des Interpreters wird der Name der auszuführenden *Klasse* als Argument angegeben, nicht der zugehörige Dateiname. Wer den Dateinamen (samt Namenserweiterung **.class**) angibt, sieht eine Fehlermeldung:

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

2.2.4 Suchpfad für class-Dateien setzen

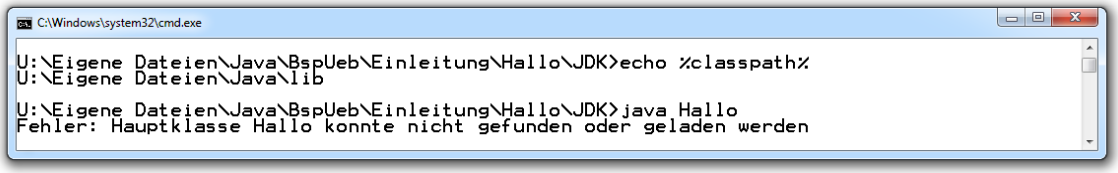
Compiler und Interpreter benötigen Zugriff auf die Bytecode-Dateien der Klassen, die im zu übersetzenden Quellcode bzw. im auszuführenden Programm angesprochen werden. Mit Hilfe der Umgebungsvariablen CLASSPATH kann man eine Liste von Verzeichnissen, JAR-Archiven (siehe Abschnitt 6.4) oder ZIP-Archiven spezifizieren, die nach **class**-Dateien durchsucht werden sollen, z.B.:



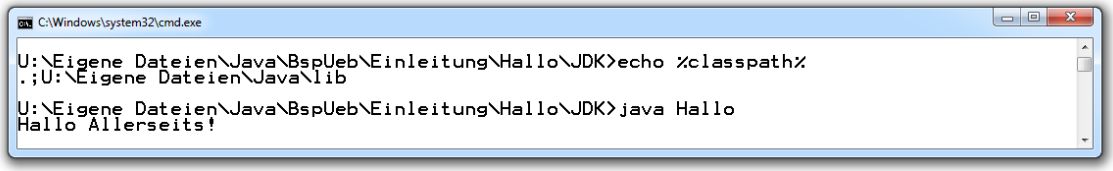
Bei einer Verzeichnisangabe sind Unterverzeichnisse *nicht* einbezogen. Sollten sich z.B. für einen Compiler- oder Interpreter-Aufruf benötigte Dateien im Ordner **U:\Eigene Dateien\Java\lib\sub** befinden, werden sie aufgrund der CLASSPATH-Definition in obiger Dialogbox nicht gefunden.

Wie man unter Windows 7 oder 10 eine Umgebungsvariable setzen kann, wird in 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 angegebenen Pfade berücksichtigt. Dies führt zu Problemen, wenn in der CLASSPATH-Definition das aktuelle Verzeichnis *nicht* enthalten ist, z.B.:



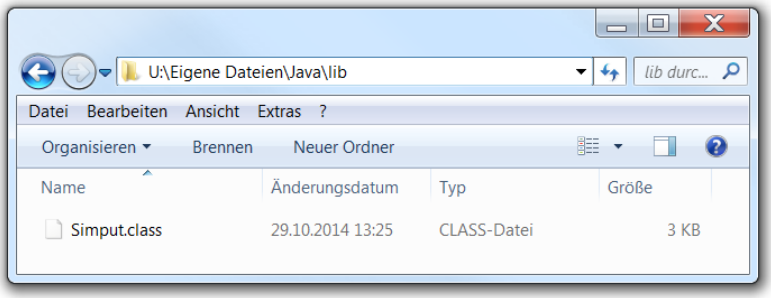
In diesem Fall muss das aktuelle Verzeichnis (z.B. dargestellt durch einen einzelnen Punkt, s.o.) in die CLASSPATH-Pfadliste aufgenommen werden, z.B.:



In vielen konsolenorientierten Beispielprogrammen des Manuskripts kommt die *nicht* zum JSE-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\Simput.class

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



Achten Sie unbedingt darauf, den aktuellen Pfad über einen Punkt in die CLASSPATH-Definition aufzunehmen.

Unsere Entwicklungsumgebung Eclipse ignoriert die CLASSPATH-Umgebungsvariable, bietet aber eine alternative Möglichkeit zur Definition eines Klassenpfads (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 **classpath**-Option (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 des **cp**-Kommandozeilenarguments gegenüber der Umgebungsvariablen CLASSPATH besteht darin, dass für jede Anwendung eine eigene Suchliste eingestellt werden kann.

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

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“).

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 verursacht 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, so dass 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.

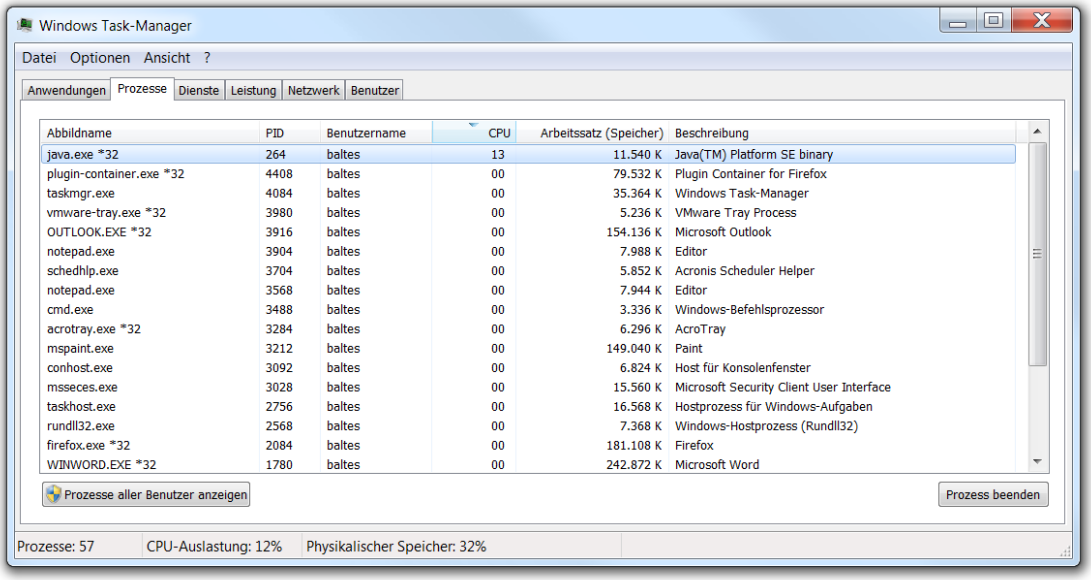
Weil sich in das simple Hallo-Beispielprogramm kaum ein Logikfehler einbauen lässt, betrachten wir die in Abschnitt 1.1 vorgestellte Klasse `Bruch`. Wird z.B. in der Methode `setzeNenner()` bei der Absicherung gegen Nullwerte das Ungleich-Operatorzeichen (!=) durch sein Gegenteil (==) ersetzt, ist keine 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 i7 mit Quad-Core - Hyper-Threading-CPU, also mit 8 logischen Kernen) belegt. Das Programm kann aufgrund seiner Single-Thread-Technik nur *einen* logischen Kern nutzen und lastet diesen voll aus, so dass 12,5% 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 Eclipse 4.5.1 mit Zubehör installieren

Im Kurs bzw. Manuskript wird eine Eclipse-Zusammenstellung verwendet, die folgende Bestandteile enthält:

- **Eclipse 4.5.1 IDE for Java EE Developers**
  Obwohl wir uns in diesem Manuskript überwiegend auf die Java Standard Edition (JSE) beschränken, leisten wir uns doch bei Eclipse die „große Lösung“, so dass (z.B. im Zusammenhang mit der Datenbankprogrammierung) auch die Werkzeuge für Lösungen aus dem Bereich der Java Enterprise Edition (JEE) verfügbar sind.
- **GUI-Designer WindowBuilder 1.8.0**
  Zur Erstellung von grafischen Bedienoberflächen wurde mit Java 8 die langjährig genutzte Swing-Technik durch eine neue Lösung namens *JavaFX* ersetzt.[1] Leider wird JavaFX von dem als Eclipse-Erweiterung verfügbaren grafischen GUI-Designer *Window Builder* nicht unterstützt. Für die bisherigen GUI-Techniken (z.B. Swing, SWT) ist der Window Builder aber gut geeignet. Weil Swing im Manuskript im Hinblick auf die zahllosen mit dieser Technik erstellten und weiterhin zu pflegenden Anwendungen neben JavaFX ebenfalls beschrieben wird, sollte der Window Builder installiert werden. Im Eclipse-Paket für die JSE ist der Window Builder enthalten, im JEE-Paket muss er (mit sehr geringem Aufwand) nachinstalliert werden. Als grafischen GUI-Designer für JavaFX werden wir das selbständige, aber gut mit Eclipse kooperierende Programm *Scene Builder* verwenden.
- **e(fx)clipse 2.1.0**
  Das Eclipse-Plugin e(fx)clipse unterstützt das Editieren von FXML-Dateien, die zur Deklaration von Bedienoberflächen im Rahmen der JavaFX-Technik dienen, und ist erforderlich für die Kooperation von Eclipse mit dem Scene Builder.

---

[1] Unter der Adresse http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6 wird von der Firma Oracle die folgende Frage eindeutig bejaht:

Is JavaFX replacing Swing as the new client UI library for Java SE?

- **Deutsche Sprachpakete (Babel-Projekt, R0.13.0)**
  Um Programmierneulingen jede mögliche Erleichterung anzubieten, werden wir eine deutsche Lokalisierung von Eclipse verwenden. Weil das zuständige Babel-Projekt bisher nur die wichtigsten Texte übersetzt hat, müssen wir eine Mischung aus Deutsch und Englisch akzeptieren. Über eine Startoption kann auch die englische Eclipse-Bedienoberfläche gewählt werden, was gelegentlich die Verwertung von Internet-Tipps erleichtert.
- **Deutsche Rechtschreibeprüfung**
  Wenn Programme ausführlich in Deutsch kommentiert werden, ist eine Rechtschreibeprüfung von Nutzen.

2.3.1 Eclipse 4.5.1 IDE for Java EE Developers

Zur (im Oktober 2015) aktuellen Eclipse-Version 4.5.1 (Mars) werden auf der Webseite

http://www.eclipse.org/downloads/

etliche Pakete angeboten. Weil wir uns auch mit Datenbankprogrammierung beschäftigen wollen, entscheiden wir uns für das Paket **Eclipse IDE for Java EE Developers**.

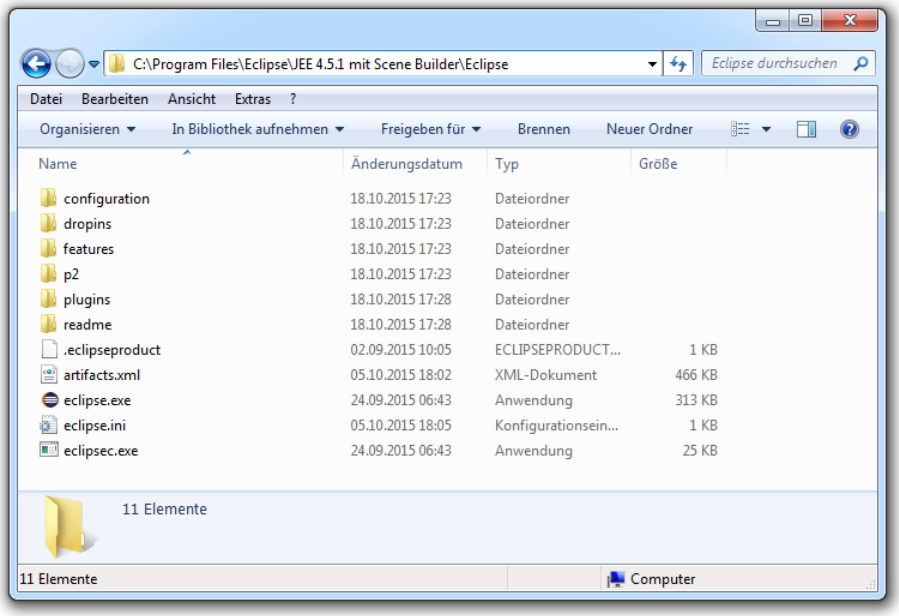
Eventuell sind Sie irritiert, weil die Pakete in einer 64- und eine 32-Bit - Variante erhältlich sind, obwohl Eclipse oben als Java-Anwendung bezeichnet wurde und daher perfekt portabel sein sollte. Vermutlich ist die mit Eclipse eingeführte GUI-Bibliothek SWT (*Standard Widget Toolkit*) mit Klassen zur Gestaltung der grafischen Bedienoberfläche dafür verantwortlich, dass Dateien mit Maschinen-Code bei Eclipse beteiligt sind, so dass sich die Versionen für x86 und x64 unterscheiden. Wir werden zwar Eclipse als Entwicklungsumgebung verwenden, bei unseren GUI-Anwendungen aber die komplett in Java realisierten Bibliotheken JavaFX und Swing einsetzen und somit eine uneingeschränkte Binärportabilität erhalten.

Eclipse 4.5.1 bringt einen eigenen Compiler mit, der kompatibel ist mit der Sprachdefinition von Java 8.

Voraussetzungen:

- JRE ab Version 7 (1.7.0)
  Als Java-Anwendung benötigt Eclipse zur Ausführung eine JRE. Weil wir bereits das JDK 8 (alias 1.8.0) installiert haben, ist auf jeden Fall eine passende JRE vorhanden.
- ca. 500 MB Massenspeicher (auf einer Festplatte oder SSD)

Von der oben angegebenen Download-Adresse erhält man eine ZIP-Datei (z.B. mit dem Namen **eclipse-jee-mars-1-win32-x86_64.zip** bei der 64-Bit - Version), die z.B. mit dem Hilfsprogramm 7-Zip (siehe Abschnitt 2.1.2) in einen beliebigen Ordner ausgepackt werden kann, z.B. mit dem folgenden Ergebnis:



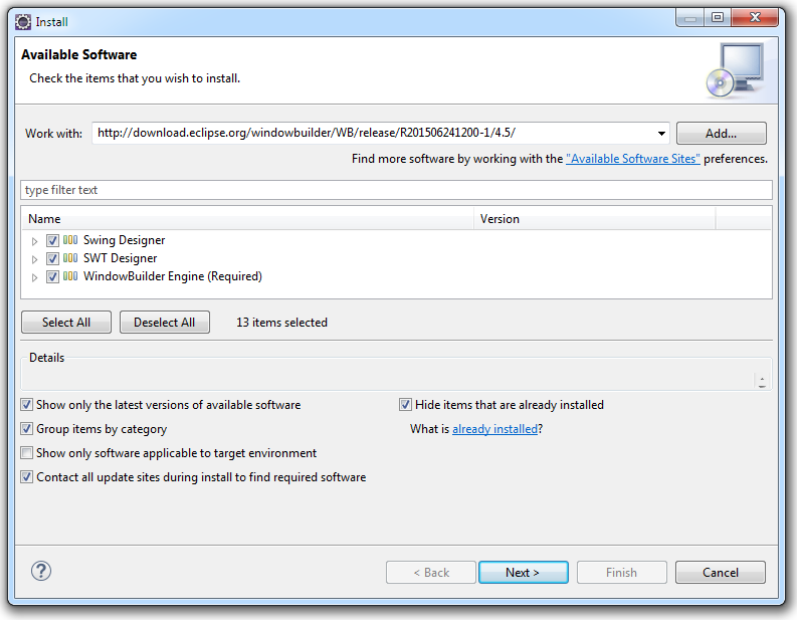
2.3.2 GUI-Designer WindowBuilder 1.8.0

Um den WindowBuilder zu installieren, kann man bei vorhandener Internetverbindung so vorgehen:

- Eclipse starten
- Menübefehl **Help > Install new Software**
- Tragen Sie im Textfeld **Work With** den folgenden Link

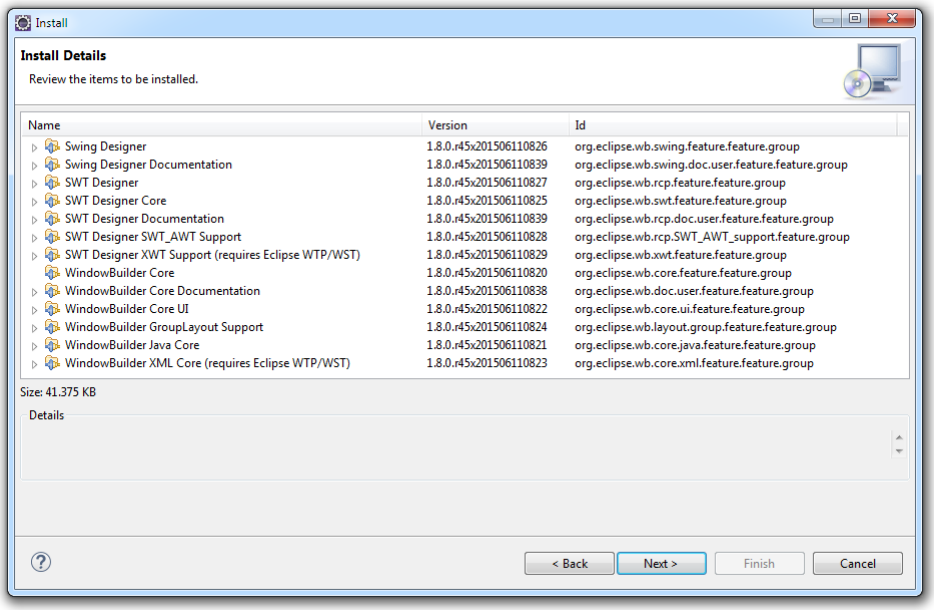
  http://download.eclipse.org/windowbuilder/WB/release/R201506241200-1/4.5/

  ein, und quittieren Sie mit **Enter**.
- Nach kurzer Wartezeit (mit Anzeige **Pending**) werden unter **Name** die installierbaren Komponenten aufgelistet. Wählen Sie alle per Mausklick auf **Select All**,



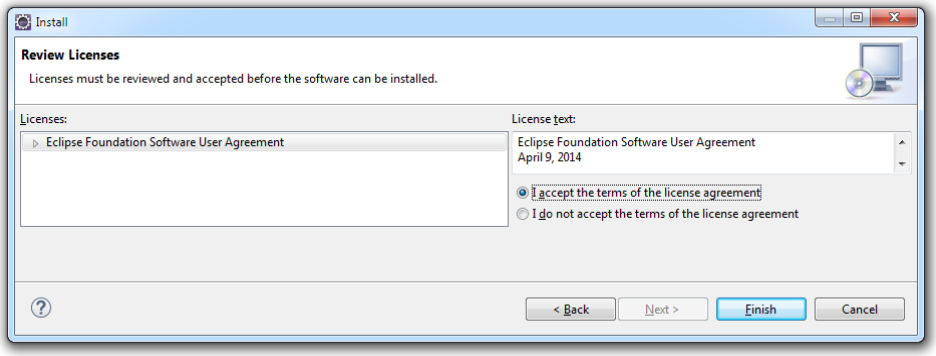
und machen Sie weiter mit dem Schalter **Next**.

- Quittieren Sie den Dialog mit den **Install Details**

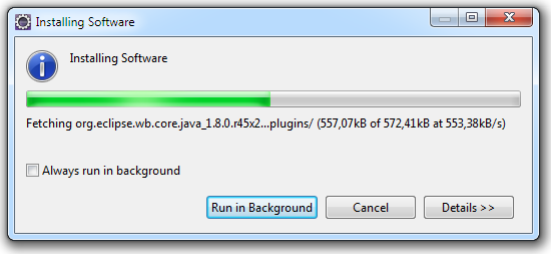


ebenfalls mit **Next**.

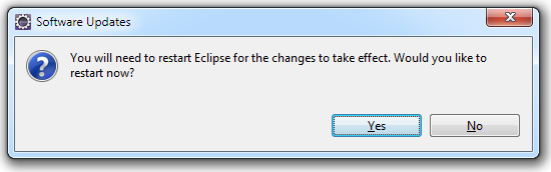
- Akzeptieren Sie die Lizenzbedingungen,



und klicken Sie auf **Finish**, um die Installation zu starten:



- Nach einem abschließenden Neustart von Eclipse



ist der WindowBuilder einsatzbereit.

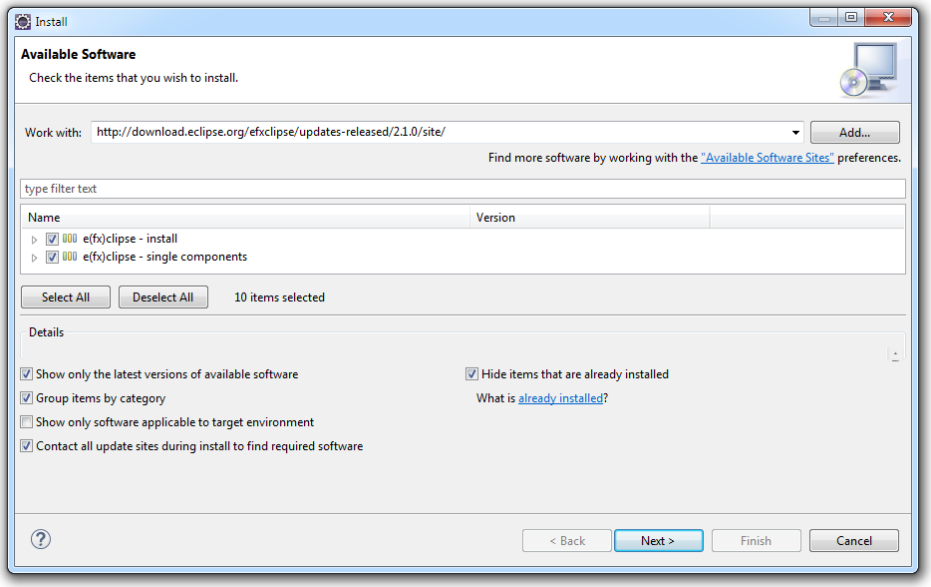
2.3.3 e(fx)clipse 2.1.0

Um mit dem Plugin e(fx)clipse die Unterstützung von JavaFX in Eclipse zu installieren, kann man bei vorhandener Internetverbindung so vorgehen:

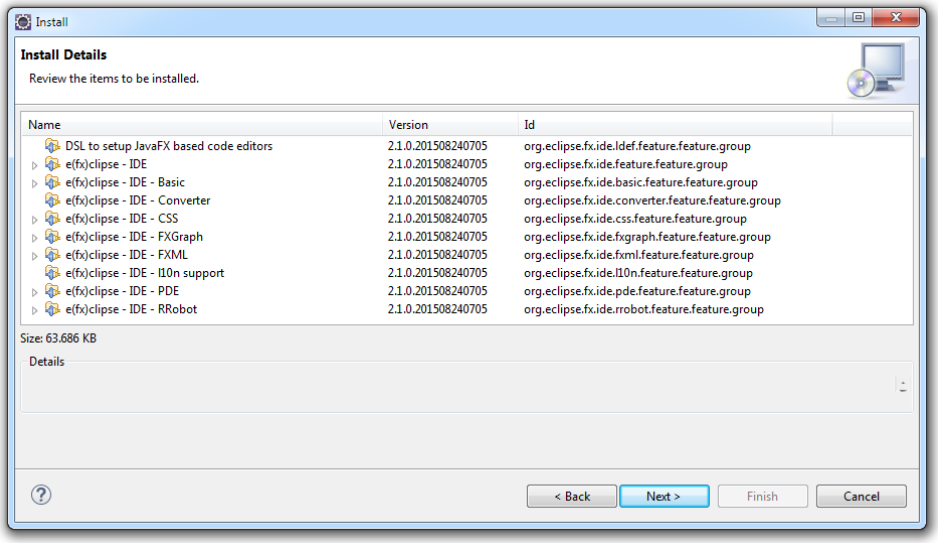
- Eclipse starten
- Menübefehl **Help > Install new Software**
- Tragen Sie im Textfeld **Work With** den folgenden Link

  http://download.eclipse.org/efxclipse/updates-released/2.1.0/site/

  ein, und quittieren Sie mit **Enter**.
- Nach kurzer Wartezeit (mit Anzeige **Pending**) werden unter **Name** die installierbaren Komponenten aufgelistet. Wählen Sie alle per Mausklick auf **Select All**,

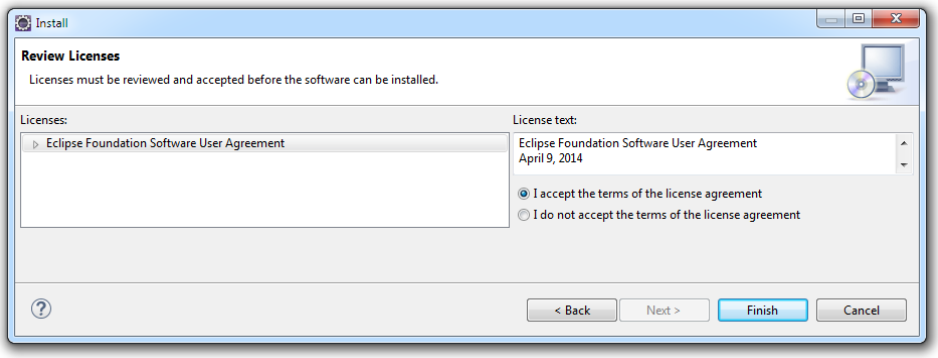


  und machen Sie weiter mit dem Schalter **Next**.
- Quittieren Sie den Dialog mit den **Install Details**

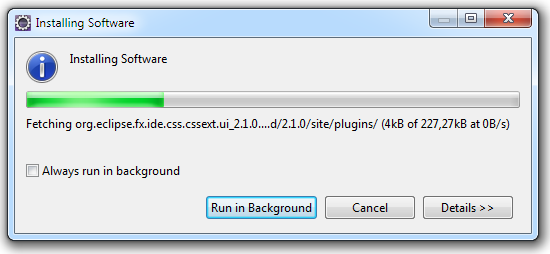


  ebenfalls mit **Next**.

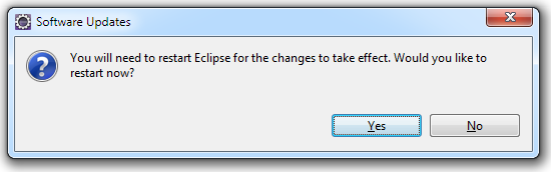
- Akzeptieren Sie die Lizenzbedingungen,



und klicken Sie auf **Finish**, um die Installation zu starten:



- Nach dem abschließenden Neustart

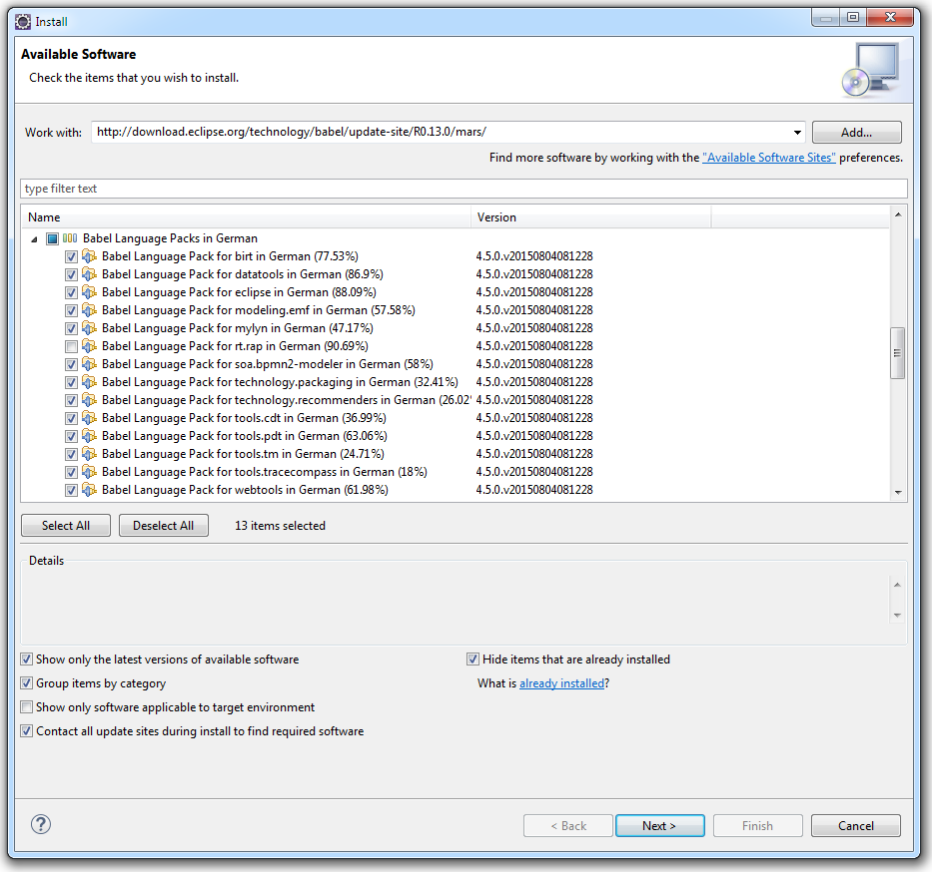


ist e(fx)clipse einsatzbereit.

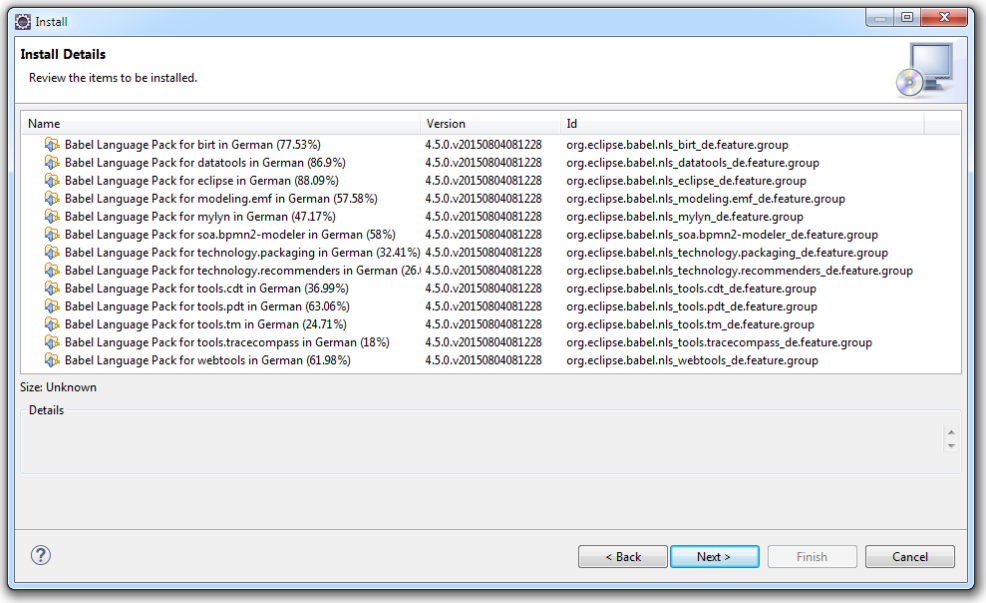
2.3.4 Deutsche Sprachpakete (Babel-Projekt, R0.13.0)

Um die deutschen Sprachpakete aus dem Babel-Projekt zu installieren, kann man bei vorhandener Internetverbindung so vorgehen:

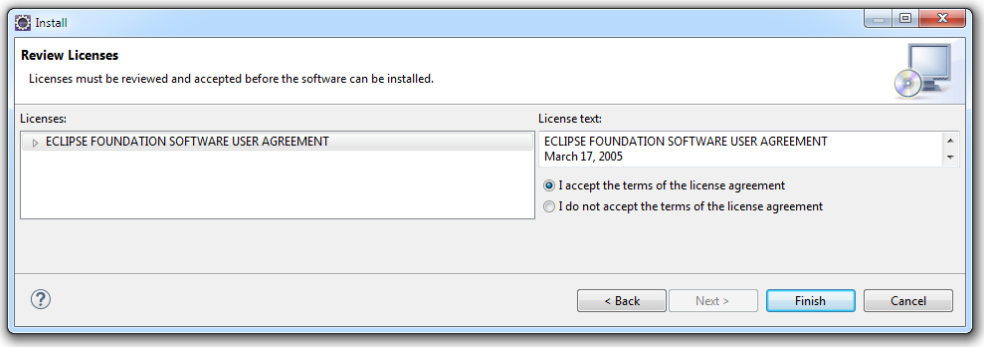
- Eclipse starten
- Menübefehl **Help > Install new Software**
- Tragen Sie im Textfeld **Work With** den folgenden Link
  http://download.eclipse.org/technology/babel/update-site/R0.13.0/mars/
  ein, und quittieren Sie mit **Enter**.
- Nach kurzer Wartezeit (mit Anzeige **Pending**) erscheint eine Liste mit den installierbaren Komponenten. Beschränken Sie sich auf die Kategorie **Babel Language Packs in German**, und wählen Sie aus dieser Kategorie alle Pakete mit Ausnahme des **Babel Language Packs for rt.rap in German**, das fehlerhaft ist den Start von Eclipse verhindert:



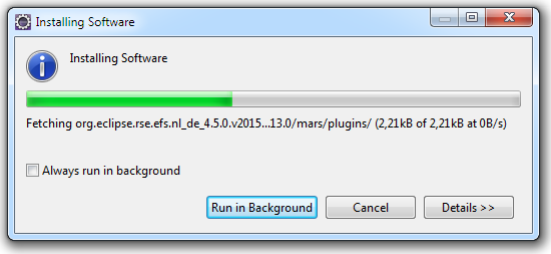
- Machen Sie weiter mit dem Schalter **Next**, und quittieren Sie ebenso den Dialog mit den **Install Details**:



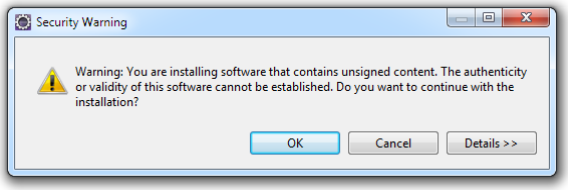
- Akzeptieren Sie die Lizenzbedingungen,



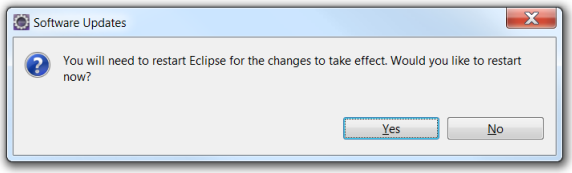
und klicken Sie auf **Finish**, um die Installation zu starten:



- Die folgende Sicherheitswarnung müssen Sie mit OK quittieren:



- Nach dem abschließenden Neustart



spricht Eclipse deutsch:

Nach einem Start mit dem Kommandozeilenargument **-nl en_US** erscheint Eclipse trotz der installierten deutschen Sprachpakete mit englischer Bedienoberfläche, z.B.:

"C:\Program Files\Eclipse\4.5\eclipse.exe" -nl en_US

2.3.5 Updatecheck

Nach der Eclipse-Installation sollten Sie über den folgenden Menübefehl

Hilfe > Auf Updates prüfen

dafür sorgen, dass ggf. vorhandene Updates für Eclipse oder eine Erweiterung installiert werden.

2.3.6 Deutsche Rechtschreibprüfung

Die Hochschule Augsburg hat freundlicherweise für Eclipse eine deutsche Rechtschreibeprüfung mit Auswirkung auf die (hoffentlich umfangreich angelegten) Quelltextkommentare zur Verfügung gestellt. Gehen Sie bei dieser optionalen Installation folgendermaßen vor:

- Eclipse beenden
- Datei german-utf8.dic herunterladen unter Verwendung der Adresse:
  http://mmprog.hs-augsburg.de/beispiele/eclipse/german-utf8.dic
- Datei im Eclipse-Installationsordner speichern unter **…\eclipse\dropins**

Konfigurationstipps:[1]

- Öffnen Sie den Konfigurationsdialog **Schreibprüfung** über

  Fenster > Benutzervorgaben > Allgemein > Editoren > Texteditoren > Schreibprüfung
- Tragen Sie ein **benutzerdefiniertes Wörterbuch** ein, indem Sie nach **Durchsuchen** die Datei **german-utf8.dic** wählen.
- Wählen Sie als **Codierung** über die Option **Sonstige** die Variante **UTF-8**.
- Quittieren Sie den Dialog mit **OK**.

2.3.7 Benutzerkonfiguration

Wenn Sie auf Ihrem Heim-PC Schreibrechte im Programmordner besitzen, wird Eclipse seine **Konfiguration** dort verwalten, z.B. in

C:\Program Files\Eclipse\4.5\configuration

Anderenfalls legt Eclipse einen **configuration**-Ordner im Benutzerprofil an, auf einem PC unter Windows 7 z.B. hier:

C:\Users\<*user*>\.eclipse\org.eclipse.platform_4.5.1_528658249_win32_win32_x86_64\configuration

2.4 Scene Builder installieren

Wer für das grafische Design von Bedienoberflächen mit JavaFX-Technik das von der Firma Oracle erstellte und unter der Oracle BSD-Lizenz zur Verfügung gestellte Programm *Scene Builder* einsatzbereit übersetzt beziehen möchte, findet auf der Download-Seite eine Warnung:[2]

> WARNING: These versions of JavaFX Scene Builder may include components that do not contain the latest security patches and are not recommended for use in production.

Es wird vermutet, dass Oracle nur noch Quellcode zur Verwendung in Entwicklungsumgebungen liefern will.[3]

Die Firma Gluon ist in die Bresche gesprungen und erstellt aus dem hier

http://hg.openjdk.java.net/openjfx

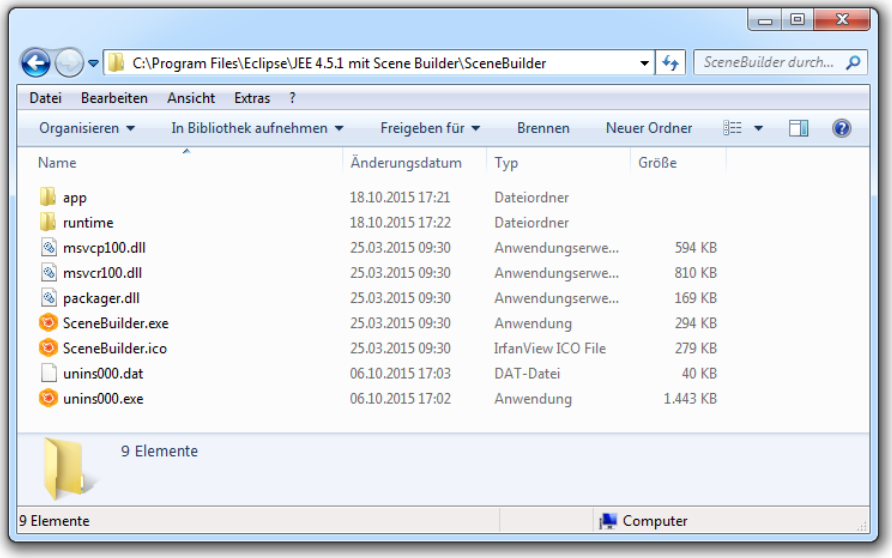
verfügbaren Quellcode des Scene Builders fertig übersetzte Programme für die Betriebssysteme Linux, Mac OS X und Windows, die auf der folgenden Webseite angeboten werden:

http://gluonhq.com/open-source/scene-builder/

---

1 Quelle: http://glossar.hs-augsburg.de/Konfiguration_von_Eclipse#W.C3.B6rterbuch
2 Quelle: http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-1x-archive-2199384.html
3 Siehe: http://stackoverflow.com/questions/28808130/where-exactly-can-i-download-the-latest-version-of-scene-builder-for-java

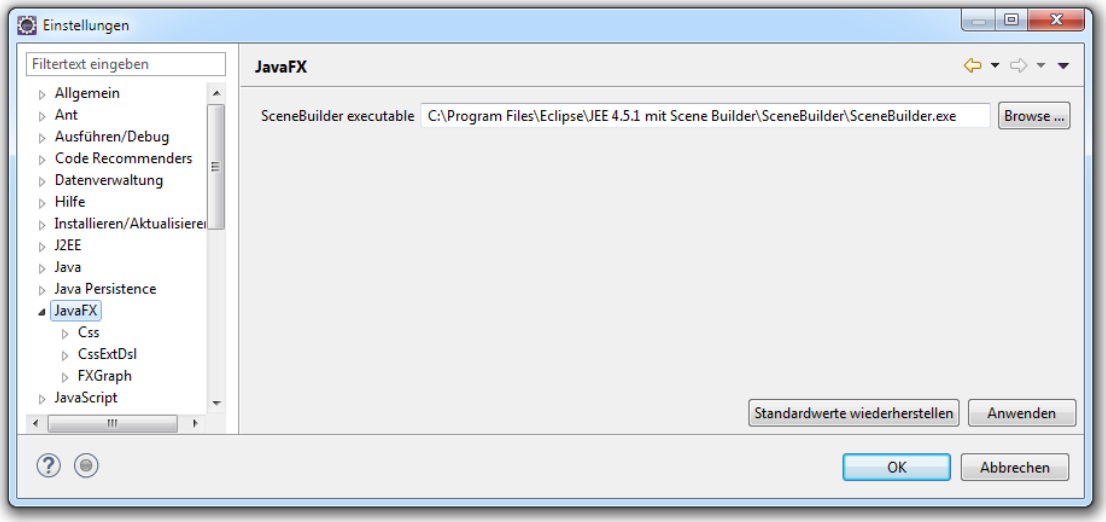
Die aktuelle Gluon-Version des Scene Builders trägt die Nummer 8.0 (passend zur aktuellen Java- bzw. JavaFX-Version) und entspricht wohl noch weitgehend der von Oracle (mit Sicherheitsbedenken) angebotenen Version 2.0. Über einen Doppelklick auf die heruntergeladene Datei **SceneBuilder-8.0.0-x64.exe** erhält man nach einer simplen Installation das ausführbare Programm, z.B.:



Damit Eclipse mit dem Scene Builder kooperieren kann, muss nach dem Menübefehl

Fenster > Benutzervorgaben > JavaFX

der Pfad zum ausführbaren Programm bekannt gegeben werden, z.B.:



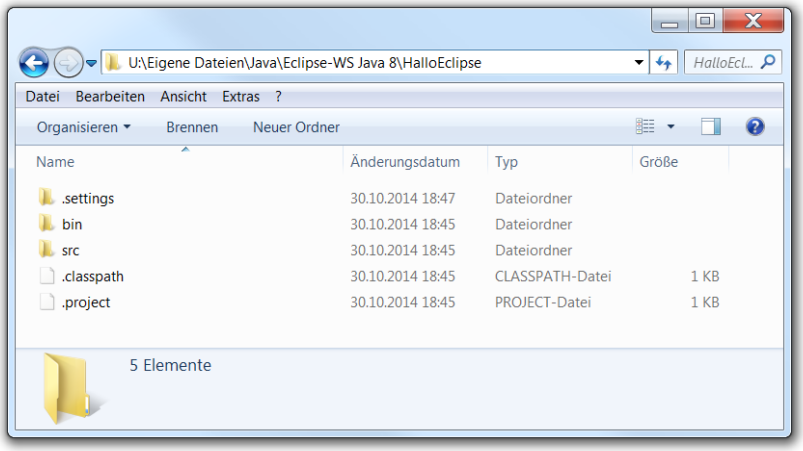
2.5 Java-Entwicklung mit Eclipse

Zu Eclipse sind umfangreiche Bücher entstanden mit einer Beschreibung der zahlreichen Profiwerkzeuge zur Softwareentwicklung (z.B. Künneth 2014). Wir beschränken uns anschließend auf elementare Informationen für Einsteiger, die später nach Bedarf ergänzt werden.

Eclipse taugt nicht nur als *Java*-Entwicklungsumgebung, sondern kann über entsprechende Erweiterung auch für andere Programmiersprachen genutzt werden (z.B. PHP, C++, Fortran). Außerdem lässt sich Eclipse aufgrund seiner modularen Struktur als **Rich Client Platform** (RCP) für eigene Java-Anwendungen verwenden.

2.5.1 Arbeitsbereich und Projekte

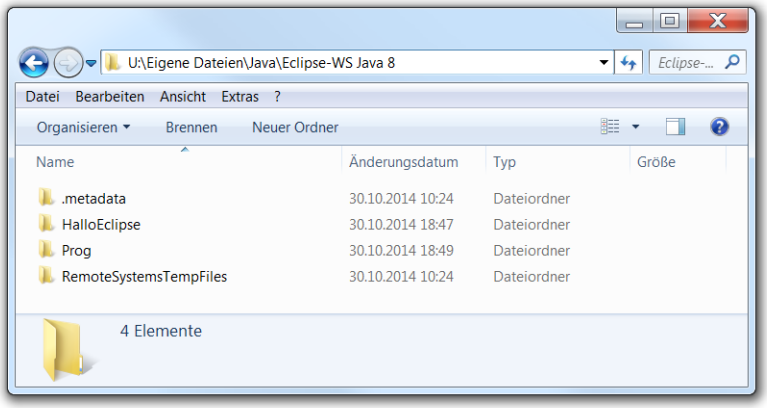
Eclipse legt für jedes Projekt einen Ordner an, der Quellcode-, Bytecode-, Konfigurations- und Hilfsdateien aufnimmt, z.B.:



Zusammengehörige Projekte bilden einen **Arbeitsbereich** (engl.: *Workspace*), und weil Eclipse zur Bearbeitung eines solchen Projekt-Ensembles konzipiert ist, verlangt es bei jedem Start den Arbeitsbereichsordner zur Sitzung. Ein Arbeitsbereichsordner enthält ...

- Konfigurationsunterordner (z.B. **.metadata**)
- und die Ordner der „internen“ Projekte des Arbeitsbereichs.
  Zu einem Arbeitsbereich können auch „externe“ Projekte gehören, deren Ordner sich *nicht* im Arbeitsbereichsordner befinden.

Hier ist ein Arbeitsbereichsordner mit zwei internen Projekten (**HalloEclipse**, **Prog**) zu sehen:



2.5.2 Eclipse starten

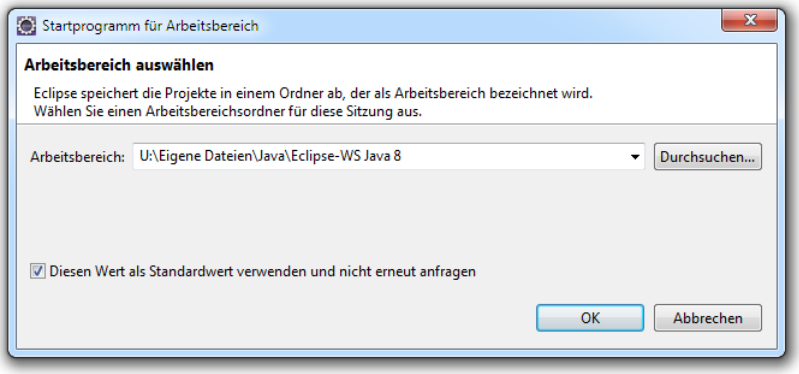
Auf den Pool-PCs an der Universität Trier können Sie die Version 4.5 der **Eclipse IDE for Java EE Developers** (vgl. Abschnitt 2.3) folgendermaßen starten:

Start > Alle Programme > Informatik > Eclipse > Eclipse JEE 4.5

Auf Ihrem eigenen Rechner starten Sie Eclipse über die Datei **eclipse.exe** im Installationsordner oder eine dazu angelegte Verknüpfung (vgl. Abschnitt 2.1).

Während Eclipse geladen wird, ist der folgende Startbildschirm zu sehen:

Beim ersten Start durch einen Benutzer kann etwas Zeit vergehen, weil Konfigurationsordner angelegt werden müssen (vgl. Abschnitt 2.3.7). Schließlich ist der Arbeitsbereichsordner zur Sitzung anzugeben, wobei Sie an einem Pool-PC der Universität Trier einen Ordner auf dem persönlichen Laufwerk U: wählen sollten, z.B.:



Eclipse warnt, wenn der gewählte Arbeitsbereich bereits von einer älteren Eclipse-Version benutzt worden ist:

In diesem Fall ist es wohl besser, den Kompatibilitätsproblemen durch die Wahl eines alternativen Arbeitsbereichsordners aus dem Weg zu gehen.

Wenn Sie Eclipse veranlasst haben, einen **Standardwert** ohne Anfrage zu verwenden, können Sie Ihre Festlegung später so modifizieren:

- Für einen spontanen Wechsel steht in Eclipse der Menübefehl

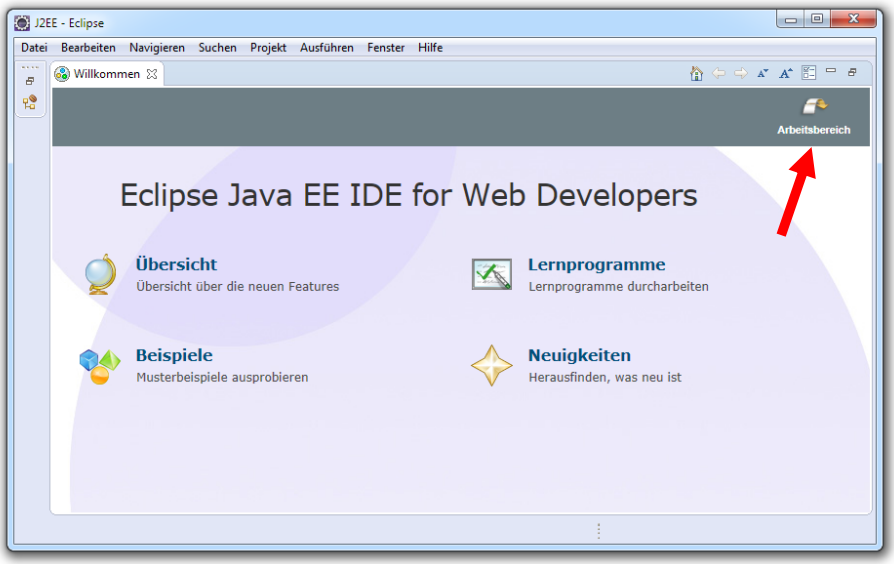
  Datei > Arbeitsbereich wechseln > Andere

  zur Verfügung. Dabei wird Eclipse beendet und mit dem gewählten Arbeitsbereich neu gestartet.
- Mit

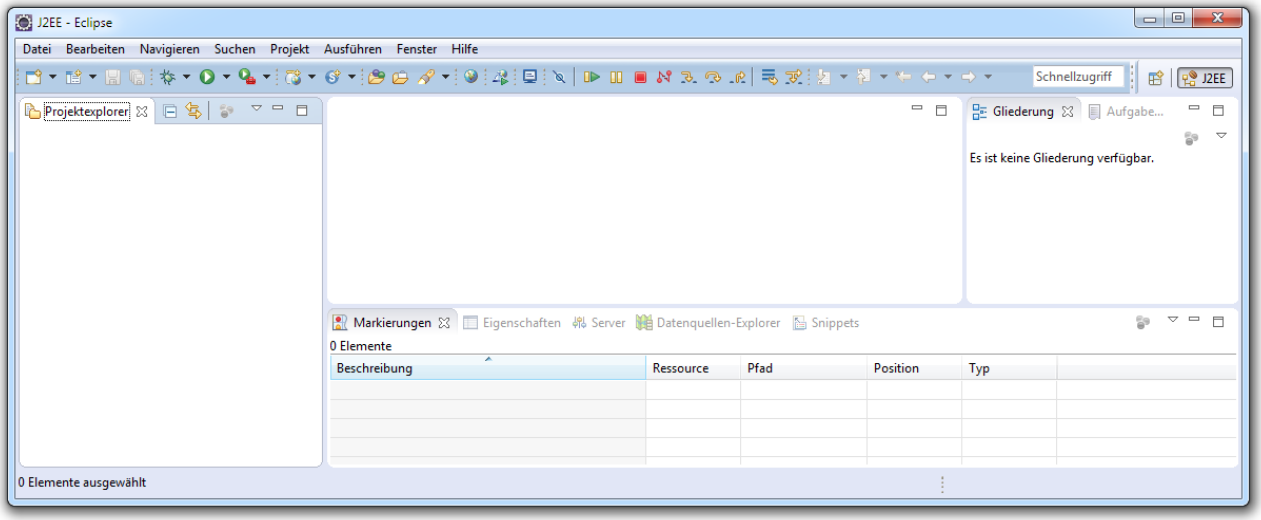
Fenster > Benutzervorgaben > Allgemein > Start und Beendigung > Arbeitsbereiche > Arbeitsbereich bei Start anfordern

reaktivieren Sie die routinemäßige Arbeitsbereichsanfrage beim Start.

Beim ersten Eclipse-Start werden Sie recht eindrucksvoll begrüßt, hier von der Version für JEE-Entwickler:



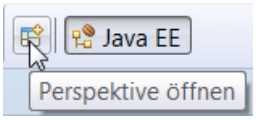
Nach einem Mausklick auf das mit **Arbeitsbereich** beschriftete Symbol erscheint ein Arbeitsplatz mit zahlreichen Werkzeugen für die komfortable und erfolgreiche Software-Entwicklung in Java:



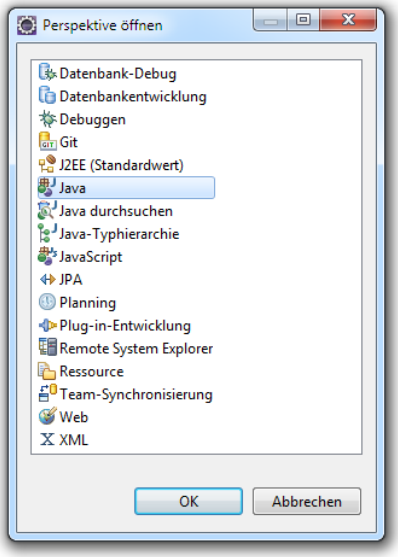
2.5.3 Eine Frage der Perspektive

Die im Eclipse-Fenster enthaltenen Werkzeuge lassen sich in **Sichten** (z.B. zur Anzeige von Projektinhalten oder Übersetzungsfehlern) sowie **Editoren** (z.B. für Quellcode oder XML-Dateien) unterteilen. Unter einer **Perspektive** versteht Eclipse eine Zusammenstellung von Sichten und Editoren.

Beim Eclipse-Paket für JEE-Entwickler ist die Perspektive **Java EE** voreingestellt. Wir wählen stattdessen nach einem Klick auf den Schalter **Perspektive öffnen** (oben rechts)



die momentan besser geeignete Perspektive **Java**:

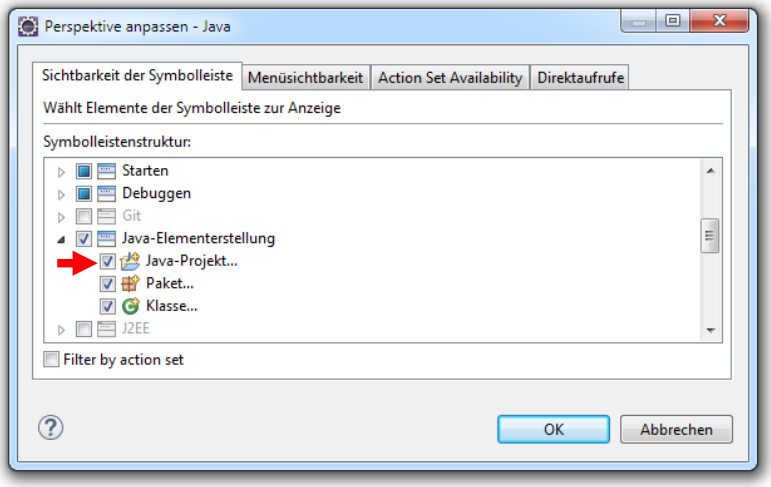


Die Sichten und Editoren einer Perspektive lassen sich flexibel in der Größe ändern, konfigurieren, verschieben oder löschen. Wir nehmen bei der Java-Perspektive zwei Veränderungen vor:

- Symbolschalter **Neues Java-Projekt** nachrüsten
  Um das Anlegen eines neuen Java-Projekts bequem per Symbolschalter veranlassen zu können, modifizieren wir die Symbolleiste **Java-Elementerstellung**. Dazu wählen wir den Menübefehl

  Fenster > Perspektive > Perspektive anpassen

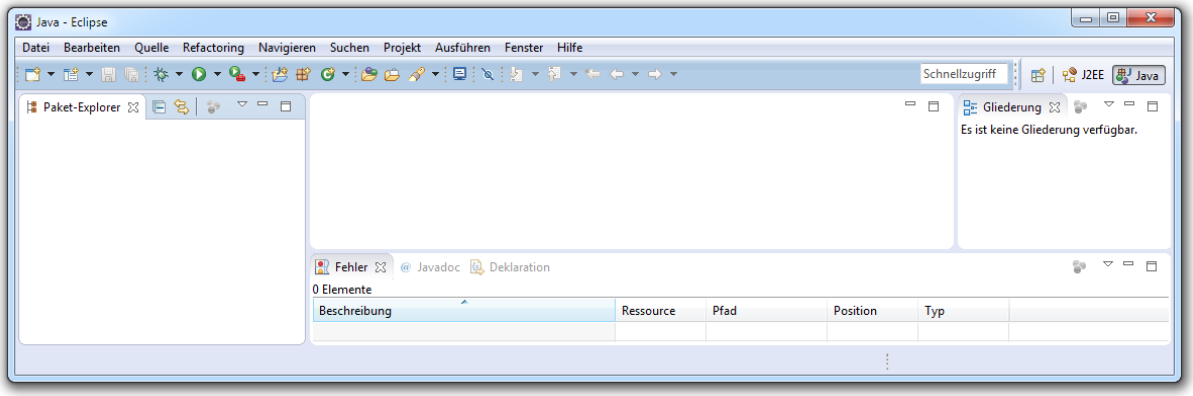
  und markieren im folgenden Dialog



  das Kontrollkästchen **Java-Projekt**.

- **Aufgabenliste** schließen
  Wir schließen die bei kleinen Projekten weniger wichtige **Aufgabenliste** (realisiert von der Eclipse-Erweiterung **Mylyn**). Sie erleichtert bei großen Projekten die Konzentration auf die aktuelle Teilaufgabe, indem irrelevante Informationen ausgeblendet werden.

So erhalten wir eine funktionale und aufgeräumte Perspektive zur Java-Entwicklung:



Nach einer verunglückten Konfiguration kann man den Originalzustand einer Perspektive über den Menübefehl

Fenster > Perspektive > Perspektive zurücksetzen

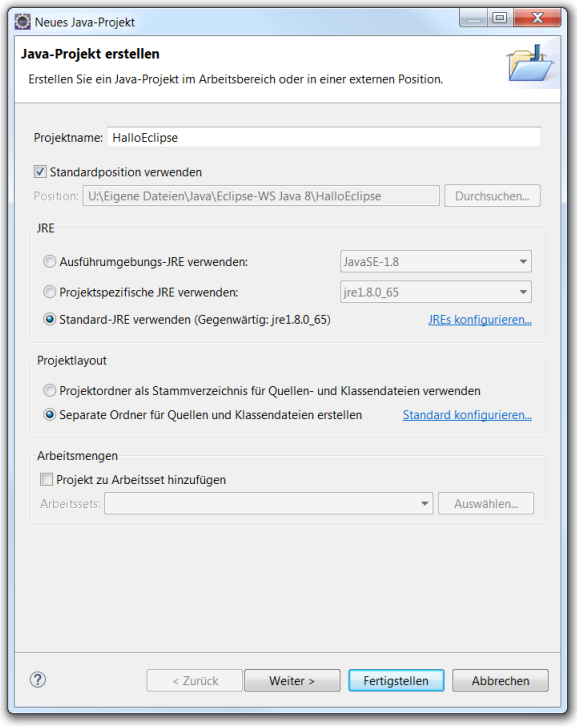
restaurieren.

2.5.4 Neues Projekt anlegen

In diesem Abschnitt erstellen wir mit Eclipse ein minimalistisches Java-Programm vom Hallo-Typ analog zum Abschnitt 2.2, wo wir dieselbe Aufgabe mit Notepad und dem JDK-Compiler **javac.exe** erledigt haben. Wir starten mit dem Symbolschalter oder dem Menübefehl

Datei > Neu > Java-Projekt

den Assistenten für neue Java-Projekte und legen im folgenden Dialog den **Projektnamen** fest, z.B.:



Im Rahmen **JRE** verwenden wir die für den Arbeitsbereich eingestellte **Standard-JRE**, also per Voreinstellung dieselbe JRE, mit der Eclipse gestartet worden ist. Damit sind der Java-

Sprachumfang (die Kompatibilitätsstufe des Compilers) und die Standardbibliothek für das neue Projekt festgelegt. Im Abschnitt 2.5.8.3 wird empfohlen, ein JDK als Standard-JRE für den Arbeitsbereich einzustellen. Bei dieser Konstellation lokalisiert Eclipse bei einem Laufzeitfehler die betroffenen Quellcodezeilen auch zu den Methoden aus den Bibliotheksklassen, was die Analyse von Programmierfehlern erleichtern kann.

Damit das entstehende Programm auf dem Rechner eines Anwenders genutzt werden kann, muss die dort installierte JRE mindestens denselben Versionsstand besitzen. Mit Rücksicht auf Kunden mit veralteter JRE-Version, die nicht zum Update gezwungen werden sollen, kann es sinnvoll sein, für ein Projekt eine ältere JRE-Version einzustellen. Im Rahmen **JRE** des Dialogs für neue Projekte können Sie zwischen den folgenden Optionen wählen:

- **Ausführungsumgebungs-JRE verwenden**
  Hier lässt sich eine JRE einstellen, die auf dem Entwicklungssystem nicht installiert ist, aber durch eine hier vorhandene, neuere und kompatible JRE-Version unterstützt wird.
- **Projektspezifische JRE verwenden**
  Man kann man zwischen den von Eclipse erkannten JRE-Installationen auf dem lokalen Rechner wählen. In Abschnitt 2.5.8.3 ist zu erfahren, wie man Eclipse über eine nicht automatisch erkannte JRE-Installation informiert.
- **Standard-JRE verwenden**
  Es wird die Standard-JRE des Arbeitsbereichs gewählt. Dies ist per Voreinstellung die JRE, mit der Eclipse gestartet worden ist. Im Abschnitt 2.5.8.3 wird beschrieben, wie die Standard-JRE des Arbeitsbereichs geändert werden kann.

Generell sind keine nennenswerten Probleme mit der JRE-Version auf den Rechnern der Anwender zu erwarten, denn:

- Die Anwender können kostenlos die passende JRE-Version installieren.
- Man kann zusammen mit einem Programm eine spezielle JRE ausliefern. Dies wird von der Firma Oracle explizit erlaubt (Oracle Binary Code License Agreement for the Java SE Platform).[1]

Im Rahmen **Projektlayout** geht es um die Struktur des Projektordners, der per Voreinstellung im Arbeitsbereichsordner angelegt wird, im Beispiel also in:

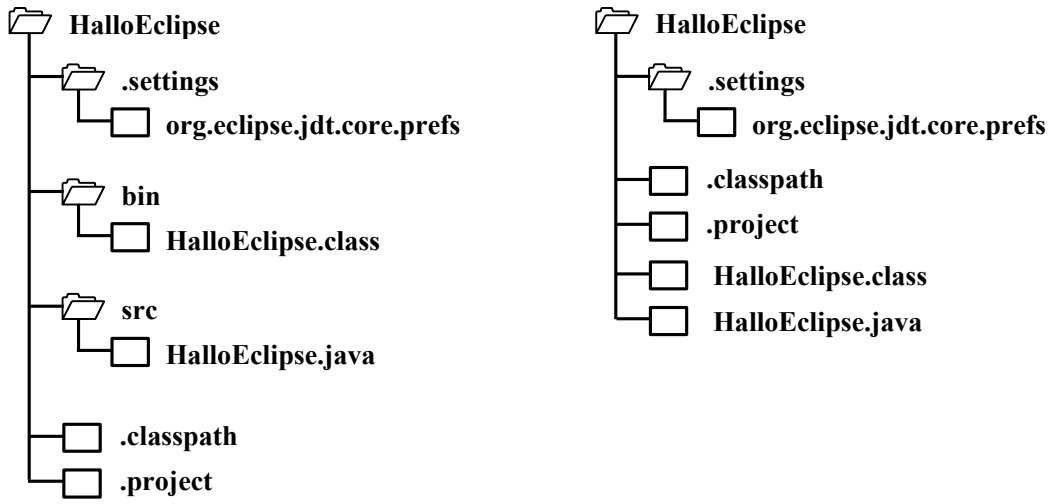
U:\Eigene Dateien\Java\ Eclipse-WS Java 8

Aus der Voreinstellung

Separate Ordner für Quellen- und Klassendateien erstellen

resultiert das linke Projektlayout:

[1] Siehe http://www.oracle.com/technetwork/java/javase/jre-8-readme-2095710.html



Aus der alternativen Option

Projektordner als Stammverzeichnis für Quellen- und Klassendateien verwenden

resultiert das rechte Projektlayout. Beim aktuell entstehenden Projekt verwenden wir das voreingestellte Layout mit Unterordnern für die Quellcode- und die Klassendateien, das aufgrund der insgesamt sehr geringen Anzahl von Dateien etwas übertrieben zergliedert wirkt. Bei vielen anderen, ähnlich simplem Beispielprojekten zum Manuskript wird das flachere Projektlayout verwendet.

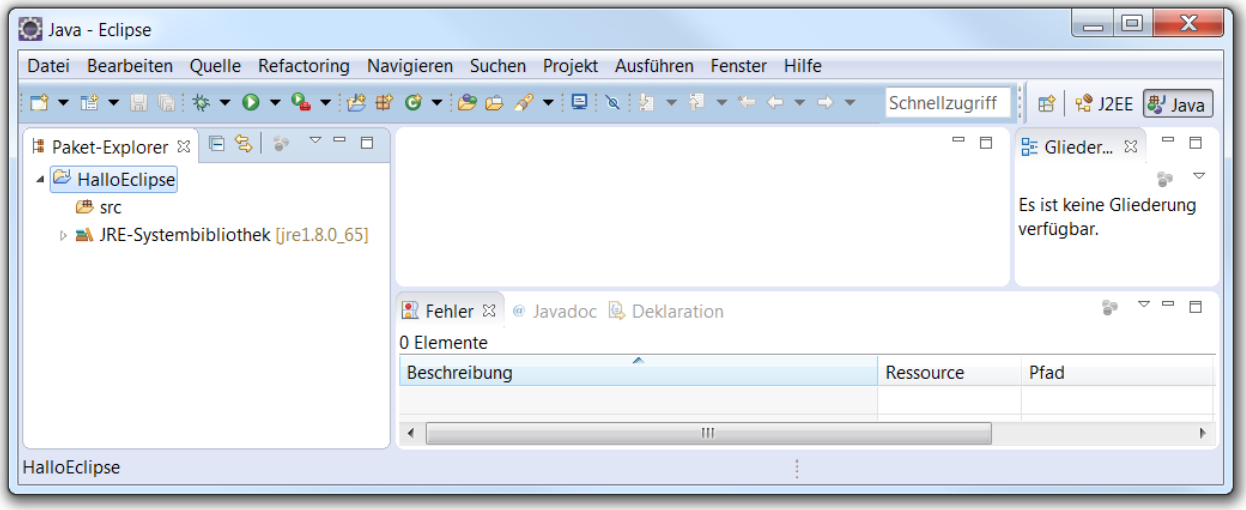
Auf die Möglichkeit, **Arbeitsmengen** (engl.: Working Sets) mit einer Teilmenge von Projekten zu definieren, um z.B. manche Suchvorgänge durch Einschränkung beschleunigen zu können, verzichten wir.

Für das Einstiegsprojekt können Sie den Assistenten jetzt mit **Fertigstellen** beenden und damit alle weiteren Dialoge ignorieren. Viele Einstellungen eines Projektes (z.B. das Compiler-Niveau) sind später über den Menübefehl

Projekt > Eigenschaften

zu ändern.

Das neue Projekt erscheint im **Paket-Explorer**:

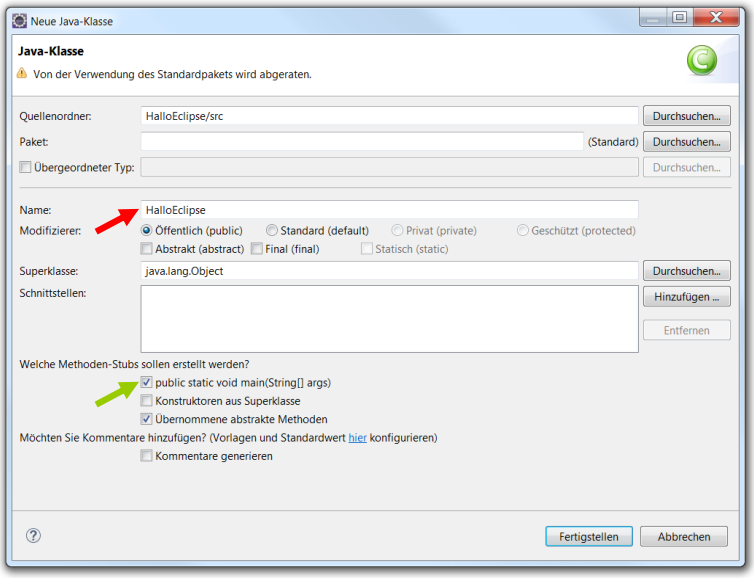


2.5.5 Klasse hinzufügen

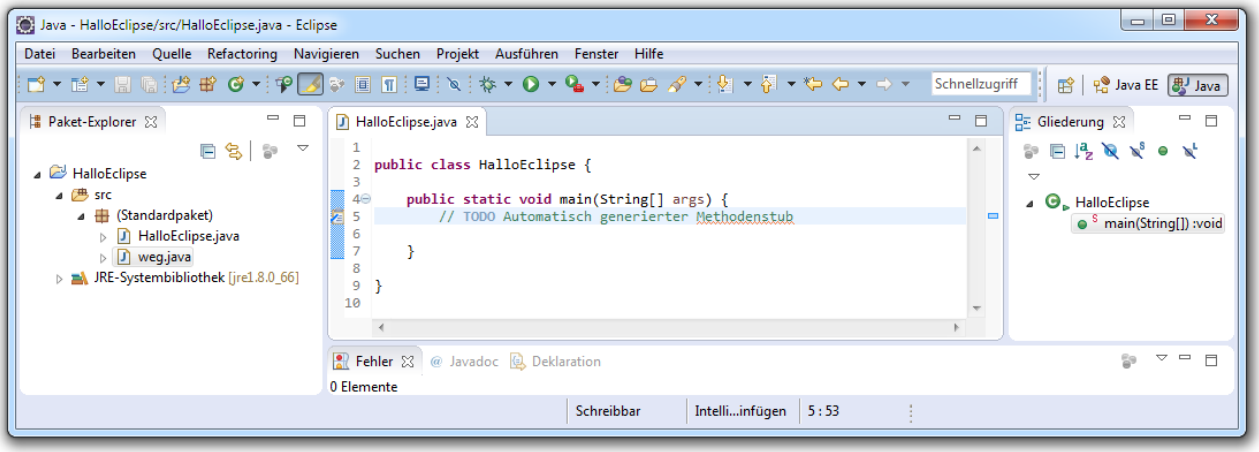
Wir starten über den Symbolschalter oder den Menübefehl

Datei > Neu > Klasse

die Definition einer neuen Klasse, legen im folgenden Dialog deren Namen fest (roter Pfeil), behalten den **Modifizierer public** sowie die **Superklasse java.lang.Object** bei, lassen einen **main()**-Methodenrohling automatisch anlegen (grüner Pfeil) und ignorieren die (bei großen Projekten sehr berechtigte) Kritik an der Verwendung des Standardpakets:[1]



Nach dem **Fertigstellen** befindet sich auf der Werkbank ein fast komplettes POO - Hallo-Programm:



Wenn Eclipse in der Kommentarzeile 5 die Textpassage „Automatisch generierter“ wegen fehlerhafter Rechtschreibung kritisiert, haben Sie das in Abschnitt 2.3.6 beschriebene deutsche Wörterbuch noch nicht installiert bzw. für den aktuellen Eclipse-Arbeitsbereich konfiguriert.

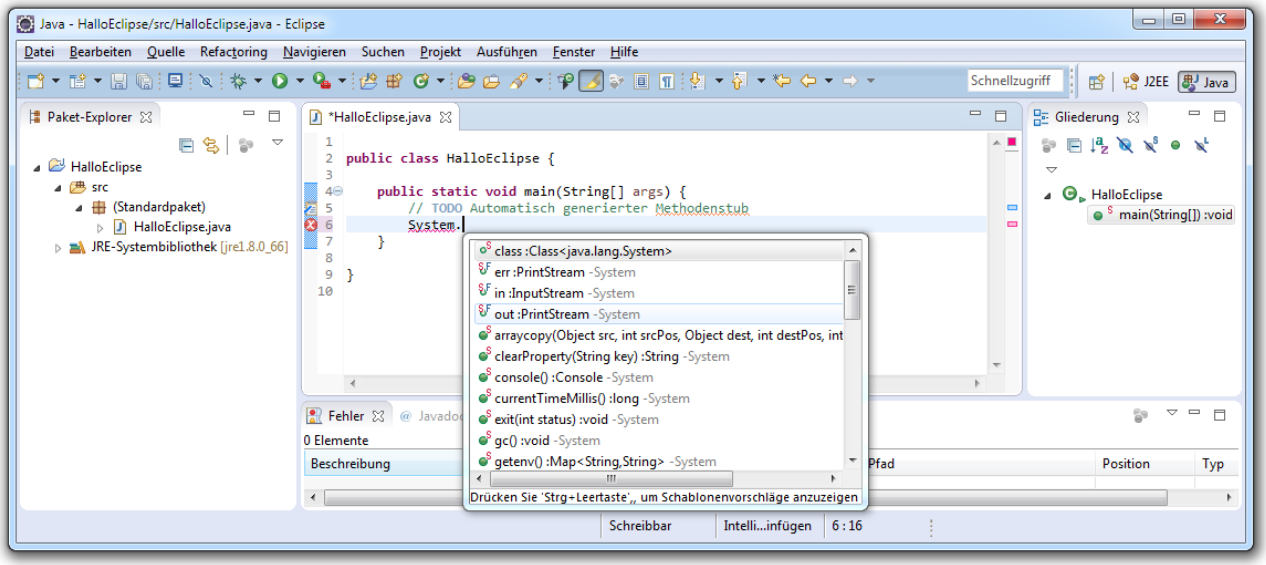
[1] Mit Paketen werden wir uns später ausführlich beschäftigen.

2.5.6 Quellcode mit Eclipse-Hilfe erstellen

Um das in Eclipse erstellte Hallo-Programm zu vollenden, müssen wir noch im Editor die Ausgabeanweisung

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

verfassen (vgl. Abschnitt 2.2.1). Dabei ist die Syntaxvervollständigung von Eclipse eine große Hilfe. Wir setzen in die Zeile 6 aus optischen Gründen zum Einrücken ein Tabulatorzeichen (mit der Taste ⇥) und schreiben den Klassennamen **System**.[1] Sobald wir einen Punkt hinter den Klassennamen setzen, erscheint eine Liste mit allen zulässigen Fortsetzungen, wobei wir uns im Beispiel für die Klassenvariable **out** entscheiden, die auf ein Objekt der Klasse **PrintStream** zeigt.



Wir übernehmen das Ausgabeobjekt per Doppelklick in den Quellcode und setzen einen Punkt hinter seinen Namen (**out**). Jetzt werden u.a. die Instanzmethoden der Klasse **PrintStream** aufgelistet, und wir wählen per Doppelklick die Variante der Methode **println()** mit einem Parameter vom Typ **String**. Ein durch doppelte Hochkommata begrenzter Text komplettiert den Methodenaufruf, den wir objektorientiert als Nachricht an das Objekt **System.out** auffassen.

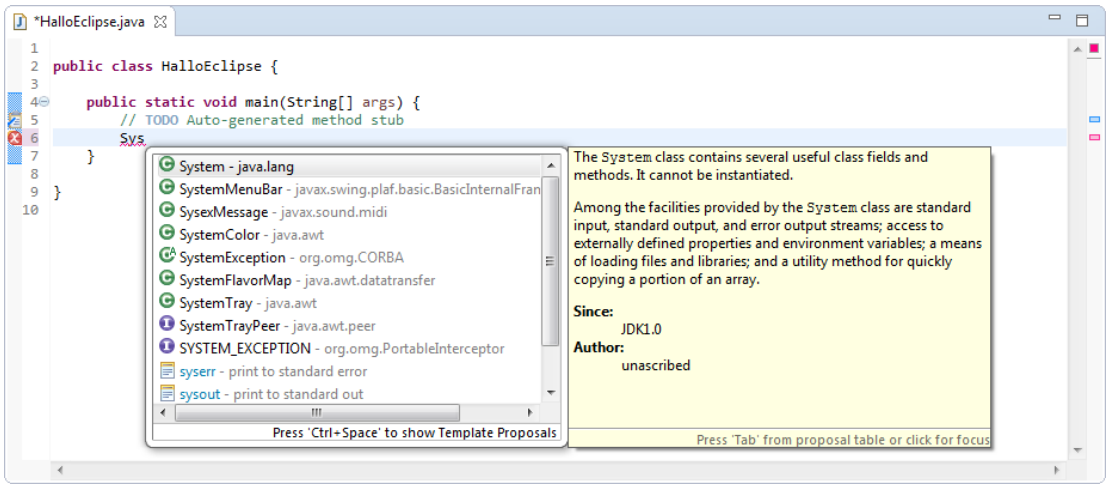
Nachdem wir einen Punkt hinter den Klassennamen gesetzt haben, hat Eclipse die Syntaxvervollständigung angeworfen spontan. Mit dem Tastaturkommando

Strg + Leertaste

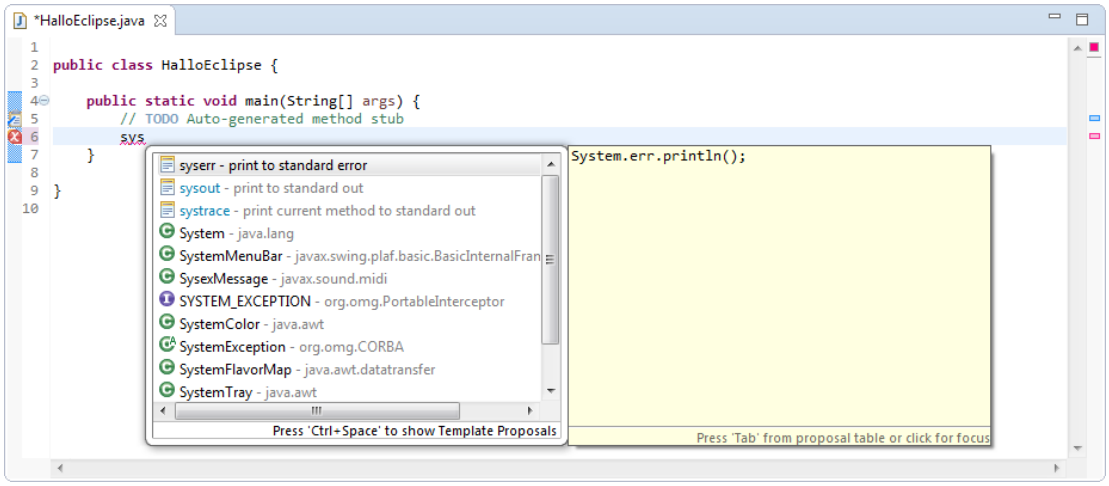
lässt sich die Syntaxvervollständigung explizit anfordern, was z.B. dann sinnvoll und erforderlich ist, wenn Eclipse zu einem Namensanfang die möglichen Fortsetzungen auflisten soll, z.B.:

---

[1] Bei Bedarf lassen sich die Zeilennummern folgendermaßen einschalten:
Fenster > Benutzervorgaben > Allgemein > Editoren > Texteditoren > Zeilennummern anzeigen



Mit den **Code-Vorlagen** bietet Eclipse eine Unterstützung beim Verfassen von längeren, routinemäßig benötigten Code-Passagen an. Man tippt den Namen bzw. Namensanfang der Vorlage an, wobei es nicht um Java-Syntax handelt, sondern um eine von Eclipse definierte Bezeichnung, und fordert mit **Strg + Leertaste** die Vervollständigung bzw. die Liste mit den kompatiblen Fortsetzungen an. Wenn Sie im letzten Beispiel die Code-Erweiterung zu „sys“ (kleines *s* am Anfang) anfordern, stehen am Anfang der angebotenen Liste möglicher Ergänzungen drei Vorlagen mit einer Vorschau des damit zu erzeugenden Codes:



Wenn Sie den Vorlagenname **sysout** komplett schreiben, erstellt Eclipse nach **Strg + Leertaste** sofort den nahezu kompletten Methodenaufruf:

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

Ihnen verbleibt nur noch die Aufgabe, die gewünschte Ausgabe in die Parameterliste zu schreiben.

Analog erstellt man z.B. mit der Vorlage **main** einen Rohling für die Startmethode **main()**:

```
public static void main(String[] args) {

}
```

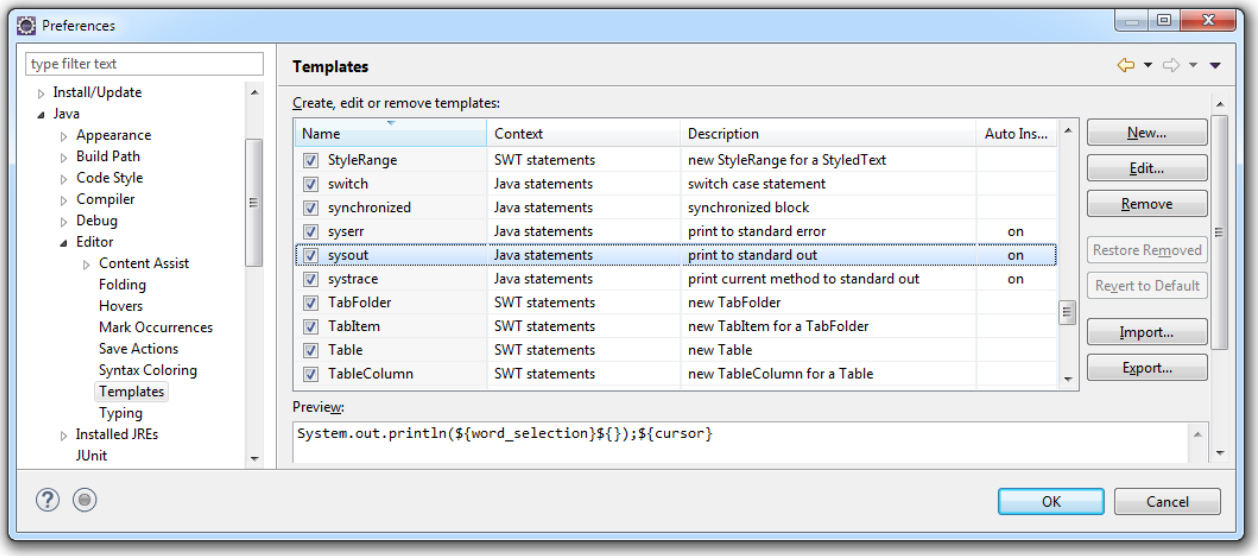
In der Eclipse-Hilfe befindet sich eine Liste mit den vordefinierten Code-Vorlagen. Weil die Hilfe zur deutschen Lokalisierung von Eclipse einige Lücken aufweist, besteht nun der erste Anlass, die in Abschnitt 2.3.4 beschriebe Möglichkeit zum Eclipse-Start in englischer Sprache zu nutzen. Auf den ZIMK-Pool-PCs ist zu diesem Zweck ein separater Link vorhanden:

Start > Alle Programme > Informatik > Eclipse > Eclipse 4.5 (Mars) > Eclipse JEE 4.5 (engl.)

Mit der US-Lokalisation von Eclipse können Sie nach dem folgenden Menübefehl

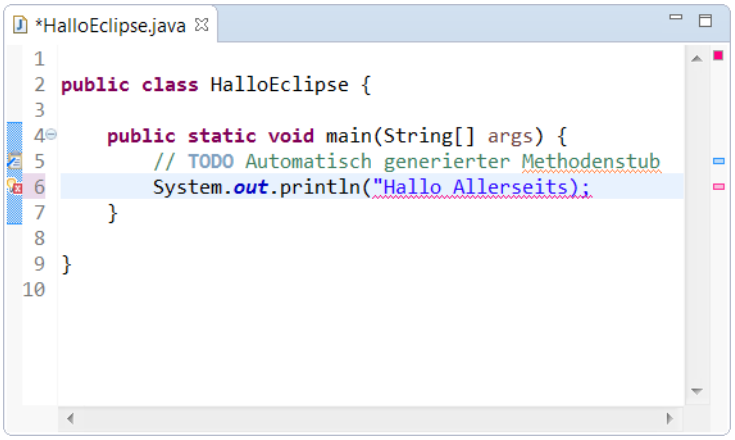
Window > Preferences > Java > Editor > Templates

die Vorlagen (engl.: *templates*) einsehen, editieren und ergänzen:



2.5.7 Übersetzen und Ausführen

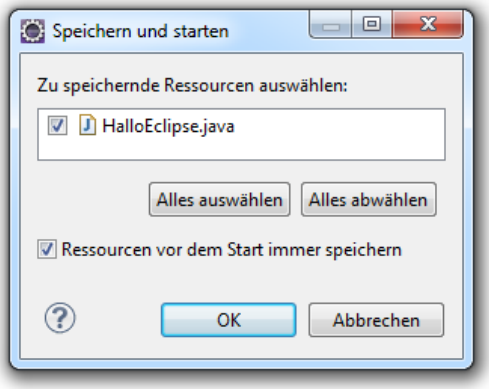
Analog zu einem Textverarbeitungsprogramm mit Rechtschreibungskontrolle informiert Eclipse während der Eingabe über Syntaxfehler, z.B.:



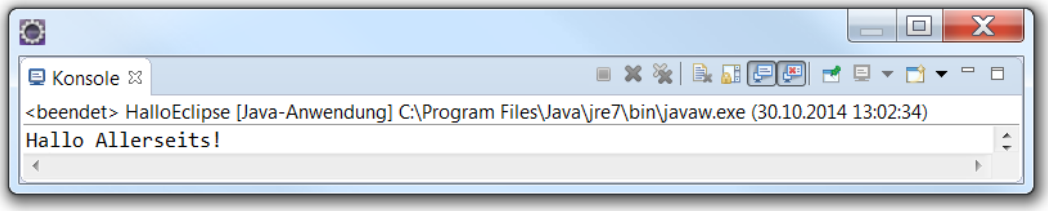
Hier ist ein inkrementeller Compiler am Werk, der schon während der Eingabe die Einhaltung der Java-Syntaxregeln überwacht.

Den Start unserer (möglichst fehlerfrei kodierten) Anwendung veranlassen wir mit Schalter oder die Tastenkombination **Strg+F11**.[1]

Falls Sie Ihr Projekt noch nicht gespeichert haben, können Sie dies jetzt tun und auch über ein Kontrollkästchen veranlassen, dass in Zukunft geänderter Quellecode grundsätzlich vor dem Programmstart gesichert wird:



Die Ausgabe des Programms erscheint in der Sicht **Konsole**, die sich wie viele andere Werkbankbestandteile verschieben oder abtrennen lässt, z.B.:

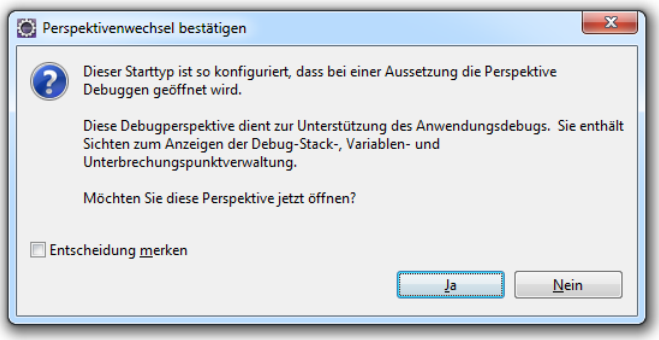


Um das Konsolenfenster wieder zu verankern, packt man seine Beschriftung (**Konsole**) mit der linken Maustaste und zieht diese an den gewünschten Ankerplatz.

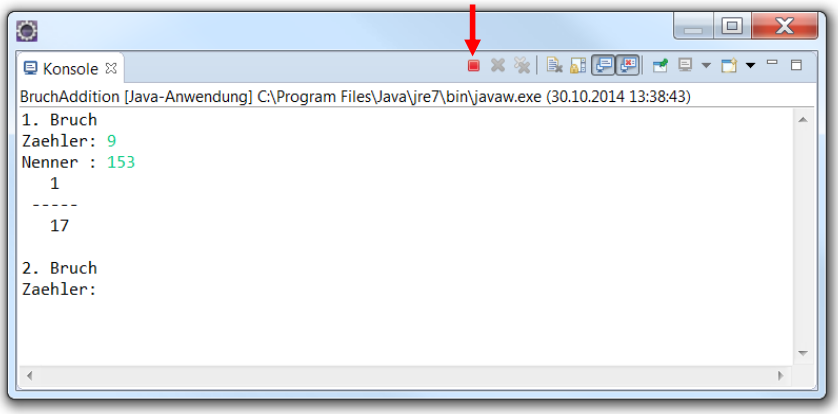
Ein im Rahmen von Eclipse gestartetes Programm (mit Konsolen und/oder GUI-Bedienoberfläche) lässt sich über den roten Stopp-Schalter in der Symbolleistenzone der Konsole beenden, z.B.:

---

[1] Beim Starten per Tastatur kann man auf die Vorschalttaste **Strg** verzichten, also nur **F11** drücken. Dies bewirkt einen Start im so genannten Debug - Modus, der zur Fehlersuche konzipiert ist (vgl. Abschnitt 4.3.3). Eclipse muss in diesem Modus einigen Zusatzaufwand betreiben, um beim Auftreten eines Problems nützliche Informationen anbieten zu können. Bei unseren Programmen ist von diesem Zusatzaufwand nichts zu spüren, so dass wir uns den bequemeren Start über die Solo-Taste **F11** erlauben können. Wenn allerdings nach einem, **F11**-Start ein Fehler auftritt, bietet Eclipse den Wechsel zu einer für die Fehlersuche optimierten Perspektive (Werkzeugausstattung) an, was auf unserem Ausbildungsstand mehr irritiert als nutzt:



Daher ist vorläufig zum Starten per Tastatur die Kombination **Strg+F11** besser geeignet.



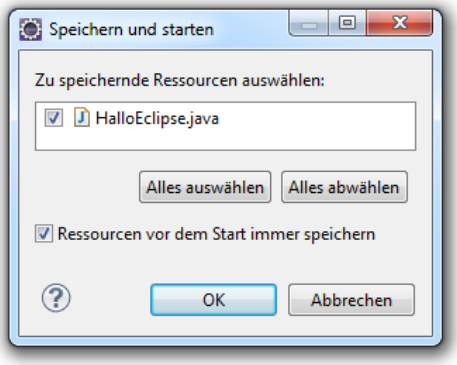
Wenn die Konsole (in der Eclipse-Terminologie eine so genannte *Sicht*) abhandengekommen ist, kann sie mit folgendem Menübefehl reaktiviert werden:

Fenster > Sicht anzeigen > Konsole

2.5.8 Einstellungen ändern

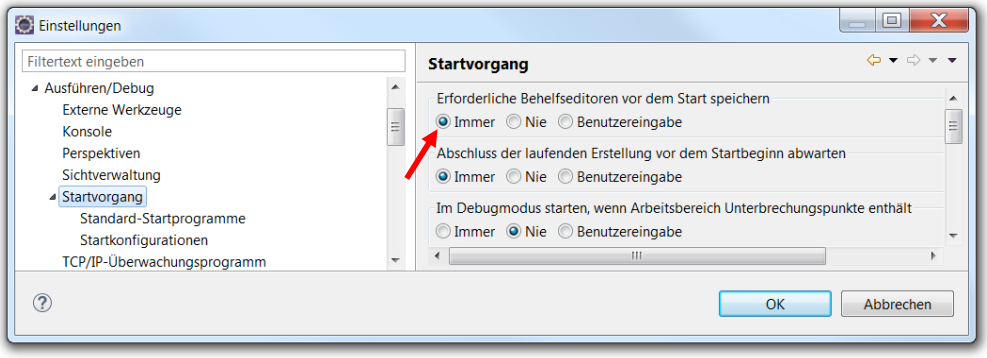
2.5.8.1 Automatische Quellcodesicherung beim Ausführen

Wie Sie eben im Zusammenhang mit einem Übungsprojekt festgestellt haben, bietet Eclipse beim Starten eines geänderten Quellcodes das vorherige Sichern an, z.B.:



Wenn Sie bei einer solchen Gelegenheit das regelmäßige Sichern veranlasst haben, können Sie diese Einstellung folgendermaßen widerrufen:

- **Fenster > Benutzervorgaben > Ausführen/Debug > Startvorgang**
- Wählen Sie im Optionsfeld **Erforderliche Befehlseditoren vor dem Starten speichern** den gewünschten Wert:

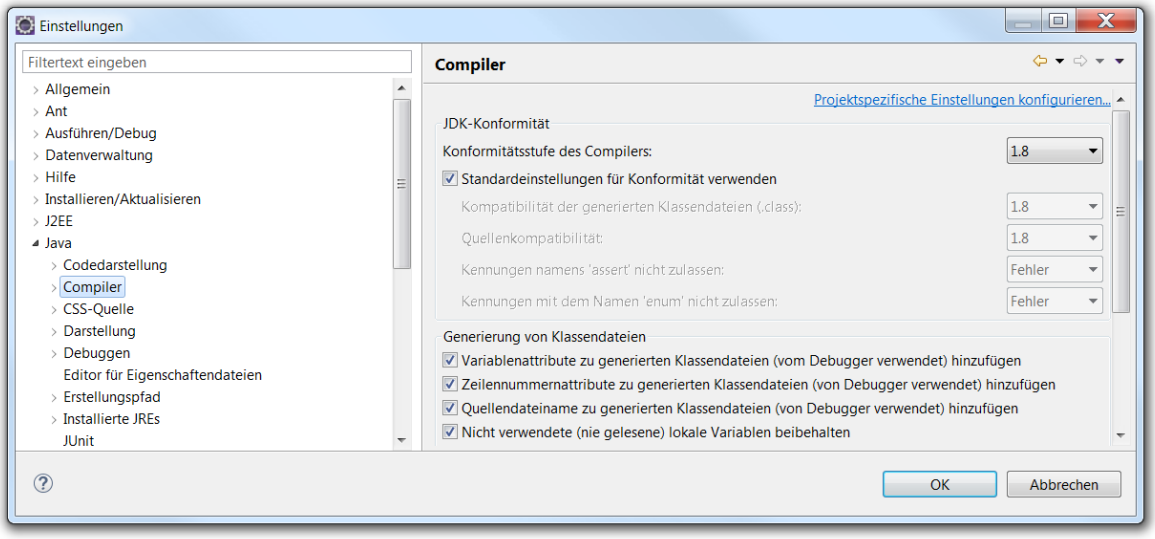


2.5.8.2 Konformitätsstufe des Compilers

Der in Eclipse 4.5.1 enthaltene Compiler unterstützt das aktuelle Niveau 8 (bzw. 1.8) der Programmiersprache Java, lässt sich aber auch auf frühere Versionen einstellen. Über

Fenster > Benutzervorgaben > Java > Compiler

wählt man eine Einstellung mit Gültigkeit für alle Projekte *im aktuellen Arbeitsbereich*, für die kein spezielles Compiler-Niveau angeordnet wird:



In unserer Situation ist es empfehlenswert, mit der Java-Version 8 zu arbeiten.

Wenn viele Kundenrechner zu versorgen sind und die dort vorhandene JVM zu verwenden ist, sollte man vor dem Einsatz einer neuen Version einige Monate verstreichen lassen. Für ein konkretes Projekt kann man über

Projekt > Eigenschaften > Java-Compiler

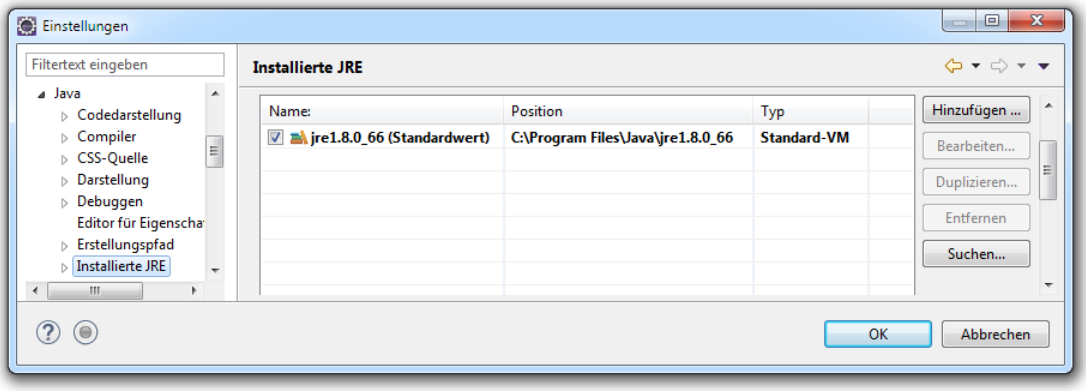
die **projektspezifischen Einstellungen aktivieren** und dann eine **Konformitätsstufe des Compilers** wählen.

2.5.8.3 JRE wählen

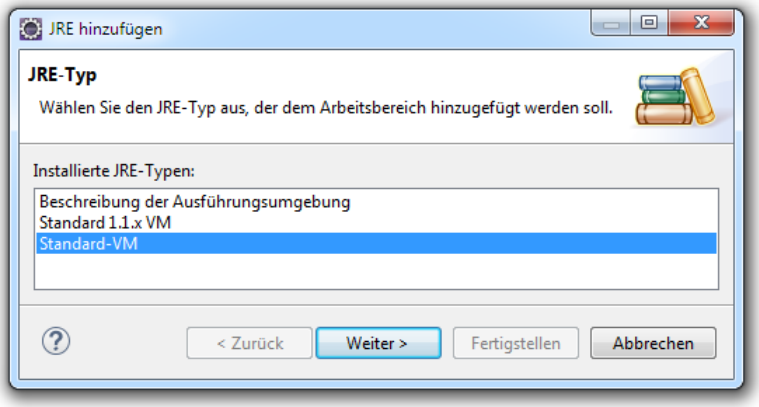
Sind auf Ihrem Rechner mehrere Java Runtime Environments (JREs) vorhanden, können Sie nach

Fenster > Benutzervorgaben > Java > Installierte JREs

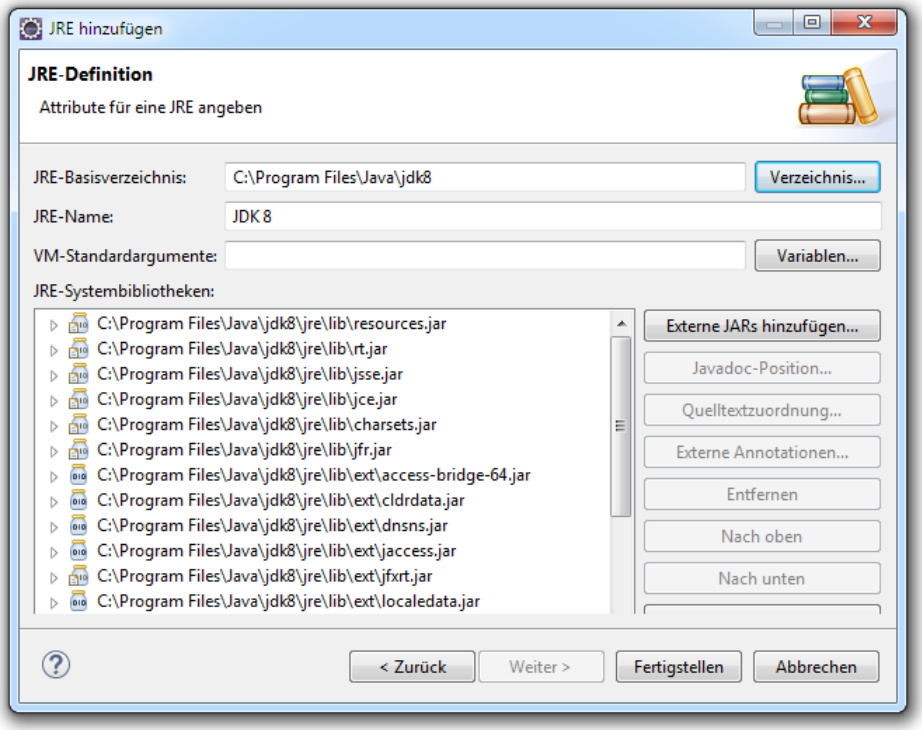
der von Eclipse automatisch erkannten Auswahl



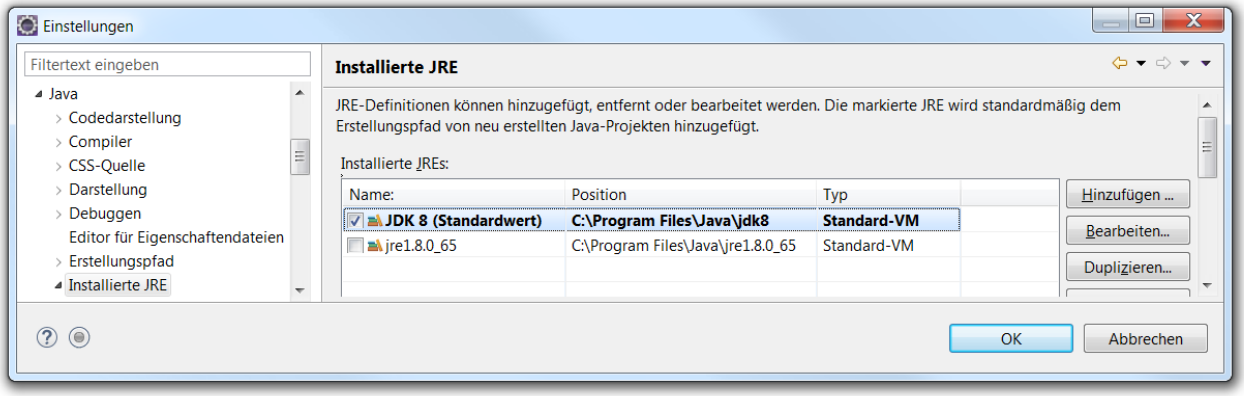
weitere Exemplare **hinzufügen**. Nach Wahl des JRE-Typs **Standard-VM**



können Sie z.B. die in einem installierten JDK (vgl. Abschnitt 2.1) enthaltene Laufzeitumgebung ergänzen:



Außerdem lässt sich die Standard-JRE zum Arbeitsbereich festlegen, z.B.:



Wird ein JDK als Laufzeitumgebung verwendet, kann bei einem Laufzeitfehler (tritt nicht beim Übersetzen auf, sondern bei der Ausführung) die Unfallstelle auch im API-Quellcode lokalisiert werden, 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 HalloEclipse.main(HalloEclipse.java:3)
```

Per Mausklick lässt sich eine Bibliotheksmethode sogar im Eclipse-Editor öffnen, um die Gründe und die Umgebung eines Fehlers zu analysieren.

Ist eine JRE *ohne* begleitenden Quellcode im Einsatz, erscheint bei API-Methoden in der Aufrufersequenz zu einem Lauzeitfehler **Unknown Source** statt einer Ortsangabe (aus Dateinamen und Zeilennummer), z.B.:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "vier"
  at java.lang.NumberFormatException.forInputString(Unknown Source)
  at java.lang.Integer.parseInt(Unknown Source)
  at java.lang.Integer.parseInt(Unknown Source)
  at HalloEclipse.main(HalloEclipse.java:3)
```

Es ist nur in seltenen Fällen erforderlich, zur Aufklärung eines Fehlers auch API-Quellcode zu inspizieren. Weil diese Option aber prinzipiell nützlich und außerdem kostenlos ist, empfiehlt es sich die Verwendung eines JDKs als Laufzeitumgebung.

2.5.8.4 Kodierung von Textdateien

Per Voreinstellung verwendet Eclipse für Textdateien (z.B. mit Java-Quellcode) eine vom Betriebssystem abhängige Kodierung:

- Linux: **UTF-8**
- MacOS: **MacRoman**
- Windows: **Cp1252**

Das führt zu Problemen, wenn die an einem Projekt beteiligten Entwickler mit verschiedenen Betriebssystemen bzw. Textkodierungen arbeiten, wie das folgende Editor-Fenster zeigt:

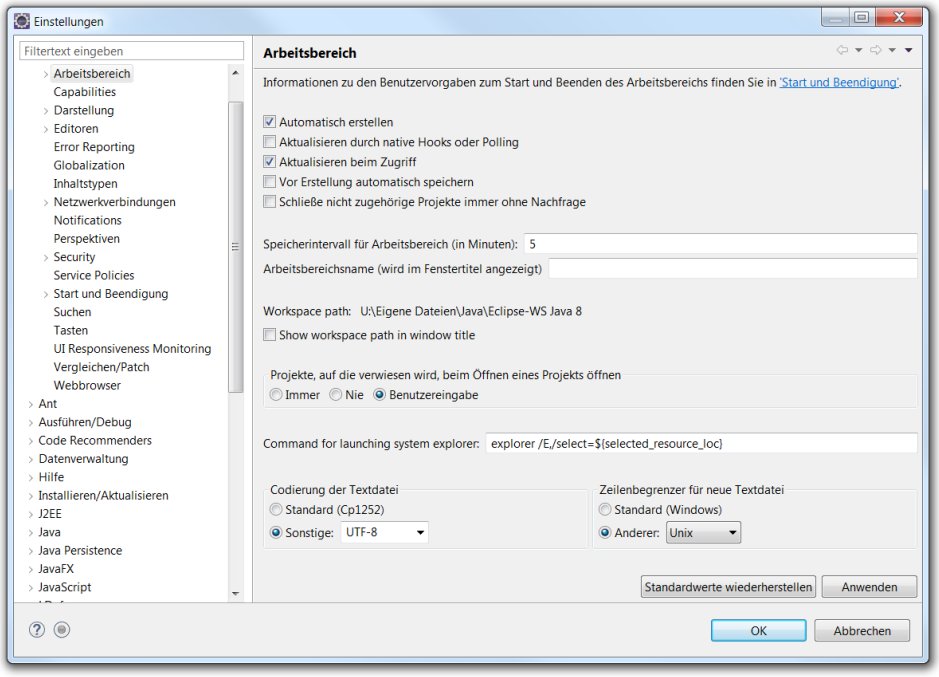
```
Fakul.java
1  import javax.swing.JOptionPane;
2  class Fakul {
3      public static void main(String[] args) {
4          int argument = -1;
5          boolean argOK = false;
6
7          do {
8              try {
9                  String s = JOptionPane.showInputDialog(null, "Argument:",
10                         "Fakult@tsberechnung", JOptionPane.QUESTION_MESSAGE);
11                 if (s == null)
12                     System.exit(0);
13                 argument = Integer.parseInt(s);
14                 if (argument < 0  || argument > 170)
15                     JOptionPane.showMessageDialog(null, "Es sind nur ganze Zahlen von 0 bis 170 erlaubt.",
16                             "Fakult@tsberechnung", JOptionPane.ERROR_MESSAGE);
17                 else
```

Neben Quellcodedateien sind z.B. auch Dateien mit Ressourcen oder Einstellungen betroffen.

Als gemeinsamer Kodierungsstandard ist UTF-8 eindeutig zu präferieren. Nach dem folgenden Menübefehl

Fenster > Benutzervorgaben > Allgemein > Arbeitsbereich

können Sie diese **Codierung der Textdatei** mit Gültigkeit *für den aktuellen Arbeitsbereich* einstellen:



Bei dieser Gelegenheit sollten Sie zur Verbesserung der plattformübergreifenden Interoperabilität die **Zeilenbegrenzer für neue Textdateien** auf den **Unix**-Standard umstellen.

Damit sind die Textdatei-Kompatibilitätsprobleme natürlich nicht universell gelöst, weil es unter Ihren jetzigen oder zukünftigen Kooperationspartnern Eclipse-Nutzer geben könnte, die unter Windows arbeiten und die Eclipse-Voreinstellungen für die Textkodierung und die Zeilenbegrenzung verwenden.

Um für ein einzelnes Projekt die angenommene Textkodierung oder Zeilenbegrenzung einzustellen, öffnen Sie über das Item **Eigenschaften** aus dem Kontextmenü zum Projekteintrag im **Paket-Explorer** den Eigenschaftsdialog und ändern auf der Seite **Ressource** die **Codierung der Textdatei** bzw. die **Zeilenbegrenzer**. Unter den **sonstigen** Textkodierungen hat **ISO-8859-1** die größte Ähnlichkeit zu **Cp1252**.

Um für ein markiertes Projekt die Zeilenbegrenzer auszutauschen, wählen Sie den Menübefehl

Datei > Zeilenbegrenzer umwandeln in

Für die Eclipse-Projekte zum Kurs bzw. Manuskript wurde die Textkodierung auf **UTF-8** und die Zeilenbegrenzung auf den **Unix**-Standard eingestellt, damit die Plattformunabhängigkeit von Java nicht durch die Entwicklungsumgebung eingeschränkt wird. Kursteilnehmer unter MacOS oder Windows müssen die Textkodierung für den benutzten Arbeitsbereich oder projektspezifisch anpassen, um Zeichensalat zu vermeiden. Bei den Zeilenbegrenzungen ist zumindest unter Windows keine Anpassung erforderlich, weil Eclipse mit beiden Normen (Unix, Windows) zurechtkommt.

2.5.9 Projekte importieren

Die Beispiele im Manuskript und Lösungsvorschläge zu vielen Übungsaufgaben sind als Eclipse-Projekte an der im Vorwort beschriebenen Stelle zu finden, wobei aber aus folgenden Gründen *keine* Arbeitsbereichsordner angeboten werden:

- Diese erreichen (insbesondere durch den Unterordner **.metadata**) eine beträchtliche Größe.
- Arbeitsbereiche sind aufgrund absoluter Pfadangaben schlecht portabel.

Erfreulicherweise sind die Projektordner klein, nicht durch absolute Pfadangeben belastet und außerdem sehr flott in einen Arbeitsbereich zu importieren. Wir üben den Import am Beispiel eines zum Einführungsbeispiel (vgl. Abschnitt 1.1) passenden Bruchadditionsprojekts, das sich im folgenden Ordner befindet:

…\BspUeb\Einleitung\Bruchaddition\KonsoleEclipse

Kopieren Sie den Projektordner auf einen eigenen Datenträger, z.B. als

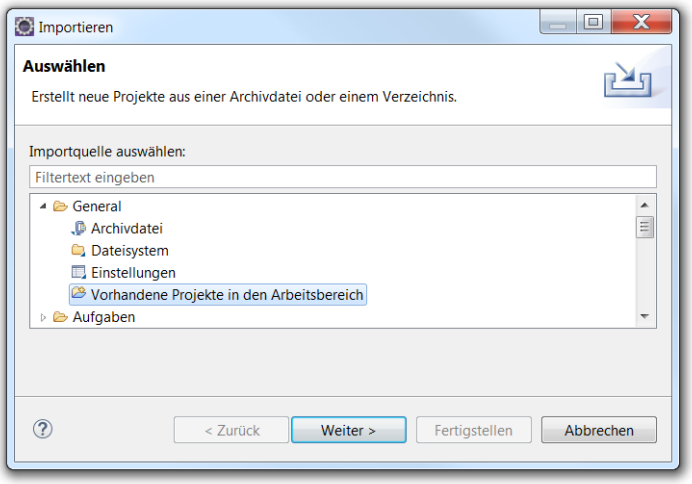
U:\Eigene Dateien\Java\BspUeb\Einleitung\Bruchaddition\KonsoleEclipse

Starten Sie Eclipse mit Ihrem persönlichen Arbeitsbereich, z.B. in

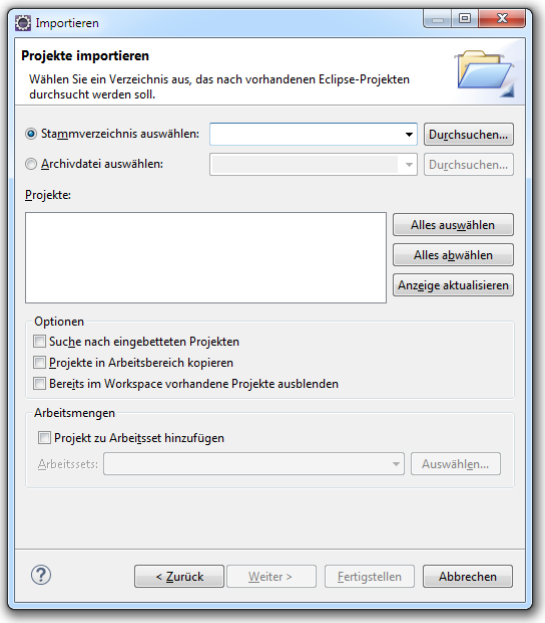
U:\Eigene Dateien\Java\Eclipse-WS Java 8

Initiieren Sie den Import mit

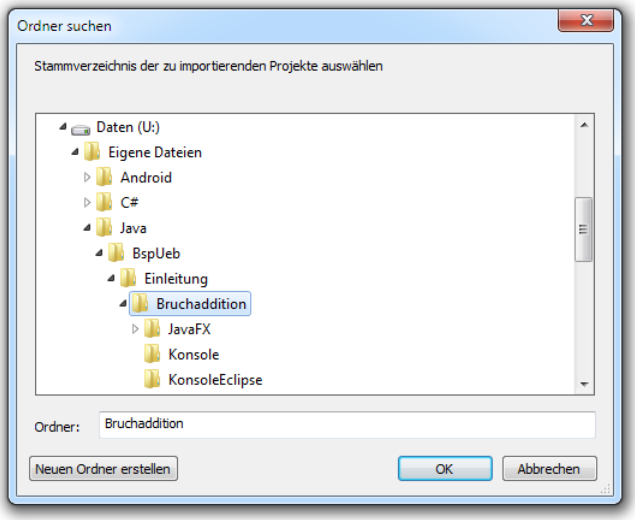
Datei > Importieren > General > Vorhandene Projekte in den Arbeitsbereich



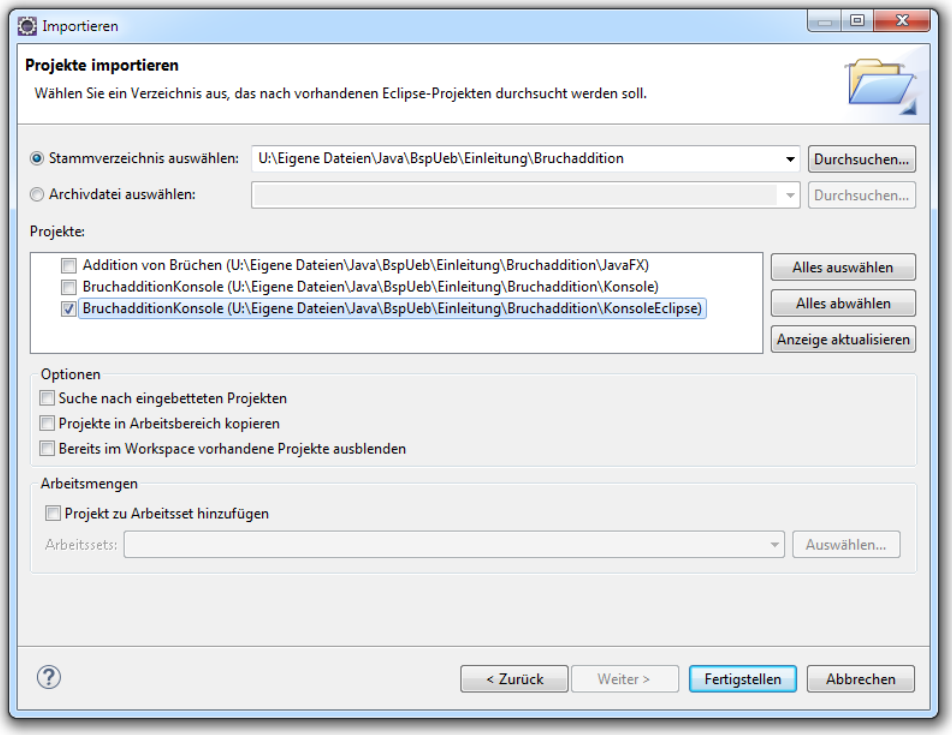
Machen Sie **weiter**, klicken Sie im folgenden Dialog



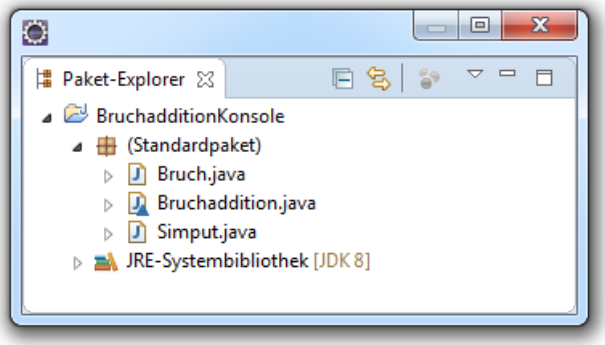
auf den Schalter **Durchsuchen**, und wählen Sie im Verzeichnisbaum einen Knoten oberhalb des zu importierenden Projektordners, z.B.:



Nach der Bestätigung mit **OK** müssen Sie eventuell in der Dialogbox **Importieren** aus mehreren importfähigen Projekten eine Teilmenge bestimmen und mit **Fertigstellen** Ihre Wahl quittieren:

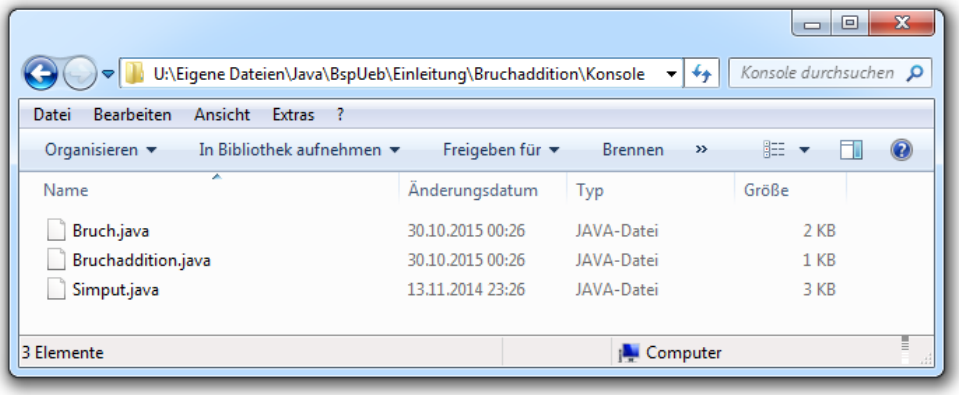


Anschließend tauchen die importierten Projekte im Paket-Explorer auf, z.B.:



2.5.10 Projekt aus vorhandenen Quellen erstellen

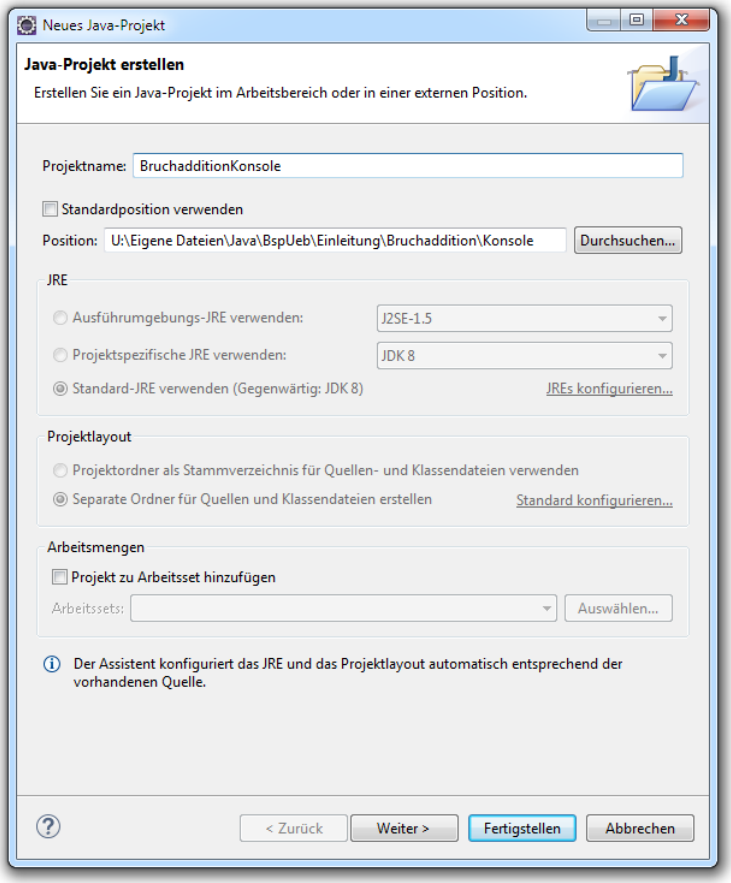
Gelegentlich soll ein neues Projekt unter Verwendung bereits existierender Quelldateien erstellt werden, z.B.:



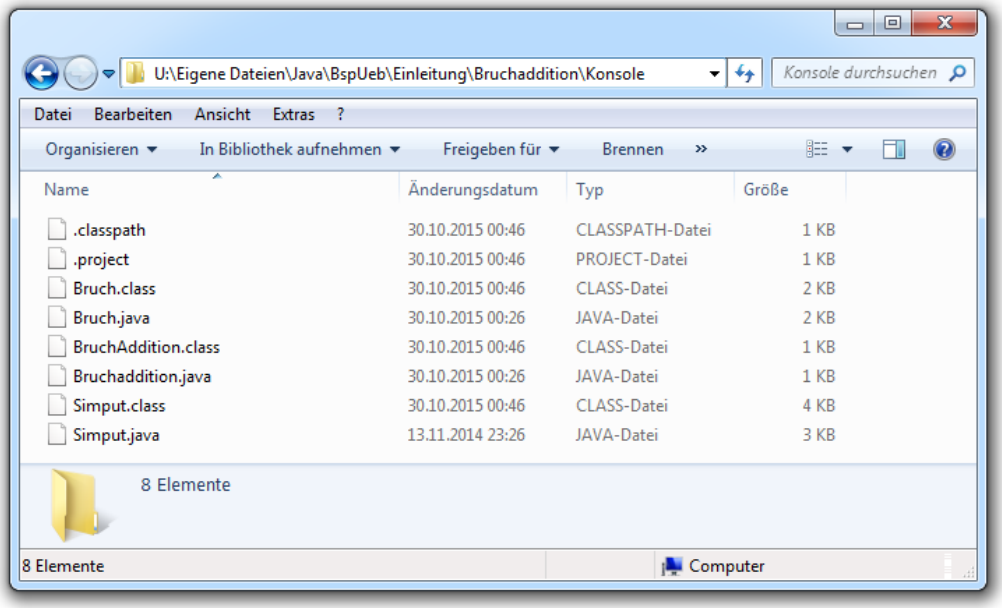
In diesem Fall trägt man in Eclipse nach

Datei > Neu > Java-Projekt

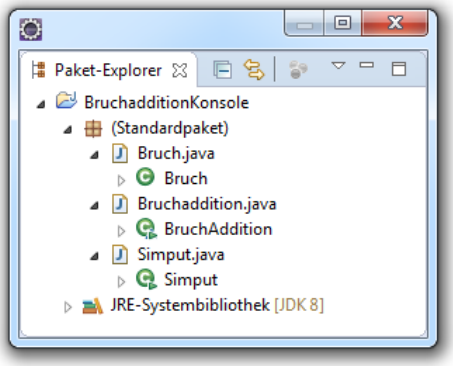
einen Projektnamen ein, entfernt die Markierung beim Kontrollkästchen **Standardposition verwenden** und wählt über den Schalter **Durchsuchen** den Ordner mit den vorhandenen Quellcodedateien, z.B.:



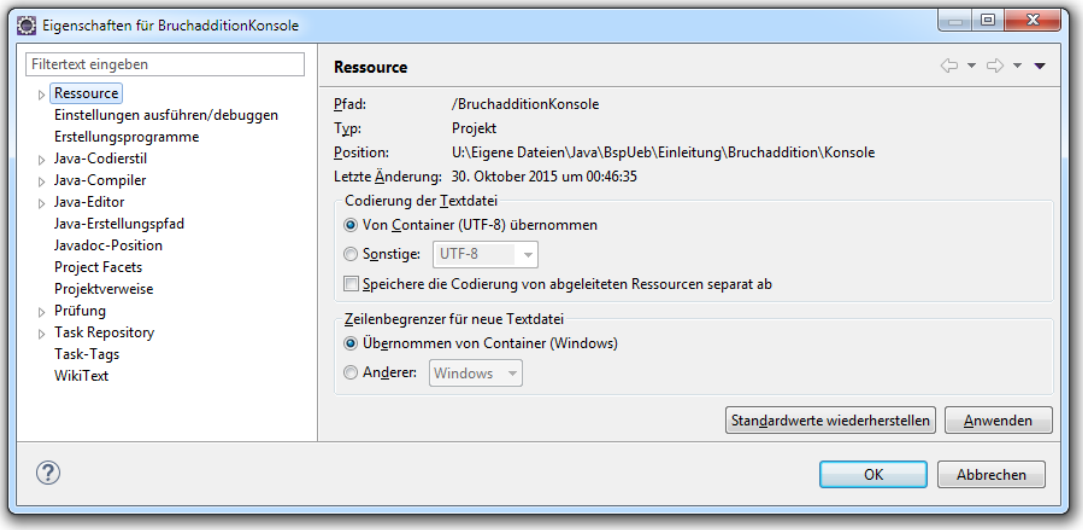
Nach dem **Fertigstellen** übernimmt Eclipse den Ordner, übersetzt automatisch die Klassen und ergänzt seine Projektdateien, z.B.:



Im Paket-Explorer von Eclipse zeigt sich das erwartete Bild:



Über das Item **Eigenschaften** aus dem Kontextmenü zum Projekt können Sie den Standort des Projekts einsehen:



2.6 Übungsaufgaben zu 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 von den folgenden Aussagen sind richtig bzw. falsch?

1. Beim Übersetzen einer Java-Quellcodedatei mit dem JDK-Compiler **javac.exe** muss man den Dateinamen samt Extension (**.java**) angeben.
2. Beim Starten eines Java-Programms muss man den Namen der auszuführenden Klasse samt Extension (**.class**) angeben.
3. Damit der Aufruf des JDK-Compilers **javac.exe** von jedem Verzeichnis aus klappt, muss unter Windows das **bin**-Unterverzeichnis der JDK-Installation in die Definition der Umgebungsvariablen PATH aufgenommen werden.

4. Damit der JDK-Compiler die Klassen der Standardbibliothek findet, müssen die zugehörigen Java-Archivdateien der JRE-Installation in die Definition der Umgebungsvariablen CLASSPATH aufgenommen werden (z.B.: **C:\Program Files\Java\jre8\lib\rt.jar**).
5. Die **main()** - Methode der Startklasse eines Java-Programms muss einen Parameter mit dem Datentyp **String[]** und dem Namen `args` besitzen, damit sie von der JRE erkannt wird.

4) Führen Sie nach Möglichkeit auf Ihrem eigenen PC die in Abschnitt 1 beschriebenen Installationen aus.

5) Kopieren Sie die Klasse …**BspUeb\Simput\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. Alternativ können Sie auch die Java-Archivdatei …**BspUeb\Simput\Simput.jar** kopieren und in den Klassenpfad aufnehmen. Mit Java-Archivdateien werden wir uns später noch ausführlich beschäftigen.

3 Elementare Sprachelemente

In Kapitel 1 wurde anhand eines halbwegs realistischen Beispiels, ein erster Eindruck von der objektorientierten Softwareentwicklung 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 einer Java-Applikation

Bevor wir im Rahmen von möglichst einfachen Beispielprogrammen elementare Sprachelemente kennen lernen, soll unser bisheriges Wissen über die Struktur von Java-Programmen[1] zusammengefasst werden:

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

  Meist verwendet man für den Quellcode einer Klasse jeweils eine eigene Textdatei mit der Namenserweiterung **.java**. Der Compiler erzeugt auf jeden Fall für jede Klasse eine eigene Bytecodedatei 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: Name der Methode, Parameterliste, Rückgabetyp und Modifikatoren (siehe Abschnitt 2.2.1). All diese Bestandteile werden noch ausführlich erläutert.

---

[1] Hier ist ausdrücklich von Java-*Programmen* (alias -*Applikationen*) die Rede. Bei den Java-*Applets*, die im Kurs *nicht* behandelt werden, ergeben sich einige Abweichungen.

  - und dem **Rumpf**
    Begrenzt durch ein Paar geschweifter Klammern befinden sich hier beliebig viele **Anweisungen**, mit denen 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 in Abschnitt 3.3 erläutert.

- 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**. Beim Bruchrechnungsbeispiel in Abschnitt 1.1 ist die Klasse `Bruchaddition` startfähig. In den POO-Beispielen (kursinterne Abkürzung für *pseudo-objektorientiert*) von Abschnitt 1 existiert jeweils nur *eine* Klasse, die infolgedessen startfähig sein muss.
- Eine **Anweisung** ist die kleinste ausführbare Einheit eines Programms.
  In Java sind bis auf wenige Ausnahmen alle Anweisungen mit einem **Semikolon** abzuschließen.

3.1.2 Projektrahmen zum Üben von elementaren Sprachelementen

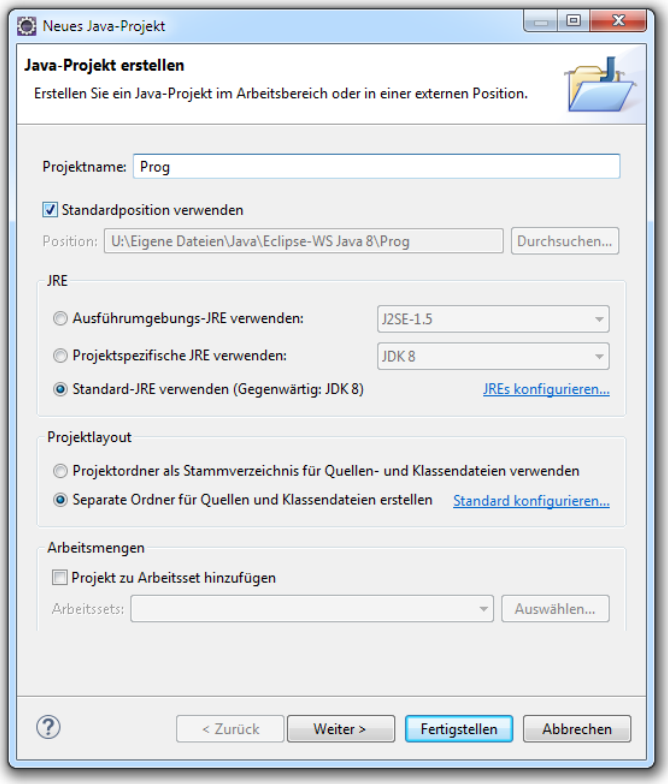
Während der Beschäftigung mit elementaren Java-Sprachelementen werden wir der Einfachheit halber mit einer relativ untypischen, jedenfalls nicht sonderlich objektorientierten Programmstruktur arbeiten, die Sie schon aus dem `Hallo`-Beispiel kennen (siehe Abschnitt 2.2.1). Es wird nur *eine* Klasse definiert, 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** 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 Eclipse-Projekt. Legen Sie für solche Zwecke mit dem Symbolschalter  oder dem Menübefehl

Datei > Neu > Java-Projekt

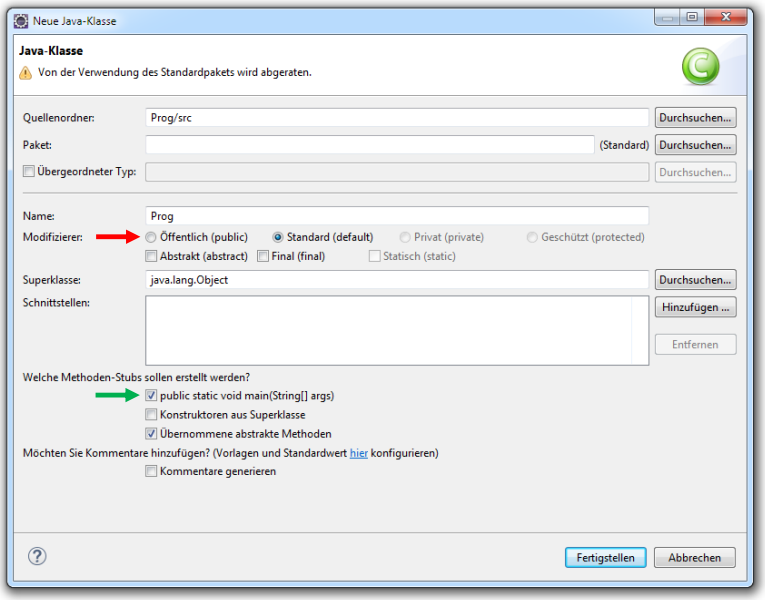
ein Projekt namens `Prog` an,



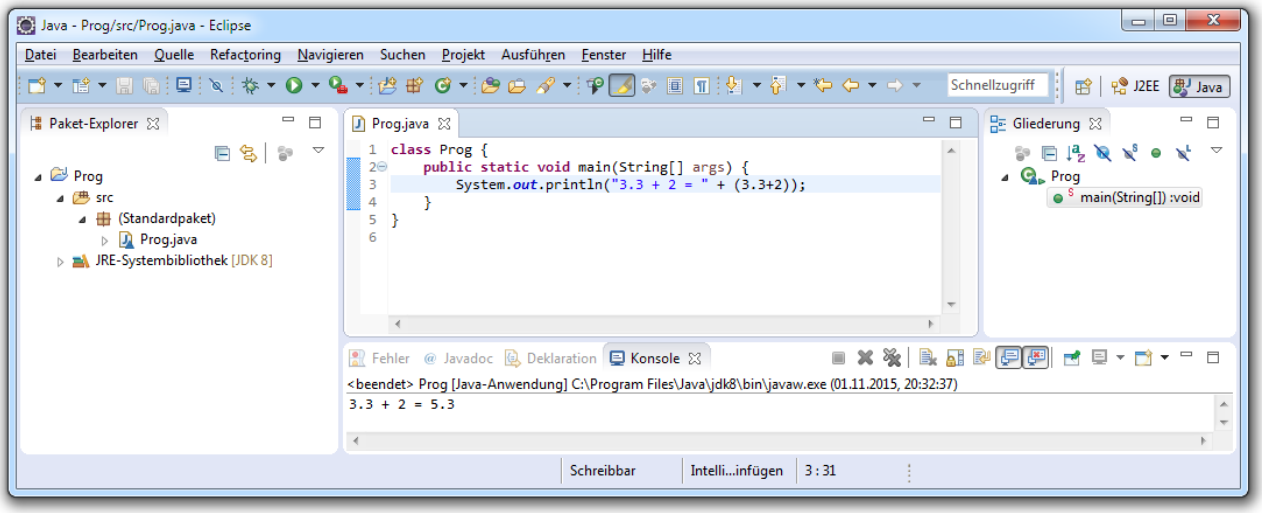
und erstellen Sie über den Symbolschalter oder den Menübefehl

Datei > Neu > Klasse

eine gleichnamige Startklasse:



Um den überflüssigen Modifikator **public** im automatisch erstellten Klassendefinitionskopf zu vermeiden, muss man eine Voreinstellung abändern (siehe roten Pfeil). Allerdings kann man sich diese Mühe eigentlich sparen, weil der Modifikator keinen Nachteil bringt. Man muss auf jeden Fall aktiv werden, damit der Assistent eine rudimentäre **main()** - Methode erstellt (siehe grünen Pfeil). Zum Üben elementarer Sprachelemente werden wir im Rumpf der **main()** - Methode passende Anweisungen einfügen, z.B.:

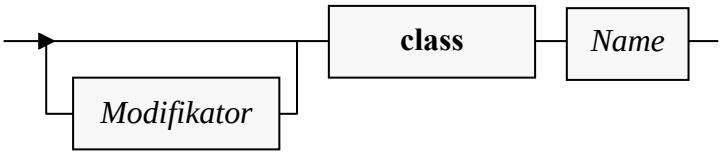


Im **Projekt-Explorer** und in der **Gliederung** enthält das Symbol zur Klasse `Prog` ein blaues Dreieck in der rechten unteren Ecke (bzw.), weil diese Klasse nicht als **public** deklariert wurde und folglich nur innerhalb des eigenen Paketes sichtbar ist. Mit Paketen und der Sichtbarkeit von Klassen werden wir uns in Abschnitt 6.3.1 beschäftigen.

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. so genannte **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, z.B.:



- 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 durch *kursive* Schrift gekennzeichnet. Im konkreten Quellcode muss anstelle des Platzhalters eine zulässige Realisation stehen, und die zugehörigen Regeln sind an anderer Stelle (z.B. in einem anderen Syntaxdiagramm) erklärt.

- Bisher kennen Sie nur den Klassenmodifikator **public**, welcher für die allgemeine Verfügbarkeit einer Klasse sorgt. Später werden Sie weitere Klassenmodifikatoren kennen lernen. Sicher kommt niemand auf die Idee, z.B. den Modifikator **public** mehrfach zu vergeben und damit gegen eine 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

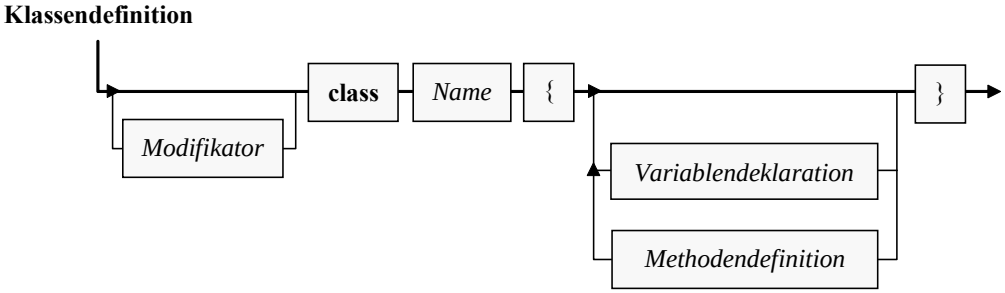
  Im Manuskript wird die zweite Lösung verwendet.

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* in der Programmiersprache Java beizutragen.

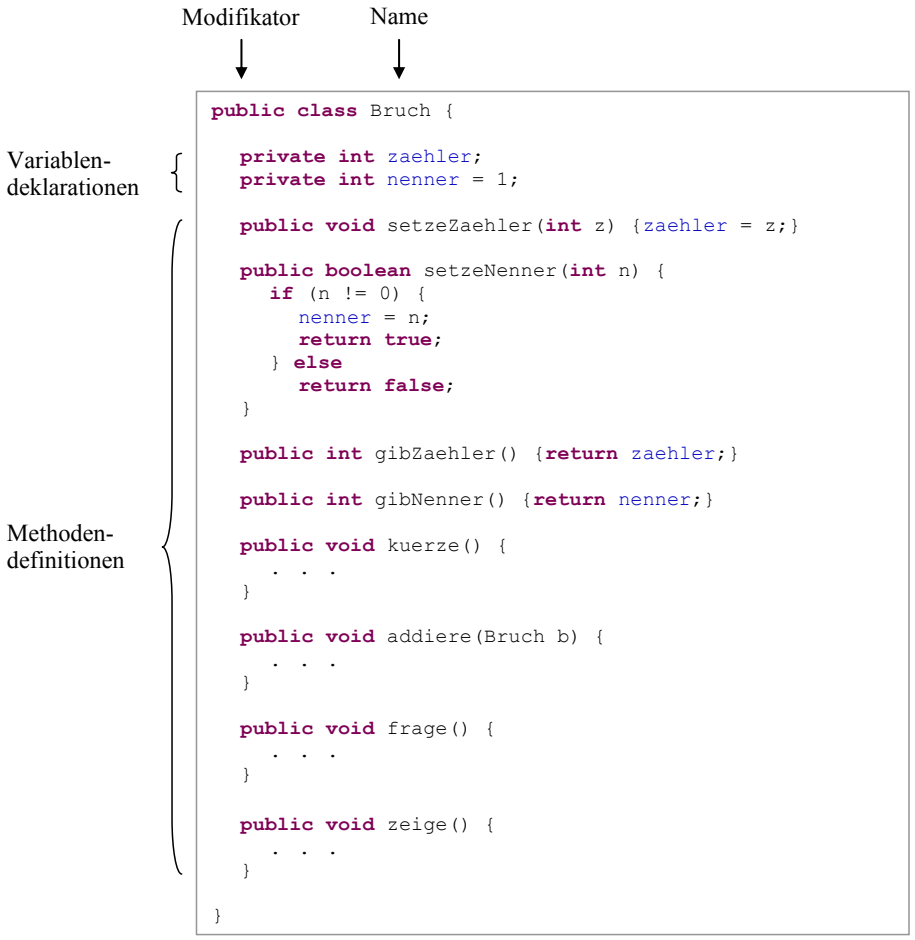
3.1.3.1 Klassendefinition

Wir arbeiten vorerst mit dem folgenden, leicht vereinfachten Klassenbegriff:



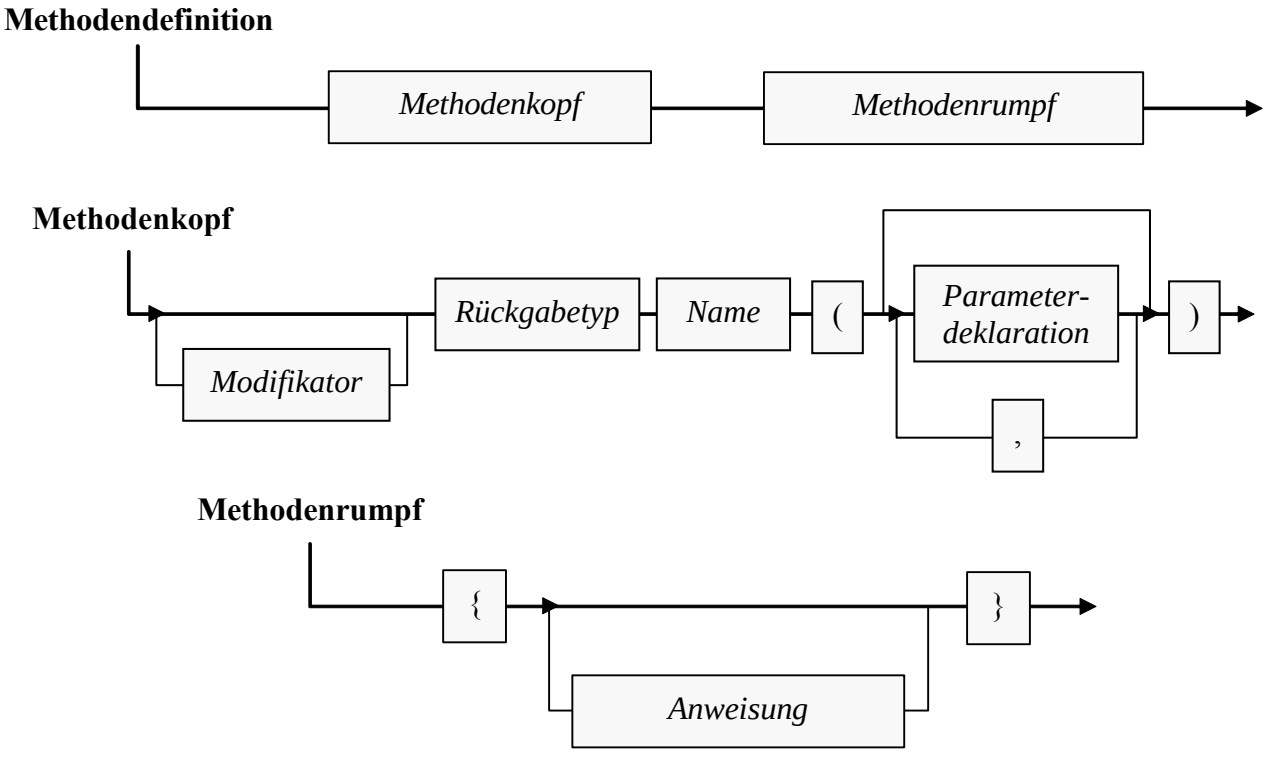
Solange man sich auf zulässigen Pfaden bewegt (immer in Pfeilrichtung, eventuell auch in Schleifen), an den Stationen (Rechtecken) entweder den terminalen Sprachbestandteil exakt übernimmt oder den Platzhalter auf zulässige (an anderer Stelle erläuterte) Weise ersetzt, sollte eine syntaktisch korrekte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

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

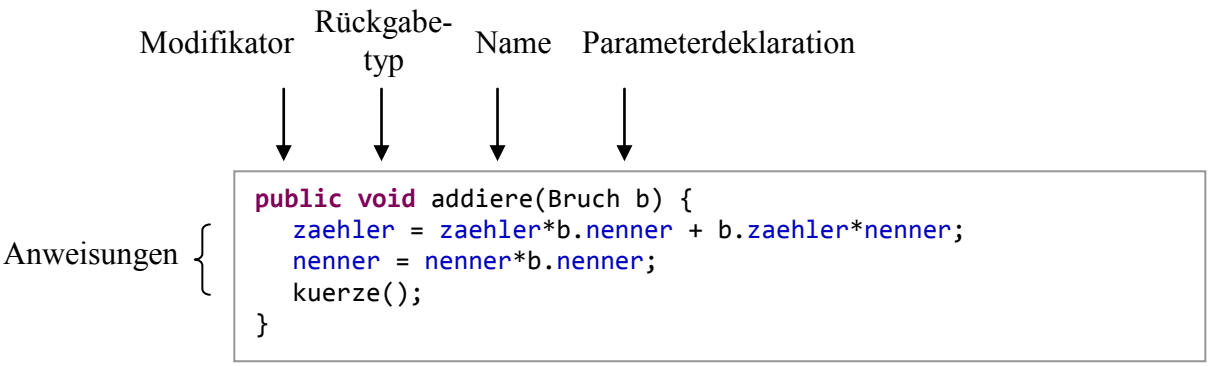


3.1.3.2 *Methodendefinition*

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



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



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

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

Weil wir bald u.a. von einer *Variablendeklarationsanweisung* sprechen 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 sehr unerwünscht.
- Zeichen mit festgelegter Bedeutung wie z.B. "{", ";", "(", "+", ">" sind **selbstbegrenzend**, 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 von öffnenden geschweiften Klammern 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):

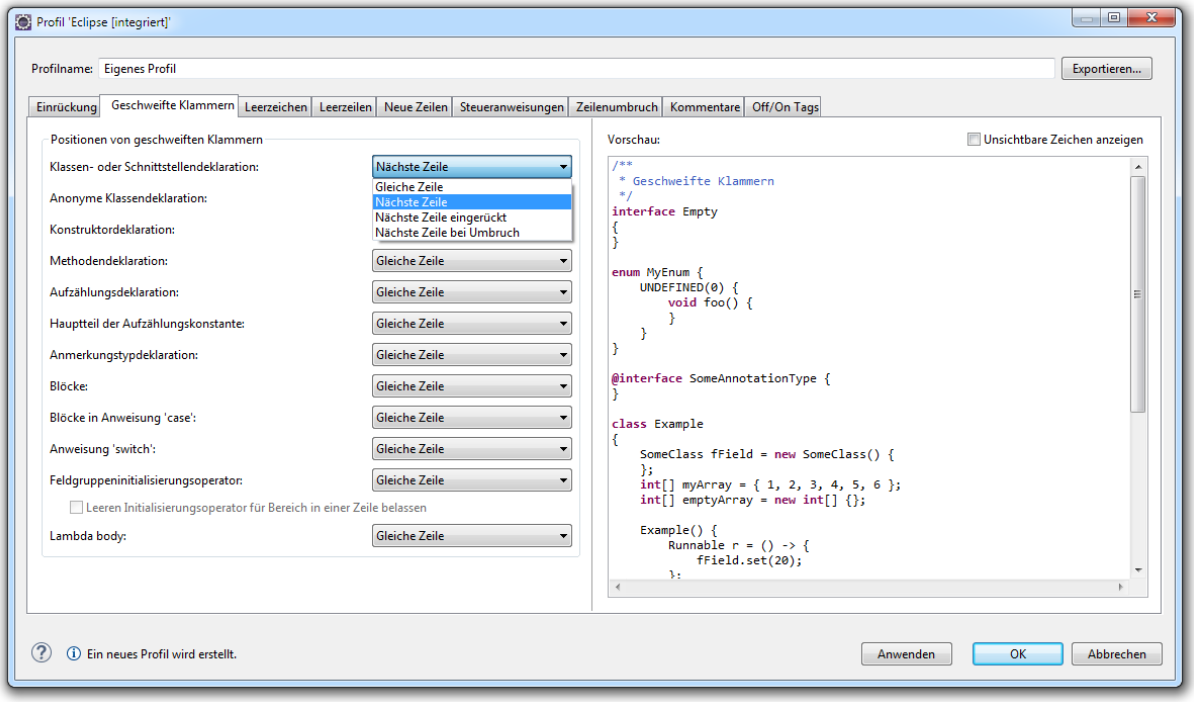
```java
class Hallo {
  public static void main(String[] par) {
    System.out.print("Hallo");
  }
}
```

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

Eclipse bevorzugt die linke Variante, könnte aber nach

Fenster > Benutzervorgaben > Java > Codedarstellung > Formatierungsprogramm > Aktives Profil = Eclipse [integriert] > Bearbeiten

in der folgenden Dialogbox umgestimmt werden:



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

  ```java
  private int zaehler; // wird automatisch mit 0 initialisiert
  ```

  Hier wird eine Variablendeklarationsanweisung in derselben Zeile kommentiert.

- **Mehrzeilenkommentar**
  Zwischen einer Einleitung durch **/*** und einer Terminierung durch ***/** kann sich ein ausführlicher Kommentar auch über mehrere Zeilen erstrecken, 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) {
 . . .
}
```

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

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

  Wenn Sie in Eclipse mehrere markierte Quellcodezeilen mit dem Menübefehl

  Quelle > Kommentar umschalten

  bzw. mit der Tastenkombination

  Strg+/ also Strg+⇧+7

  gemeinsam als Kommentar deklarieren, werden doppelte Schrägstriche vor jede Zeile gesetzt. Bei Anwendung des Menübefehls auf einen zuvor mit Doppelschrägstrichen auskommentierten Block entfernt Eclipse die Kommentar-Schrägstriche.

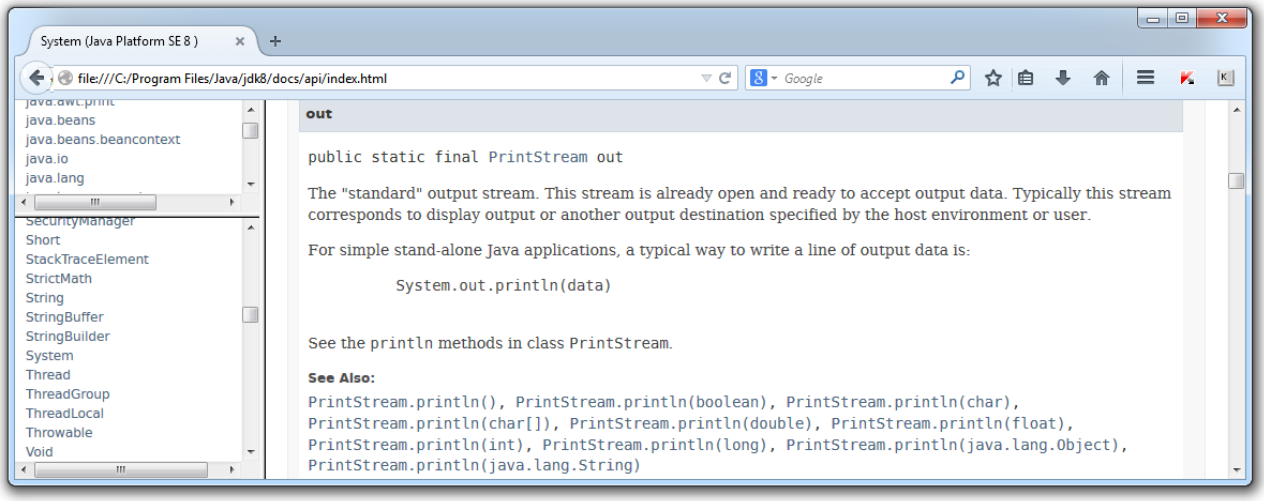
- **Dokumentationskommentar**
  Vor der Definition bzw. Deklaration von Klassen, Interfaces (siehe unten), Methoden oder Variablen darf ein Dokumentationskommentar stehen, eingeleitet mit **/**** und beendet mit ***/**. Er kann mit dem JDK-Werkzeug **javadoc** in eine HTML-Datei extrahiert werden. Die systematische Dokumentation wird über Tags für Methodenparameter, Rückgabewerte usw. unterstützt. Nähere Informationen finden Sie in der JDK - Dokumentation (vgl. Abschnitt 2.1.3) über den Link **javadoc**.

  Im Quellcode der wichtigen API-Klasse **System** findet sich z.B. der folgende Dokumentationskommentar zum Ausgabeobjekt **out**, das Sie schon kennen gelernt haben:[1]

[1] Die Quellcodedatei **System.java** steckt im API-Quellcodearchiv **src.zip**. In Abschnitt 2.1 wurde empfohlen, das Quellcodearchiv als JDK-Bestandteil zu installieren und anschließend in den Unterordner **src** der JDK-Installation auszupacken.

```
/**
 * 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>
 * <p>
 * See the <code>println</code> methods in class <code>PrintStream</code>.
 *
 * @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 final static PrintStream out = null;
```

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



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.
- 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 unterstützt.
- Namen müssen innerhalb ihres Kontexts (siehe unten) eindeutig sein.

3.1.7 Vollständige Klassennamen und Import-Deklaration

Jede Java-Klasse gehört zu einem sogenannten **Paket** (siehe Kapitel 6), und dem Namen der Klasse ist grundsätzlich der jeweilige Paketname voranzustellen. Dies gilt natürlich auch für die API-Klassen, also z.B. für die im folgenden Beispiel verwendete Klasse **Random** aus dem Paket **java.util** zum Erzeugen von Pseudozufallszahlen:

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 so genannten **Standardpakets**, zu dem alle keinem Paket explizit zugeordneten Klassen gehören, weil dieses Paket unbenannt bleibt,
- bei den Klassen aus dem API-Paket **java.lang** (z.B. **Math**), weil die Klassen dieses Pakets automatisch in jede Quellcodedatei importiert werden.

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

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.

Mit der Anzahl importierter Bezeichner steigt das Risiko für eine Namenskollision. Der Compiler meckert, ...

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

Wenn aufgrund der Platzhaltersyntax aus mehreren Paketen namensgleiche Klassen importiert worden sind, 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 vom Compiler (beim Übersetzen) und von der JVM (bei der Ausführung des Programmes) gefunden werden Bei den Klassen aus dem Java-API ist dies garantiert. Damit das Importieren anderer Klassen klappt, müssen Compiler und Runtime darüber informiert werden, an welchen Orten gesucht werden soll (siehe Abschnitt 2.2.4).

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 in 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**, seine **print()** - oder seine **println()** - Methode auszuführen.[1] Im Unterschied zu **print()** schließt **println()** die Ausgabe automatisch mit einer Zeilenschaltung ab, so dass die nächsten Aus- oder Eingabe in einer neuen Zeile erfolgt. Folglich ist **print()** zu bevorzugen, …

[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**. Deren Felder sind statisch (klassenbezogen), können also verwendet werden, ohne ein Objekt aus der Klasse **System** zu erzeugen. U.a. befindet sich unter den **System** - Membern ein Objekt namens **out** aus der Klasse **PrintStream**. Es beherrscht u.a. die Methoden **print()** und **println()**, die jeweils ein einziges Argument von beliebigem Datentyp erwarten und zur Standardausgabe befördern.

- wenn eine Benutzereingabe unmittelbar hinter einer Ausgabe in derselben Zeile ermöglicht werden soll,
- wenn mehrere Ausgaben in einer Zeile hintereinander erscheinen sollen.
  Allerdings ist es durchaus möglich, eine zusammengesetzte Ausgabe mit *einer* **print()** - oder **println()** - Anweisung zu erzeugen.

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 **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**, sonst **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` in eine Zeichenfolge gewandelt und dann mit `"Ergebnis: "` verknüpft.

3.2.2 Formatierte Ausgabe

Der Methodenaufruf **System.out.printf()**[1] erlaubt eine *formatierte* Ausgabe von mehreren Ausdrücken, z.B.:[2]

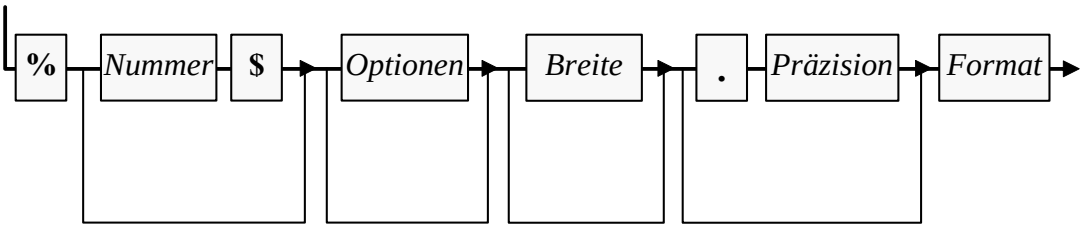
Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.printf("Pi = %12.3f", Math.PI);` `  System.out.println();` `  System.out.printf("e  = %12.7f", Math.E);` `  }` `}`	`Pi =        3,142` `e  =    2,7182818`

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

---

[1] Es handelt sich um eine Instanzmethode der Klasse **PrintStream** (siehe Fußnote in Abschnitt 3.2.1).
[2] Alternativ kann auch der äquivalente Methodenaufruf **System.out.format()** benutzt werden.

Platzhalter für die formatierte Ausgabe



Darin bedeuten:

Nummer	Fortlaufende Nummer des auszugebenden Arguments, bei 1 beginnend
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**
c	ein einzelnes Zeichen	`// x ist eine char-` `// Variable` `("Inhalt von x: %c", x)`	`Inhalt von x: h`
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	`("%5.2f", 4.711)`	` 4,71`
e	Rationale Zahl in wissenschaftlicher Notation Präzision: Anzahl Stellen in der Mantisse	`("%e", 47.11)` `("%.2e", 47.11)` `("%12.2e", 47.11)`	`4,711000e+01` `4,71e+01` `    4.71e+01`

Wie **print()** produziert auch **printf()** *keinen* automatischen Zeilenwechsel nach der Ausgabe. Im obigen Beispielprogramm wird daher nach dem ersten **printf()** - Aufruf mit der Methode **println()** für einen Zeilenwechsel gesorgt.

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

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.println(Math.PI);` `  System.out.printf("%-12.7f", Math.PI);` `  }` `}`	`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** (im Paket **java.util**[1]). Wie man die Dokumentation zu einer API-Klasse findet, wurde zwar schon einmal (in Abschnitt 2.1.3) beschrieben, soll aber wegen der großen praktischen Bedeutung am aktuellen Beispiel erneut demonstriert werden:

- Öffnen Sie die HTML-Startseite der JDK-Dokumentation, je nach Installationsort z.B. über die Datei

  C:\Program Files\Java\jdk8\docs\index.html

- Wechseln Sie per Mausklick auf den Link **Java SE API** zur Dokumentation der **Java Platform, Standard Edition 8 API Specification**.
- Klicken Sie im linken oberen Frame auf **All Classes** oder (zur Verkürzung der Liste im linken unteren Frame) auf das Paket **java.util**.
- Klicken Sie im linken unteren Frame auf den Klassennamen **Formatter**. Anschließend erscheinen im rechten Frame detaillierte Informationen über die Klasse **Formatter**.

3.3 Variablen und Datentypen

Während ein Programm läuft, müssen zahlreiche Daten im Arbeitsspeicher des Rechners abgelegt werden, um anschließend mehr oder weniger lange für lesende und schreibende Zugriffe verfügbar zu 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 Zahl) wird eine so genannte **Variable** verwendet, worunter Sie sich einen **benannten Speicherplatz für einen Wert mit einem bestimmten Datentyp** (z.B. Ganzzahl) vorstellen können.

Eine Variable erlaubt über ihren Namen den lesenden oder schreibenden Zugriff auf die zugehörige Stelle im Arbeitsspeicher, z.B.:

---

[1] Mit den Paketen der Standardklassenbibliothek werden wir uns später ausführlich beschäftigen. An dieser Stelle dient die Angabe der Paketzugehörigkeit dazu, das Lokalisieren der Informationen zu einer Klasse in der API-Dokumentation zu erleichtern.

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

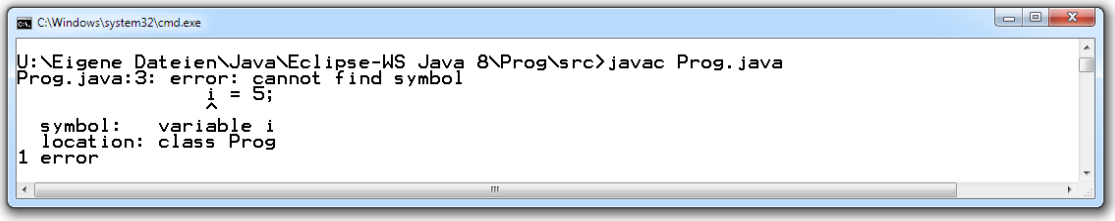
3.3.1 Strenge Compiler-Überwachung bei Java-Variablen

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:

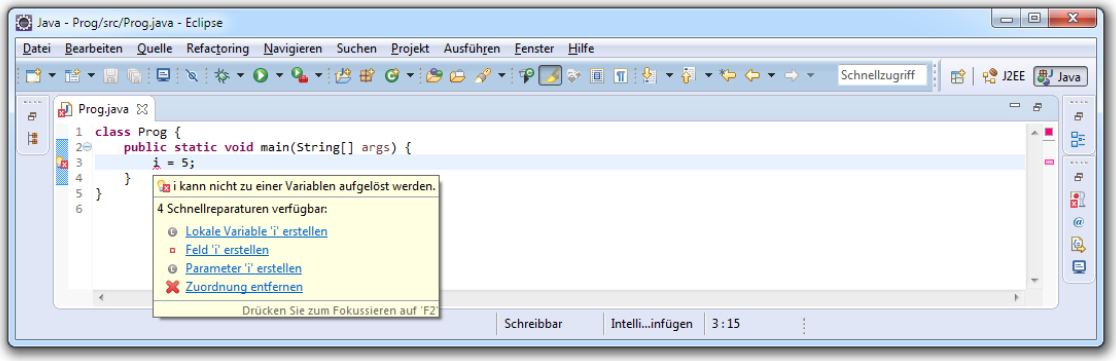
- Variablen müssen **explizit deklariert** werden, z.B.:

  ```
  int ivar = 4711;
  ```

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



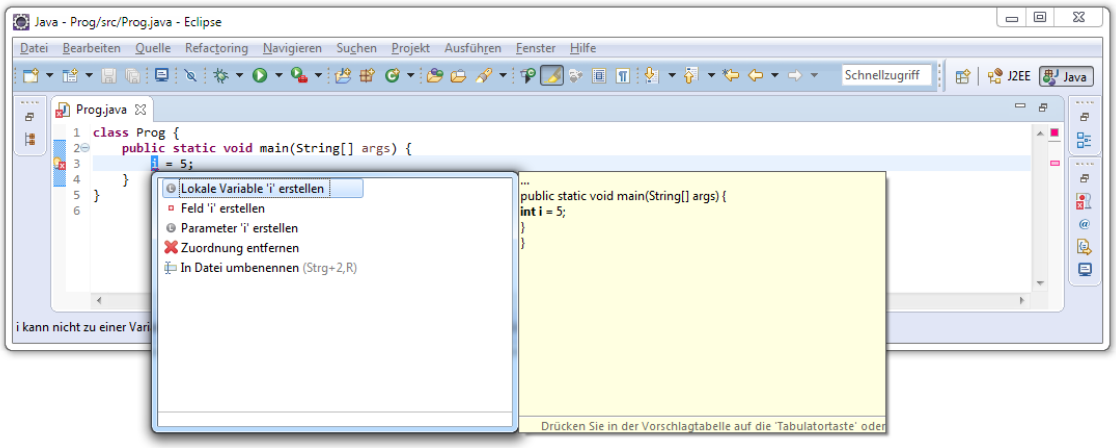
  Der inkrementelle Compiler in Eclipse erkennt und dokumentiert das Problem unmittelbar nach der Eingabe im Editor:[1]



  Wenn Sie den Mauszeiger kurz über der rot unterschlängelten Fehlerstelle ruhen lassen, erscheint ein Überlagerungsfenster mit Schnellreparaturvorschlägen. Nach einem Mausklick auf das Fehlersymbol am linken Zeilenrand (Glühbirne mit Kreuz) erscheinen zwei Fenster, wobei zu dem im linken Fenster markieren Reparaturvorschlag im rechten Fenster eine Vorschau auf den korrigierten Quellcode erscheint:

---

[1] Um das Editorfenster zu ermuntern, alle anderen Sichten und Editoren vorübergehend zu verdrängen, setzt man einen Doppelklick auf seine Titelzeile. Mit einem weiteren Doppelklick am selben Ort stellt man den alten Zustand wieder her.



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

- 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 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 manchen Skriptsprachen arbeitet Java mit einer *statischen* Typisierung, so dass der einer Variablen zugewiesene Typ nicht mehr geändert werden kann.

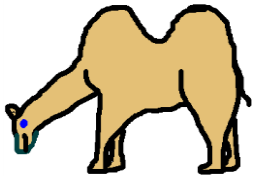
In der obigen Anweisung erhält die Variable `ivar` beim Deklarieren gleich den **Initialisierungswert** 4711. Auf diese oder andere Weise müssen Sie jeder *lokalen*, d.h. innerhalb einer Methode deklarierten, Variablen einen Wert zuweisen, bevor Sie zum ersten Mal lesend darauf zugreifen (vgl. Abschnitt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

[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, so dass es mir nicht sinnvoll erschien, eine andere Bezeichnung vorzunehmen, um die Doppelbedeutung zu vermeiden.

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:

- Variablennamen beginnen mit einem Kleinbuchstaben (siehe z.B. Gosling et al. 2015, Abschnitt 6.1).
- 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).

3.3.3 Primitive Typen und Referenztypen

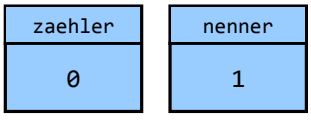
Bei der objektorientierten Programmierung werden neben den traditionellen (elementaren, primitiven) Variablen zur Aufbewahrung von Zahlen, Zeichen oder Wahrheitswerten auch Variablen benötigt, welche die Adresse eines Objekts aufnehmen und so die Kommunikation mit dem Objekt ermöglichen. Wir unterscheiden 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), 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. Sie werden in den folgenden Anweisungen deklariert, wobei `nenner` auch noch einen expliziten Initialisierungswert erhält:[1]

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

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



[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.

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

```
int ggt = 0;
```

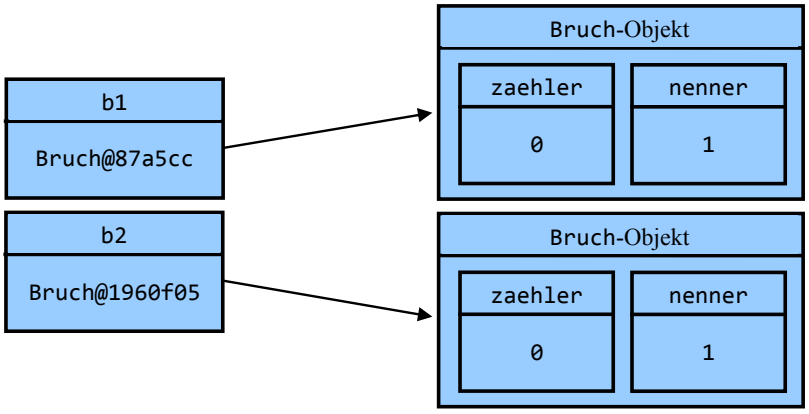
In Abschnitt 3.3.6 werden zahlreiche weitere primitive Datentypen vorgestellt.

- **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 (Objektadresse)
  - und durch ihre Rolle bei der Kommunikation mit Objekten.

  Man kann jede Klasse (aus der Java-Standardbibliothek übernommen 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 plagen uns nicht mit solchen Adressen, sondern sprechen die dort abgelegten Objekte über Referenzvariablen 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), 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** 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, 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 beschränkt sich auf die Methode bzw. auf einen Block innerhalb der Methode (siehe Abschnitt 3.3.9).

  Solange eine Methode ausgeführt wird, befinden sich ihre Variablen in einem Speicherbereich, den man als **Stack** (deutsch: *Stapel*) bezeichnet.

- **Instanzvariablen (nicht-statische Felder)**
  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`.

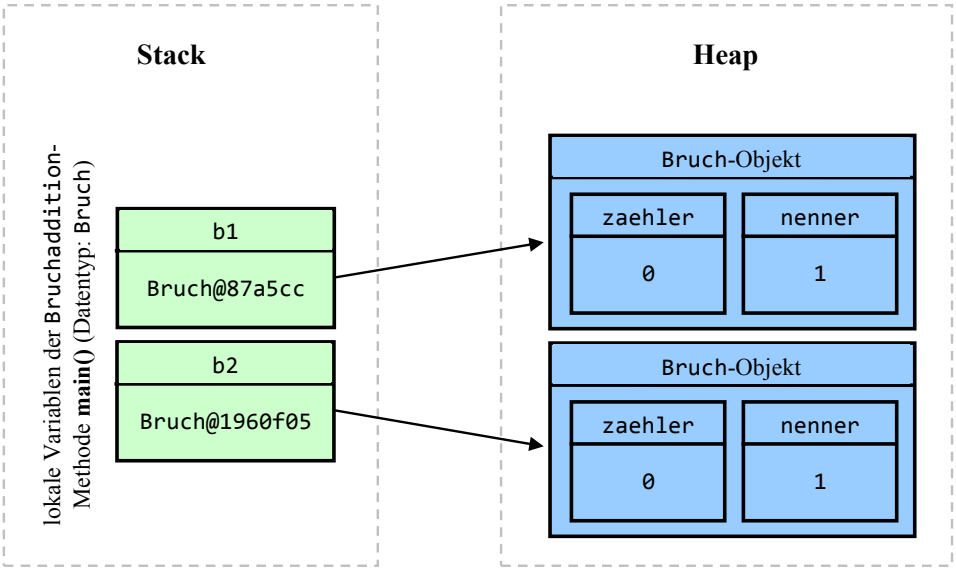
  Solange ein Objekt existiert, befinden es sich mit all seinen Instanzvariablen in einem Arbeitsspeicherbereich, den man als **Heap** (deutsch: *Haufen*) bezeichnet.

- **Klassenvariablen (statische Felder)**
  Diese Variablen beziehen sich auf eine Klasse insgesamt, nicht auf einzelne Instanzen der Klasse. Oft hält man z.B. hält in einer Klassenvariablen fest, 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 schon 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.

  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 so genannten **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:



Die lokalen Referenzvariablen `b1` und `b2` der Methode **main()** befinden sich im Stack-Bereich des 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 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.

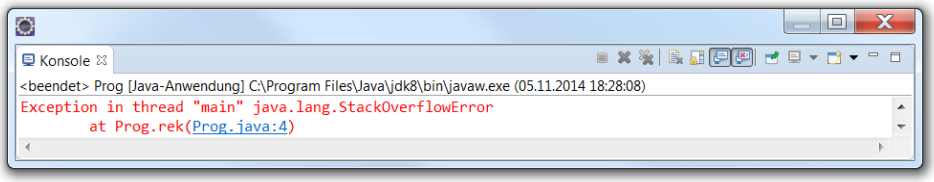
Im Kapitel 3 werden wir überwiegend mit *lokalen* Variablen arbeiten, aber z.B. auch das statische Feld **out** der Klasse **System** benutzen, das auf ein Objekt der Klasse **PrintStream** zeigt. Im Zu-sammenhang mit der systematischen Behandlung der objektorientierten Programmierung werden die Instanz-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

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

3.3.5 Eigenschaften einer Variablen

Als wichtige 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 von der JVM ein **StackOverflowError** gemeldet wird:



- **Datentyp**
  Damit sind festgelegt: Wertebereich, 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 Variable auf, die zur Methode **main()** gehört, vom primitiven Typ **int** ist, den Namen `ivar` besitzt und den Wert 5 hat:

```
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 Wertebereich und Speicherplatzbedarf 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	Speicherbedarf 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. Beispiel: `float pi = 3.141593f;` **float**-Literale (s. u.) benötigen den Suffix **f** (oder **F**).	Minimum: $-3{,}4028235 \cdot 10^{38}$ Maximum: $3{,}4028235 \cdot 10^{38}$ Kleinster Betrag: $1{,}4 \cdot 10^{-45}$	32 1 für das Vorz., 8 für den Expon., 23 für die Mantisse

Typ	Beschreibung	Werte	Speicher-bedarf in Bit
double	Variablen vom Typ **double** speichern Gleitkommazahlen nach der Norm IEEE-754 mit einer Genauigkeit von mindestens 15 Dezimalstellen. Beispiel: `double pi = 3.1415926535898;`	Minimum: $-1{,}7976931348623157 \cdot 10^{308}$ Maximum: $1{,}7976931348623157 \cdot 10^{308}$ Kleinster Betrag: $4{,}9406564584124654 \cdot 10^{-324}$	64 1 für das Vorz., 11 für den Expon., 52 für die Mantisse
char	Variablen vom Typ **char** dienen zum Speichern *eines* Unicode-Zeichens. Im Speicher landet aber nicht die Gestalt eines Zeichens, sondern seine Nummer im Unicode-Zeichensatz. Daher zählt **char** zu den *integralen* (ganzzahligen) Datentypen. Beispiel: `char zeichen = 'j';` **char**-Literale (s. u.) sind durch *einfache* Anführungszeichen zu begrenzen.	Unicode-Zeichen Tabellen mit allen Unicode-Zeichen sind z.B. auf der Webseite http://www.unicode.org/charts/ des Unicode-Konsortiums verfügbar.	16
boolean	Variablen vom Typ **boolean** können Wahrheitswerte aufnehmen. Beispiel: `boolean cond = false;`	**true**, **false**	1

Eine *Gleitkommazahl* (synonym: *Gleitpunkt-* oder *Fließkommazahl*, englisch: *floating point number* dient zur approximativen Darstellung einer reellen Zahl in der EDV. 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):

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

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

Ein Vorteil von Java besteht darin, dass die Wertebereiche der elementaren Datentypen auf allen Plattformen identisch sind, worauf man sich bei anderen Programmiersprachen (z.B. C/C++) nicht verlassen kann.

Im Vergleich zu den Programmiersprachen C, C++ und C# fällt auf, dass der Einfachheit halber auf *vorzeichenfreie* Ganzzahltypen verzichtet wurde.

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 Norm **IEEE-754** zu erklären, wobei jede Zahl als Produkt aus drei getrennt zu speichernden Faktoren dargestellt wird:[1]

$$\textbf{Vorzeichen} \cdot \textbf{Mantisse} \cdot 2^{\textbf{Exponent}}$$

Im ersten Bit einer **float**- und **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. Besitzt z.B. eine **float**-Zahl den Exponenten 0, dann wird für ihren Exponenten der Wert 127 gespeichert, und für den Exponenten -2 landet der Wert 125 im Speicher.

---

[1] Die Norm wurde veröffentlicht vom ***I**nstitute of **E**lectrical and **E**lectronics **E**ngineers* (IEEE).

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. Weil die führende 1 der normalisierten Mantisse *nicht* abgespeichert wird (*hidden bit*), 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. Das *i*-te Mantissen-Bit (von *links* nach *rechts* mit 1 beginnend nummeriert) hat die Wertigkeit 2^{-i}, so dass sich der *dezimale* Mantissenwert folgendermaßen ergibt:

$$1+m \quad \text{mit} \quad m = \sum_{i=1}^{23\,\text{bzw.}\,52} b_i 2^{-i},\ b_i \in \{0,1\}$$

Eine **float**- bzw. **double**-Variable mit dem Vorzeichen *v* (0 oder 1), dem Exponenten *e* und dem dezimalen Mantissenwert (1+ *m*) speichert also bei normalisierter Darstellung den Wert:

$$\mathbf{(-1)^v \cdot (1+m) \cdot 2^{e-127}} \quad \text{bzw.} \quad \mathbf{(-1)^v \cdot (1+m) \cdot 2^{e-1023}}$$

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

$$\begin{aligned} 0{,}25 &= 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \\ &= 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} \end{aligned}$$

perfekt dargestellt werden kann, gelingt dies bei der Restmantisse 0,3 nur approximativ:

$$\begin{aligned} 0{,}3 &= 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + \ldots \\ &= 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 0 \cdot \frac{1}{16} + 1 \cdot \frac{1}{32} + \ldots \end{aligned}$$

Sehr aufmerksame Leser werden sich darüber wundern, wieso die Tabelle mit den elementaren Datentypen in Abschnitt 3.3.6 z.B.

$$1{,}4 \cdot 10^{-45}$$

als betragsmäßig kleinsten **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* 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 dezimalen Mantissenwert *m* speichert also bei denormalisierter Darstellung die Zahl:

$$\mathbf{(-1)^{v} \cdot 2^{-126} \cdot m} \quad \text{bzw.} \quad \mathbf{(-1)^{v} \cdot 2^{-1022} \cdot m}$$

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 relative Genauigkeit mit der Annäherung an die Null.

Eclipse- 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 10000000000000000000000
```

Die beschriebene Technik, eine reelle Zahl approximativ durch ein Tripel aus Vorzeichen, Mantisse und Exponent zu speichern, wurde übrigens von Konrad Zuse ausgetüftelt.[1] Im Vergleich zur Festkommadarstellung erhält man bei gleichem Speicherplatzbedarf einen erheblich größeren Wertebereich.

Zur Verarbeitung von binären Gleitkommazahlen wurde die *binäre Gleitkommaarithmetik* entwickelt, normiert und zur Verbesserung der Verarbeitungsgeschwindigkeit sogar teilweise in Computer-Hardware realisiert.

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

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 (siehe API-Dokumentation). Objekte dieser Klasse können Dezimalzahlen mit beliebiger Genauigkeit speichern und verwenden eine dezimale Gleitkommaarithmetik mit einstellbarer Rechengenauigkeit.

Gespeichert werden:

- 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 bd13 = new BigDecimal("1.3");
     System.out.println(bd13);
 }
}
```

belegt zunächst als Nachtrag zum Abschnitt 3.3.7.1, dass auch eine **double**-Variable den Wert 1,3 nur approximativ speichern kann:[1]

```
1.3000000000000000444089209850062616169452667236328125
1.3
```

Diese Folge der binären Gleitkommadarstellung tritt bei einem Objekt der Klasse **BigDecimal** nicht auf, 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,

$$1700000000 \quad - \sum_{i=1}^{1000000000} 1{,}7$$

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] 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.

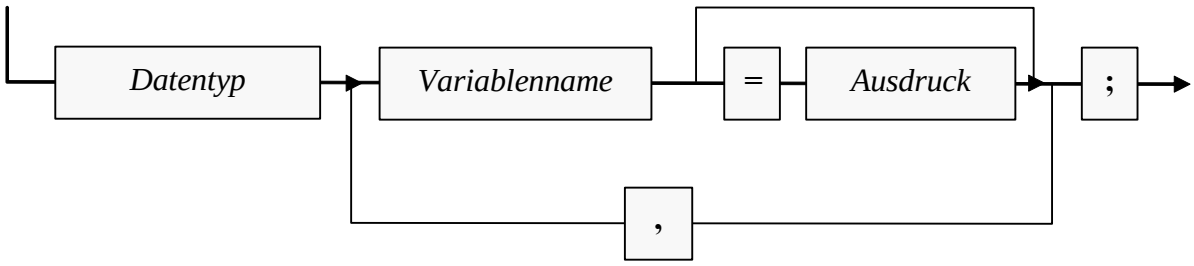
```
double:
 Abweichung:            -29.96745276451111
 Zeit in Millisekunden: 988

BigDecimal:
 Abweichung:            0.0
 Zeit in Millisekunden: 9739
```

Die gut bezahlten Verantwortlichen bei den deutschen Landesbanken und anderen Instituten, 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, so dass man bei finanzmathematischen Aufgaben trotz des erhöhten Zeitaufwands (im Beispiel: Faktor 10) die Klasse **BigDecimal** verwenden sollte.

3.3.8 Variablendeklaration, Initialisierung und Wertzuweisung

In einem Java-Programm muss jede Variable vor ihrer ersten Verwendung deklariert werden, wobei auf jeden Fall der Name und der Datentyp 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. Es folgt das Syntaxdiagramm zur Deklaration einer lokalen Variablen:

Deklaration einer lokalen Variablen



Als Datentypen kommen in Frage (vgl. Abschnitt 3.3.3):

- Primitive Datentypen, z.B.
  ```
  int i;
  ```
- Referenztypen, also Klassen (aus dem Java-API oder selbst definiert), z.B.
  ```
  Bruch b;
  ```

Neu deklarierte Variablen kann man optional auch gleich **initialisieren**, also auf einen gewünschten Wert bringen, z.B.:

```
int i = 4711;
Bruch b = new Bruch();
```

Im zweiten Beispiel wird per **new**-Operator ein `Bruch`-Objekt erzeugt und dessen Adresse in die neue Referenzvariable `b` 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), so dass man sie im Quelltext z.B. gut von den Namen von Klassen unterscheiden kann.

Weil *lokale* Variablen *nicht* automatisch initialisiert werden, muss man ihnen unbedingt 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. Dieses Verhalten wird durch folgendes Programm demonstriert:

```
class Prog {
   public static void main(String[] args) {
      int argument;
      System.out.print("Argument =  " + argument);
   }
}
```

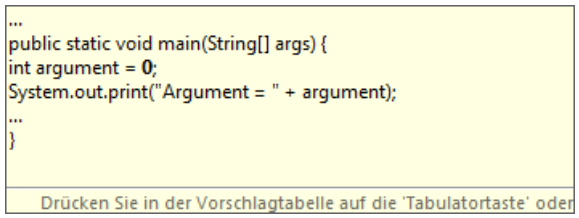
Der JDK-Compiler meint dazu:

```
Prog.java:4: variable argument might not have been initialized
                System.out.print("Argument =  " + argument);
                                                  ^
1 error
```

Ähnlich äußert sich auch der Eclipse-Compiler:

```
Exception in thread "main" java.lang.Error: Unaufgelöstes Kompilierungsproblem:
 Die lokale Variable argument ist möglicherweise nicht initialisiert

 at Prog.main(Prog.java:4)
```

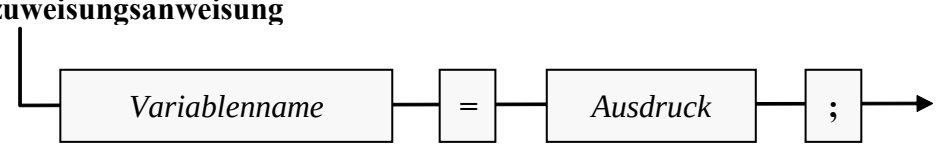
Eclipse gibt nach einem Mausklick auf das Fehlersymbol neben der betroffenen Zeile sogar konkrete Hinweise zur Verbesserung des Quellcodes:



Weil Instanz- und Klassenvariablen automatisch mit dem typspezifischen Nullwert initialisiert werden (siehe unten), kann in einem Java-Programm kein Zugriff auf undefinierte Werte stattfinden.

Um den Wert einer Variablen im weiteren Programmablauf zu verändern, verwendet man eine **Wertzuweisung**, die zu den einfachsten und am häufigsten benötigten Anweisungen gehört:

Wertzuweisungsanweisung



Beispiel: `ggt = az;`
Durch diese Wertzuweisungsanweisung aus der `kuerze()` - Methode unserer `Bruch`-Klasse (siehe Abschnitt 1.1.2) erhält die **int**-Variable `ggt` den Wert der **int**-Variablen `az`.

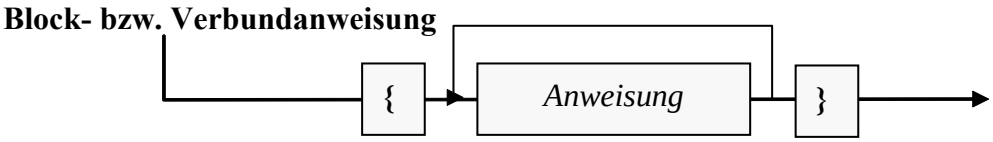
Es wird sich bald herausstellen, dass auch ein Ausdruck stets einen Datentyp besitzt. Bei der Wertzuweisung muss dieser Typ kompatibel zum Datentyp der Variablen sein.

U.a. haben Sie mittlerweile zwei Sorten von Java-Anweisungen kennen gelernt:

- Variablendeklaration
- Wertzuweisung

3.3.9 Blöcke und Gültigkeitsbereiche 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 Blockanweisungen 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 extrem lästig. Stattdessen fasst man die Anweisungen zu einem *Block* zusammen, der als *eine* Anweisung gilt, so dass die Bedingung nur einmal formuliert werden muss. Dieses extrem häufige 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 Gültigkeit der darin deklarierten Variablen: Eine lokale Variable ist verfügbar von der deklarierenden Zeile bis zur schließenden Klammer des lokalsten Blocks. Nur in diesem **Gültigkeitsbereich** (engl. *scope*) kann sie über ihren Namen angesprochen werden. Beim Versuch, das folgende (weitgehend sinnfreie) Beispielprogramm

```
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("Wert2 = " + wert2);
  }
 System.out.println("Wert2 = " + wert2);
 }
}
```

zu übersetzen, erhält man vom Eclipse-Compiler die Fehlermeldung:

```
Exception in thread "main" java.lang.Error: Unaufgelöstes Kompilierungsproblem:
  wert2 cannot be resolved to a variable

  at Prog.main(Prog.java:9)
```

Bei hierarchisch geschachtelten Blöcken ist es in Java *nicht* erlaubt, auf mehreren Stufen Variablen mit identischem Namen zu deklarieren. Diese kaum sinnvolle Option ist in der Programmiersprache C++ vorhanden und erlaubt dort Fehler, die schwer aufzuspüren sind. In Java gehören die eingeschachtelten Blöcke zum Gültigkeitsbereich der umgebenden Blocks.

Bei der 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);
}
```

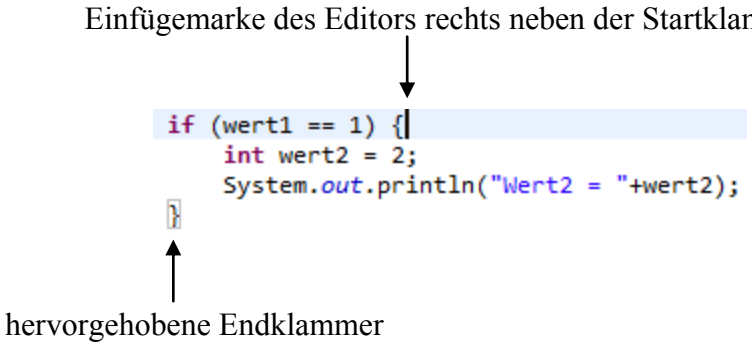
In Eclipse kann ein markierter Block aus mehreren Zeilen mit

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:

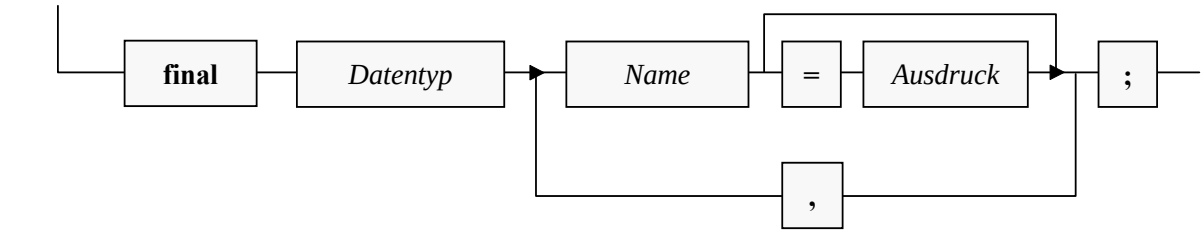
- Bei einer späteren Änderung des Wertes ist nur die Quellcodezeile mit der Variablendeklaration und -initialisierung betroffen.
- Der Quellcode ist leichter zu lesen, wenn Variablennamen an Stelle von „magischen Zahlen“ stehen.

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 Wertänderung mehr erlaubt.

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 (für mehrere Beteiligte) 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).[1]

Durch den systematischen Gebrauch des **final**-Modifikators für lokale Variablen wirken Beispielprogramme allerdings 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.[2] Nur bei Klassenvariablen von primitivem Typ mit **final**- und **public**-Modifikator ist es üblich, den Namen komplett in Großbuchstaben zu schreiben (siehe Abschnitt 4.5.1).

---

[1] 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.
- Das gelegentlich anzutreffende Argument, **final**-Deklarationen würden Code-Optimierungen durch den Compiler ermöglichen, ist vermutlich wenig relevant.
- Im funktionalen Programmierstil, den Java seit der Version 8 ermöglicht, werden finalisierte (unveränderliche) Variablen strikt bevorzugt.

[2] Siehe z.B. https://google-styleguide.googlecode.com/svn/trunk/javaguide.html

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 Nachschlage-Charakter, so dass 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 kw = 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.

Etwas 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, so dass der Eclipse-Compiler meldet:

```
Das Literal 2147483648 des Typs int liegt außerhalb des gültigen Bereichs
```

Der Kleinbuchstabe **l** ist leicht mit der Ziffer **1** zu verwechseln und daher als Suffix wenig geeignet.

Seit Java 7 dürfen bei Ganzzahlliteralen zwischen zwei Ziffern Unterstriche zur optischen Gruppierung gesetzt werden, z.B.:

```
final int kw = 4_711;
```

Weil **int**-Literale als Bestandteile der im nächsten Abschnitt behandelten Gleitkommaliterale auftreten, lässt sich die Zifferngruppierung 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 (kaum jemals erforderlichen) Suffix **D** oder **d** wird der Datentyp **double** „optisch betont".

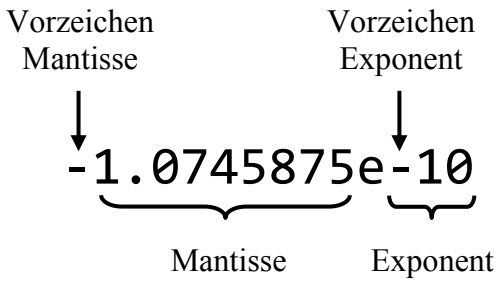
Die Java-Compiler achten bei Wertzuweisungen streng auf die Typkompatibilität. Z.B. führt die folgende Zeile:

```
final float mwst = 1.19;
```

zur folgenden Fehlermeldung des Eclipse-Compilers:[1]

```
Typabweichung: Konvertierung von double auf float nicht möglich
```

Neben der alltagsüblichen Schreibeweise erlaubt Java bei Gleitkommaliteralen auch die Exponentialnotation (mit der Basis 10), z.B. bei der Zahl -0,00000000010745875):



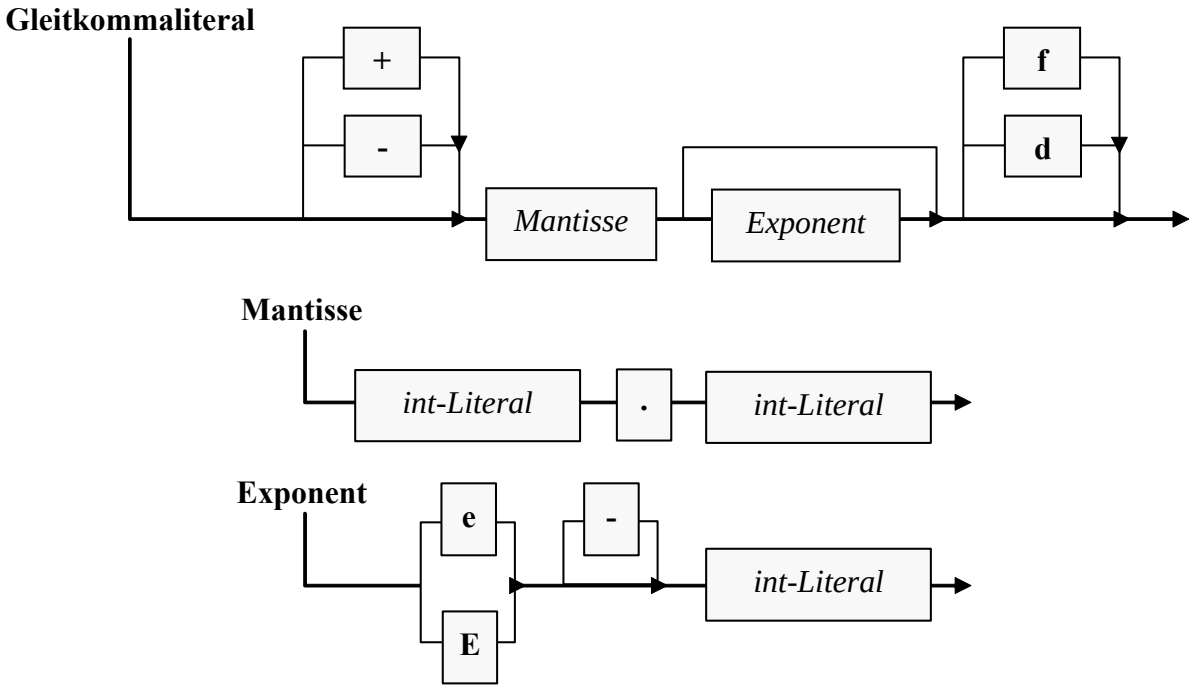
Erst diese wissenschaftliche Notation erlaubt das Gleiten des Dezimaltrennzeichens, das die Bezeichnung *Gleitkommaliteral* (engl.: *floating-point literal*) begründet.

In den folgenden Syntaxdiagrammen werden die die wichtigsten Regeln für Gleitkommaliterale beschrieben:

---

[1] Bei den Ganzzahltypen arbeitet der Compiler mit mehr Sachverstand bzw. Großzügigkeit. Im folgenden Beispiel kann das **int**-Literal 4 einer Variablen vom Typ **byte** zugewiesen werden, weil keine Wertbereichsüberschreitung stattfindet:

```
final byte b4 = 4;
```



Die in der Mantisse und im Exponenten auftretenden Ganzzahlliterale müssen das dezimale Zahlensystem verwenden und den Datentyp **int** besitzen, so dass die in 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.

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 so genannte *Escape-Sequenz*:
- **Escape-Sequenzen**
  Indem man ein Zeichen hinter einen einleitenden Rückwärts-Schrägstrich setzt (z.B. `\'`, `\n`) und damit eine so genannte *Escape-Sequenz* bildet, kann man …

- Zeichen von ihrer besonderen Bedeutung befreien (z.B. Hochkomma zur Begrenzung eines **char**-Literals) und wie ein einfaches Zeichen behandeln:

  ```
  \'
  \"
  \\
  ```

- Steuerzeichen für die Textausgabe im Konsolenfenster ansprechen, z.B.:[1]

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.

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** oder **\x**. 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 Zeichenfolgenliterale keine Sonderrolle, z.B.:

```
System.out.println("Otto's Welt");
```

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 von allen Editoren als Zeilenschaltung interpretiert wird (siehe Abschnitt 15.8.1).

Das folgende Programm enthält einen Aufruf der statischen Methode **showMessageDialog()** der Klasse **JOptionPane** aus dem Paket **javax.swing** 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:

3.3.11.6 *Referenzliteral null*

Einer Referenzvariablen kann das Referenzliteral **null** zugewiesen werden, z.B.:[2]

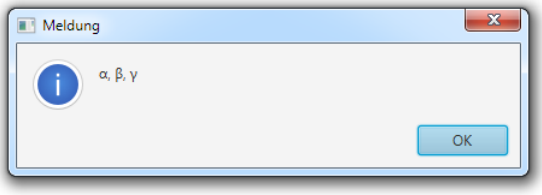
```
Bruch b1 = null;
```

---

[1] Im Manuskript wird überwiegend an Stelle des betagten GUI-Frameworks Swing die moderne Alternative JavaFX 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:



[2] 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. 2015, S. 42) 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.

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 frei gegeben.

3.4 Eingabe bei Konsolenprogrammen

Konsolenprogramme sind ein geeignetes Umfeld, um die Programmiersprache Java zu erlernen und mit der Standardbibliothek vertraut zu werden. Später 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 entgegen zu nehmen. 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** (aus dem Paket **java.util**) verwendet werden.[1] Im folgenden Beispielprogramm zur Berechnung der Fakultät zu einer ganzen Zahl wird ein **Scanner**-Objekt per **nextInt()** - Methodenaufruf gebeten, vom Benutzer eine **int**-Ganzzahl entgegen zu nehmen:

```
import java.util.Scanner;
class Prog {
 public static void main(String[] args) {
  int i, argument;
  double fakul = 1.0;
  Scanner input = new Scanner(System.in);
  System.out.print("Argument:  ");
  argument = input.nextInt();
  for (i = 1; i <= argument; i++)
   fakul = fakul * i;
  System.out.println("Fakultaet: " + fakul);
 }
}
```

Zwei Hinweise zum Quellcode:

- 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 in Abschnitt 3.7.3 behandelt.

Bei einer gültigen Eingabe arbeitet das Programm wunschgemäß, z.B.:

```
Argument:  4
Fakultaet: 24.0
```

Auf ungültige Benutzereingaben reagiert die Methode **nextInt()** mit einer so genannten Ausnahme, und das Programm „stürzt ab“, z.B.:

---

[1] Mit den Paketen der Standardbibliothek werden wir uns später ausführlich beschäftigen. An dieser Stelle dient die Angabe der Paketzugehörigkeit dazu, das Lokalisieren der Informationen zu einer Klasse in der API-Dokumentation zu erleichtern.

```
Argument:  vier
Exception in thread "main" java.util.InputMismatchException
 at java.util.Scanner.throwFor(Scanner.java:864)
 at java.util.Scanner.next(Scanner.java:1485)
 at java.util.Scanner.nextInt(Scanner.java:2117)
 at java.util.Scanner.nextInt(Scanner.java:2076)
 at Prog.main(Prog.java:8)
```

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 können, wurde für den Kurs eine Klasse namens **`Simput`** erstellt. Die zugehörige Bytecode-Datei **Simput.class** findet sich bei den Übungs- und Beispielprogrammen zum Kurs (Verzeichnis **…\BspUeb\Simput**, weitere Ortsangaben im Vorwort).

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) {
  int i, argument;
  double fakul = 1.0;
  System.out.print("Argument:  ");
  argument = Simput.gint();
  for (i = 1; i <= argument; i++)
   fakul = fakul * i;
  System.out.println("Fakultaet: " + 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, so dass die Klasse `Simput` als bequemes Hilfsmittel zur Verfügung steht. Für ernsthafte Projekte werden Sie jedoch eigene Pakete definieren (siehe Kapitel 6), so dass die (kompilierte) Klasse `Simput` dort *nicht* verwendbar ist. Diese Einschränkung ist aber leicht durch eine Änderung des Quellcodes in der Datei **Simput.java** (Verzeichnis **…\BspUeb\Simput**, weitere Ortsangaben im Vorwort) zu beheben.

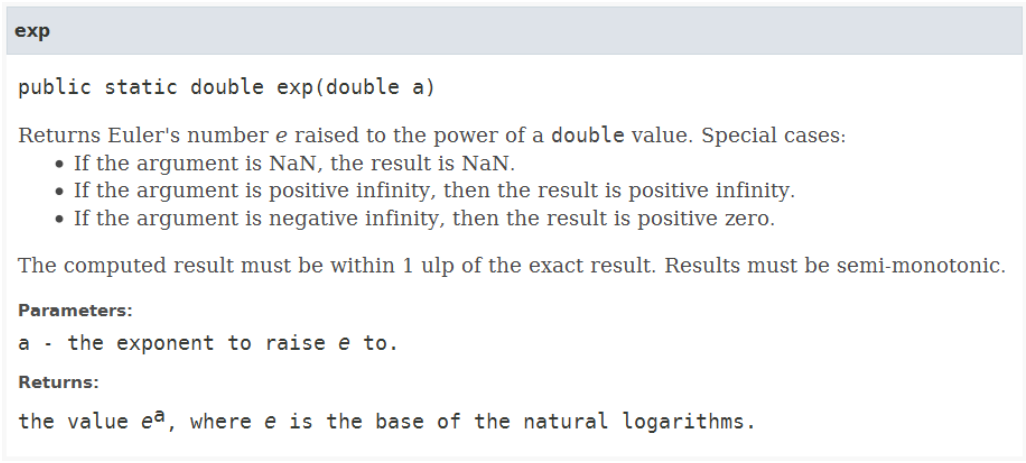
Die statische `Simput`-Methode `gint()` erwartet vom Benutzer eine per **Enter**-Taste quittierte Eingabe und versucht, diese als **int**-Wert zu interpretieren. Im Erfolgsfall erhält die aufrufende Methode das Ergebnis als `gint()` - Rückgabewert. Anderenfalls sieht der Benutzer eine Fehlermeldung, und die aufrufende Methode erhält den (Verlegenheits-)Rückgabewert 0, z.B.

```
Argument:  vier
Falsche Eingabe!

Fakultaet: 1.0
```

Die `Simput`-Klassenmethode `gint()` liefert also eine Rückgabe vom Typ **int**. Ihre Verwendung kann durch den Methodenkopf beschrieben werden:

```
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.`

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 leicht irreführende 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 in Abschnitt 3.7.2 behandelt.

Quellcode	Ein- und Ausgabe
`class Prog {` `  public static void main(String args[]) {` `  int i, argument;` `  double fakul = 1.0;` `  System.out.print("Argument:  ");` `  argument = Simput.gint();` `  if (!Simput.checkError()) {` `  for (i = 1; i <= argument; i += 1)` `  fakul = fakul * i;` `  System.out.println("Fakultaet: " +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 Dezimaltrenneichen 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

Außerdem sind Methoden mit einer Fehlerbehandlung über die Ausnahmetechnik (vgl. Kapitel 11) vorhanden (wie bei der Klasse **Scanner**).

3.4.2 Simput-Installation für die JRE, den JDK-Compiler und Eclipse

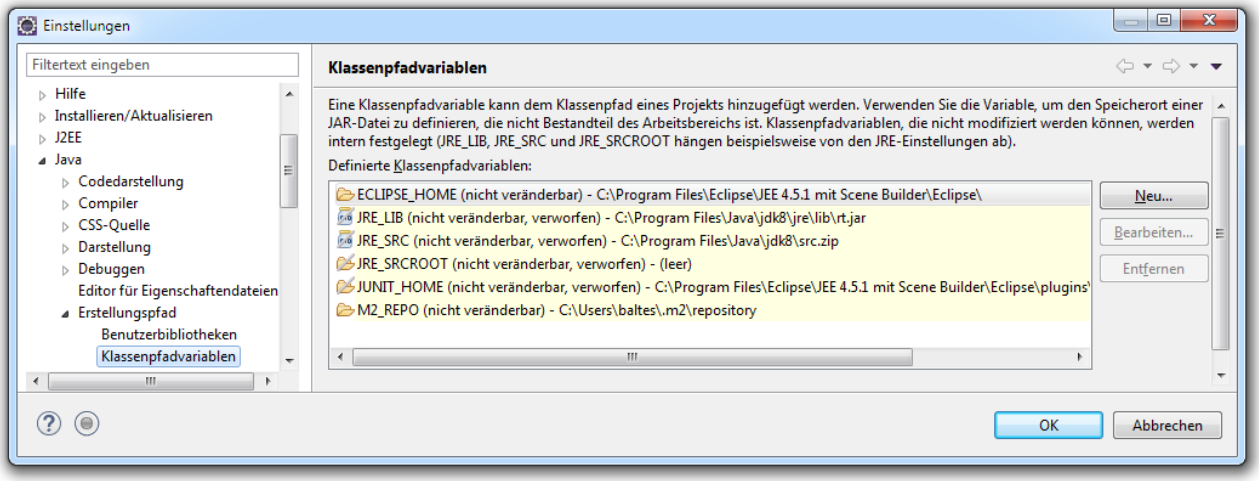
Damit beim Übersetzen durch den JDK-Compiler (**javac.exe**) und/oder beim Ausführen durch die JRE (**java.exe**) die `Simput`-Klasse mit ihren Methoden verfügbar ist, muss die Datei **Simput.class** entweder im aktuellen Verzeichnis liegen oder über die CLASSPATH-Umgebungsvariable (vgl. Abschnitt 2.2.4) auffindbar sein, wenn sie nicht bei jedem Compiler- oder Interpreter-Aufruf per **classpath**-Kommandozeilenargument zugänglich gemacht werden soll.

Unsere Entwicklungsumgebung Eclipse ignoriert die CLASSPATH-Umgebungsvariable, bietet aber alternative Möglichkeiten zur Definition eines Klassenpfads. Es hat sich als günstig erwiesen, wenn die benötigten Klassen in einer Java-Archivdatei vorliegen. Im selben Ordner wie die Bytecode-Datei **Simput.class** finden Sie daher auch die Archivdatei **Simput.jar**. Wir werden uns in Kapitel 6 mit Java-Archivdateien beschäftigen.

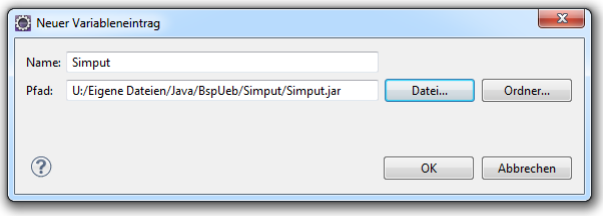
Namen und Pfad einer Archivdatei hinterlegt man am besten in einer **Klassenpfadvariablen** auf Arbeitsbereichsebene, damit das Archiv in einzelnen Projekten ohne Pfadangabe angesprochen werden kann. Nach dem Menübefehl

Fenster > Benutzervorgaben > Java > Erstellungspfad > Klassenpfadvariablen

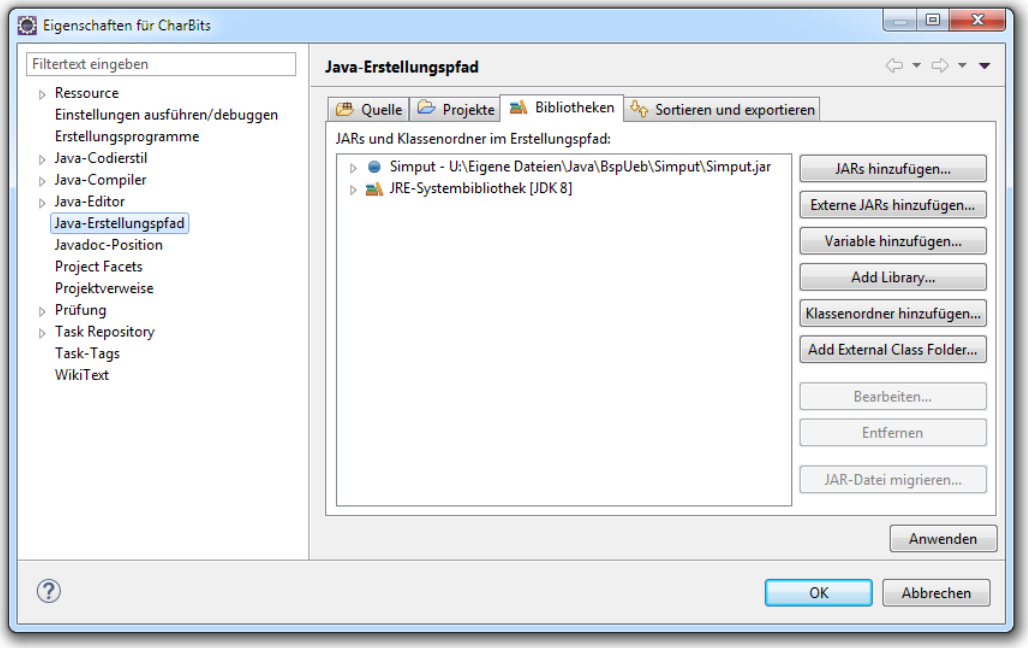
kann man in der folgenden Dialogbox



über den Schalter **Neu** die Definition einleiten, z.B.:



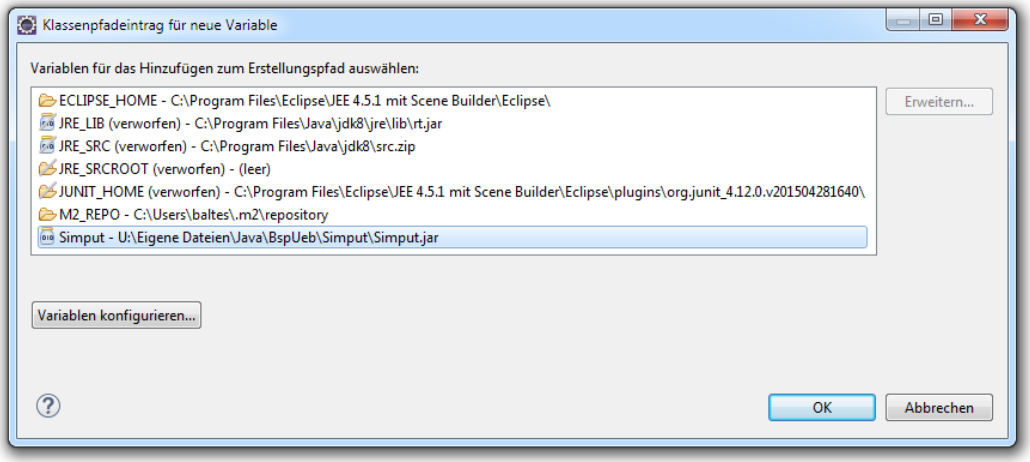
Soll ein konkretes Projekt die Klassenpfadvariable nutzen, muss diese im Eigenschaftsdialog des Projekts (z.B. erreichbar via Kontextmenü zum Projekteintrag im Paket-Explorer)



über

Java-Erstellungspfad > Bibliotheken > Variable hinzufügen

hinzugefügt werden, z.B.:

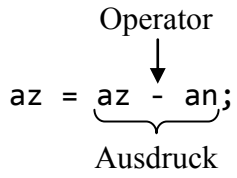


3.5 Operatoren und Ausdrücke

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 so genannten Operatoren mit ihren Argumenten veranstaltet 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. Hier werden Algorithmen bzw. 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 Wertes geeigneten, aus Operatoren und zugehörigen Argumenten aufgebauten Teil einer Anweisung, bezeichnet man als **Ausdruck**, z.B. in der folgenden Wertzuweisung:



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.[1]

Beispiele:

- `1.5`
  Dies ist ein Ausdruck mit dem Typ **double** und dem Wert 1,5.
- `Simput.gint()`
  Dieser Methodenaufruf ist ein Ausdruck mit 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).

Aus vorhandenen Ausdrücken entsteht mit Hilfe eines Operators ein komplexerer Ausdruck, wobei Typ und Wert des neuen Ausdrucks von den Argumenten und vom Operator abhängen.

Beispiele:

- `2 * 1.5`
  Hier resultiert der **double**-Wert 3,0.
- `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

---

[1] Besteht ein Ausdruck aus einem Methodenaufruf mit dem Pseudorückgabetyp **void**, dann liegt allerdings *kein* Wert vor.

haben jedoch zusätzlich einen **Nebeneffekt** auf eine als Argument fungierende *Variable*, z.B. der **Postinkrementoperator**:

```
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 gesetzt.

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 betrachten wir den **Negationsoperator**, der mit einem „**!**“ bezeichnet wird, *ein* Argument vom Typ **boolean** erwartet und dessen Wahrheitswert umdreht (**true** und **false** vertauscht):

```
!cond
```

Wir werden auch noch einen *dreistelligen* Operator kennen lernen.

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 vertrauten 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, z.B.:

Quellcode	Ausgabe
`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);` `  }` `}`	`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 kennen lernen.

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 (vgl. Abschnitt 3.5.7).

Wie der vom Compiler gewählte Arithmetiktyp und der Ergebnisdatentyp von den Datentypen der Argumente abhängen, ist der folgenden Tabelle zu entnehmen:

Datentypen der Operanden	**Verwendete Arithmetik**	**Datentyp des Ergebniswertes**
Beide Operanden haben den Typ **byte**, **short**, **char** oder **int**.	**Ganzzahlarithmetik**	**int**
Beide Operanden haben einen integralen Typ, und mind. ein Operand hat den Datentyp **long**.		**long**
Mindestens ein Operand hat den Typ **float**, keiner hat den Typ **double**.	**Gleitkomma-arithmetik**	**float**
Mindestens ein Operand hat den Datentyp **double**.		**double**

In der nächsten Tabelle sind alle arithmetischen Operatoren beschrieben, wobei die Platzhalter *Num*, *Num1* und *Num2* für Ausdrücke mit einem numerischen Typ stehen, und *Var* eine numerische *Variable* vertritt:

Operator	**Bedeutung**	**Beispiel**	
		Programmfragment	**Ausgabe**
-Num	**Vorzeichenumkehr**	`int i = 2, j = -3;` `System.out.printf("%d %d",-i,-j);`	`-2 3`
Num1 + Num2	**Addition**	`System.out.println(2 + 3);`	`5`
Num1 – Num2	**Subtraktion**	`System.out.println(2.6 - 1.1);`	`1.5`
*Num1 * Num2*	**Multiplikation**	`System.out.println(4 * 5);`	`20`
Num1 / Num2	**Division**	`System.out.println(8.0 / 5);` `System.out.println(8 / 5);`	`1.6` `1`
Num1 % Num2	**Modulo (Divisionsrest)** Sei *GAD* der ganzzahlige Anteil aus dem Ergebnis der Division (*Num1* / *Num2*). Dann ist *Num1 % Num2* def. durch *Num1 - GAD · Num2*	`System.out.println(19 % 5);` `System.out.println(-19 % 5.25);`	`4` `-3.25`
++Var *--Var*	**Präinkrement bzw. -dekrement** Als Argumente sind hier nur Variablen erlaubt. *++Var* liefert *Var* + 1 *und* erhöht *Var* um 1 *--Var* liefert *Var* – 1 *und* reduziert *Var* um 1	`int i = 4;` `double a = 0.2;` `System.out.println(++i +"\n"+` `--a);`	`5` `-0.8`

Operator	Bedeutung	Beispiel	
		Programmfragment	Ausgabe
Var++ *Var*−−	**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	`int i = 4;` `System.out.println(i++ +"\n"+ i);`	4 5

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
`class Prog {` `  public static void main(String[] args) {` `  int i = 12;` `  i++;` `  System.out.println(i);` `  }` `}`	13

Ein (De)inkrementoperator bietet keine eigenständige mathematische Funktion, sondern 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 Zwei teilbar) ist. Dazu prüft man, ob der Rest aus der Division durch Zwei gleich Null 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 = 7.1248239;` `double ganz = a - a % 1.0;` `double rest = a - ganz;` `System.out.printf("%f = %1.0f + %f", a, ganz, rest);`	`7,124824 = 7 + 0,124824`

Bei einer Modulo-Operation sind meist beide Argumente ganzzahlig, so dass nach der Tabelle auf Seite 123 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 in 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 in 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. Bei der `setzeNenner()` - Methode aus unserer Klasse `Bruch` (siehe Abschnitt 1.1.2) informiert die Rückgabe vom Typ **boolean** darüber, ob die beantragte Änderung des Nenners ausgeführt wurde.

In syntaktischer Hinsicht stellen wir fest, 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), wie Sie aus den zahlreichen Einsätzen der Methode **println()** in unseren Beispielprogrammen bereits wissen. Ein Methodenausdruck vom Typ **void** taugt also „immerhin“ zur Bildung einer Ausdrucksanweisung.

Mit den in 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 Methoden der Klasse **Math** im API-Paket **java.lang** zur Verfügung. Im folgenden Programm wird die Methode **pow()** zur Potenzberechnung genutzt:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.println(4 * Math.pow(2, 3));` `  }` `}`	`32.0`

Alle Methoden der Klasse **Math** sind als **static** deklariert, werden also von der Klasse selbst ausgeführt. Später werden wir uns ausführlich mit der Verwendung von Klassen aus den API-Paketen befassen.[1]

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 vom 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] An dieser Stelle dient die Angabe der Paketzugehörigkeit vor allem dazu, das Lokalisieren der Informationen zu einer Klasse in der API-Dokumentation zu erleichtern. 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 größeren Programms recht unangenehm werden kann.

3.5.4 Vertiefung: Gleitkommawerte vergleichen

Bei den *binären* Gleitkommatypen (**float** und **double**) muss man beim Identitätstest unbedingt technisch bedingte Abweichungen von der reinen Mathematik berücksichtigen, 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 in 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. Folglich steckt im zwischengespeicherten Berechnungsergebnis 10,0 - 9,9 ein anderer Fehler als im Speicherabbild der Zahl 0,1. Weil die Vergleichspartner nicht Bit für Bit identisch sind, meldet der Identitätsoperator ein **false**.

Mit den Objekten der in 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.

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 häufig anzutreffenden Vorschlag,

$$|d_1 - d_2|$$

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:

$$d_1 = (1 + m + e_1)\ 2^k \quad \text{und} \quad d_2 = (1 + m + e_2)\ 2^k$$

Für den Betrag des technisch bedingten relativen Fehlers gilt bei normalisierten Werten (mit einer Mantisse im Intervall [1, 2)) mit der oberen Schranke ε für den absoluten Mantissenfehler einer einzelnen **double**-Zahl die Abschätzung:

$$\left|\frac{d_1 - d_2}{d_1}\right| = \left|\frac{e_1 - e_2}{1 + m + e_1}\right| \le \frac{|e_1| + |e_2|}{|1 + m + e_1|} \le \frac{2 \cdot \varepsilon}{|1 + m + e_1|} \le 2 \cdot \varepsilon \quad (\text{wegen } (1 + m + e_1) \in [1, 2))$$

Bei normalisierten **double**-Werten (mit 52 Mantissen-Bits) ist aufgrund der begrenzten Speichergenauigkeit mit Fehlern im Bereich des halben Abstands zwischen zwei benachbarten Mantissenwerten zu rechnen:

$$2^{-53} \approx 1{,}1 \cdot 10^{-16}$$

Die vorgeschlagene Schwelle $1{,}0 \cdot 10^{-14}$ berücksichtigt über den Speicherfehler hinaus auch eingeflossene Rechnungsungenauigkeiten. Mit welcher Fehlerkumulation bzw. -verstärkung zu rechnen ist, hängt vom konkreten Algorithmus ab, so dass 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 Indifferenzrelation 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 Null 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 Grenzfällen von *einseitigen* 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 Müller (2004) oder Strey (2003).

3.5.5 Logische Operatoren

Durch Anwendung der logischen Operatoren auf bereits vorhandene logische Ausdrücke kann man neue, komplexere logische Ausdrücke erstellen. Die Wirkungsweise der logischen Operatoren wird in **Wahrheitstafeln** beschrieben (*La1* und *La2* seien logische Ausdrücke):

Argument *La1*	**Negation** **!***La1*
true	**false**
false	**true**

Argument 1 *La1*	Argument 2 *La2*	Logisches UND *La1* **&&** *La2* *La1* **&** *La2*	Logisches ODER *La1* || *La2* *La1* | *La2*	Exklusives ODER *La1* ^ *La2*
true	**true**	**true**	**true**	**false**
true	**false**	**false**	**true**	**true**
false	**true**	**false**	**true**	**true**
false	**false**	**false**	**false**	**false**

In der folgenden Tabelle gibt es noch wichtige Erläuterungen und Beispiele:

Operator	Bedeutung	Beispiel	
		Programmfragment	**Ausgabe**
!*La1*	**Negation** Der Wahrheitswert wird umgekehrt.	`boolean erg = true;` `System.out.println(!erg);`	`false`
La1 && *La2*	**Logisches UND (mit bedingter Auswertung)** *La1* && *La2* ist genau dann wahr, wenn beide Argumente wahr sind. Ist *La1* falsch, wird *La2* **nicht** ausgewertet.	`int i = 3;` `boolean erg = false && i++ > 3;` `System.out.println(erg + "\n"+i);` `erg = true && i++ > 3;` `System.out.println(erg + "\n"+i);`	`false` `3` `false` `4`
La1 & *La2*	**Logisches UND (mit *un*bedingter Auswertung)** *La1* & *La2* ist genau dann wahr, wenn beide Argumente wahr sind. Es werden auf jeden Fall **beide** Ausdrücke ausgewertet.	`int i = 3;` `boolean erg = false & i++ > 3;` `System.out.println(erg + "\n"+i);`	`false` `4`
La1 || *La2*	**Logisches ODER (mit bedingter Auswertung)** *La1* || *La2* ist genau dann wahr, wenn mindestens ein Argument wahr ist. Ist *La1* wahr, wird *La2* **nicht** ausgewertet.	`int i = 3;` `boolean erg = true || i++ == 3;` `System.out.println(erg + "\n"+i);` `erg = false || i++ == 3;` `System.out.println(erg + "\n"+i);`	`true` `3` `true` `4`
La1 | *La2*	**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.	`int i = 3;` `boolean erg = true | i++ == 3;` `System.out.println(erg + "\n"+i);`	`true` `4`
La1 ^ *La2*	**Exklusives logisches ODER** *La1* ^ *La2* ist genau dann wahr, wenn genau *ein* Argument wahr ist, wenn also die Argumente verschiedene Wahrheitswerte haben.	`boolean erg = true ^ true;` `System.out.println(erg);`	`false`

Der Unterschied zwischen den beiden logischen UND-Operatoren **&&** und **&** bzw. zwischen den beiden logischen ODER-Operatoren || und | ist für Einsteiger vielleicht etwas unklar, 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 UND-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 UND-Arguments das Gesamtergebnis **false** feststeht:

```
bv && i++ > 3
```

Das vom Programmierer nicht erwartete Ausbleiben einer Auswertung (z.B. bei „`i++`“) kann erhebliche Auswertungen auf ein Programm haben.

Mit der Entscheidung, grundsätzlich die unbedingten Operatorvarianten zu verwenden, nimmt man (mehr oder weniger relevante) Leistungseinbußen in Kauf. Eher empfehlenswert ist der Verzicht auf Nebeneffekt-Konstruktionen im Zusammenhang mit logischen Operatoren.

Dank der *bedingten* Auswertung des Operators **&&** kann man sich im rechten Operanden darauf verlassen, dass der linke Ausdruck 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:[1]

```
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.

Wie der Tabelle auf Seite 142 zu entnehmen ist, unterscheiden sich die beiden UND-Operatoren **&&** und **&** bzw. die beiden ODER-Operatoren || und | auch hinsichtlich der 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** voraussetzen. Folglich kann der Compiler mühelos 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 2015) beschränken wir uns auf ein Beispielprogramm, das zudem nützliche Einblicke in die Speicherung von **char**-Werten im Computerspeicher vermittelt. Allerdings sind Beispiel und zugehörige Erläuterungen mit einigen technischen Details belastet. Wenn Ihnen der Sinn momentan nicht da-

[1] Herkunft des Beispiels: http://introcs.cs.princeton.edu/java/11precedence/

nach 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 in 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 in 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`):

Quellcode	Ausgabe
`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);` `  }` `}`	`Zeichen:  x` `Unicode:  0000000001111000` `int-Wert: 120`

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 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 Eins enthält, wenn *beide* Argumentmuster an dieser Stelle eine 1 besitzen und anderenfalls eine 0. Bei `cbit = 'x'` ist das Unicode-Bitmuster

```
0000000001111000
```

beteiligt, 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 << i & cbit) != 0
```

Dieser logische Ausdruck wird bei einem Schleifendurchgang genau dann wahr, wenn das zum aktuellen im `i`-Wert korrespondierende Bit in der Binärdarstellung des untersuchten Zeichens den Wert 1 hat.

3.5.7 Typumwandlung (Casting) bei primitiven Datentypen

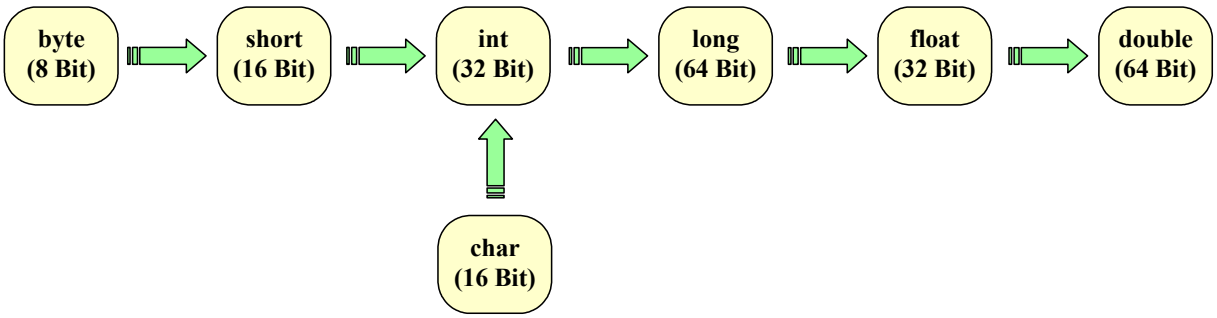
3.5.7.1 Implizite Typanpassung

Beim der Auswertung des Ausdrucks

```
2.0/7
```

trifft der Divisionsoperator auf ein **double**- und ein **int**-Argument, so dass nach der Tabelle in Abschnitt 3.5.1 (Seite 123) 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.:

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 112): 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 Typkonvertierung

Gelegentlich gibt es gute Gründe, über den so genannten **Casting-Operator** eine *explizite* Typumwandlung zu erzwingen. Im nächsten Beispielprogramm wird mit

```
(int)'x'
```

die **int**-erpretation des kleinen „x“ ermittelt, 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 ein ganzzahliger Datentyp benötigt wird. Dabei werden die Nachkommastellen abgeschnitten. Soll stattdessen ein Runden stattfinden, addiert man vor der Typumwandlung 0,5 zum Gleitkommawert.

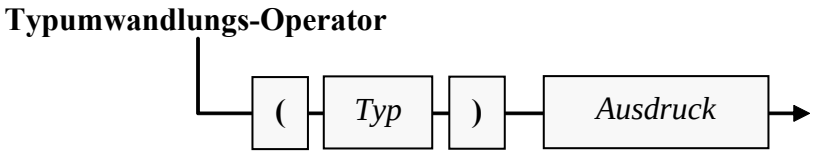
Es ist auf jeden Fall zu beachten, dass eine **einschränkende Konvertierung** stattfindet, so dass die zu erwartenden Gleitkommazahlen im Wertebereich des Ganzzahltyps liegen müssen. Wie die letzte Ausgabe zeigt, sind kapitale Programmierfehler möglich, wenn die Wertebereiche der beteiligten Variablen bzw. 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.

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. 2015, Abschnitt 5.1).

Die Java-Syntax zur expliziten Typumwandlung:



Am Rand soll noch erwähnt werden, dass die Wandlung in einen Ganzzahltyp *keine* sinnvolle Technik ist, um die Nachkommastellen in einem Gleitkommawert zu entfernen. Dazu kann man den Modulo-Operator verwenden (vgl. Abschnitt 3.5.1), ohne ein Wertebereichsproblem befürchten zu müssen, z.B.:

Quellcode	Ausgabe
`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);` `  }` `}`	85347483648,13 2147483647 85347483648,00

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.

In folgendem 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, *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 in Abschnitt 3.5.10), so dass die Anweisung

```
i = j = j * i;
```

folgendermaßen ausgeführt wird:

- Weil der Multiplikationsoperator eine höhere Bindungskraft besitzt als der Zuweisungsoperator, 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;
```

stammen übrigens *nicht* aus einem Kuriositätenkabinett, sondern sind in Java - Programmen oft 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 die Prä- und Postinkrementausdrücke (vgl. Abschnitt 3.5.1) sowie für Methodenaufrufe, jedoch *nicht* für die anderen Ausdrücke, die in 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 *kombinierte 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 einschränkende 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 in Abschnitt 3.5.1), akzeptiert der Compiler den folgenden Ausdruck 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. 2015, Abschnitt 15.26.2) findet sich die folgende Erläuterung zu der ungewohnte Laxheit des Java-Compilers:

> 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);
```

3.5.9 Konditionaloperator

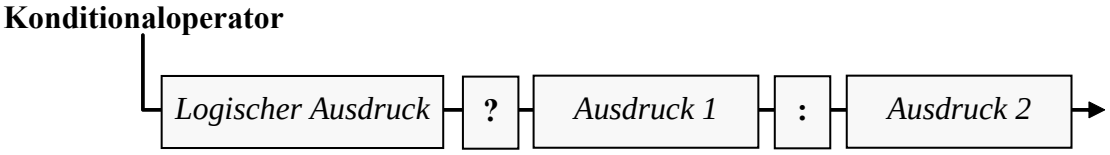
Der **Konditionaloperator** erlaubt eine sehr kompakte Schreibweise, wenn beim neuen Wert einer Zielvariablen bedingungsabhängig zwischen zwei Ausdrücken zu entscheiden ist, z.B.

$$i = \begin{cases} i + j & \text{falls } k > 0 \\ i - j & \text{sonst} \end{cases}$$

In Java ist für diese Zuweisung mit Fallunterscheidung nur eine einzige Zeile erforderlich:

Quellcode	Ausgabe
`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);` `  }` `}`	`3`

Eine Besonderheit des Konditionaloperators besteht darin, dass er *drei* Argumente verarbeitet, welche durch die Zeichen **?** und **:** getrennt werden:



Ist der logische Ausdruck *wahr*, liefert der Konditionaloperator den Wert von *Ausdruck 1*, anderenfalls den Wert von *Ausdruck 2*.

Die Frage nach dem Typ eines Konditionalausdrucks ist etwas knifflig, und in der Java 8 - Sprachspezifikation werden zahlreiche Fälle unterschieden (Gosling et al. 2015, Abschnitt 15.25). Es liegt an Ihnen, sich auf den einfachsten und wichtigsten Fall zu beschränken: Wenn der zweite und der dritte Operator denselben Typ haben, ist dies auch der Typ 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 Fehlergefahren hauptverantwortlich:

- Komplexität des Ausdrucks (Anzahl der Operatoren, Schachtelungstiefe)
- Operatoren mit Nebeneffekten

Um Problemen aus dem Weg zu gehen, sollte man also übertriebene Komplexität vermeiden und auf Nebeneffekte weitgehend verzichten.

3.5.10.1 Regeln

Nun werden die Regeln vorgestellt, nach denen der Java-Compiler einen Ausdruck mit mehreren Operatoren auswertet.

1) Bindungskraft

Bei konkurrierenden Operatoren entscheidet die Bindungskraft (siehe Tabelle unten) darüber, wie die Operanden den Operatoren zugeordnet werden. Mit `a`, `b` und `c` als Platzhaltern für Operanden (z.B. Zahlen oder Variablen) wird

```
a + b * c
```

nach der Regel „*Punktrechnung geht vor Strichrechnung*“ interpretiert als

```
a + (b * c)
```

Die hohe Bindungskraft des Postinkrementoperators führt im folgenden Beispiel

```
b * b++
```

zur Operandenzuordnung

```
b * (b++)
```

Man darf sich aber *nicht* zum voreiligen Schluss verleiten lassen, der Postinkrementoperator werde wegen seiner hohen „Priorität" vor dem Multiplikationsoperator ausgeführt. Der Postinkrementoperator hat seinen linken Nachbarn als Operanden an sich gebunden, und der resultierende Teilausdruck wird zum rechten Operanden der Multiplikation. Nach einer gleich vorzustellenden Regel wird in Java der linke Operand eines binären Operators stets vor dem rechten ausgeführt. Damit bleibt der Nebeneffekt des rechten Multiplikationsoperanden ohne Einfluss auf den linken Operanden, und wir erhalten als Wert des Ausdrucks `b`^{2}. Außerdem wird die Variable `b` inkrementiert.[1]

2) Assoziativität

Stehen mehrere Operatoren gleicher Bindungskraft zur Auswertung an, dann entscheidet deren Assoziativität über die Zuordnung der Operanden:

- Mit Ausnahme der Zuweisungsoperatoren sind alle binären Operatoren *links-assoziativ*. Z.B. wird

  ```
  x - y - z
  ```

  ausgewertet als

  ```
  (x - y) - z
  ```

- Die Zuweisungsoperatoren sind *rechts-assoziativ*. Z.B. wird

  ```
  a += b -= c = d
  ```

  ausgewertet als

  ```
  a += (b -= (c = d))
  ```

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 Assoziativitätsgesetz, so dass die Reihenfolge der Auswertung mathematisch 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 EDV-Gründen *nicht* für die die Addition und die Multiplikation von *Gleitkommatypen* (Gosling et al 2015, Abschnitt 15.7.3).

---

[1] Oft wird *falsch* angenommen, die Postinkrementoperation würde zuerst ausgeführt, und der gesamt Ausdruck würde den Wert `b*(b+1)` annehmen.

3) Links vor Rechts bei der Auswertung der Argumente eines binären Operators

Bevor ein Operator ausgeführt werden kann, müssen erst seine Argumente (Operanden) ausgewertet werden. Bei jedem binären Operator kann man sich in Java darauf verlassen, dass erst der linke Operand ausgewertet wird, dann der rechte. Kommt es bei der Auswertung des linken Operanden zu einem Ausnahmefehler (siehe unten), dann unterbleibt die Auswertung des rechten Operanden. Bei den logischen Operatoren mit bedingter Ausführung (**&&**, ||) verhindert ein bestimmter Wert des linken Operanden die Auswertung des rechten Operanden.

Das folgende, schon im Zusammenhang mit Regel 1) verwendete Beispiel zeigt, dass die hohe Bindungskraft des Postinkrementoperators (siehe Tabelle unten) *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);` `  }` `}`	`4 3`

Die Auswertung des Ausdrucks verläuft so:

- Zuerst wird der linke Operand der Multiplikation ausgewertet (Ergebnis: 2)
- Dann wird der rechte Operand der Multiplikation ausgewertet, wobei die Postinkrementoperation ausgeführt wird (Ergebnis: 2, Nebeneffekt auf die Variable `ivar`).
- Die Ausführung der Multiplikationsoperation liefert schließlich das Endergebnis 4.

Wie eine leichte Variation des letzten Beispiels zeigt, kann sich ein Nebeneffekt im *linken* Operanden einer binären Operation sehr wohl 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 mit drei Operanden (`a`, `b`, `c`) und zwei Operatoren (*, +)

```
a + b * c
```

resultiert aus der Bindungskraftregel die folgende Zuordnung der Operanden:

```
a + (b * c)
```

Zusammen mit der Links-vor-Rechts - Regel ergibt sich für die Auswertung der Operanden und die Ausführung der Operatoren die folgende Reihenfolge:

`a`, `b`, `c`, *, +

Wenn die Operanden-Platzhalter (z.B. `a`, `b`, `c`) für Zahlen oder numerische Variablen stehen, wird bei der „Auswertung“ eines Operanden lediglich sein Wert ermittelt, und die Reihenfolge der Operandenauswertung ist belanglos. Im letzten Beispiel eine falsche Auswertungsreihenfolge zu unter-

stellen (z.B. `b`, `c`, `*`, `a`, `+`), bleibt ungestraft. Wenn Operanden *Nebeneffekte* enthalten (Zuweisungen, In- bzw. Deinkrementoperationen oder Methodenaufrufe), ist die Reihenfolge der Auswertung jedoch relevant, und eine falsche Vermutung kann gravierende Fehler verursachen. Der Übersichtlichkeit halber sollte ein Ausdruck maximal *einen* Nebeneffekt enthalten.

Auch bei Beteiligung von rechts-assoziativen Operatoren erfolgt die Auswertung *der Operanden* von links nach rechts, so dass im folgenden Beispiel

```
a = b += c
```

diese Auswertungs- bzw. Ausführungsreihenfolge resultiert:

```
a, b, c, +=, =
```

Allerdings müssen die Operanden `a` und `b` Variablen sein, so dass bei deren Auswertung nichts passiert außer der Zwischenspeicherung des alten Wertes.

4) Runde Klammern

Wenn aus der Prioritäts- und der Assoziativitätsregel nicht die gewünschte Operandenzuordnung resultiert, greift man mit runden Klammern steuernd ein, wobei auch eine Schachtelung erlaubt ist. Durch Klammern werden Terme zu *einem* Operanden zusammengefasst, so dass die *internen* Operationen ausgeführt sind, bevor der Klammerausdruck von einem *externen* Operator verarbeitet wird.

Die oft anzutreffende Behauptung, Klammerausdrücke würden generell zuerst ausgewertet, ist hingegen 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 verläuft so:

- Wegen der Links-vor-Rechts - Regel wird zuerst wird 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 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 in absteigender Priorität (Bindungskraft) aufgelistet. Gruppen von Operatoren mit gleicher Priorität sind durch horizontale Linien voneinander abgegrenzt. 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-Shift	*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*

Operator	Bedeutung	Operanden
=	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 in 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, so dass 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 streng genommen nicht zu den elementaren Sprachelementen gehören. In diesem Sinn wurde schon im Abschnitt 3.3.7.2 demonstriert, dass die Klasse **BigDecimal** bei finanzmathematischen Anwendungen wegen ihrer beliebigen Genauigkeit zu bevorzugen ist.

3.6.1 Überlauf bei Ganzzahltypen

Ohne besondere Vorkehrungen stellt ein Java-Programm im Falle eines Ganzzahlüberlaufs keinesfalls seine Tätigkeit (z.B. mit einem Ausnahmefehler) ein, sondern arbeitet munter weiter.[1] Dieses Verhalten ist beim Programmieren von Pseudozufallszahlgeneratoren willkommen, ansonsten aber eher bedenklich. Das folgende Programm

```
class Prog {
   public static void main(String[] args) {
      int i = 2147483647, j = 5, k;
      k = i + j; // Überlauf!
      System.out.println(i + " + " + j + " = " + k);
   }
}
```

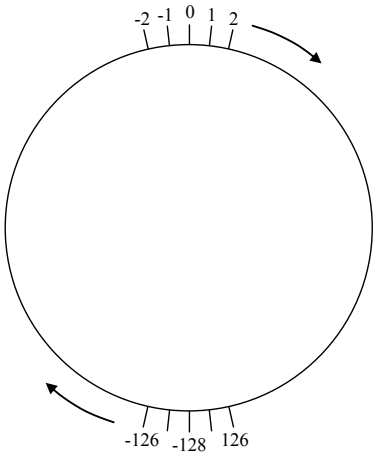
liefert ohne jede Warnung das fragwürdige Ergebnis:

```
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

[1] Ein Entsprechung zur **checked**-Option in C# (siehe Baltes-Götz 2011) steht in Java leider noch nicht zur Verfügung.

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]

Oft kann ein Überlauf durch Wahl eines geeigneten Datentyps verhindert werden. Mit den Deklarationen

```
long i = 2147483647, j = 5, k;
```

erhält man das korrekte Ergebnis, weil neben `i`, `j` und `k` nun auch der Ausdruck `i+j` den Typ **long** hat:

```
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 = 2147483647, 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, 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. 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.

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.println("Double.MAX_VALUE =\t" + bigd);
  bigd = Double.MAX_VALUE * 10.0;
  System.out.println("Double.MaxValue * 10 =\t" + bigd);
  System.out.println("Unendlich + 10 =\t" + (bigd + 10));
  System.out.println("Unendlich * (-1) =\t" + (bigd * -1));
  System.out.println("13.0/0.0 =\t\t" + (13.0 / 0.0));
 }
}
```

liefert die Ausgabe:

```
Double.MAX_VALUE =        1.7976931348623157E308
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.println("Unendlich – Unendlich =\t"+(bigd-bigd));
  System.out.println("Unendlich / Unendlich =\t"+(bigd/bigd));
  System.out.println("Unendlich * 0.0 =\t" + (bigd * 0.0));
  System.out.println("0.0 / 0.0 =\t\t" + (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
`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));` `  }` `}`	`true` `true`

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 der Paket **java.math** noch rechnen:

Quellcode	Ausgabe
`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);` `  }` `}`	`Very Big: 1.057066e+3000`

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 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 **BigDecimal**-Objekten aber auch ein erhöhter Speicher- und Rechenaufwand (siehe Abschnitt 3.3.7.2), so dass 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 JRE kann Über- bzw. Unterlaufprobleme reduzieren. Andererseits geht aber die Plattformunabhängigkeit der Rechenergebnisse verloren.

Nach Gosling et al (2015, Abschnitt 15.4) ist einer JRE 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 JRE 8 ist es mir auf einem Rechner mit Intel-CPU (Core i7; 2,8 GHz) unter Windows 7 (64 Bit) *nicht* gelungen, einen Effekt des **strictfp**-Modifikators zu beobachten. Das folgende Beispielprogramm[1]

```
public strictfp class FpDemo3  {
   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 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.[2] 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**:

```
public static strictfp double toRadians(double angdeg) {
  return angdeg / 180.0 * PI;
}
```

---

[1] Das Programm stammt von einer ehemaligen Webseite der Firma Sun Microsystems, die nicht mehr abrufbar ist.

[2] Diese überzeugende Schlussfolgerung stammt von der Webseite http://stackoverflow.com/questions/22562510/does-java-strictfp-modifier-have-any-effect-on-modern-cpus.

3.7 Anweisungen (zur Ablaufsteuerung)

Wir haben uns im Kapitel 1 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** 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 (Verzweigungen und Schleifen) kennen lernen.

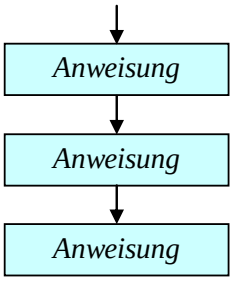
3.7.1 Überblick

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 Variablendeklarationsanweisung wurde schon in 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(la1);`
    Besitzt die aufgerufene Methode einen Rückgabewert (siehe unten), 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 in 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 gemeinsamen 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 in Kapitel 1 bestanden meist aus einer *Sequenz* von Anweisungen, die bei jedem Programmlauf komplett und linear durchlaufen wurde:



  Oft möchte man jedoch z.B.
  - die Ausführung einer Anweisung (eines Anweisungsblocks) von einer *Bedingung* abhängig machen
  - oder eine Anweisung (einen Anweisungsblock) *wiederholt* ausführen lassen.

  Für solche Zwecke stellt Java etliche Anweisungen zur Ablaufsteuerung zur Verfügung, 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.

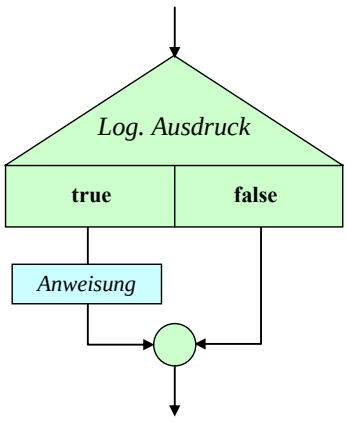
Anweisungen werden durch ein **Semikolon** abgeschlossen, sofern sie nicht mit einer schließenden Blockklammer enden.

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.

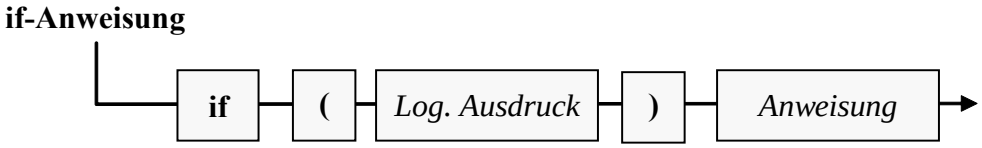
3.7.2.1 if-Anweisung

Nach dem folgenden **Programmablaufplan** (**PAP**) bzw. **Flussdiagramm** soll eine (Block-)Anweisung nur dann ausgeführt werden, wenn ein logischer Ausdruck den Wert **true** besitzt:



Wir werden diese Darstellungstechnik ab jetzt verwenden, um einen Algorithmus oder einen Programmablauf zu beschreiben. Die verwendeten Symbole sind hoffentlich anschaulich, entsprechen aber keiner strengen Normierung.

Das folgende Syntaxdiagramm beschreibt die zur Realisation einer bedingten Ausführung geeignete **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:

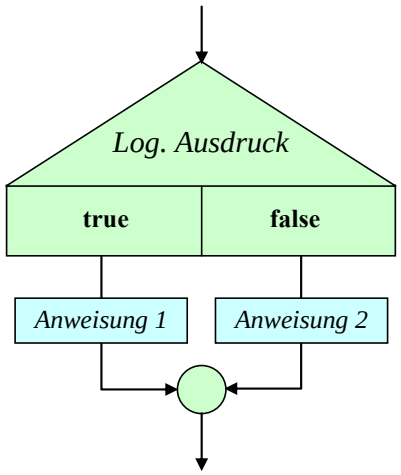
```
if (anz == 0)
  System.out.println("Die Anzahl muss > 0 sein!");
```

Der Zeilenumbruch zwischen dem logischen Ausdruck und der Anweisung dient nur der Übersichtlichkeit und ist für den Compiler irrelevant.

Selbstverständlich kommt als Anweisung auch ein *Block* in Frage.

3.7.2.2 if-else - Anweisung

Soll auch etwas passieren, wenn der steuernde logische Ausdruck den Wert **false** besitzt,



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.

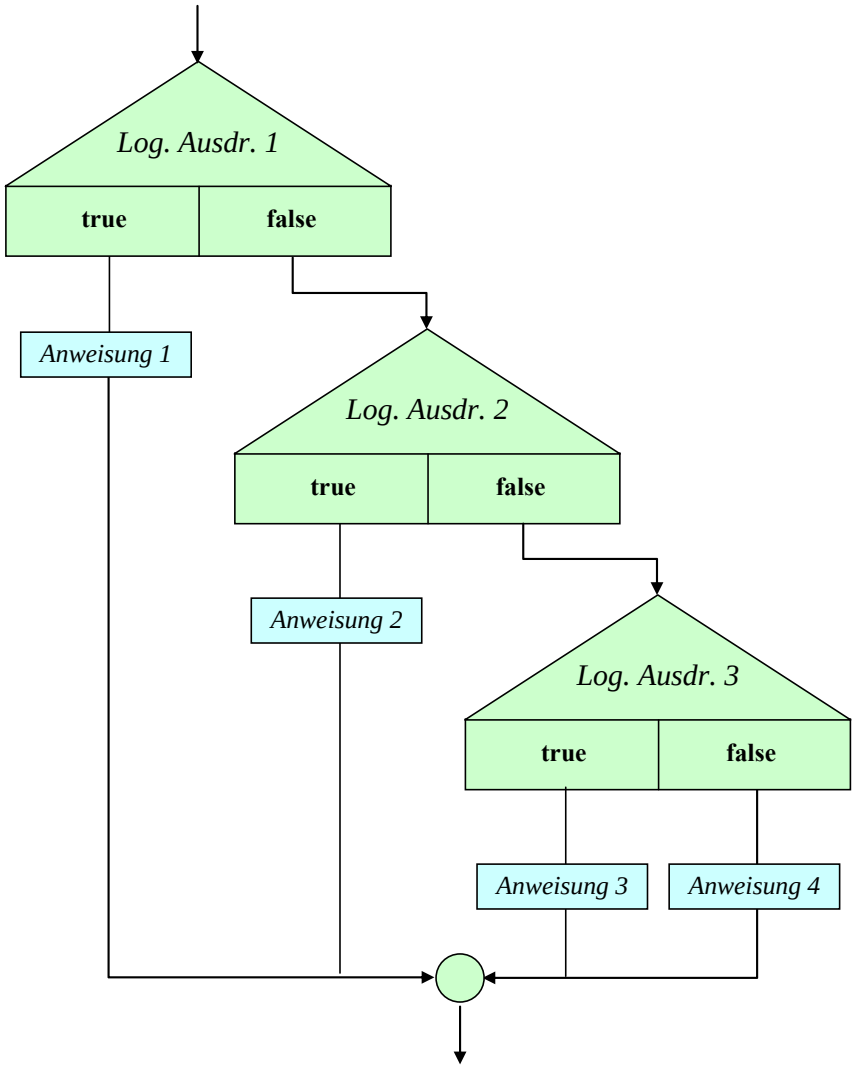
Während die Syntaxbeschreibung im Quellcode-Layout relativ einfache Bildungsregeln (mit einer einzigen zulässigen Sequenz) sehr anschaulich beschreiben kann, bietet das Syntaxdiagramm den Vorteil, bei komplizierter, variantenreicher Syntax alle zulässigen Sequenzen kompakt und präzise als Pfade durch das Diagramm zu beschreiben.

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).

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 <= 0!");` `  }` `}`	`Argument: 2,4` `ln(2,400) = 0,875`

Eine bedingt auszuführende Anweisung darf durchaus wiederum vom **if-** bzw. **if-else** - Typ sein, so dass sich mehrere, *hierarchisch geschachtelte* Fälle unterscheiden lassen. Den folgenden Programmablauf mit „sukzessiver Restaufspaltung“



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.

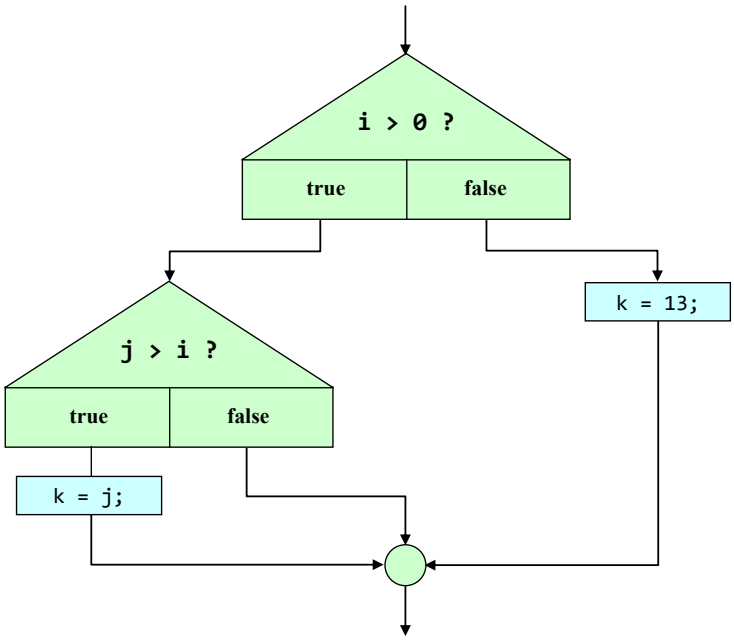
Gerade wurde eine zusammengesetzte Anweisung mit spezieller Bauart als Beispiel vorgeführt. Es ist z.B. keinesfalls allgemein vorgeschrieben, dass alle beteiligten **if**-Anweisungen eine **else**-Klausel haben müssen.

Bei einer Mehrfallunterscheidung ist die in 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 Compiler und Programmierer hinsichtlich der Zuordnung einer **else**-Klausel besteht. Im folgenden Code-Fragment

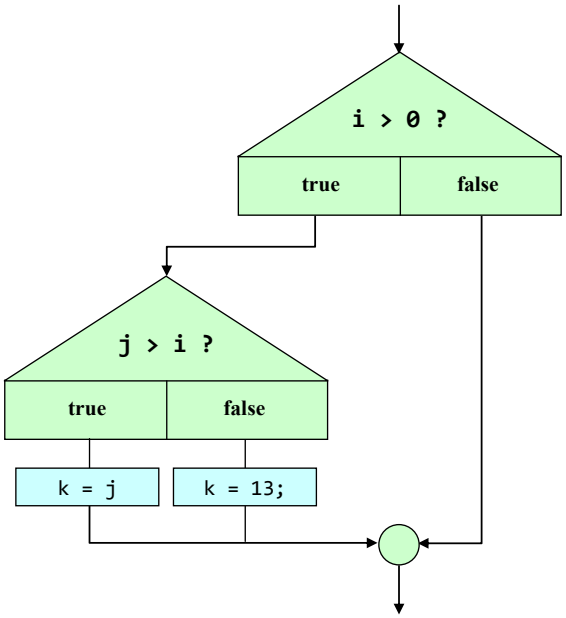
```
if (i > 0)
  if (j > i)
    k = j;
else
  k = 13;
```

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, so dass de facto folgender Programmablauf resultiert:

[1] Deutsche Übersetzung von *dangling*: *baumelnd*.



Bei `i` ≤ 0 geht der Programmierer vom neuen `k`-Wert 13 aus, der beim tatsächliche Programmablauf 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:

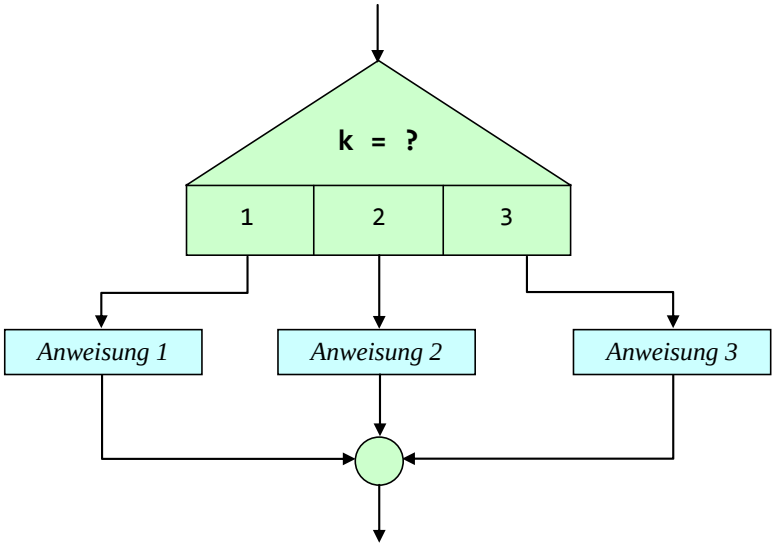
```
if (i > 0)
  if (j > i)
    k = j;
  else
    ;
else
  k = 13;
```

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 in Abhängigkeit vom Wert *eines* Ausdrucks vorgenommen werden soll,

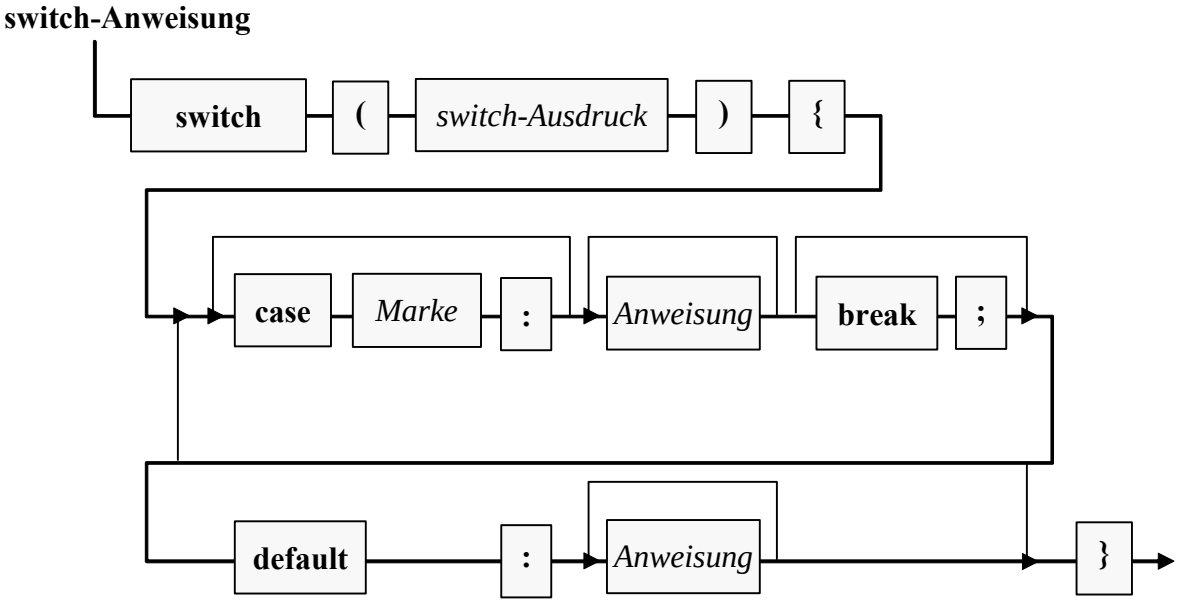


dann ist eine **switch**-Anweisung weitaus handlicher als eine verschachtelte **if-else** - Konstruktion.

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**!)
- Aufzählungstypen (siehe unten)
- Verpackungsklassen (siehe unten) für integrale primitive Datentypen mit maximal 4 Bytes:
  Byte, **Short**, **Character** oder **Integer** (nicht **Long**!)
- Ab Java 7 sind auch Zeichenfolgen (Objekte der Klasse **String**) erlaubt.

Der Genauigkeit halber 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.



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:

Quellcode	Ausgabe
`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");` `  }` `  }` `}`	`Fall 2 (mit Durchfall)` `Fälle 3 und 4`

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 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. 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.

Soll für mehrere Werte des **switch**-Ausdrucks dieselbe Anweisung ausgeführt werden, setzt man die zugehörigen **case**-Marken 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 die Lieblingszahl berücksichtigt:

```
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 ein emotionaler Typ.");
    break;
   case "schwarz":
    System.out.println("Nehmen Sie nicht alles so tragisch.");
    break;
   default: {
    System.out.println("Sie scheinen ein sachlicher Typ zu sein.");
    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, 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):

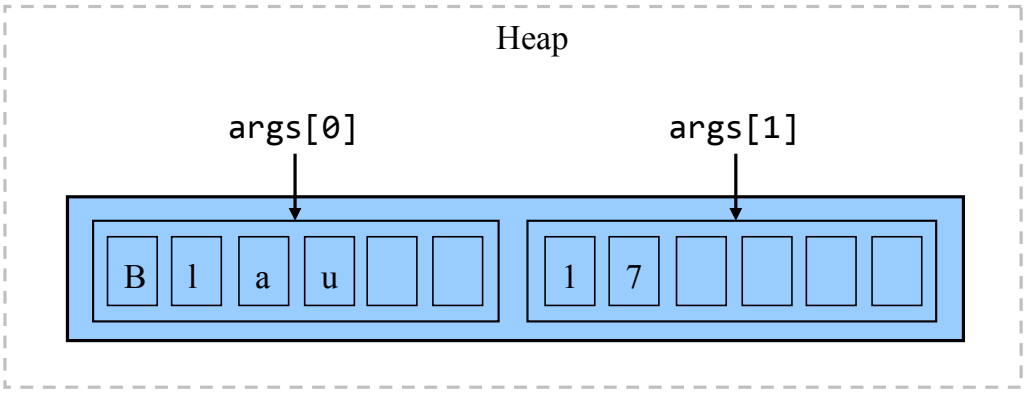
- Bei einem **Array** handelt es sich um ein Objekt, das eine Serie von Elementen desselben Typs aufnimmt, auf die man per Index, d.h. durch die mit eckigen Klammern begrenzte Elementnummer, zugreifen kann.
- In unserem Beispiel kommt ein Array mit Elementen vom Datentyp **String** zum Einsatz, wobei 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 lokale Variablen.
- Beim Programmstart werden der Methode **main()** von der Java Runtime Environment (JRE) 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 **toLower()** auszuführen. 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 **toLower()** - 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.

Nach einem Programmstart mit dem Aufruf

```
java PerST Blau 17
```

kann man sich den **String**-Array `args`, der als Objekt im Heap-Bereich des programmeigenen Speichers abgelegt wird, ungefähr so vorstellen:

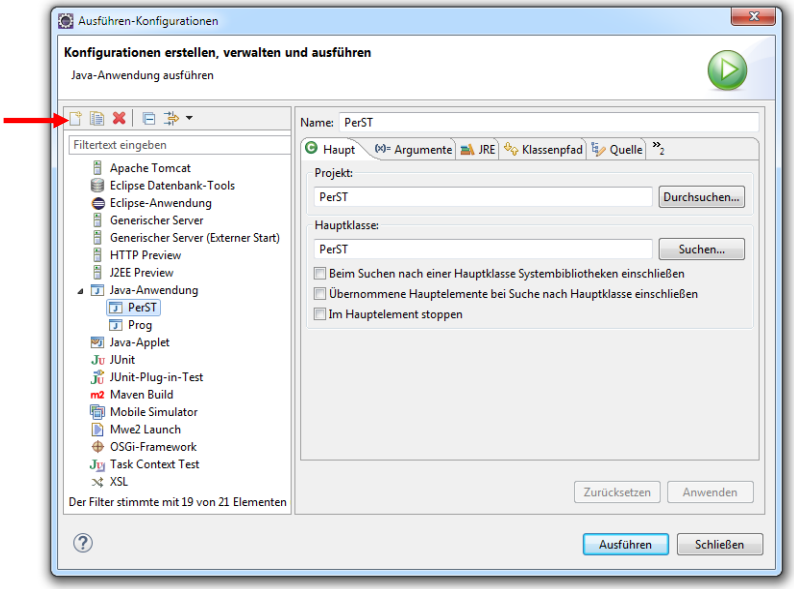


3.7.2.4 Eclipse-Startkonfigurationen

Um das im letzten Abschnitt vorgestellte Programm `PerST` in der Eclipse-Entwicklungsumgebung starten zu können, müssen Sie eine neue **Startkonfiguration** anlegen und dort die benötigten Programmargumente eingetragen. Auch bei unseren früheren Eclipse-Projekten entstand jeweils eine neue Startkonfiguration, wobei aber lediglich beim ersten Start der Projekttyp **Java-Anwendung** anzugeben war.

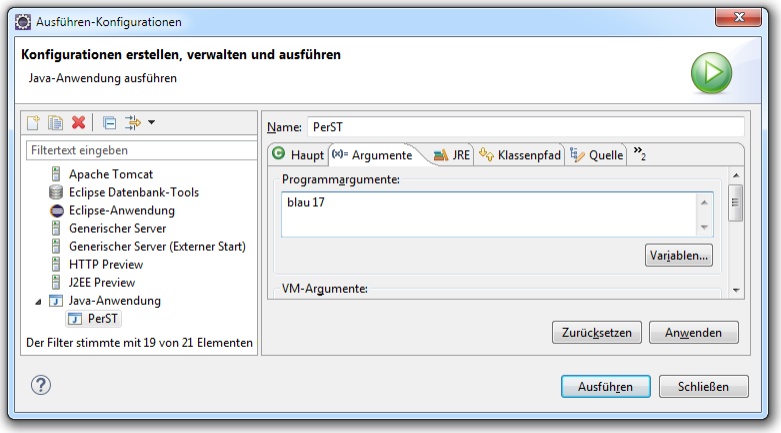
Gehen Sie folgendermaßen vor, nachdem Sie das Java-Projekt `PerST` mit der gleichnamigen Startklasse erstellt (siehe Abschnitt 2.5.4) oder importiert (siehe Abschnitt 2.5.9) haben:

- Öffnen Sie im Editor die Quellcodedatei **PerST.java** (siehe Paket-Explorer, Standardpaket zum Projekt `PerST`).
- Menübefehl **Ausführen > Ausführungskonfigurationen**
- In der Dialogbox **Ausführen-Konfigurationen** muss zunächst über den Befehlsschalter eine **neue Startkonfiguration** angefordert werden:



Weil das Projekt `PerST` in Bearbeitung ist, nimmt Eclipse passende Eintragungen vor.

- Tragen Sie auf der Registerkarte **Argumente** die benötigten **Programmargumente** ein, z.B.:



- Nun können Sie das Programm `PerST` mit der neuen Startkonfiguration gleich **ausführen** lassen.

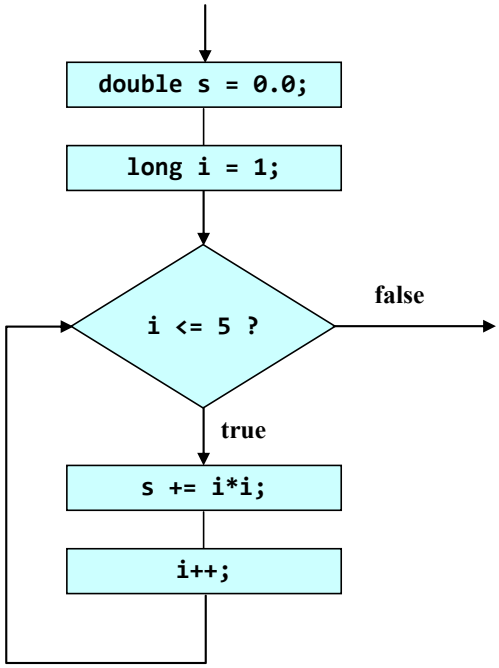
Für spätere Starts genügt bei geöffneter Quellcodedatei **PerST.java** ein Klick auf den Schalter  .

Über die Dialogbox **Ausführen Konfigurationen** ist es jederzeit möglich, eine Startkonfiguration zu ändern, zu löschen oder zu duplizieren.

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 bietet Java verschiedene Wiederholungsanweisungen (jeweils bestehend aus einer Schleifensteuerung und der 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.
- **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 unten) ausführen zu lassen.

[1] Das Verzweigungssymbol sieht aus darstellungstechnischen Gründen etwas anders aus als in 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) kurz 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.

- **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 oder nicht.

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, z.B.:
    ```
    i = 1
    ```
  - eine Variablendeklaration mit Initialisierung, z.B.
    ```
    long i = 1
    ```

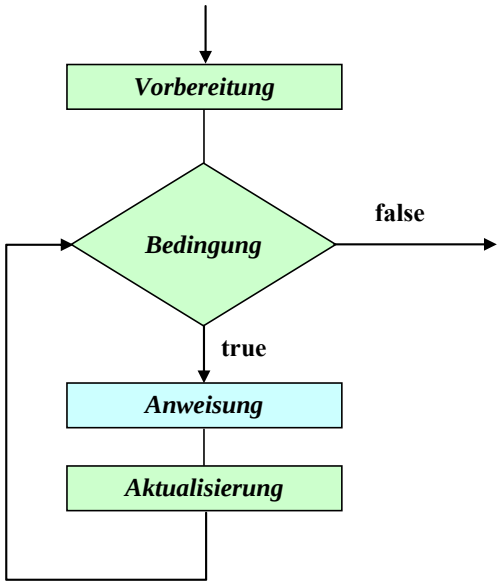
  Im folgenden Programm, das die Summe der quadrierten natürlichen Zahlen von 1 bis 5 berechnet, kommt die zweite Variante zum Einsatz:

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, steht also nur in deren Anweisung(sblock) zur Verfügung.

- ***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 der Aktualisierungsteil ausgeführt. Hier 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 umsetzt, gehören **for**-Schleifenköpfe ohne Vorbereitung oder ohne Aktualisierung, wobei die trennenden Strichpunkte trotzdem zu setzen sind.

3.7.3.2 Iterieren über die Elemente einer ***Kollektion***

Obwohl wir uns bisher mit *Arrays* (Objekten, die eine feste Anzahl von Elementen desselben Datentyps enthalten) nur anhand eines Beispiels und mit anderen Kollektionen noch gar nicht beschäftigt haben, soll die mit Java 5 (bzw. 1.5) eingeführte **for**-Schleifen - Variante für Kollektionen doch hier im Kontext mit den übrigen Wiederholungsanweisungen behandelt werden. Konzentrieren Sie sich also 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` in 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 **for**-Schleife für Kollektionen jedes Element im **String**-Array `args` mit den Programmargumenten ausgegeben:

Quellcode	Ausgabe nach einem Start mit `java Prog eins zwei drei`
`class Prog {` `  public static void main(String[] args) {` `  for (String s : args)` `  System.out.println(s);` `  }` `}`	`eins` `zwei` `drei`

Die Syntax der **for**-Variante für Kollektionen:

for (*Elementtyp Iterationsvariable* **:** *Kollektion***)**
Anweisung

Als Kollektion erlaubt der Compiler:

- einen Array (siehe Abschnitt 5.1)
- ein Objekt einer Klasse, welche das Interface **Iterable<T>** implementiert

Im Schleifenkopf wird eine Iterationsvariable vom Datentyp der Kollektionselemente deklariert. Die Anweisung wird nacheinander für jedes Element der Kollektion ausgeführt, wobei die Iterationsvariable das gerade in Bearbeitung befindliche Kollektionselement anspricht.

3.7.3.3 Bedingungsabhängige Schleifen

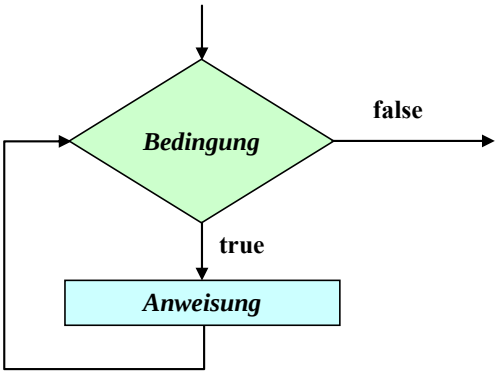
Wie die Erläuterungen zur **for**-Schleife gezeigt haben, ist die Überschrift dieses Abschnitts nicht sehr trennscharf, weil bei der **for**-Schleife ebenfalls eine beliebige Terminierungsbedingung angegeben werden darf. In vielen Fällen ist es eine Frage des persönlichen Geschmacks, welche Wiederholungsanweisung man zur Lösung eines konkreten Iterationsproblems benutzt. Unter der aktuellen Abschnittsüberschrift diskutiert man traditionsgemäß die **while**- und die **do**-Schleife.

3.7.3.3.1 while-Schleife

Die **while**-Anweisung kann als vereinfachte **for**-Anweisung beschreiben 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:

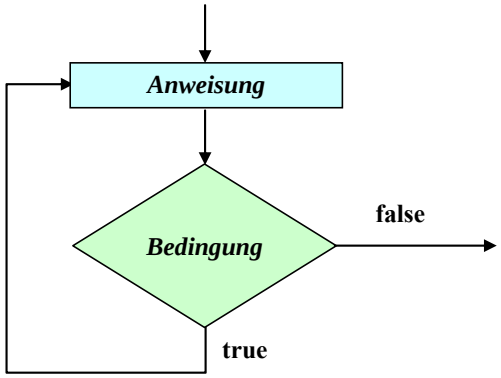


Das in Abschnitt 3.7.3.1 vorgestellte Beispielprogramm zur Quadratsummenberechnung mit Hilfe einer **for**-Schleife kann leicht auf die Verwendung einer **while**-Schleife umgestellt werden:

Quellcode	Ausgabe
`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);` `  }` `}`	`Quadratsumme = 55.0`

3.7.3.3.2 do-Schleife

Bei der **do**-Schleife wird die Fortsetzungsbedingung *am Ende* der Schleifendurchläufe geprüft, so dass wenigstens *ein* Durchlauf stattfindet:



Das Schlüsselwort **while** tritt auch in der Syntax zur **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' );
```

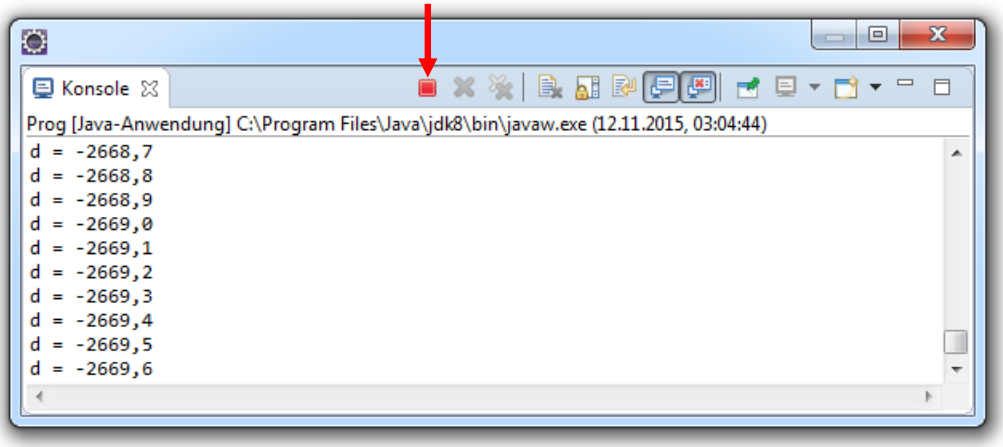
Bei einer **do**-Schleife mit Anweisungs*block* sollte man die **while**-Klausel unmittelbar hinter die schließende Blockklammer setzen (in dieselbe Zeile), um sie optisch von einer selbständigen **while**-Anweisung abzuheben (siehe Beispiel).

3.7.3.4 Endlosschleifen

Bei einer Wiederholungsanweisung (**for**, **while** oder **do**) kann es in Abhängigkeit von der verwendeten Bedingung passieren, dass der Anweisungsteil unendlich oft ausgeführt wird. In folgendem Beispiel resultiert eine Endlosschleife aus einer ungeschickten Identitätsprüfung bei **double**-Werten (vgl. Abschnitt 3.5.4):

```
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 > 1.0e-14)
  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 aus Eclipse gestartet, klickt man stattdessen auf den roten Knopf im Konsolenfenster:



3.7.3.5 Schleifen(durchgänge) vorzeitig beenden

Mit der **break**-Anweisung, die uns schon in Abschnitt 3.7.2.3 als Bestandteil der **switch**-Anweisung begegnet ist, kann eine Schleife vorzeitig verlassen werden. Mit der **continue**-Anweisung veranlasst man Java, den aktuellen Schleifendurchgang zu beenden und sofort mit dem nächsten zu beginnen (bei **for** und **while** nach Prüfung der Fortsetzungsbedingung). In der Regel kommen **break** und **continue** im Rahmen einer **if**-Anweisung zum Einsatz, z.B. in folgendem 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);  //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 fuer den Einsatz dieser Software!");
 }
}
```

Die zu untersuchende Zahl erfragt das Programm mit der statischen Methode `glong()` der in Abschnitt 3.4.2 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 `Simput`-Methode `checkError()`. Ist kein Fehler aufgetreten, liefert `checkError()` den Wert **false** zurück.

Bei einer irregulären Eingabe erscheint eine Fehlermeldung auf der Konsole, und der aktuelle Durchgang der **while**-Schleife wird per **continue** verlassen. Durch Eingabe der Zahl 0 kann das Programm beendet werden, wobei die absichtlich konstruierte **while** - „Endlosschleife“ per **break** verlassen wird.

Man hätte die **continue**- und die **break**-Anweisung zwar vermeiden können (siehe Übungsaufgabe auf Seite 177), doch werden bei dem 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 also einstellen, und z ist eine Primzahl.

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.

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 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 in Abschnitt 3.3.11.5 verwendet.

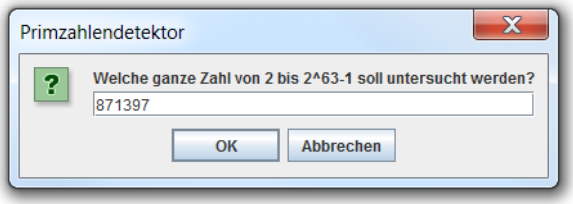
Die Klasse **JOptionPane** aus dem Paket **javax.swing** an dieser Stelle trotz der erklärten Absicht zum Wechsel von der traditionellen GUI-Lösung *Swing* auf die moderne Alternative *JavaFX* zu verwenden, wirkt inkonsequent. Allerdings wurde schon in Abschnitt 3.3.11.5 demonstriert, dass es nur mit Swing 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 sehr vielen vorhandenen Programmen anzutreffen ist.

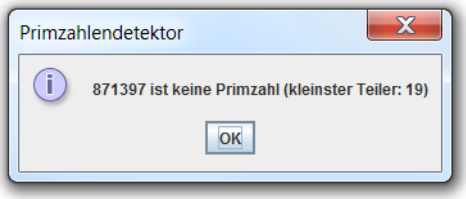
Wir erstellen zum Primzahldiagnoseprogramm aus Abschnitt 3.7.3.5 mit erstaunlich geringem Aufwand die folgende Variante

```
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);   //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 den 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.[1]

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(argument) + " 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:

[1] Derartige Konvertierungsmethoden werden in Abschnitt 5.3.3 offiziell behandelt.

<table>
<tr><th>Name</th><th>Erläuterung</th></tr>
<tr><td>parentComponent</td><td>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.</td></tr>
<tr><td>message</td><td>Dieser Text erscheint in der Dialogbox.</td></tr>
<tr><td>title</td><td>Dieser Text erscheint in der Titelzeile der Dialogbox.</td></tr>
<tr><td>messageType</td><td>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:
<table>
<tr><th>JOptionPane-Konstante</th><th>int</th></tr>
<tr><td>JOptionPane.PLAIN_MESSAGE</td><td>-1</td></tr>
<tr><td>JOptionPane.ERROR_MESSAGE</td><td>0</td></tr>
<tr><td>JOptionPane.INFORMATION_MESSAGE</td><td>1</td></tr>
<tr><td>JOptionPane.WARNING_MESSAGE</td><td>2</td></tr>
<tr><td>JOptionPane.QUESTION_MESSAGE</td><td>3</td></tr>
</table>
</td></tr>
</table>

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. „3,14“, „9223372036854775808“).
- Der Benutzer hat den Input-Dialog abgebrochen (auf die Schaltfläche **Abbrechen** 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.lang.Long.parseLong(Long.java:552)
 at 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 JRE-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 in Abschnitt 1.2.4 beschrieben.

Von den zahlreichen weiteren Möglichkeiten der Klasse **JOptionPane** (siehe API-Dokumentation) soll noch die statische Methode **showConfirmDialog()** erwähnt werden:

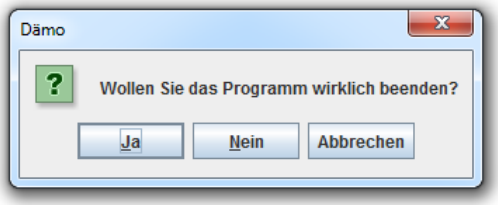
public static int showConfirmDialog(Component *parentComponent*,
Object *message*, **String** *title*, **int** *optionType*)

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 Benut-

zerwunsch 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 ansprechbar:

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	**JOptionPane.CLOSED_OPTION**	-1
Ja	**JOptionPane.YES_OPTION**	0
Nein	**JOptionPane.NO_OPTION**	1
Abbrechen	**JOptionPane.CANCEL_OPTION**	2

3.9 Übungsaufgaben zu Kapitel 3

Abschnitt 3.1 (Einstieg)

1) Welche **main()** - Varianten sind zum Starten einer Applikation geeignet?

```
public static void main(String[] irrelevant) { … }
public void main(String[] argz) { … }
public static void main() { … }
static public void main(String[] argz) { … }
public static void Main(String[] argz) { … }
```

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, so dass 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) In folgendem 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?

Quellcode	Ausgabe
`class Prog {` `  public static void main(String args[]) {` `  char z = 99;` `  int i = z;` `  System.out.println("z = "+z+"\ni = "+i);` `  }` `}`	`z = c` `i = 99`

Wie kann man das Zeichen *c* über eine Unicode-Escape-Sequenz ansprechen?

3) Wieso klagt der Eclipse-Compiler über ein unbekanntes Symbol, obwohl die Variable `i` deklariert worden ist?

Quellcode	Fehlermeldung des Eclipse-Compilers
`class Prog {` `  public static void main(String[] args) {{` `  int i = 2;` `  }` `  System.out.println(i);` `  }` `}`	`i cannot be resolved to a variable`

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 Flaecheninhalt betraegt: +PI*radius*radius);
  }
}
```

Abschnitt 3.4 (Eingabe bei Konsolen)

1) Führen Sie die in Abschnitt 3.4.2 beschriebene Eclipse-Konfiguration aus, und lassen Sie das in 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 in folgendem 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);
 }
}
```

Tipp: Die Negation von zusammengesetzten Ausdrücken ist etwas unangenehm. Mit Hilfe der Regeln von **DeMorgan** kommt man zu äquivalenten Ausdrücken, die leichter zu interpretieren sind:

$!(La1 \;\&\&\; La2) \quad = \quad !La1 \;||\; !La2$

$!(La1 \;||\; La2) \quad = \quad !La1 \;\&\&\; !La2$

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 (vgl. 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.

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 entgegen zu nehmen.

Abschnitt 3.6 (Über- und Unterlauf bei numerischen Variablen)

1) Welche von den 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?

Quellcode	Ausgabe
`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.ausg.: b ist "+b);` `  }` `}`	`b ist true` `Kontr.ausg.: b ist false`

2) Das folgende Programm soll Buchstaben nummerieren:

Quellcode	Ausgabe
`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+` `  " gehoert die Nummer "+nr);` `  }` `}`	`Zu a gehoert die Nummer 3`

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()`) benutzten Algorithmus (siehe Abschnitt 1.1.2). Sein Problem besteht darin, dass bei stark unterschiedlichen Zahlen *u* und *v* sehr viele Subtraktions-Operationen erforderlich werden. 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 ist gemeinsamer Teiler von *u* und *v*

$\Leftrightarrow$

x ist gemeinsamer Teiler von *u* 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: 52 Millisek.
```

```
ggT-Bestimmung mit Euklid (Modulo)

Erste Zahl:   999000999

Zweite Zahl:  36

ggT:          9

Benötigte Zeit: 0 Millisek.
```

7) Wer kann ein Programm erstellen, das zur Berechnung der Summe der natürlichen Zahlen von 1 bis $10^{12}+1$ (1 Billion + 1)

$$\sum_{i=1}^{1000000000001} i \quad = 1+2+3+...+1000000000001$$

weniger als 1 Millisekunde benötigt?

8) Bei einem **double**-Wert sind mindestens 15 signifikante Dezimalstellen garantiert (siehe Abschnitt 3.3.6). Folglich 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 Softwareentwicklung besteht 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. 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 vordefinierte (z.B. der Standardbibliothek entstammende) oder selbst erstellte Klassen zur Erledigung von Aufgaben verwendet. Für ein gerade agierendes (eine Methode ausführendes) Objekt bzw. eine agierende Klasse kommen als Ansprechpartner zur Erledigung eines Auftrags in Frage:

- eine Klasse mit passenden Handlungskompetenzen, z.B.:
  ```
  double res = Math.exp(arg);
  ```
- ein Objekt, das beim Laden einer Klasse automatisch entsteht und über eine statische (klassenbezogene) Referenzvariable ansprechbar ist, z.B.:
  ```
  System.out.println(arg);
  ```
- ein explizit im Programm erstelltes Objekt, z.B.:
  ```
  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.

Ein Programm besteht aus ...

- Klassen und Objekten,
- die jeweils einen Zustand haben
- und Botschaften empfangen sowie senden können.

In der Hoffnung, dass die bisher präsentierten Eindrücke von der objektorientierten Programmierung (OOP) neugierig gemacht und nicht abgeschreckt haben, kommen wir nun zur systematischen Behandlung dieser Softwaretechnologie. Für die in Kapitel 1 speziell für größere Projekte empfohlene objektorientierte *Analyse*, z.B. mit Hilfe der Unified Modeling Language (UML), ist dabei leider keine Zeit.

4.1 Überblick, historische Wurzeln, Beispiel

4.1.1 Einige Kernideen und Vorzüge der OOP

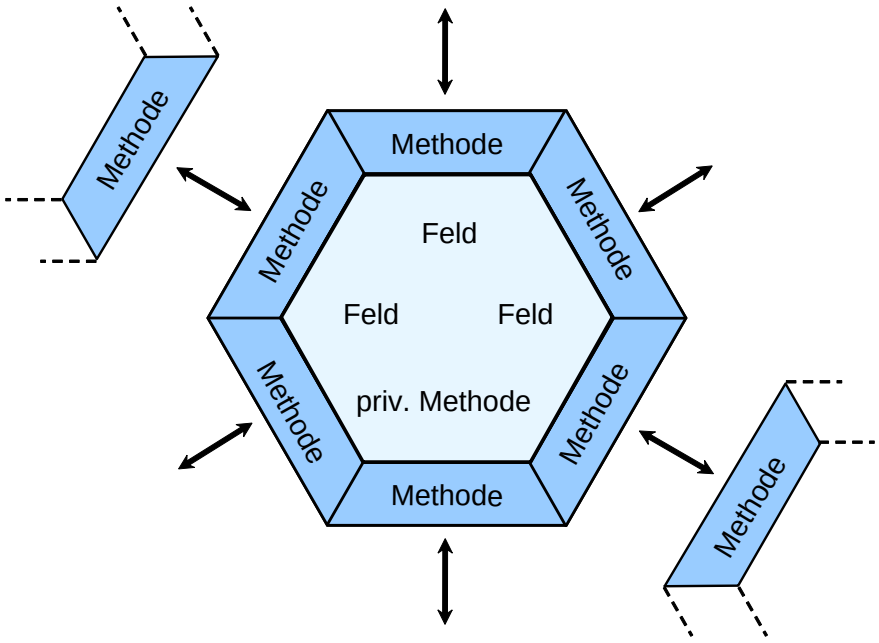
Lahres & Rayman (2009, Abschnitt 2) nennen in ihrem Buch *Praxisbuch Objektorientierung* unter Berufung auf **Alan Kay**, der den Begriff *Objektorientierte Programmierung* geprägt und die objektorientierte Programmiersprache *Smalltalk* entwickelt hat, als unverzichtbare OOP-Grundelemente:

- **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 Implementationsdetails ohne Nebenwirkungen auf andere Klassen ändern. Mit der Datenkapselung haben wir uns schon in 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 Modifikationen vorzunehmen, also z.B. Methoden situationsangepasst zu implementieren,
- **Polymorphie**
  Über Referenzvariablen vom Typ einer Basisklasse lassen sich auch Objekte von abgeleiteten Klassen verwalten, wobei selbstverständlich nur solche Methode aufgerufen werden dürfen, die schon in der Basisklasse definiert sind. Ist eine solche Methode in abgeleiteten Klassen unterschiedlich implementiert, führt jedes angesprochene Objekt sein angepasstes Verhalten aus. Derselbe Methodenaufruf hat also unterschiedliche (polymorphe) Verhaltensweisen zur Folge.

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. Hier besteht ein Programm 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**, **information hiding**) und so vor ungeschickten Zugriffen schützen. Die meisten Methoden einer Klasse sind hingegen von außen ansprechbar und bilden ihre **Schnittstelle**. Dies kommt in der folgenden Abbildung zum Ausdruck, die Sie schon aus Abschnitt 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 kennen gelernt.

Klassen mit Datenkapselung realisieren besser als frühere Software-Technologien (siehe Abschnitt 4.1.2) das Prinzip der **Modularisierung**. Die Modularisierung ist ein unverzichtbares Mittel der Software-Entwickler zur Bewältigung von umfangreichen Projekten.

Im Sinne einer gelungenen Modularisierung sind Klassen mit hoher Komplexität (also vielfältigen Aufgaben) und auch Methoden mit hoher Komplexität zu vermeiden. Als eine Leitlinie für den Entwurf von Klassen genießt das von **Robert C. Martin**[1] erstmals formulierte **Prinzip einer einzigen Verantwortung** (engl.: ***S**ingle **R**esponsibility **P**rinciple*, SRP) (Martin 2002) bei den Vordenkern der objektorientierten Programmierung hohes Ansehen (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 Datenbankverwaltung kümmert (Verbindung zum Datenbankserver herstellen, Fälle lesen und ablegen).

Aus der Datenkapselung und der Modularisierung ergeben sich gravierende Vorteile für die Softwareentwicklung:

[1] Der als *Uncle Bob* bekannte Software-Berater und Autor erläutert auf der folgenden Webseite seine Vorstellungen von objektorientierter Software: http://www.objectmentor.com/omSolutions/oops_what.html

- **Vermeidung von Fehlern**
  Direkte Schreibzugriffe auf die Felder (Variablen) einer Klasse bleiben den klasseneigenen Methoden vorbehalten, die vom Designer der Klasse sorgfältig entworfen wurden. Damit sollten Programmierfehler seltener werden. In unserem `Bruch`-Beispiel haben wir dafür gesorgt, dass unter keinen Umständen der Nenner eines Bruches auf 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 kann es wichtig sein, dass für eine Gruppe von Feldern bei jeder Änderung gewissen 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. Bei der Softwareentwicklung 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 bei gekapselten Details einer Klassenimplementation**
  Verborgene Details einer Klassenimplementation kann der Designer ändern, ohne die Kooperation mit anderen Klasen zu gefährden.
- **Produktivität durch wiederholt und bequem verwendbare Klassen**
  Selbständig agierende Klassen, die ein Problem ohne überflüssige Anhängigkeiten 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, so dass 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**:

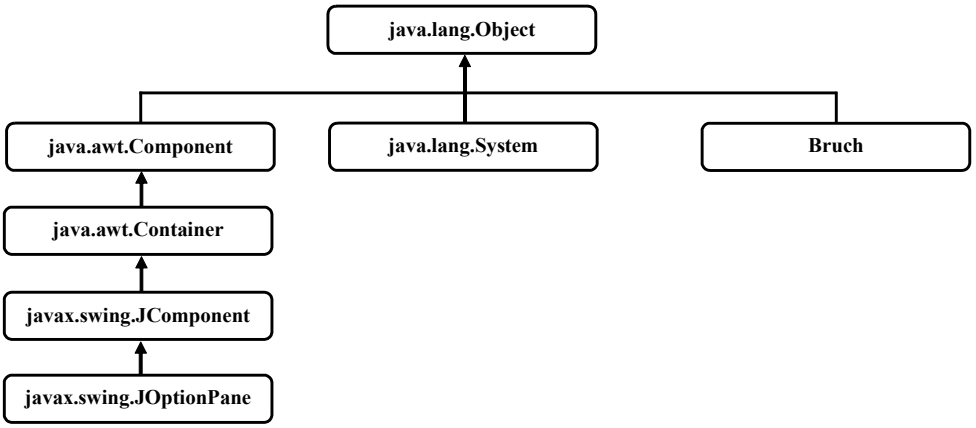
- Vermeidung bzw. schnelles Aufklären von Programmierfehlern
- gute Chancen für die Wiederverwendung von Software
- gute Voraussetzungen für die Kooperation in Teams

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 Softwaresystemen bei rationeller **Wiederverwendung** der bisherigen Code-Basis schafft: Bei der Definition einer neuen Klasse werden alle Eigenschaften (Felder) und Handlungskompetenzen (Methoden) einer *Basisklasse* übernommen. Es ist also leicht möglich, ein Softwaresystem 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 vor-

handener 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`):

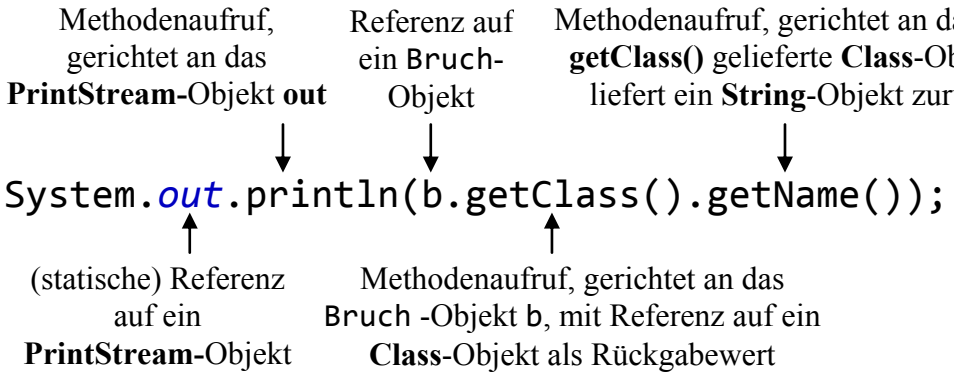


Zu jeder Klasse ist auch ihre Paketzugehörigkeit angegeben.

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**, welches 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 die Methode **getClass()**, obwohl in der `Bruch`-Klassendefinition nichts davon zu sehen ist.

4.1.1.3 Polymorphie

Obwohl in unseren bisherigen Beispielen von Polymorphie noch nichts zu sehen war, 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 auslassen.

Beim Klassendesign ist generell das **Open-Closed - Prinzip** zu beachten, das zwei auf den ersten Blick schwer vereinbare Forderungen enthält:[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.

Es ist klar, dass für neue Aufgaben zusätzlicher Quellcode erforderlich wird. Es darf aber nicht passieren, dass bei der Anpassung eines Programms an neue Anforderungen vorhandener Quellcode modifiziert werden muss. In ungünstigen Fällen zieht jede Änderung weitere nach sich, so dass eine Kaskade von Anpassungen unter Beteiligung von vielen Klassen resultiert. Bei einem solchen Programm verursacht eine Anpassung an neue Aufgaben hohe Kosten und oft ein instabiles Ergebnis, so dass man in der Regel auf eine Anpassung verzichten wird.

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.

---

[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://www.objectmentor.com/resources/articles/ocp.pdf

Dank Polymorphie kann eine Klasse nicht nur mit gleich alten oder älteren Klassen, die beim Design schon bekannt waren, zusammenarbeiten, sondern auch mit jüngeren Klassen. Um diese Offenheit für neue Aufgaben zu ermöglichen, verwendet man beim Klassendesign für Felder und Parameter, die auf Objekte zeigen, einen möglichst allgemeinen Datentyp, der die benötigten Verhaltenskompetenzen vorschreibt, aber keine darüber hinausgehende Einschränkung enthält.

In objektorientierten Programmiersprachen 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 Voraussetzungen und Bedürfnisse anzupassen. Erfolgen Methodenaufrufe 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 wurden, verhalten sie sich unterschiedlich. Genau in dieser Situation spricht man von *Polymorphie*.

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, 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, verzichtet man auf eine Implementation, wobei eine so genannte *abstrakte Methode* entsteht. Enthält eine Klasse mindestens eine abstrakte Methode, ist sie 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 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 erlauben auch eine gute Modellierung des Anwendungsbereichs. In der zentralen Projektphase der objektorientierten Analyse und Modellierung sprechen Software-Entwickler und Auftraggeber dieselbe Sprache, so dass 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 der virtuellen Welt des Computers repräsentieren (z.B. Bildschirmfenster, Mausereignisse).

---

[1] Neben den abstrakten Klassen, die mindestens *eine* abstrakte Methode (Definitionskopf ohne Implementation) enthalten, spielen bei der Polymorphie auch die so genannten *Schnittstellen* eine wichtige Rolle als veränderungsoffene Datentypen. Eine Schnittstelle kann näherungsweise als Klasse mit *ausschließlich* abstrakten Methoden charakterisiert werden. Abstrakte Klassen und Schnittstellen (Interfaces) werden später ausführlich behandelt.

4.1.2 Strukturierte Programmierung und OOP

In vielen klassischen Programmiersprachen (z.B. C oder 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 kompaktere und übersichtlichere Programme, die leichter erstellt, analysiert, korrigiert und erweitert werden können. Praktisch alle traditionellen 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.

- **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 Elementen 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 Fortschritte:

- 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.
- Rationelle (Weiter-)Entwicklung von Software nach dem Open-Closed - Prinzip durch Vererbung und Polymorphie
- Bessere Abbildung des Anwendungsbereichs
- Mehr Bequemlichkeit 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.

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 in 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 einer elementaren Ausbaustufe des Programms lediglich eine Klasse zur Repräsentation von Brüchen benötigt wird. Später sind weitere Klassen (z.B. Aufgabe, Übungsaufgabe, Testaufgabe, Schüler, Lernepisode, Testepisode, Fehler) zu ergänzen.

Wir nehmen nun bei der `Bruch`-Klassendefinition im Vergleich zur Variante in Abschnitt 1.1 einige Verbesserungen vor:

- Als 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()` neben anderen Eigenschaften auf dem Bildschirm erscheint. Objekte der erweiterten `Bruch`-Klasse besitzen also auch eine Instanzvariable mit *Referenztyp* (neben den Feldern `zaehler` und `nenner` vom primitiven Typ **int**).
- Weil die `Bruch`-Klasse ihre Eigenschaften systematisch kapselt, also fremden Klassen keine direkten Zugriffe erlaubt, muss sie auch für das `etikett` zum Lesen bzw. Setzen 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 178).

Im folgenden Quellcode der erweiterten `Bruch`-Klasse 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() {
      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");
  }
}
```

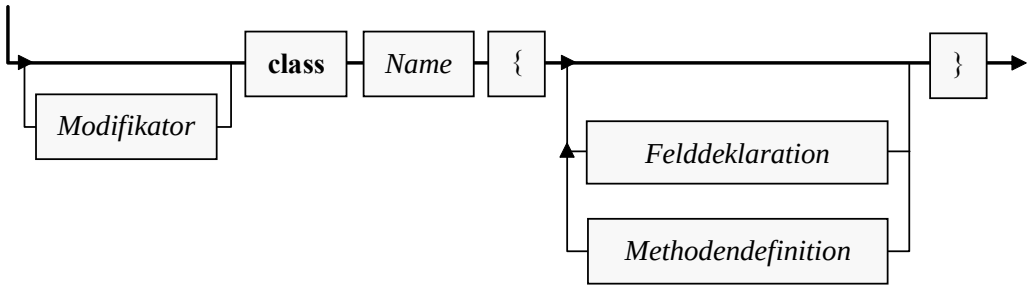
Für die bei diversen Demonstrationen in den folgenden Abschnitten verwendeten Startklassen (mit jeweils spezieller Implementierung) werden wir den Namen `Bruchrechnung` verwenden, z.B.:

Quellcode	Ausgabe
`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();` `  }` `}`	`  1` `Der gekürzte Bruch: -----` `  4`

Im Unterschied zur Präsentation in Abschnitt 1.1 wird die `Bruch`-Klassendefinition anschließend gründlich erläutert. Dabei machen die in 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, so dass wir uns in 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 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 JRE *nicht* erforderlichen) Zugriffsmodifikator **public** verzichtet. Im Zusammenhang mit den 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]

Hinsichtlich der **Dateiverwaltung** ist zu beachten:

---

[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.

[2] Bei einer startfähigen Klasse ist ein komplizierter Name zu vermeiden, wenn dieser vom Benutzer beim Programmstart eingetippt werden muss (mit korrekt eingehaltener Groß-/Kleinschreibung!).

- 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.
- Dateien mit Java-Quellcode benötigen auf jeden Fall die Namenserweiterung **.java**.

4.2 Instanzvariablen

Die Instanzvariablen (Felder) einer Klasse besitzen viele Gemeinsamkeiten mit den lokalen Variablen, die wir im Kapitel 2 über elementare Sprachelemente ausführlich behandelt haben, doch gibt es auch wichtige Unterschiede, die im Mittelpunkt des aktuellen Abschnitts stehen. Unsere Klasse `Bruch` besitzt nach der Erweiterung um ein beschreibendes Etikett folgende 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 Gültigkeitsbereich, Existenz und Ablage im Hauptspeicher

Von den lokalen Variablen 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 Gültigkeitsbereich (synonym: Sichtbarkeitsbereich), die Lebensdauer und die Ablage im Hauptspeicher:

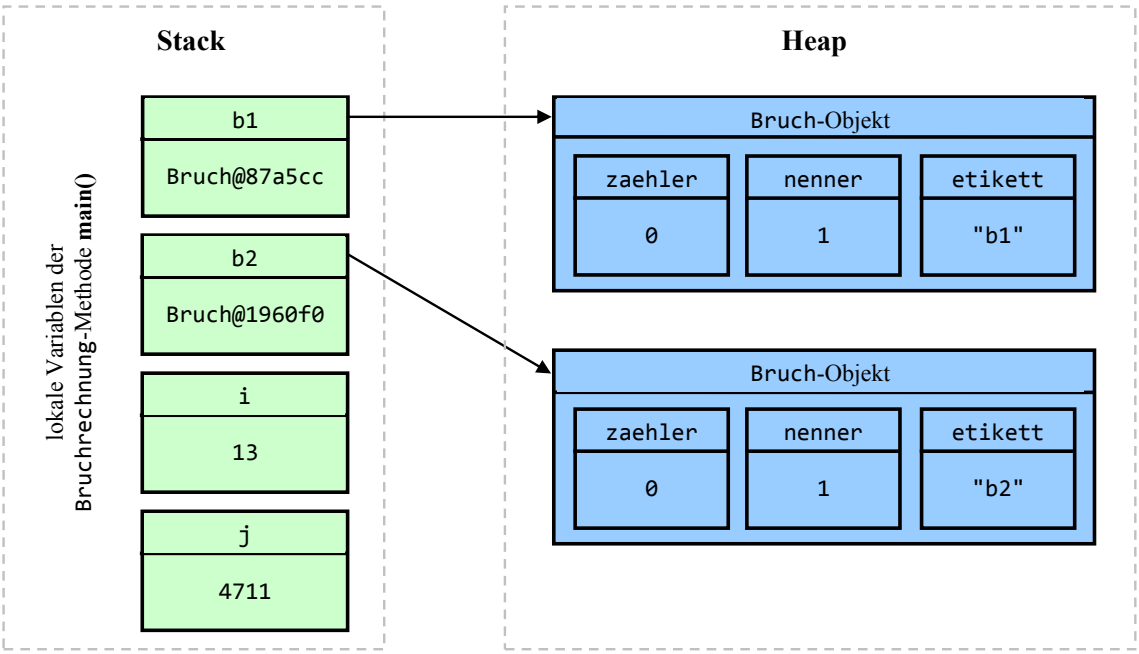
	lokale Variable	**Instanzvariable**
Gültigkeit, **Sichtbarkeit**	Eine lokale Variable ist nur in ihrer eigenen Methode gültig (sichtbar). Nach der Deklarationsanweisung kann sie in den restlichen Anweisungen des lokalsten Blocks angesprochen werden. Ein eingeschachtelter Block gehört zum Gültigkeitsbereich des umgebenden Blocks.	Die Instanzvariablen eines existenten Objekts sind in einer Methode ansprechbar, wenn ... • der Zugriff erlaubt • 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).

	lokale Variable	Instanzvariable
Lebensdauer	Sie existiert nur bei 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 vorhanden ist.
Ablage im Speicher	Sie wird auf dem so genannten **Stack** (deutsch: *Stapel*) abgelegt. Innerhalb des programmeigenen Speichers dient dieses Segment zur Verwaltung 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]



[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.

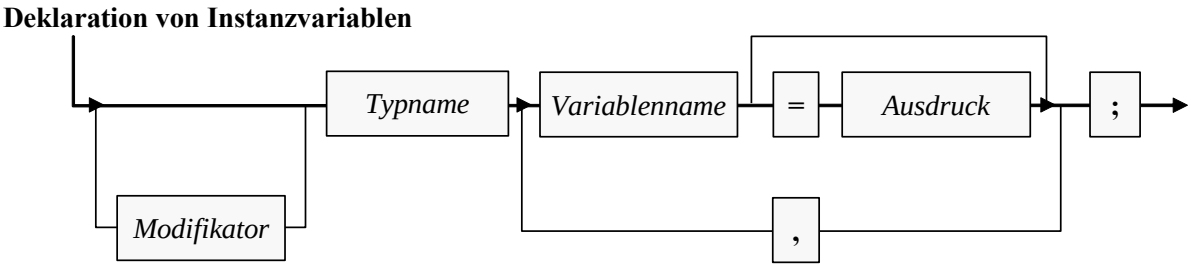
Hier wird aus didaktischen Gründen ein wenig gemogelt: Die beiden Etiketten sind selbst Objekte und liegen „neben“ den `Bruch`-Objekten auf dem Heap. In jedem `Bruch`-Objekt befindet sich eine Referenz-Instanzvariable namens `etikett`, die auf das zugehörige **String**-Objekt zeigt.

Dass in der `Bruchrechnung`-Methode **main()** keine Zugriffe auf Instanzvariablen der Klasse `Bruch` zu sehen sind, liegt vor allem am Prinzip der Datenkapselung. In Abschnitt 4.2.4 wird der Zugriff auf Instanzvariablen in klasseneigenen und fremden Methoden behandelt.

4.2.2 Deklaration mit Wahl der Schutzstufe

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.

Für die Deklaration einer lokalen Variablen haben wir **final** als einzigen Modifikator kennen gelernt und in einem Syntaxdiagramm explizit berücksichtigt (vgl. Abschnitt 3.3.10). Dieser Modifikator kommt auch bei Instanzvariablen in Frage, wird aber hier in seiner Bedeutung weit übertroffen durch die Zugriffsmodifikatoren, wobei zur Realisation der Datenkapselung meist die Schutzstufe **private** vergeben wird. Insgesamt ist es sinnvoll, in das Syntaxdiagramm zur Deklaration von Instanzvariablen den allgemeinen Begriff des Modifikators aufzunehmen:[1]



Im `Bruch`-Beispiel wird im Sinne einer perfekten Datenkapselung für alle Instanzvariablen mit dem Modifikator **private** angeordnet, dass nur klasseneigenen Methoden ein direkter 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 etliche Zugriffsmethoden (z.B. `setzeNenner()`, `gibNenner()`).

Gibt man bei der Deklaration einer Instanzvariablen *keine* Schutzstufe an, haben alle anderen Klassen *im selben Paket* (siehe unten) 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:

---

[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 verbieten.

- Zugunsten einer übersichtlichen Bildschirmausgabe soll das Etikett auf 40 Zeichen beschränkt bleiben. Mit Hilfe der Zugriffsmethode `setzteEtikett()` kann dies gewährleistet werden.
- Abgeleitete (erbende) Klassen (siehe unten) 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 der überzeugenden Vorteile soll die Datenkapselung nicht zum Dogma erhoben werden. Sie ist überflüssig, wenn bei einem Feld Lese- und Schreibzugriffe uneingeschränkt erlaubt sein sollen und auch die eben im Beispiel angedachte Option, bestimmte Wertzuweisungen zu einem Ereignis zu machen, nicht von Interesse ist. Um allen Klassen den Direktzugriff auf eine Instanzvariable zu erlauben, wird in der Deklaration der Modifikator **public** angegeben, z.B.:

```
public int zaehler;
```

Im Zusammenhang mit den Paketen (siehe Kapitel 6) werden wir uns noch ausführlich mit dem Thema *Zugriffsschutz* beschäftigen. Die wichtigsten Regeln sind Ihnen aber vermutlich mittlerweile schon ziemlich vertraut:

- 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 (auch die im selben Paket).
  - **public**
    Alle Klassen dürfen zugreifen.

In Bezug auf die Namenskonventionen gibt es keine Unterschiede zwischen den Instanzvariablen und den lokalen Variablen (vgl. Abschnitt 3.3). Insbesondere sollten folgende Regeln eingehalten werden:

- 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*)

4.2.3 Initialisierung

Während bei lokalen Variablen der Programmierer für die Initialisierung verantwortlich ist, erhalten die Instanzvariablen eines neuen Objekts automatisch folgende Startwerte, falls der Programmierer nicht eingreift:

Datentyp	Initialisierung
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 (z.B. der Klarheit halber) sinnvoll sein, den Instanzvariablennamen über das Schlüsselwort **this** (vgl. Abschnitt 4.4.6.2) eine Referenz auf das aktuell handelnde Objekt voranzustellen, z.B.:

```
System.out.println(" " + luecke  + "   " + this.zaehler + "\n" +
                   " " + this.etikett + " -----\n" +
                   " " + luecke  + "   " + this.nenner + "\n");
```

Beim Zugriff auf eine Instanzvariablen 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 Anwei-

sung 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;
```

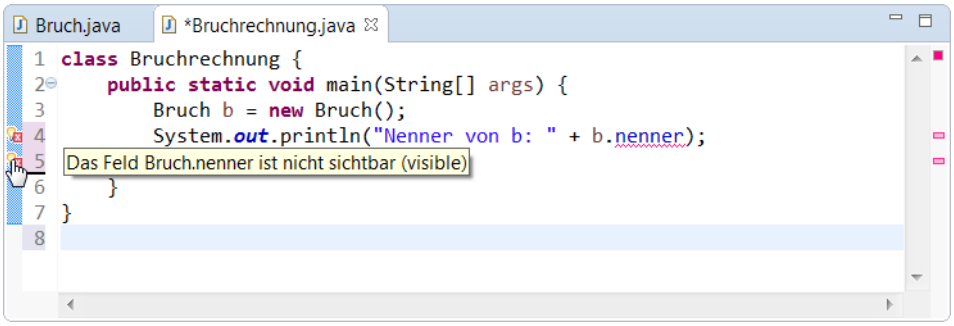
Direkte Zugriffe auf die Instanzvariablen eines Objekts in Methoden *fremder* Klassen sind zwar nicht grundsätzlich verboten, verstoßen aber gegen das Prinzip der Datenkapselung, das in der OOP von zentraler Bedeutung ist. Würden die `Bruch`-Instanzvariablen ohne den Modifikator **private** deklariert, dann könnte z.B. der Nenner eines Bruches in der **main**() - Methode der (fremden!) Klasse `Bruchrechnung`, die sich im selben Paket (siehe unten) befindet, direkt angesprochen werden, z.B.:

```
class Bruchrechnung {
 public static void main(String[] args) {
  Bruch b = new Bruch();
  System.out.println("Nenner von b: " + b.nenner);
  b.nenner = 0;
 }
}
```

In der von uns tatsächlich realisierten `Bruch`-Definition werden solche Zugriffe jedoch verhindert. Der JDK-Compiler meldet:

```
Bruchrechnung.java:4: error: nenner has private access in Bruch
                System.out.println("Nenner von b: " + b.nenner);
                                                         ^
Bruchrechnung.java:5: error: nenner has private access in Bruch
                b.nenner = 0;
                 ^
2 errors
```

Unsere Entwicklungsumgebung Eclipse signalisiert die Problemstellen sehr deutlich im Quellcode-Editor:



Besteht man trotzdem auf einem Übersetzungsversuch, resultiert die Fehlermeldung:

```
Exception in thread "main" java.lang.Error: Unaufgelöste Kompilierungsprobleme:
        Das Feld Bruch.nenner ist nicht sichtbar (visible)
        Das Feld Bruch.nenner ist nicht sichtbar (visible)

        at Bruchrechnung.main(Bruchrechnung.java:4)
```

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, so dass 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 Fehler verhindern, die ansonsten als gravierende Logikfehler großen Ärger bei den Kunden und großen Aufwand beim Software-Hersteller verursachen können (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 oder in einem Konstruktor (siehe Abschnitt 4.4.3) erfolgen, wobei die erste Technik zu identischen Werten für alle Objekte führen würde und daher kaum in Frage kommt.

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, so dass oft eine **public**-Deklaration wie im folgenden Beispiel sinnvoll ist:

```
public final int nummer;
```

Für die obligatorische initiale Wertzuweisung sorgt man bei einer finalisierten Instanzvariablen am besten in den Konstruktoren der Klasse, weil bei *jeder* Objektkreation ein Konstruktor abläuft und hier eine *individuelle* Wertvergabe möglich ist, z.B.

```
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 in 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.

4.3 Instanzmethoden

In einer Bauplan-Klassendefinition werden Objekte entworfen, die eine Anzahl von Handlungskompetenzen (Methoden) besitzen, die von anderen Klassen oder Objekten 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 Objekte (bzw. Methoden) fremder Klassen unsichtbar (*information hiding*). Um anderen Klassen trotzdem (kontrollierte) Zugriffe auf eine Instanzvariable 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 so genannte **Parameter** die gewünschte Verhaltensweise festzulegen, und bei 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 sollen. Wenn ein anderer Programmierer z.B. ein Objekt aus unserer Klasse `Bruch` verwendet, kann er es mit einen Aufruf der Methode `addiere()` veranlassen, einen per Parameter benannten zweiten Bruch mit dem Hauptnennerverfahren 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;
 this.kuerze();
}
```

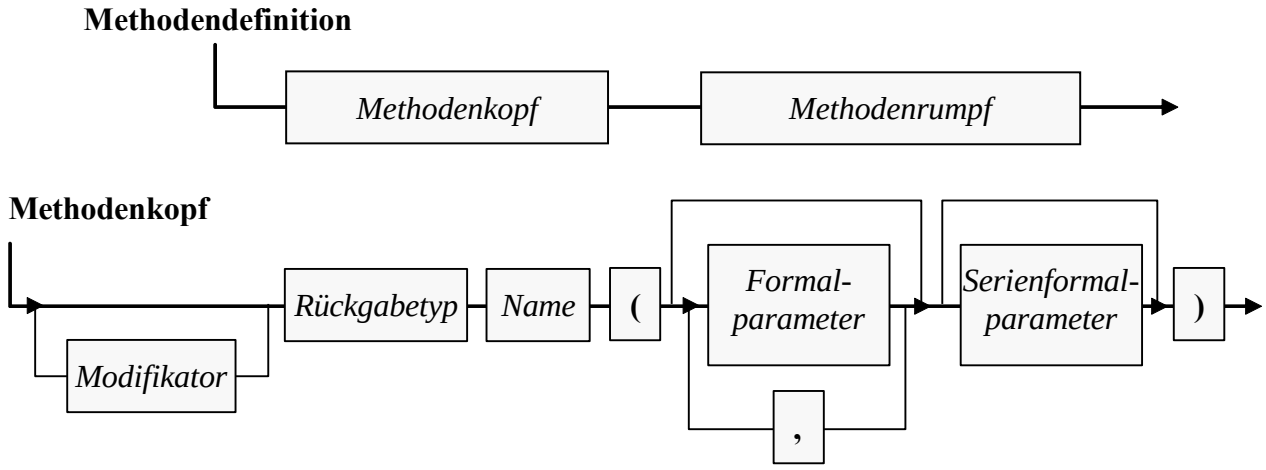
Weil diese Methode auch für fremde Klassen verfügbar sein soll, wird per Modifikator die Schutzstufe **public** gewählt.

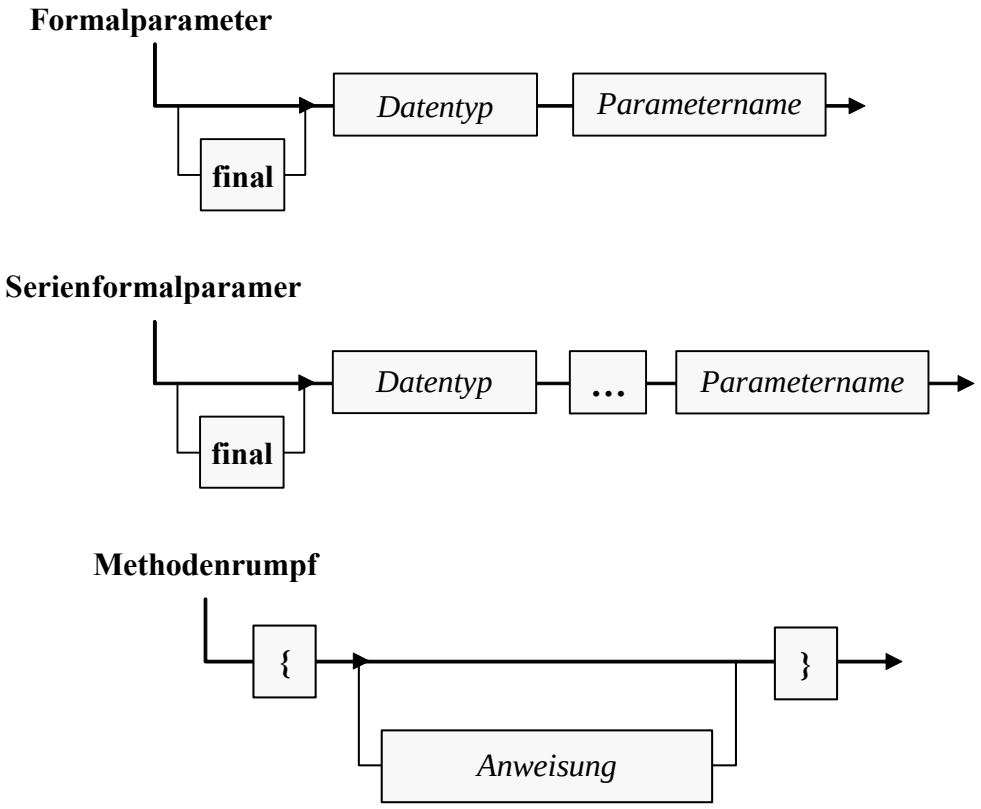
Da es vom Verlauf der Auftragserledigung nichts zu berichten gibt, liefert `addiere()` keinen Rückgabewert. Folglich ist im Kopf der Methodendefinition der Rückgabetyp **void** angegeben.

Während sich jedes Objekt mit seinem eigenen vollständigen Satz von Instanzvariablen auf dem Heap befindet, ist der Bytecode der Instanzmethoden jeweils nur *einmal* im Speicher vorhanden und wird von allen Objekten 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 in Abschnitt 3.1.3.2 präsentierten Variante durch eine genauere Erklärung der (in Abschnitt 4.3.1.3 zu behandelnden) Formalparameter:





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 beim Aufruf zu erkennen.

Für die Namen von Methoden sind in Java dieselben Konventionen üblich wie bei den Namen von lokalen Variablen und Feldern:

- Sie beginnen mit einem Kleinbuchstaben.
- Besteht ein Name aus mehreren Wörtern (z.B. `setzeNenner()`), schreibt man ab dem zweiten Wort die Anfangsbuchstaben groß (*Camel Casing*).

4.3.1.1 Modifikatoren

Bei einer Methodendefinition kann per Modifikator u.a. der voreingestellte Zugriffsschutz verändert werden. Wie für Instanzvariablen gelten auch für Instanzmethoden beim Zugriffsschutz folgende Regeln:

- Per Voreinstellung ist der Zugriff allen Klassen im selben Paket (siehe unten) erlaubt.
- Mit einem Modifikator lassen sich alternative Schutzstufen wählen, z.B.:
  - **private**
    Alle fremden Klassen werden ausgeschlossen.
  - **public**
    Alle Klassen dürfen zugreifen.

In unserer Beispielklasse `Bruch` haben alle Methoden den Zugriffsmodifikator **public** erhalten. Damit die 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 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, so dass auch ein komplexes Informationsobjekt geliefert werden kann.

Wir haben schon in 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.

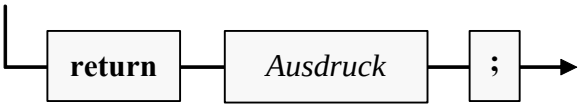
Bei der Definition einer Methode muss festgelegt werden, von welchem Datentyp ihr Rückgabewert ist. Erfolgt *keine* Rückgabe, ist der Ersatztyp **void** anzugeben.

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;
}
```

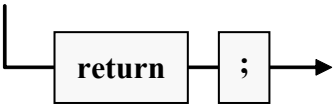
Ist der Rückgabetyp einer Methode von **void** verschieden, dann muss im Rumpf der Methode dafür gesorgt werden, dass jeder mögliche Ausführungspfad mit einer **return**-Anweisung endet, die einen Rückgabewert von passendem Typ liefert:

return-Anweisung für Methoden *mit* Rückgabewert



Bei Methoden *ohne* Rückgabewert 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



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 ggt = ggTr(zaehler, nenner);
  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 ggt = ggTr(zaehler, nenner);
 zaehler /= ggt;
 nenner  /= ggt;
}
```

4.3.1.3 Formalparameter

Methodenparameter wurden Ihnen bisher vereinfachend als Informationen über die gewünschte Arbeitsweise einer Methode vorgestellt. Tatsächlich ermöglichen Parameter aber den Informationsaustausch zwischen einem Aufrufer und einer angeforderten Methode in *beide* Richtungen.

Im Kopf der Methodendefinition werden über so genannte **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 4.3.1.3.1). Werden einer Methode Referenzen übergeben, kann sie jedoch im Rahmen ihrer Zugriffsrechte auf die zugehörigen Objekte einwirken (siehe Abschnitt 4.3.1.3.2) 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. Um Namenskonflikte zu vermeiden, hängen manche Programmierer an Parameternamen ein Suffix an, z.B. *par* oder einen Unterstrich. Weil Formalparameter im Methodenrumpf wie lokale Variablen zu behandeln sind, …

  - können Namenskonflikte mit anderen lokalen Variablen derselben Methode auftreten,
  - werden namensgleiche Instanz- bzw. Klassenvariablen überdeckt.
    Diese bleiben jedoch über ein geeignetes Präfix weiter ansprechbar:
    - **this** bei Instanzvariablen
    - Klassenname bei statischen Variablen

  Solche Namenskonflikte sollte man vermeiden.

- **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.

Ein Formalparameter kann wie jede andere lokale Variable mit dem Modifikator **final** auf den Initialisierungswert fixiert werden. Auf diese Weise lässt sich die (ohnehin 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 in Abschnitt 3.3.10 erläutert.

4.3.1.3.1 Parameter mit primitivem Datentyp

Über einen Parameter mit primitivem Datentyp werden Informationen in eine Methode kopiert, um diese mit Daten zu versorgen 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:

Quellcode	Ausgabe
`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);` `  }` `}`	`4712` `4711`

Die Klasse `Prog` ist startfähig, besitzt also eine statische Methode namens **main()**. Dort wird ein Objekt der Klasse `Prog` erzeugt und beauftragt, die Instanzmethode `primParDemo()` auszuführen. Mit dieser (auch in den folgenden Abschnitten anzutreffenden) 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).

4.3.1.3.2 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 eventuell 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, so dass die aufgerufene Methode über ihre lokale Referenzvariable auf das Originalobjekt zugreifen und dort Veränderungen vornehmen kann, sofern sie dazu berechtigt ist.

Die Originalversion 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;
 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 `Bruch`-Methode 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 miteinander kommunizieren sollen.

Beim Aufruf der Methode `addiere()` bleibt das per Referenzparameter ansprechbare Objekt unverändert. Sofern entsprechende Zugriffsrechte vorliegen, was bei Referenzparametern vom Typ der eigenen Klasse stets der Fall ist, kann eine Methode das Referenzparameterobjekt aber durchaus 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:

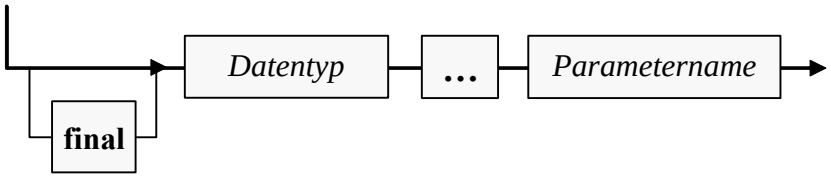
Quellcode	Ausgabe
`class Bruchrechnung {` `  public static void main(String[] args) {` `  Bruch b1 = new Bruch(), b2 = new Bruch();` `  b1.setzeZaehler(1); b1.setzeNenner(2);` `  b1.setzeEtikett("b1 = ");` `  b2.setzeZaehler(5); b2.setzeNenner(6);` `  b2.setzeEtikett("b2 = ");` `  b1.zeige();` `  b2.zeige();` `  b1.dupliziere(b2);` `  System.out.println("b2 nach dupliziere():\n");` `  b2.zeige();` `  }` `}`	`  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.

4.3.1.3.3 Serienparameter

Seit der Version 5.0 (bzw. 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



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 ein trennendes Leerzeichen setzt oder wie im Beispiel der Konvention folgend 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 benutzt, um jedes Element im Array `bar` mit `Bruch`-Objekten durch Aufruf der originalen `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, für einen Serienformalparameter beim Aufruf überhaupt keinen Aktualparameter anzugeben, z.B.:

```
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 5.1.6 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:[1]

public PrintStream printf(String *format***, Object...** *args***)**

[1] Alternativ kann auch die funktionsgleiche Methode **format()** benutzt werden.

Dass die Methode **printf()** eine Referenz auf das handelnde **PrintStream**-Objekt als (meist ignorierten) Rückgabewert liefert, kann uns momentan gleichgültig sein.

4.3.1.4 Methodenrumpf

Über die Verbundanweisung, 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, so dass dieser den Ablauf der Methode beeinflussen kann.
- Die **return**-Anweisung dient zur Rückgabe eines Wertes an den Aufrufer und/oder zum Beenden der Methodenausführung.

Ansonsten können beliebige Anweisungen unter Verwendung von elementaren und objektorientierten Sprachelementen eingesetzt werden, um den Zweck einer Methode zu implementieren.

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).

Verschachtelte Methoden*definitionen* sind verboten, z.B.:

*Prog.java

```
1  class Prog {
2      public void mitLokalerMethode() {
3          void lokaleMethode() {
4              System.out.println("Hallo");
5          }
6          lokaleMethode();
7      }
8
9      public static void main(String[] args) {
10         Prog p = new Prog();
11         p.mitLokalerMethode();
12     }
13 }
```

4.3.2 Methodenaufruf und Aktualparameter

Beim Aufruf einer Instanzmethode, z.B.:

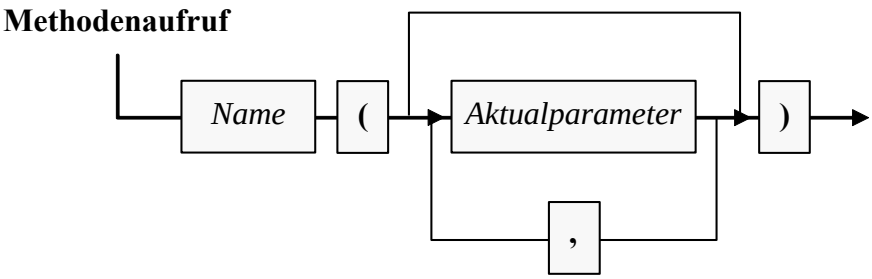
```
b1.zeige();
```

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 kennen gelernt.

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 Serie 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 in Abschnitt 4.3.1.3.1 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 erweiternd in diesen Typ gewandelt werden kann.

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 Werten versorgt werden. Man darf also keinen Parameter in der Hoffnung auf geeignete Voreinstellungswerte weglassen. 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).

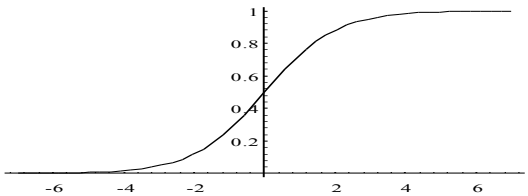
Liefert eine Methode einen Wert zurück, stellt ihr Aufruf einen **Ausdruck** dar und kann als Argument in komplexeren Ausdrücken auftreten, z.B.:

Quellcodesegment	Ausgabe
`double arg = 0.0, logist;` `logist = Math.exp(arg)/(1+Math.exp(arg));` `System.out.println(logist);`	`0.5`

Hier 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 4.3.1.3.1 paramtergesteuert) 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();
}
```

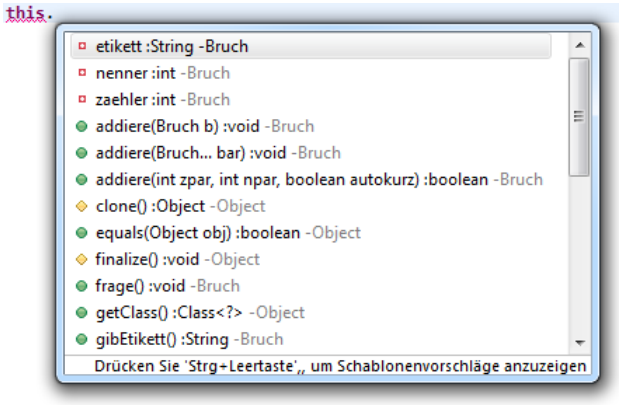
Wer auch solche Methodenaufrufe nach dem Schema

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 Eclipse 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.

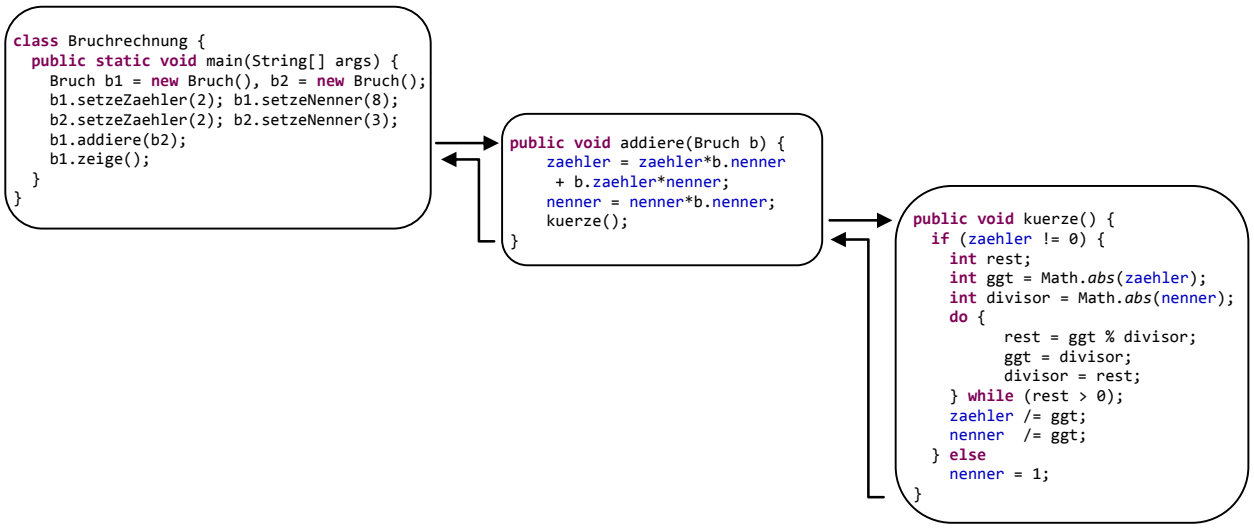
4.3.3 Debug-Einsichten zu (verschachtelten) Methodenaufrufen

Verschachtelte Methodenaufrufe stellen keine Besonderheit, sondern den selbstverständlichen Normalfall dar. Gerade deswegen ist es angemessen, das Geschehen etwas genauer zu betrachten. Anhand der folgenden Bruchrechnungsstartklasse

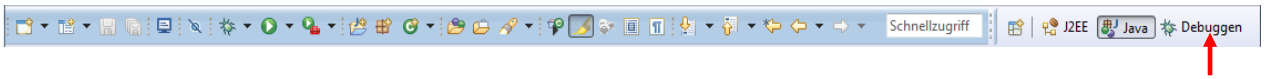
```
class Bruchrechnung {
 public static void main(String[] args) {
  Bruch b1 = new Bruch(), b2 = new Bruch();
  b1.setzeZaehler(2); b1.setzeNenner(8);
  b2.setzeZaehler(2); b2.setzeNenner(3);
  b1.addiere(b2);
  b1.zeige();
 }
}
```

soll mit Hilfe unserer Entwicklungsumgebung Eclipse untersucht werden, was bei folgender Aufrufverschachtelung geschieht:

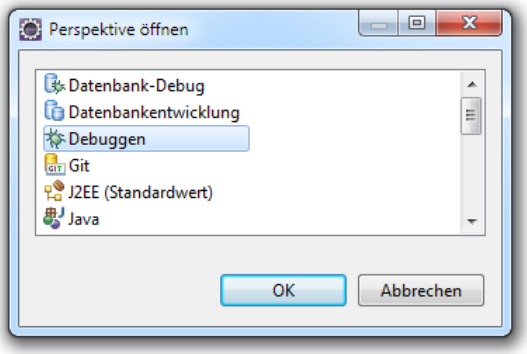
- Die statische Methode **main()** der Klasse `Bruchrechnung` ruft die `Bruch`-Instanzmethode `addiere()`.
- Die `Bruch`-Instanzmethode `addiere()` ruft die `Bruch`-Instanzmethode `kuerze()`.



Wir verwenden die zur Fehlersuche konzipierte **Debug**-Technik von Eclipse und wechseln daher von der Eclipse-Perspektive **Java** zur Perspektive **Debuggen**. Damit erhalten wir eine zur Fehlersuche optimierte Zusammenstellung von Eclipse-Werkzeugen (Sichten und Editoren). War die Perspektive **Debuggen** bereits einmal im Einsatz, ist sie über eine Schaltfläche in der Symbolleistenzone wählbar:



Anderenfalls ist sie über den Schalter zum Öffnen einer Perspektive erreichbar:

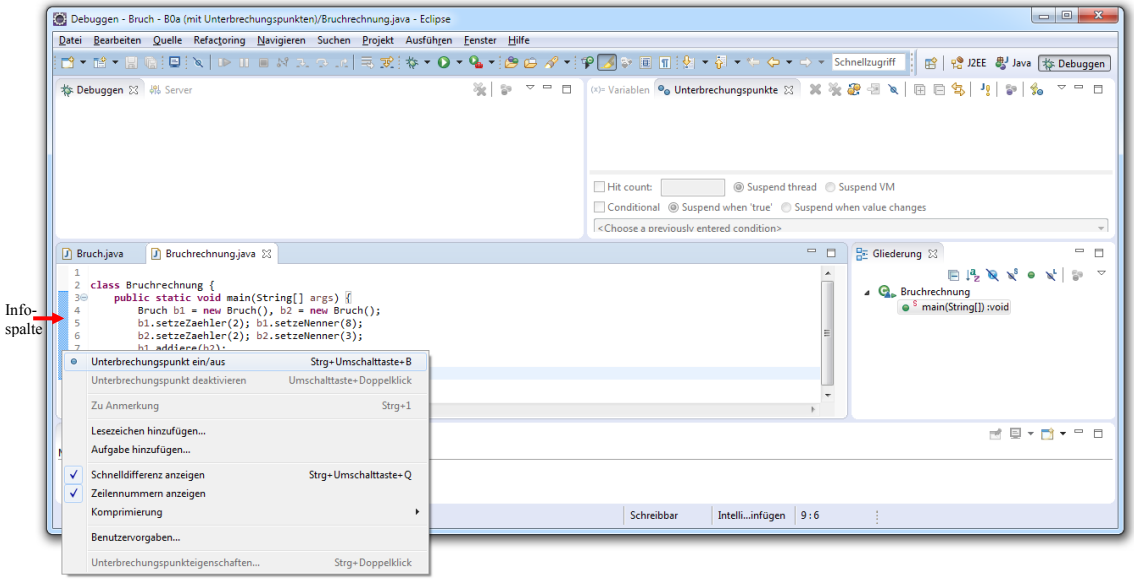


Das Programm soll an mehreren Stellen durch einen so genannten **Unterbrechungspunkt** (engl. *breakpoint*) angehalten werden, so dass wir jeweils die Lage im Hauptspeicher inspizieren können. Um einen Unterbrechungspunkt festzulegen, …

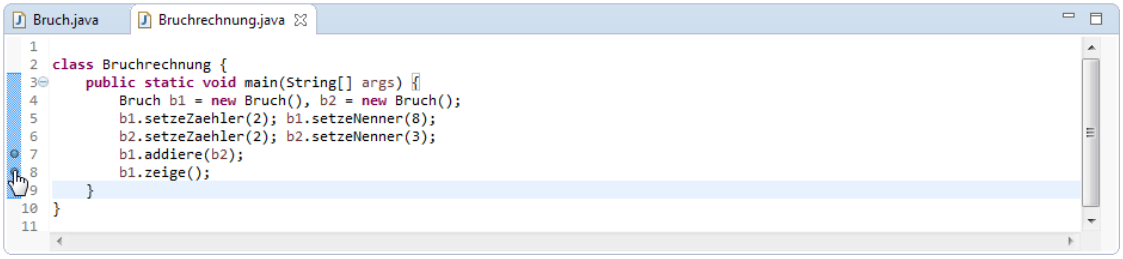
- setzt man in der Infospalte des Editors einen Rechtsklick in Höhe der betroffenen Zeile
- und wählt im Kontextmenü das Item **Unterbrechungspunkt ein/aus**

Zum Entfernen eines Unterbrechungspunkts wählt man das Kontextmenü-Item erneut.

Hier wird in der **main()** - Methode ein Unterbrechungspunkt eingefügt:



Noch bequemer klappt das Setzen bzw. Entfernen eines Unterbrechungspunkts per Mausdoppelklick in die Infospalte neben der betroffenen Anweisung, z.B.:



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;
  ```

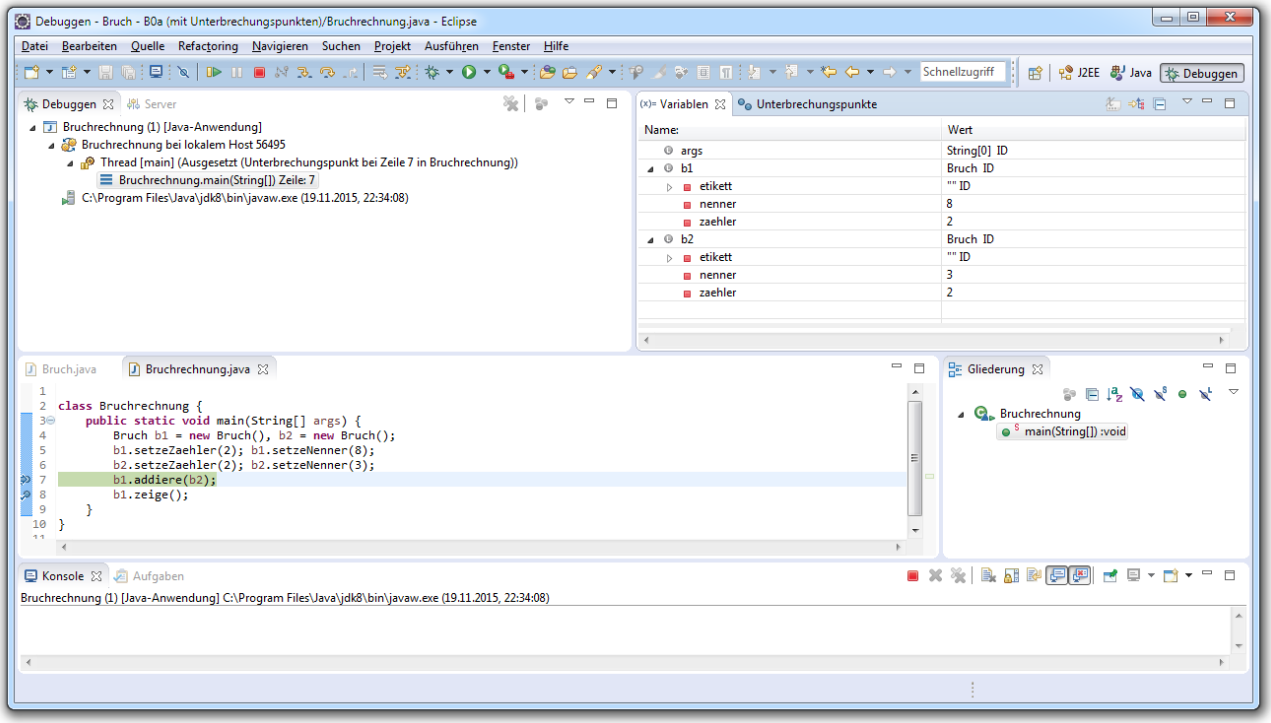
  im Block der **do-while** - Schleife.

Starten Sie die Klasse `Bruchrechnung` im Debug-Modus mit der Funktionstaste **F11**, über den Menübefehl

Ausführen > Debug

oder über das Steuerelement  , das analog zum bekannten Startknopf  funktioniert.

Das **Debuggen**-Fenster zeigt im Zweig **Thread [main]**, welche **Stack Frames** mit den Daten eines Methodenaufrufs sich derzeit auf dem Stack zum Haupt-Thread befinden.[1] Bei Erreichen des ersten Unterbrechungspunkts (Anweisung `b1.addiere(b2);` in **main()**) ist nur der Stack Frame der Methode **main()** vorhanden:

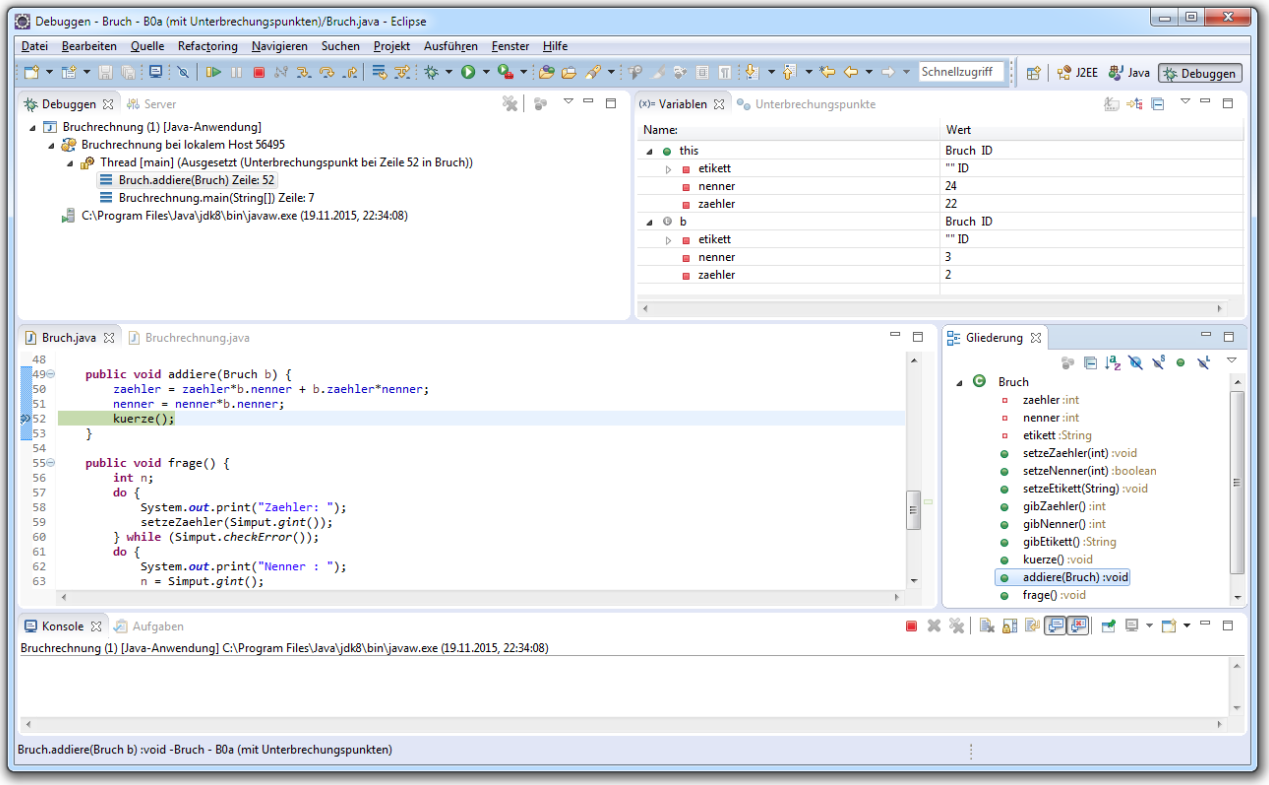


Im **Variablen**-Fenster der **Debuggen**-Perspektive sind die lokalen Variablen der Methode **main()** zu sehen:

- Parameter `args`
- die lokalen Referenzvariablen `b1` und `b2`
  Es werden auch die Instanzvariablen der referenzierten `Bruch`-Objekte angezeigt.

Lassen Sie das Programm mit dem Schalter oder der Taste **F8** fortsetzen. Beim Erreichen des zweiten Unterbrechungspunkts (Anweisung `kuerze();` in der Methode `addiere()`) liegen die Stack Frames der Methoden `addiere()` und **main()** übereinander:

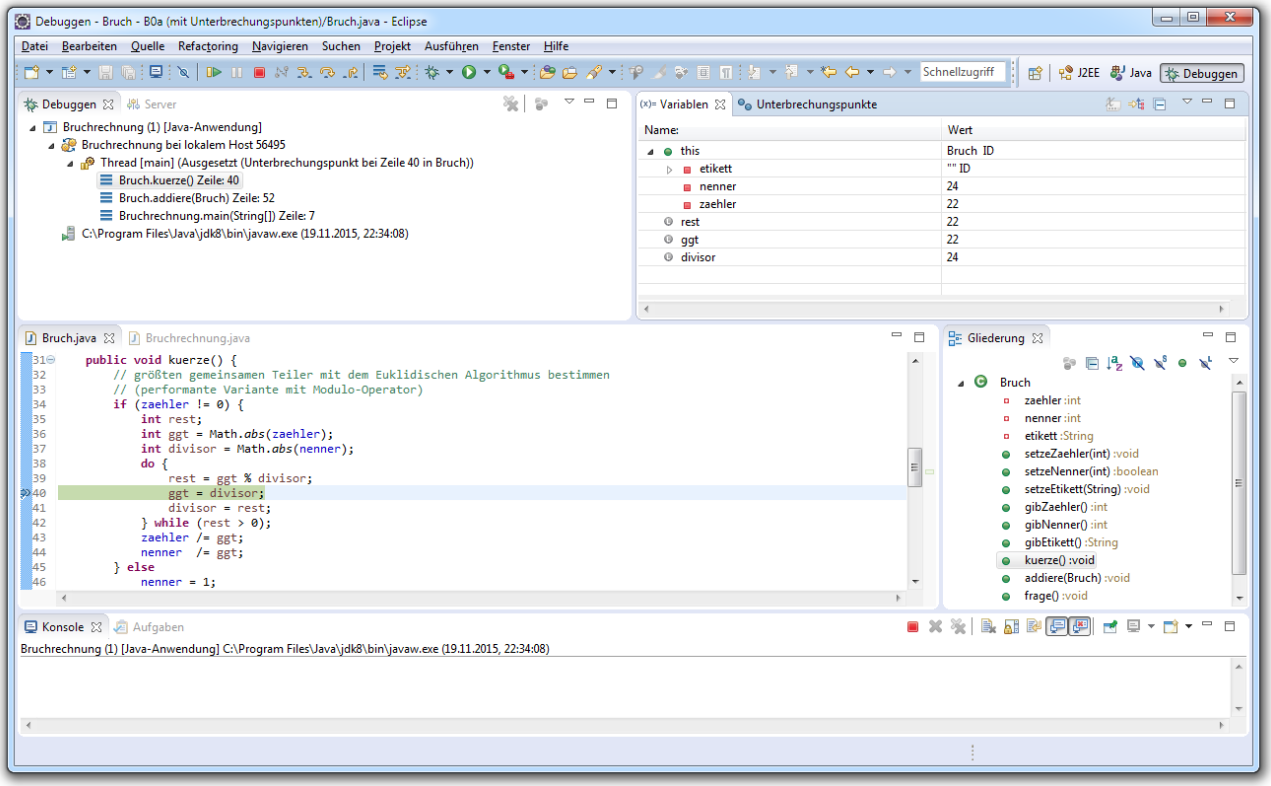
[1] Dass in einem Programm mehrere Threads (Ausführungsfäden) existieren können, wird uns später noch ausführlich beschäftigen.



Das **Variablen**-Fenster zeigt als lokale Variablen der Methode `addiere()`:

- **this** (Referenz auf das handelnde `Bruch`-Objekt)
- Parameter `b`

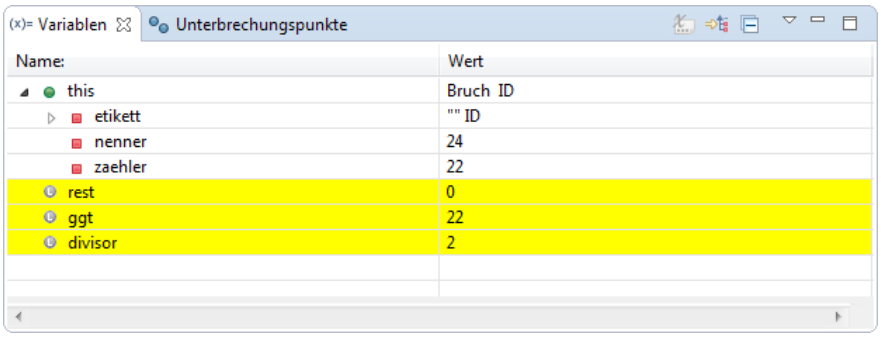
Beim Erreichen des dritten Unterbrechungspunkts (Anweisung `ggt = divisor;` in der Methode `kuerze()`) liegen die Stack Frames der Methoden `kuerze()`, `addiere()` und **main()** übereinander:



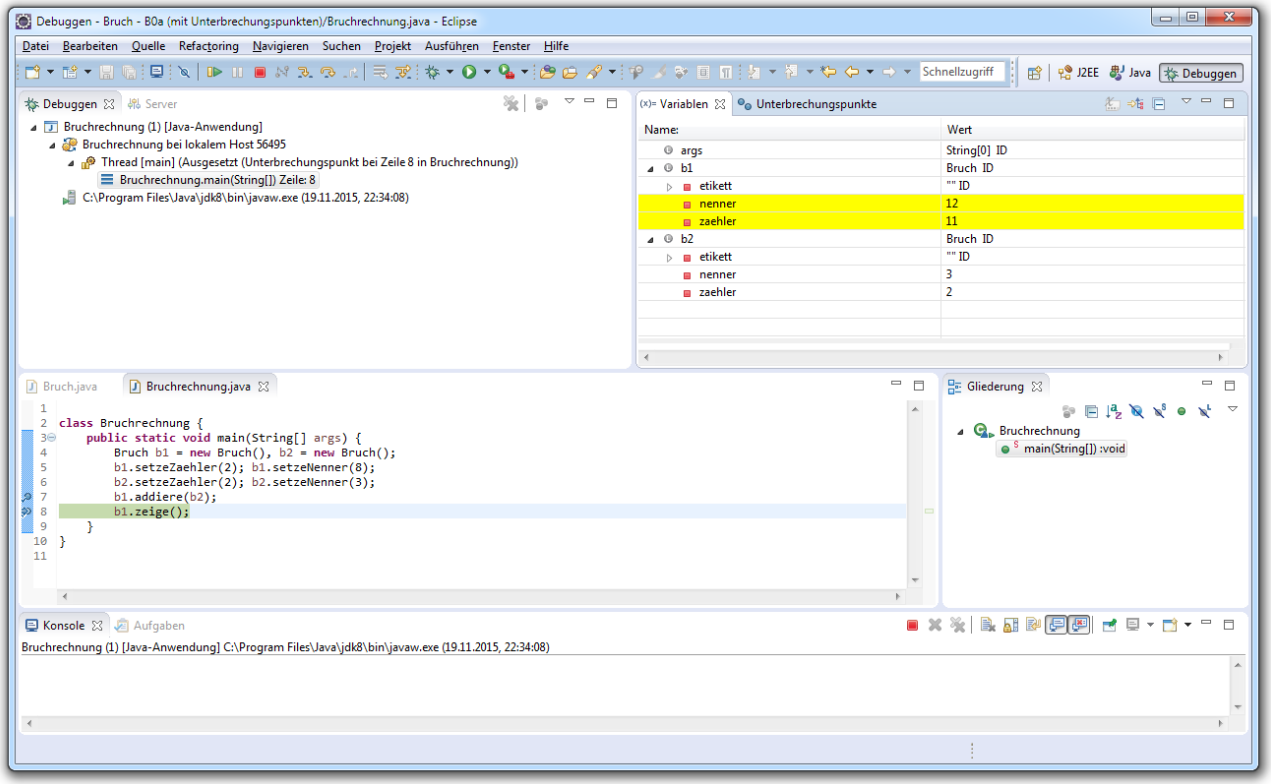
Das **Variablen**-Fenster zeigt als lokale Variablen der Methode `kuerze()`:

- **this** (Referenz auf das handelnde `Bruch`-Objekt)
- die lokalen (im Block zur **if**-Anweisung deklarierten) Variablen `rest`, `ggt` und `divisor`.

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.:

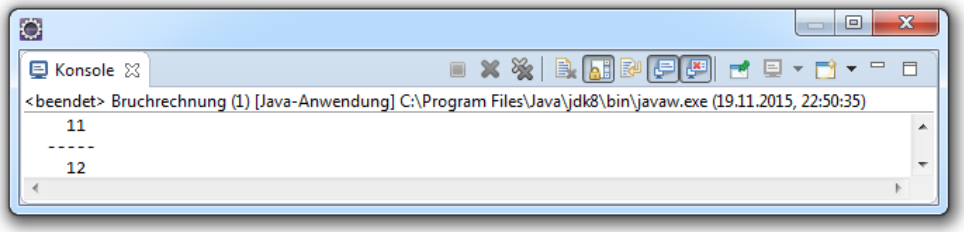


Bei Erreichen des letzten Unterbrechungspunkts (Anweisung `b1.zeige();` in **main()**) ist nur noch der Stack Frame der Methode **main()** vorhanden:

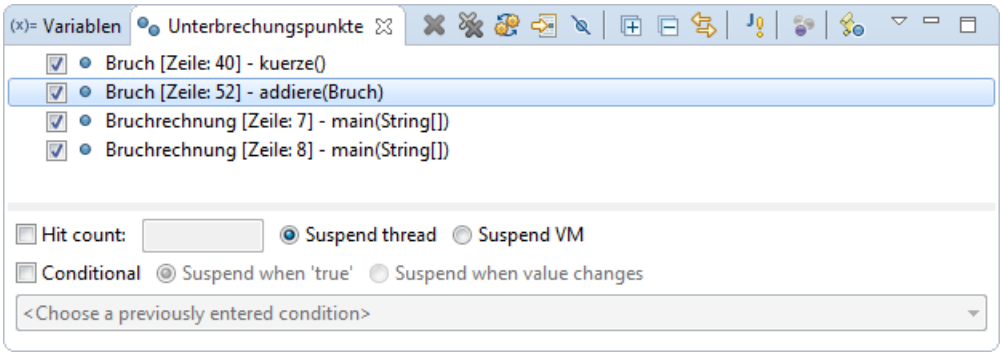


Die anderen Stack Frames sind verschwunden, und die dort ehemals vorhandenen lokalen Variablen existieren nicht mehr.

Beenden Sie das Programm durch einen letzten Fortsetzungsklick auf den Schalter , wobei die Ausgabe der Methode `zeige()` im Konsolenfenster erscheint:



Im Fenster (bzw. in der Sicht) **Unterbrechungspunkte** sind alle Unterbrechungspunkte aufgelistet:



Über die Symbolleiste oder das Kontextmenü dieses Fensters kann man z.B. alle Unterbrechungspunkte löschen.

Kehren Sie per Mausklick auf den Symbolleistenschalter Java zur Eclipse-Perspektive **Java** zurück.

Weil der verfügbare Speicher endlich ist, kann es bei der 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 in Abschnitt 4.3.1.3 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 so genannte *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 sind identisch.
- Die Formalparameterlisten sind gleich lang, und die Typen korrespondierender Parameter stimmen überein.

Für die Signatur ist der Rückgabetyp einer Methode ebenso irrelevant wie die Namen ihrer Formalparameter. Die fehlende Signaturrelevanz des Rückgabetyps resultiert daraus, dass der Rückgabewert einer Methode in Anweisungen oft keine Rolle spielt (ignoriert wird). Folglich muss unabhän-

[1] Bei den später zu behandelnden *generischen* Methoden muss die Liste der Kriterien für die Identität von Signaturen erweitert werden.

gig vom Rückgabetyp entscheidbar sein, welche Methode aus einer Überladungsfamilie zu verwenden ist.

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 Eclipse-Compiler äußert sich so:

```
Exception in thread "main" java.lang.Error: Unaufgelöstes Kompilierungsproblem:
 Die Methode addiere(long, int) ist für den Typ Bruch mehrdeutig (ambiguous)
 at Bruchrechnung.main(Bruchrechnung.java:6)
```

Bei einem vernünftigen 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 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 Typen zu einem deutlich höheren Zeitaufwand bei der Methodenausführung, so dass 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]
- **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 vorhandenen 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 (siehe unten) 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 er hier als Name eines *Konstruktors* (siehe Abschnitt 4.4.3) aufzufassen ist. Als Wert des Ausdrucks resultiert eine Referenz (Speicheradresse), die einen Zugriff auf das neue Objekt (seine Instanzvariablen und -methoden) erlaubt.

In der **main()** - Methode der folgenden Startklasse

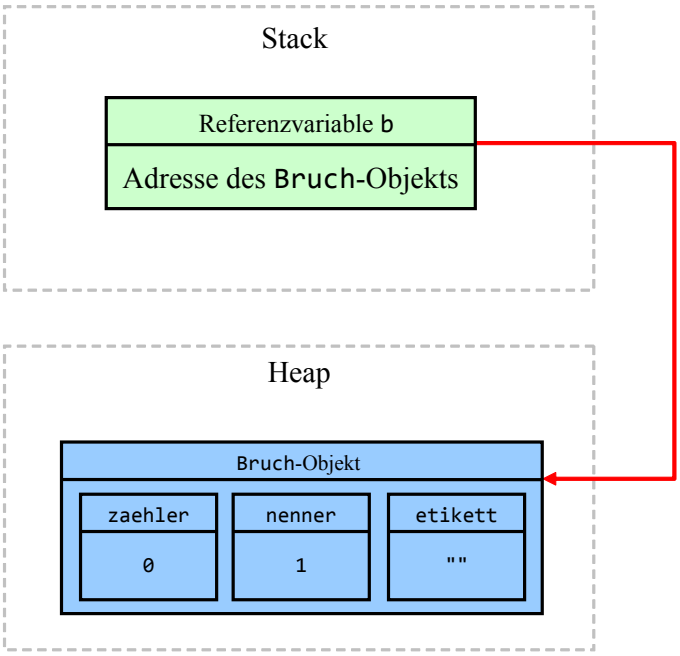
```
class Bruchrechnung {
  public static void main(String[] args) {
    Bruch b = new Bruch();
      . . .
  }
}
```

wird die vom **new**-Operator gelieferte Adresse in die lokale Referenzvariable `b` geschrieben. Es resultiert die folgende Situation im programmeigenen Hauptspeicher:[2]

---

[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.

[2] 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`-Referenzinstanzvariablen `etikett` befindet sich die Adresse des **String**-Objekts.



Während lokale Variablen im Stack-Bereich des Hauptspeichers angelegt werden, entstehen Objekte mit ihren Instanzvariablen auf dem Heap .

In einem Programm können *mehrere* Referenzvariablen auf *dasselbe* Objekt zeigen, z.B.:

Quellcode	Ausgabe

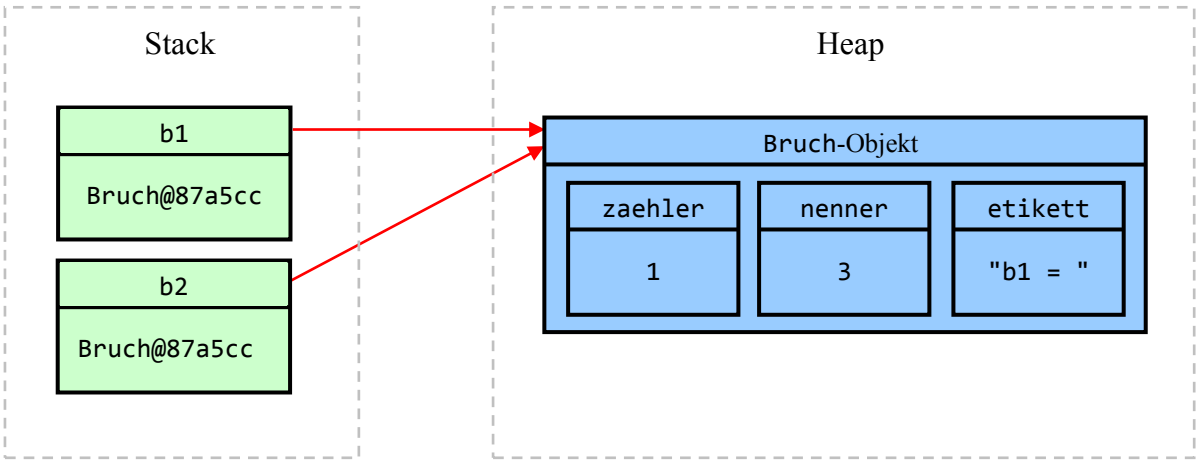
```
class Bruchrechnung {
    public static void main(String[] args) {
        Bruch b1 = new Bruch();
        b1.setzeZaehler(1);
        b1.setzeNenner(3);
        b1.setzeEtikett("b1 = ");
        Bruch b2 = b1;
        b2.zeige();
    }
}
```

```
        1
b1 =  -----
        3
```

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.

4.4.3 Objekte initialisieren über Konstruktoren

In diesem Abschnitt werden spezielle Methoden behandelt, die beim Erzeugen von neuen Objekten aufgerufen 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 anzugeben.

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 b2 = new Bruch();
```

Der Standardkonstruktor beschränkt sich auf den Aufruf des parameterfreien Konstruktors der Basisklasse (siehe unten) und hat dieselbe Schutzstufe wie die Klasse, so dass beim Standardkonstruktor der Klasse `Bruch` die Schutzstufe **public** resultiert.

Eventuell empfinden manche Leser den doppelten Auftritt des Klassennamens bei einer Referenzvariablendeklaration mit Initialisierung als störend redundant. Hier sind aber zwei Sprachbestandteile (Variablendeklaration und Objektkreation) involviert, die beide den Klassennamen enthalten müssen:

---

[1] Man ist geneigt, beim Erzeugen eines neuen Objekts der Klasse eine aktive Rolle zuzuschreiben. Allerdings lassen sich in einem Konstruktor die Instanz-Member des neuen Objekts genauso verwenden wie in einer Instanzmethode (siehe unten), was (wie die Abwesenheit des Modifikators **static**, vgl. Abschnitt 4.5.3) den Konstruktor in die Nähe einer Instanzmethode rückt. Laut Sprachbeschreibung zu Java 8 ist ein Konstruktor allerdings überhaupt kein Member, also weder Instanz- noch Klassenmethode (Gosling et al. 2015, Abschnitt 8.2). Für die Praxis der Programmierung ist es irrelevant, welchem Akteur man die Ausführung des Konstruktors zuschreibt.

- Für die Variablendeklaration ist die Angabe des Datentyps (also des Klassennamens) unverzichtbar.
- Wenn der **new**-Operator ein Objekt bestimmten Typs kreieren soll, kommt man um die Nennung des Klassennamens nicht herum.

Bei der Referenzvariablendeklaration mit Initialisierung stehen beide Sprachbestandteile unmittelbar hintereinander. Es wäre wohl kaum sinnvoll, per Ausnahmeregel eine Kürzung zu erlauben.

Um Polymorphie zu ermöglichen, sind auch Basisklassen, abstrakte Klassen und Schnittstellen als Datentypen erlaubt und sinnvoll. Nutzt man solche Datentypen, stimmen bei der Referenzvariablendeklaration mit Initialisierung der deklarierte Datentyp und der Klassenname im **new**-Operator *nicht* überein.

In der Regel ist es beim Klassendesign sinnvoll, mindestens einen Konstruktor *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.
- Der Konstruktor liefert grundsätzlich *keinen* Rückgabewert, und es wird *kein* Rückgabetyp angegeben, auch nicht der Ersatztyp **void**, mit dem wir bei gewöhnlichen Methoden den Verzicht auf einen Rückgabewert dokumentieren müssen.
- Wie bei einer gewöhnlichen Methodendefinition ist eine Parameterliste anzugeben, ggf. eine leere.
- Sobald man einen expliziten Konstruktor definiert, 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**).
- Während der Standardkonstruktor die Schutzstufe der Klasse übernimmt, gelten für selbstdefinierte 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 *nicht* die Konstruktoren ihrer Basisklasse. Allerdings wird bei jeder Objektkreation ein Basisklassenkonstruktor aufgerufen. Wenn dies nicht explizit über das Schlüsselwort **super** als Bezeichnung eines Basisklassenkonstruktors geschieht, wird der parameterfreie Basisklassenkonstruktor automatisch aufgerufen. Mit Fragen zur Objektkreation, die im Zusammenhang mit der Vererbung stehen, werden wir uns in Abschnitt 7.4 beschäftigen.
- Es 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.

[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 unten).

Es ist unbedingt zu vermeiden, dass durch öffentlich zugängliche Konstruktoren das Prinzip der Datenkapselung ausgehebelt wird, indem Instanzvariablen auf beliebige Werte gesetzt und somit defekte Objekte erzeugt werden können.

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 sprich hier von *Fabrikmethoden* (engl.: *factory methods*). Im Fall der Klasse **Box** ist ein öffentlicher Konstruktor verfügbar, so dass 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 verwendet, z.B.:

```
public static Box createHorizontalBox() {
 return new Box(BoxLayout.X_AXIS);
}
```

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 natürlich 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:

---

[1] Eine sinnvolle Reaktion auf den Versuch, ein defektes Objekt zu erstellen, kann darin bestehen, im Konstruktor eine so genannte *Ausnahme* zu werfen und dadurch den Aufrufer über das Scheitern seiner Absicht zu informieren. Mit der Kommunikation über Ausnahmeobjekte werden wir uns später beschäftigen.

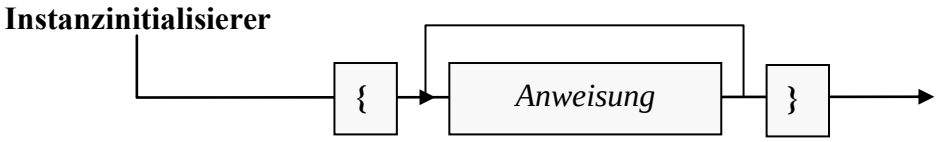
Quellcode	Ausgabe
`class Bruchrechnung {` `  public static void main(String[] args) {` `  Bruch b1 = new Bruch(1, 2, "b1 = ");` `  Bruch b2 = new Bruch();` `  b1.zeige();` `  b2.zeige();` `  }` `}`	`1` `b1 = -----` `2` `0` `-----` `1`

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");
}
```

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.



Ein Instanzinitialisierer kommt z.B. in Frage, wenn für eine Instanzvariable eine komplexe (nicht per Wertzuweisungsausdruck zu erledigende) Initialisierung unmittelbar hinter der Deklaration stehen soll (statt in einem später folgenden Konstruktor). Im folgenden Beispiel wird mit Hilfe der statischen Methode `gint()` aus unserer Bequemlichkeitsklasse `Simput` (vgl. Abschnitt 3.4) der Wert einer Instanzvariablen beim Benutzer erfragt:

```
public class InstInit {
 private int alter;
 {
  System.out.print("Ihr Alter: ");
  alter = Simput.gint();
 }
  . . .
}
```

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.

Bei gewöhnlichen Klassen werden Instanzinitialisierer nur selten verwendet (Evans & Flanagan 2015, S. 110). In den später vorzustellenden anonymen Klassen (siehe Abschnitt 12.1.1.2) werden sie aber gelegentlich benötigt, weil dort mangels Klassenname keine Konstruktoren möglich sind.

4.4.5 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 vorhanden ist.

Bei 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. Es ist jedoch durchaus möglich (und normal), dass ein Objekt die erzeugende Methode überlebt, weil eine Referenz nach Außen transportiert worden ist (z.B. per Rückgabewert, vgl. Abschnitt 4.4.6).

Andererseits kann man ein methodenintern erzeugtes Objekt schon während der Methodenausführung „aufgeben“, indem man die Referenz auf das Objekt entfernt. Dazu setzt man die entsprechende Referenzvariable entweder auf den Wert **null** oder weist ihr eine andere Referenz zu, z.B.:

```
b1 = null;
```

Vermutlich sind Programmiereinsteiger vom Garbage Collector nicht sonderlich beeindruckt. Schließlich war im Manuskript noch nie die Rede davon, dass man sich um den belegten Speicher nach Gebrauch kümmern müsse. Der in einer Methode von lokalen Variablen belegte Speicher wird bei *jeder* Programmiersprache frei gegeben, sobald die Ausführung der Methode beendet ist. Demgegenüber muss der von *Objekten* belegte Speicher bei älteren Programmiersprachen (z.B. C++) nach Gebrauch explizit wieder frei gegeben werden. In Anbracht 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 vernichtete 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, so dass der genaue Zeitpunkt für die Entsorgung eines Objekts kaum vorhersehbar ist.

Mehr müssen Programmiereinsteiger über die Arbeitsweise des Garbage Collectors nicht wissen. Wer sich trotzdem dafür interessiert, findet im Rest dieses Abschnitts noch einige Details.[1]

Sollen die Objekte einer Klasse vor dem Entsorgen durch den Garbage Collector noch spezielle Aufräumaktionen durchführen, dann muss eine Methode namens **finalize()** nach folgendem Muster definiert werden, die ggf. vom Garbage Collector aufgerufen wird, z.B.:

---

[1] Es soll nicht verschwiegen werden, dass bei großen Anwendungen die im Hintergrund ablaufenden GC-Aktivitäten zu spürbaren temporären Leistungseinbußen führen können, so dass Programmierer über Maßnahmen zur GC-Optimierung nachdenken müssen. Auf der folgenden Webseite finden sich diesbezügliche Erläuterungen und Tipps der Firma Oracle:

http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html

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.

```
protected void finalize() throws Throwable {
  super.finalize();
  System.out.println(this + " finalisiert");
}
```

In dieser Methodendefinition tauchen einige Bestandteile auf, die bald ausführlich zur Sprache kommen und hier ohne großes Grübeln hingenommen werden sollten:

- `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.
- `protected`
  In der Klasse **Object** ist für **finalize()** die Schutzstufe **protected** festgelegt, und dieser Zugriffsschutz darf beim Überschreiben der Methode nicht verschärft werden. Die ohne Angabe eines Modifikators voreingestellte Schutzstufe *Paket* enthält gegenüber **protected** eine Einschränkung und ist daher verboten.
- `throws Throwable`
  Die **finalize()** - Methode der Klasse **Object** löst ggf. eine Ausnahme aus der Klasse **Throwable** aus. 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.
- `this`
  In der aus didaktischen Gründen eingefügten Kontrollausgabe wird mit dem Schlüsselwort **this** (vgl. Abschnitt 4.4.6.2) das aktuell handelnde Objekt angesprochen. Bei der automatischen Konvertierung der Referenz in eine Zeichenfolge wird die vom Laufzeitsystem verwaltete Objektbezeichnung zu Tage fördert.

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 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();
 }
}
```

Dass anschließend der Garbage Collector aufgrund der expliziten Aufforderung tatsächlich tätig wird, ist an den Kontrollausgaben der `finalize()` - Methode zu erkennen. Bei der Entsorgungsreihenfolge treten beide Varianten ohne erkennbare Regel auf:

`b1: Bruch@21a722ef, b2: Bruch@63e68a2b` `Bruch@21a722ef finalisiert` `Bruch@63e68a2b finalisiert`	`b1: Bruch@60f32dde, b2: Bruch@7d487b8b` `Bruch@7d487b8b finalisiert` `Bruch@60f32dde finalisiert`

Im Normalfall müssen Sie sich um das Entsorgen überflüssiger Objekte *nicht* kümmern, also weder eine **finalize()** - Methode für eigene Klassen definieren, noch die **System**-Methode **gc()** aufrufen.

4.4.6 Objektreferenzen verwenden

Methodenparameter mit Referenztyp wurden schon in Abschnitt 4.3.1.3.2 behandelt. In diesem Abschnitt geht es um Methodenrückgabewerte mit Referenztyp und um das Schlüsselwort **this**, mit dem in einer Methode das aktuell handelnde Objekt angesprochen werden kann.

4.4.6.1 Rückgabewerte 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 abzuliefern:

```
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:

Quellcode	Ausgabe
`class Bruchrechnung {` `  public static void main(String[] args) {` `  Bruch b1 = new Bruch(1, 2, "b1 = ");` `  b1.zeige();` `  Bruch b2 = b1.klone();` `  b2.zeige();` `  }` `}`	`  1` `b1 = -----` `  2` `  1` `b1 = -----` `  2`

4.4.6.2 this als Referenz auf das aktuelle Objekt

Gelegentlich ist es sinnvoll oder erforderlich, dass ein handelndes Objekt sich selbst ansprechen bzw. seine Adresse als Methodenaktualparameter verwenden kann. Dies ist mit dem Schlüsselwort **this** möglich, das innerhalb einer Instanzmethode wie eine Referenzvariable funktioniert. In folgendem 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 werden soll. Später werden Sie noch weit relevantere **this**-Verwendungsmöglichkeiten kennen lernen.

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 Deklaration bzw. Definition kaum Unterschiede zwischen einem Instanz- und dem analogen Klassenmitglied.

Auch bei den statischen Klassen-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 dürfen zugreifen.

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 = "";

   static private int anzahl;

   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 *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), so dass im Beispiel die Variable `anzahl` mit dem **int**-Wert 0 startet.

Im Java-Editor der Entwicklungsumgebung Eclipse 4.x werden statische Variablen per Voreinstellung durch *kursive Schrift* gekennzeichnet (siehe obigen Quellcode).

In Instanz- oder Klassenmethoden der eigenen Klasse lassen sich Klassenvariablen ohne jeden 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 ist vom Referenztyp und zeigt auf ein Objekt der Klasse **PrintStream**, dem wir unsere 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 der typspezifischen Null. Die somit erforderliche explizite Initialisierung kann bei der Deklaration oder im statischen Initialisierer (siehe Abschnitt 4.5.4) erfolgen. Im weiteren Programmverlauf ist bei finalisierten Klassenvariablen keine Wertänderung mehr möglich.

Bei häufig benötigten Konstanten bewährt sich eine Variablendeklaration mit den drei Modifikatoren **public**, **static** und **final**, z.B. beim **double**-Feld **PI** in der API-Klasse **Math** (Paket **java.lang**), das die Kreiszahl π enthält:

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

In diesem Beispiel wird eine von Sun/Oracle vorgeschlagene Konvention beachtet, im Namen einer *finalisierten statischen* Variablen ausschließlich Großbuchstaben zu verwenden.[1] Besteht ein Name aus mehreren Wörtern, sollen diese der Lesbarkeit halber durch einen Unterstrich getrennt werden, z.B.:

```
public final static int DEFAULT_SIZE = 100;
```

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 kennen gelernt, 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:

- **Lokale Variablen ...**
  werden in Methoden vereinbart,
  landen auf dem Stack,
  werden **nicht** automatisch initialisiert,
  sind nur in den Anweisungen des innersten Blocks verwendbar.

---

[1] Siehe: http://www.oracle.com/technetwork/java/codeconventions-135099.html
Finalisierte und statische *Referenz*variablen (z.B. **System.out**) sind bei diesem Benennungsvorschlag wohl nicht einbezogen.

[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 2015, Abschnitt 12.7). Eine vom regulären Klassenlader der JRE geladene Klasse wird nicht vor dem Ende des Programms entladen (Ullenboom 2012a, Abschnitt 11.5).

- **Instanzvariablen ...**
  werden außerhalb jeder Methode deklariert,
  landen (als Bestandteile von Objekten) auf dem Heap,
  werden 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,
  werden 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` `static private 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 einer Java-Klasse deren Methode **main()** 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 kaum 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. So wird im **Math**-Quellcode das Instanzieren verhindert:

```
private Math() {}
```

In Abschnitt 3.5.2 wurde demonstriert, wie die **Math**-Klassenmethode **pow()** von einer fremden Klasse aufgerufen werden kann:

```
System.out.println(4 * Math.pow(2, 3));
```

Vor den Namen der gewünschten 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.

Oft ist es sinnvoll, klassenbezogene Kompetenzen mit objektbezogenen zu kombinieren. 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:

Quellcode	Ausgabe

```
class Bruchrechnung {
 public static void main(String[] args) {
  System.out.println(Bruch.hanz() + " Brueche 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() + " Brueche erzeugt");
 }
}
```

```
0 Brueche erzeugt
          1
 Bruch 1 -----
          2

          5
 Bruch 2 -----
          6

2 Brueche erzeugt
```

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, um eine laufende Nummer zum neu erzeugten `Bruch`-Objekt auszugeben:

```
public Bruch klone() {
 Bruch b = new Bruch(zaehler, nenner, etikett);
 System.out.println("Neuer Bruch mit der Nr. " + hanz() + " erzeugt");
 return b;
}
```

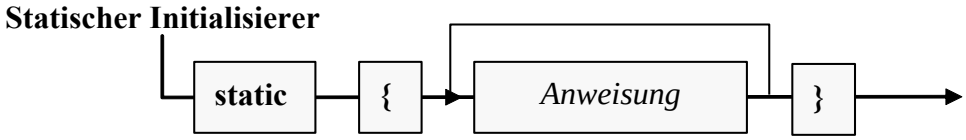
Gelegentlich wird missverständlich behauptet, in einer statischen Methode könnten keine Instanzmethoden aufgerufen werden, z.B. (Mössenböck 2005, S. 153):

„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 Instanzkonstruktoren, die beim Erzeugen eines Objekts ausgeführt werden (siehe Abschnitt 4.4.3), 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. 2015, Abschnitt 8.7). Ein syntaktischer Unterschied zu den Instanzkonstruktoren besteht darin, dass bei einem statischen Initialisierer *kein* Name angegeben wird:



Außerdem sind keine Zugriffsmodifikatoren erlaubt. Diese werden auch nicht benötigt, weil ein statischer Konstruktor 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 des `Bruch`-Beispiels soll der parameterfreie Instanzkonstruktor zufallsabhängige, aber pro Programmlauf identische Werte zur Initialisierung der Felder `zaehler` und `nenner` verwenden:

```
public Bruch() {
 zaehler = zaehlerVoreinst;
 nenner = nennerVoreinst;
 anzahl++;
}
```

Dazu erhält die `Bruch`-Klasse private statische Felder, die vom statischen Initialisierer beim Laden der Klasse auf Zufallswerte gesetzt werden sollen:

```
private static int zaehlerVoreinst;
private static int nennerVoreinst;
```

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 Null bis Vier beauftragt. Daraus entstehen Startwerte für die Felder `zaehler` und `nenner`:

```
public class Bruch {
    . . .

  static {
   java.util.Random zuf = new java.util.Random();
   zaehlerVoreinst = zuf.nextInt(5)+1;
   nennerVoreinst = zuf.nextInt(5)+zaehlerVoreinst;
   System.out.println("Klasse Bruch geladen");
  }
   . . .
}
```

Außerdem protokolliert der statische Initialisierer noch das Laden der Klasse, z.B.:

Quellcode	Ausgabe
`class Bruchrechnung {` `  public static void main(String[] args) {` `   Bruch b = new Bruch();` `   b.zeige();` `  }` `}`	`Klasse Bruch geladen` `    5` `  -----` `    9`

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 natürlichen Zahlen, die in der `Bruch`-Methode `kuerze()` benötigt wird. Sie haben bereits zwei *iterative* (mit einer Schleife realisierte) Varianten des Euklidischen Lösungsverfahrens kennen gelernt: In Abschnitt 1.1 wurde ein sehr einfacher Algorithmus benutzt, den Sie später in einer Übungsaufgabe (siehe Seite 178) durch einen effizienteren Algorithmus (unter Verwendung der Modulo-Operation) 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:[1]

[1] Wenn die Methode `ggTi()` oder die gleich darzustellende Methode `ggTr()` beim Aufruf eine 0 als Wert für den zweiten Parameter erhält, 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 eine 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 die Division durch 0 sehr 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, und der Algorithmus endet mit der Rückgabe des Betrags von `b`:

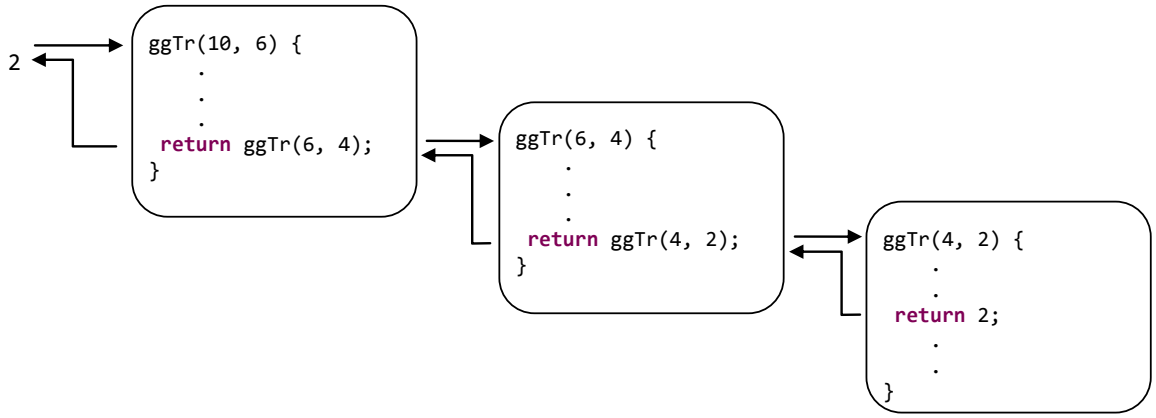
  ```
  return Math.abs(b);
  ```

- 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:

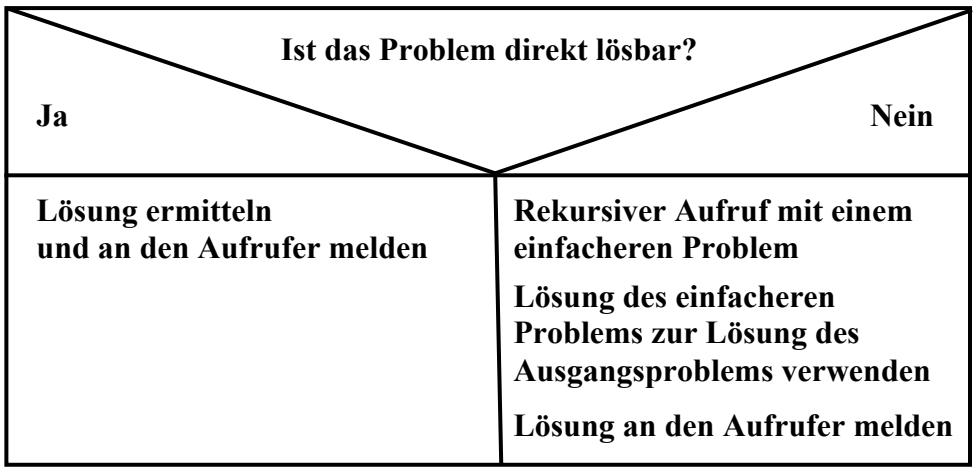
  ```
  return ggTr(b, rest);
  ```

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 in 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). In der aktuellen `Bruch`-Definition ist z.B. 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*) von Klassen eine effektive Technik zur Wiederverwendung von Software bzw. zum Aufbau von komplexen Softwaresystemen. Außerdem ist sie im Sinne einer realitätsnahen Modellierung unverzichtbar, denn auch ein reales Objekt (z.B. eine Fir-

ma) enthält andere Objekte[1] (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 ist, 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 dem recht länglichen Beispiel zur Komposition überspringen.

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;
  }
```

[1] Die betroffenen Personen mögen den Fachterminus *Objekt* nicht persönlich nehmen.

```
  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 Zaehler 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 Methode `frage()`) im Rahmen einer Aufgabenstellung vorgelegt, z.B. zum Addieren.
- In `antwort` landet der Lösungsversuch des Anwenders.
- In `lsg` steht das korrekte (und gekürzte) Ergebnis.

Im folgenden Programm wird die Klasse `Aufgabe` für ein Bruchrechnungstraining eingesetzt:

```
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();
 }
}
```

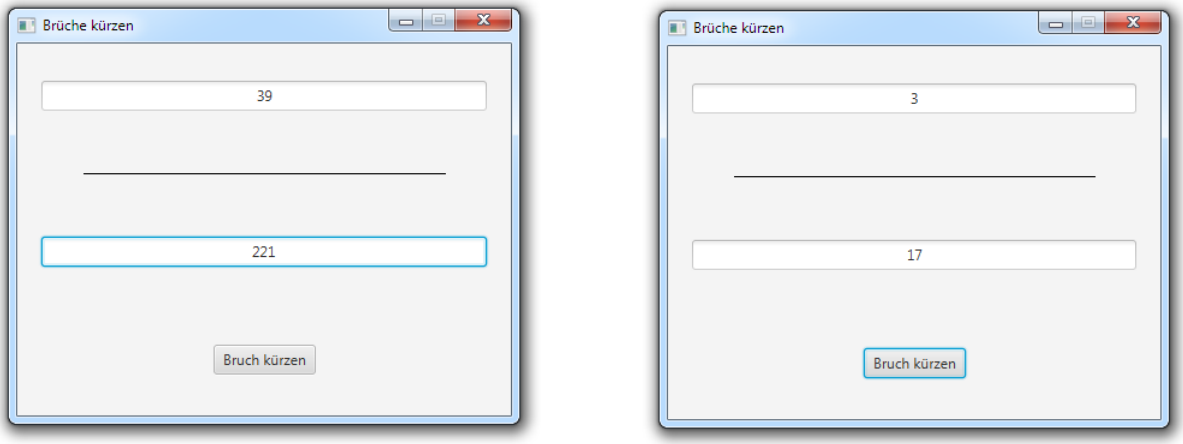
Man kann immerhin schon ahnen, wie die praxistaugliche Endversion des Programms einmal arbeiten wird:

```
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 Zaehler hat Dein Ergebnis:     3
                                      ------
Welchen Nenner hat Dein Ergebnis:      7

 Falsch!
                          9
 Das korrekte Ergebnis: -----
                          10
```

4.8 Bruchrechnungsprogramm mit JavaFX-GUI

Nachdem Sie nun wesentliche Teile der objektorientierten Programmierung mit Java kennen gelernt haben, ist vielleicht 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 Programmieren Spaß machen kann.

Wir erstellen unter Verwendung der Klasse `Bruch` mit Hilfe des Eclipse - Plugins **e(fx)clipse** und des externen GUI-Designers **Scene Builder** ein Bruchkürzungsprogramm mit grafischer Benutzerschnittstelle in JavaFX-Technik:



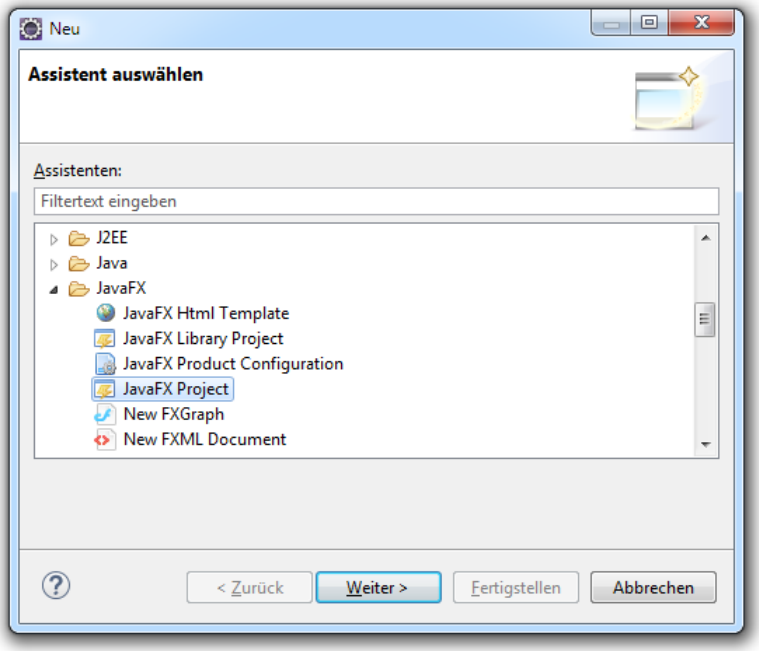
Aufgrund der individuellen Oberflächengestaltung kommt die in Abschnitt 3.8 vorgestellte Klasse **JOptionPane** mit statischen Methoden zur bequemen Realisation von Standarddialogen *nicht* in Frage.

4.8.1 JavaFX-Projekt mit FXML-Nutzung anlegen

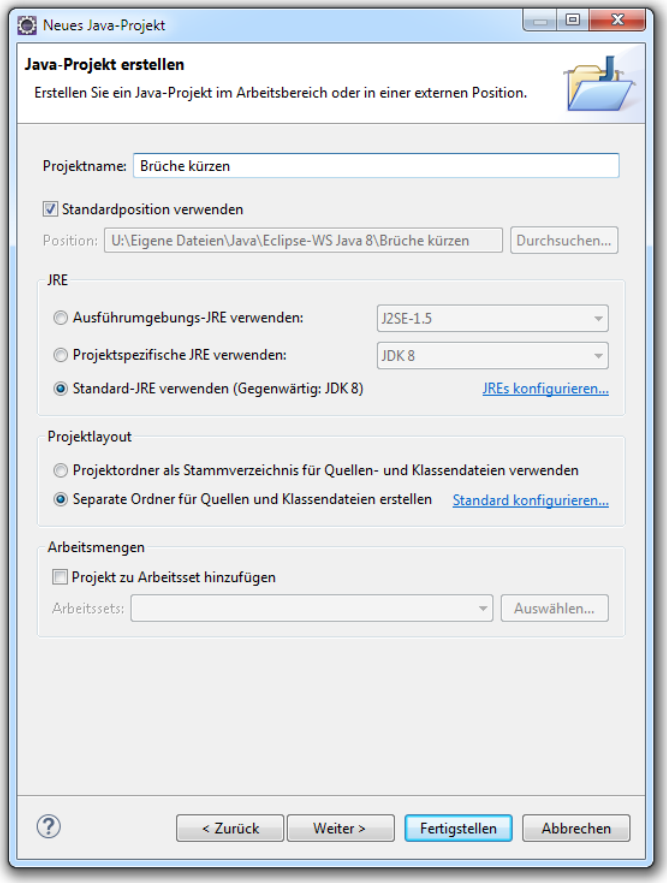
Wir starten mit dem Menübefehl

Datei > Neu > Andere

und wählen im folgenden Dialog



den Assistenten für ein neues **JavaFX Project**. Wir benötigen nur den ersten Assistentendialog und geben dort den Projektnamen `Brüche Kürzen` an:

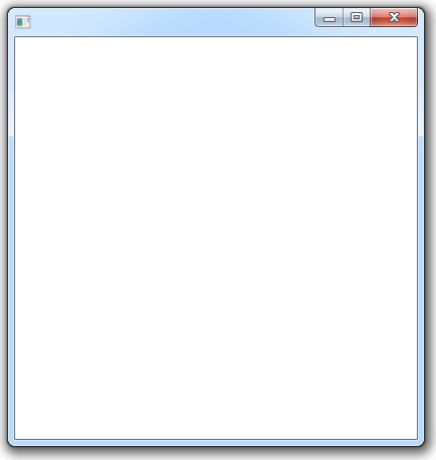


Wir tragen den **Projektnamen** ein und quittieren mit einem Kick auf den Schalter **Fertigstellen**.

Der Assistent hat bereits eine Hauptklasse mit dem Namen `Main` angelegt:

```
package application;

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;


public class Main extends Application {
 @Override
 public void start(Stage primaryStage) {
  try {
    .  .  .
  } catch(Exception e) {
   e.printStackTrace();
  }
 }

 public static void main(String[] args) {
  launch(args);
 }
}
```

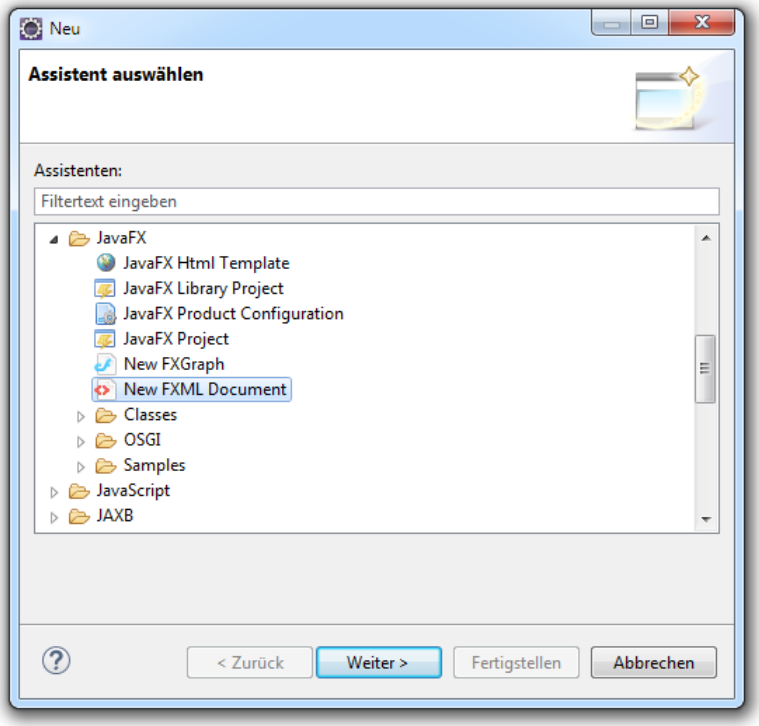
Das Programm ist schon startfähig, zeigt aber bisher nur eine leere Szenerie:



Wir wollen beim JavaFX-Einsatz die Bedienoberfläche in einer FXML-Datei deklarieren und zu deren Gestaltung das von der Firma Oracle entwickelte Programm **Scene Builder** verwenden. Um eine leere FXML-Datei anzulegen, wählen wir aus dem Kontextmenü zum `application`-Eintrag im Projekt den Befehl[1]

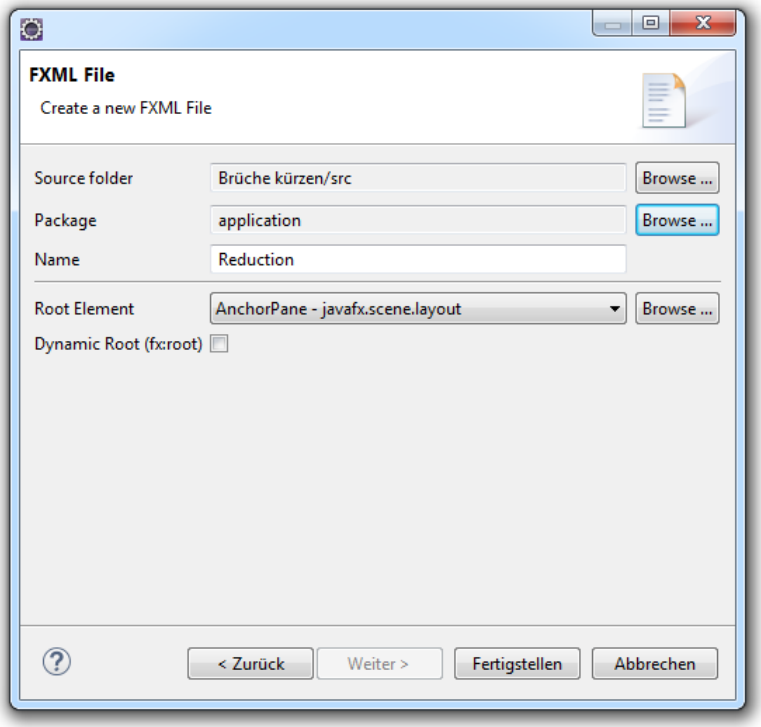
Neu > Andere

und wählen im folgenden Dialog

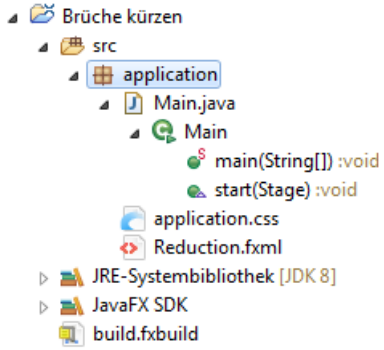


den Assistenten für ein neues **FXML-Dokument**. Wir benötigen nur den ersten Assistentendialog, wo wir den Namen **Reduction** für die FXML-Datei angeben (mit großem Anfangsbuchstaben!). Nach dem **Fertigstellen**

[1] Weil sich die Klasse `Main` aufgrund einer Assistentenentscheidung im Paket `application` befindet (siehe erste Quellcodezeile), sollte auch die FXML-Datei hier angelegt werden.



ist die Datei **Reduction.fxml** im Paket-Explorer zu sehen



und im Editor geöffnet:

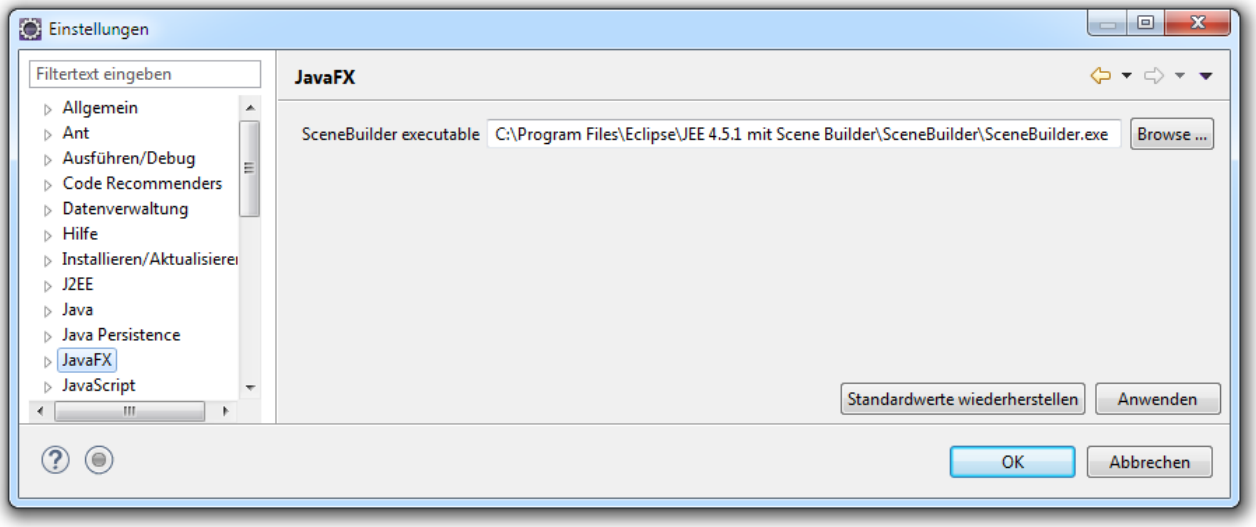
```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.AnchorPane?>

<AnchorPane xmlns:fx="http://javafx.com/fxml/1">
 <!-- TODO Add Nodes -->
</AnchorPane>
```

4.8.2 FXML-Datei mit dem Scene Builder gestalten

Damit der Scene Builder bequem aus Eclipse gestartet werden kann, muss nach

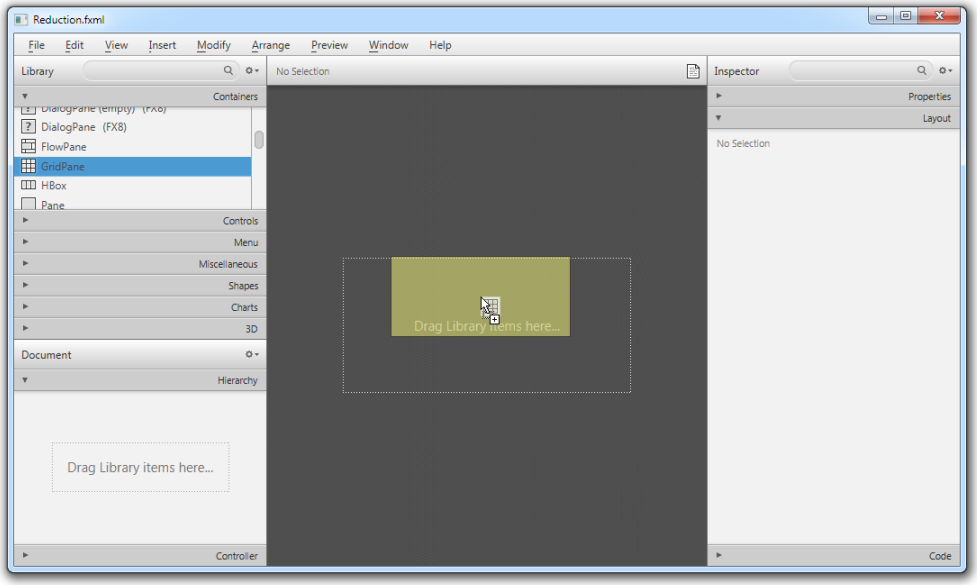
Fenster > Benutzervorgaben > JavaFX

der Pfad zur Datei **SceneBuilder.exe** angegeben werden, z.B.:

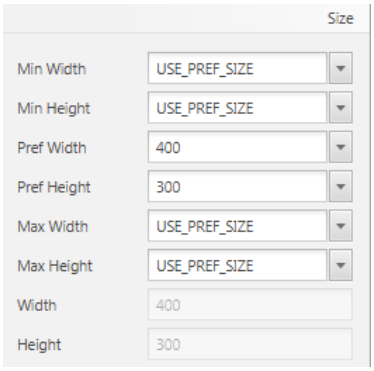


Um die FXML-Datei mit dem Scene Builder zu öffnen, genügt es nach dieser Vorbereitung, aus dem Kontextmenü zu dieser Datei das Item **Open with Scene Builder** zu wählen.

Wir löschen das vorgegebene **AnchorPane**-Wurzelelement (siehe **Hierarchy**-Segment der **Document**-Zone am linken Fensterrand) über das Item **Delete** aus seinem Kontextmenü und ziehen aus der **Containers**-Abteilung der **Library** eine Komponente vom Typ **GridPane** per Drag & Drop auf die Inhaltzone im Mittelteil des Scene Builders:



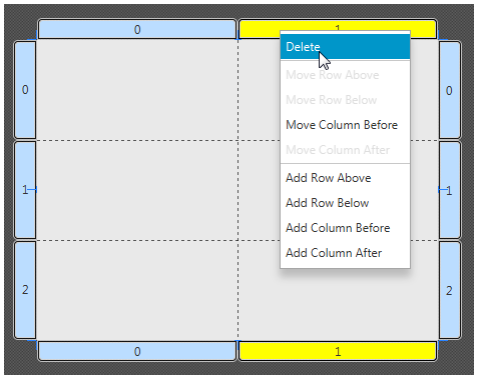
Über das **Layout**-Segment der **Inspector**-Zone setzen wir die bevorzugte Breite bzw. Höhe des Wurzelcontainers (**Pref Width** bzw. **Pref Height**) auf 400 bzw. 300:



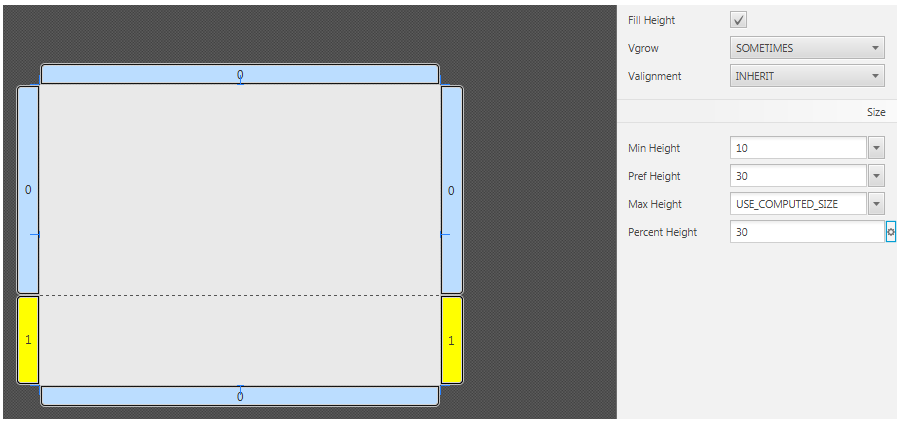
Für die Fensterstruktur soll eine äußere **GridPane**-Komponente mit einer Spalte und zwei Zeilen verwendet werden:

- In die obere Zelle soll eine innere **GridPane**-Komponente für die Bruchbestandteile eingefügt werden.
- In die untere Zelle soll ein Befehlsschalter eingefügt werden, mit dem die Kürzung angefordert wird.

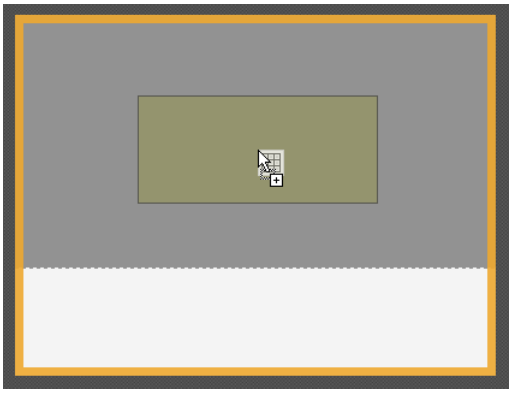
Um die überflüssige Zeile bzw. Spalte der äußeren **GridPane**-Komponente zu löschen, markieren wir sie per Mausklick auf die angeflanschte Markierfläche und wählen dann das Item **Delete** aus dem Kontextmenü der Markierfläche, z.B.:



Die verfügbare Höhe wird im Verhältnis 70:30 auf die beiden Zeilen aufgeteilt. Nach dem Markieren einer Zeile kann der gewünschte Eintrag im **Layout**-Segment der **Inspector**-Zone vorgenommen werden (**Percent Height**), z.B.:



In die obere Zelle soll ein internes **GridPane**-Element zur Aufnahme der Bruchbestandteile (zwei Textfelder für Zähler und Nenner sowie ein Bruchstrich) aufgenommen werden. Dazu übernehmen wir aus der **Containers**-Liste der **Library** eine **GridPane**-Komponente per Drag & Drop an den durch graue Hintergrundfarbe hervorgehobenen Einsatzort:



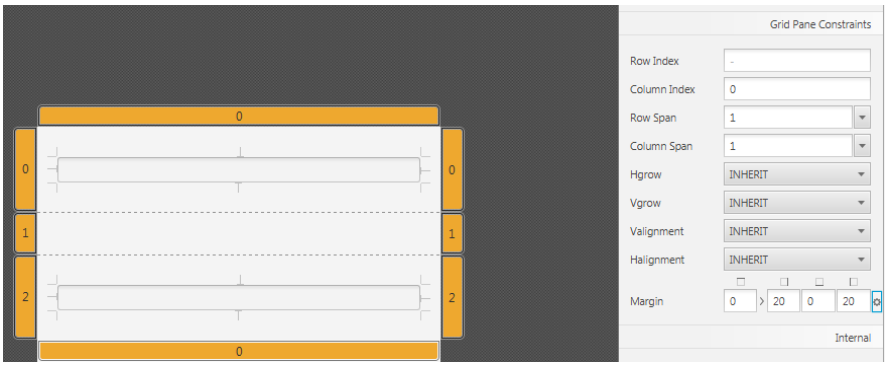
Auch das innere **GridPane**-Komponente hat initial 3 Zeilen und 2 Spalten. Während die Zahl der Zeilen zu unserer Einsatzplanung passt, müssen wir eine Spalte löschen:

- Spalte markieren
- Item **Delete** aus dem Kontextmenü der Markierungsfläche wählen

Für die drei Zeilen der inneren **GridPane**-Komponente vereinbaren wir über das **Layout**-Segment der **Inspector**-Zone die vertikale Platzaufteilung 40:20:40.

Im nächsten Schritt übertragen wir aus der **Library**-Abteilung **Controls** zwei **TextField**-Komponenten per Drag & Drop in die oberste bzw. unterste Zelle des inneren **GridPane**-Containers. Wenn versehentlich *beide* Textfelder in der obersten Zelle landen, kann eine Komponente per Drag & Drop verschoben werden.

Wir markieren beide Textfelder und verpassen ihnen über das **Layout**-Segment der **Inspector**-Zone einen linken sowie rechten Rand der Breite 20 (**Margin**):



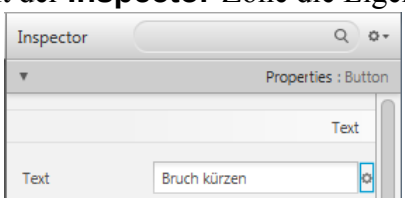
Außerdem sollte über das **Properties**-Segment der **Inspector**-Zone die Eigenschaft **Alignment** auf den Wert **CENTER** gesetzt werden, damit im aktiven Programm die vom Benutzer eingetragenen Zeichenfolgen zentriert werden.

Nun fügen wir aus der **Library**-Abteilung **Shapes** eine **Line**-Komponente per Drag & Drop in die mittlere Zelle des inneren **GridPane**-Containers ein und setzen zwei Eigenschaften über das **Layout**-Segment der **Inspector**-Zone:

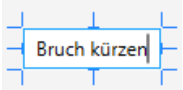
- **Halignment** erhält den Wert **CENTER**
- **End X** erhält den Wert 200, um die Linie zu verlängern

Schließlich befördern wir aus der **Library**-Abteilung **Controls** eine **Button**-Komponente per Drag & Drop in die unterste Zelle des äußeren **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:



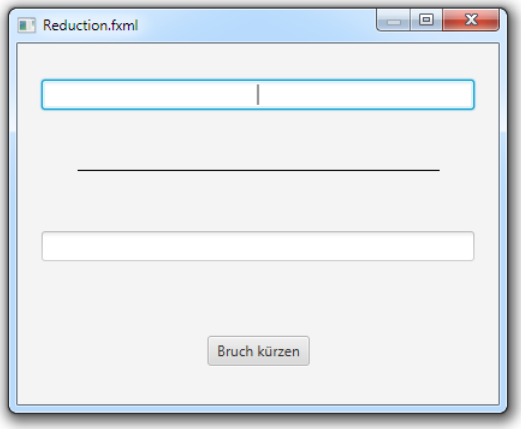
- Nach einem Doppelklick auf den Schalter in der Inhaltzone die Beschriftung editieren:



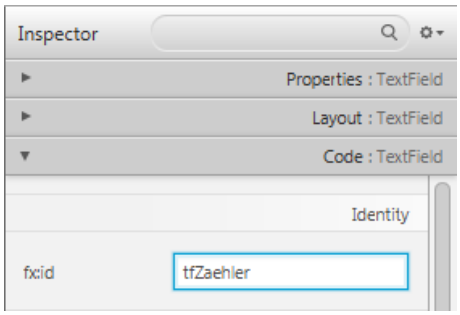
Über den Menübefehl

Preview > Show Preview in Window

können wir unser Werk begutachten:



Wir werden bald eine Ereignisbehandlungsmethode erstellen, um auf das Betätigen des Schalters regieren zu können. Darin werden wir auf GUI-Komponenten Bezug nehmen müssen. Um dies zu ermöglichen, erhalten die betroffenen Komponenten nun eine Kennung, die später als Variablenname zu verwenden ist. 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 und die Kennung `btnKuerzen` an den Befehlsschalter.

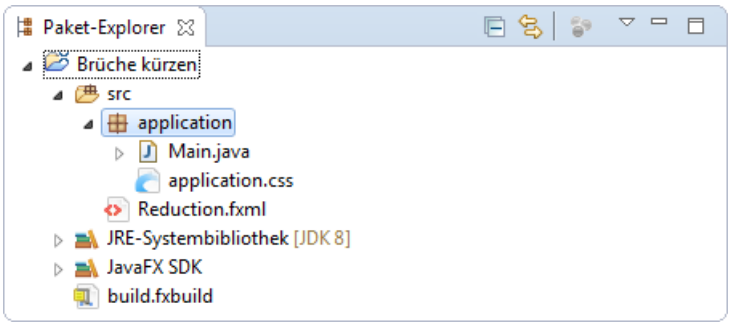
Wenn wir die per Scene Builder gestaltete FXML-Datei speichern (z.B. mit dem Menübefehl **File > Save**), ist in Eclipse nach einem Mausklick auf das betroffenen Editorfenster der aktuelle Inhalt verfügbar.

4.8.3 Klasse `Bruch` einbinden

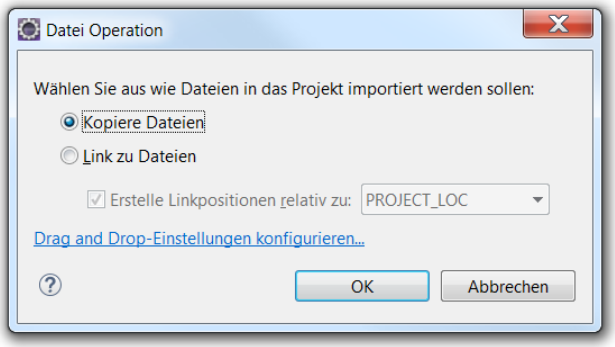
Aus den Benutzereingaben in die Textfelder des oben entworfenen Anwendungsfensters soll ein Objekt unserer Klasse `Bruch` entstehen, das bei einem Mausklick auf den Schalter mit dem Kürzen beauftragt wird. Leider stellen sich nun Designschwächen der Klasse `Bruch` heraus, die am einfachsten zu umgehen sind, indem wir den *Quellcode* der Klasse `Bruch` in das aktuelle Projekt aufnehmen und dann modifizieren, was generell keine gute Idee ist. Ziehen Sie aus einem Fenster des Windows-Explorers die Datei

…\BspUeb\Klassen und Objekte\Bruch\B3 (mit Konstruktoren)

auf das Paket `application` des Projekts:



Wählen Sie als Importmodus das Kopieren:



Alternativ können Sie die Datei **Bruch.java** mit den Mitteln des Betriebssystems in den Quellenordner des Projekts kopieren (z.B. **...\src**) und anschließend Eclipse auffordern, sich mit dem Dateisystem zu synchronisieren, z.B. über die Funktionstaste **F5** bei aktivem Paket-Explorer.

Nach der Aufnahme der Klasse `Bruch` wird das Projekt als fehlerhaft markiert, weil in **Bruch.java** die Klasse `Simput` genutzt wird, die über den Klassenpfad des aktuellen Projekts nicht auffindbar ist. Wir beseitigen den Fehler, in dem 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 in Abschnitt 4.1.1.1 angesprochene *Prinzip einer einzigen Verantwortung* (***S**ingle **R**esponsibility **P**rinciple*, SRP) interpretieren. Eventuell sollte sich die Klasse `Bruch` 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 das per Assistentenhilfe angelegte JavaFX-Projekt ein Paket namens `application` verwendet, muss auch die Klasse `Bruch` in dieses Paket aufgenommen werden. Dazu ist am Anfang des Quellcodes die folgende Zeile einzufügen:

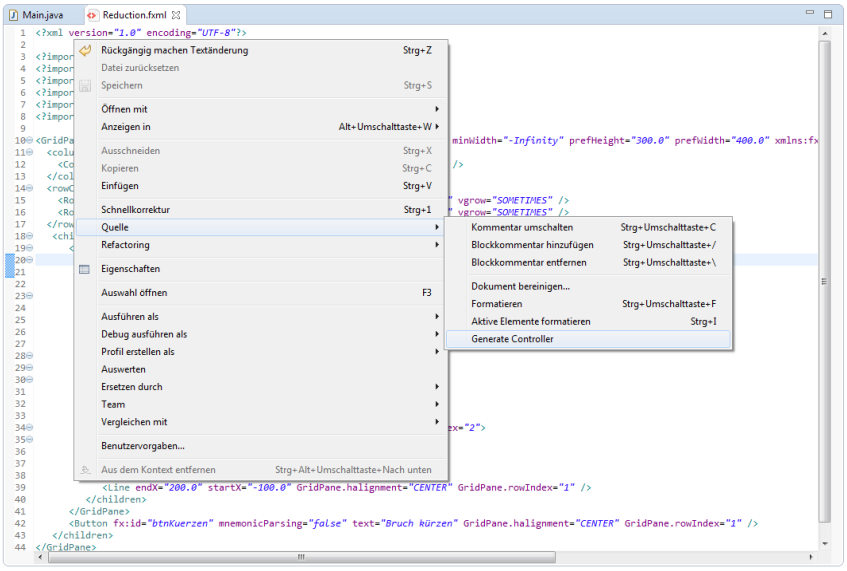
```
package application;
```

4.8.4 Controller-Klasse erstellen

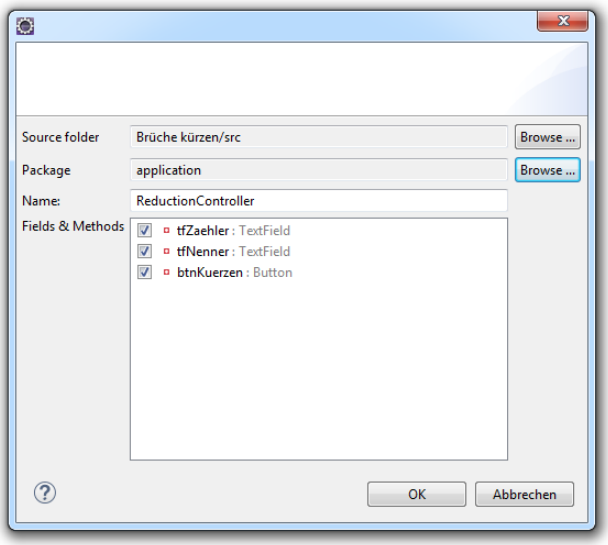
In einem JavaFX-Projekt erstellt man eine so genannte *Controller-Klasse*, die für die Ereignisbehandlung zuständig ist. Das Plugin **e(fx)clipse** macht es möglich, einen Rohling für die Controller-Klasse über den Befehl

Quelle > Generate Controller

aus dem Kontextmenü zum Editor-Fenster mit der geöffneten FXML-Datei erstellen zu lassen:



Es wird ein sinnvoller Name vorgeschlagen, und die im Scene Builder vereinbaren Kennungen sind berücksichtigt:



Schließlich erhalten wir den folgenden Rohling für die Controller-Klasse:

```
package application;

import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;

public class ReductionController {
 @FXML
 private TextField tfZaehler;
 @FXML
 private TextField tfNenner;
 @FXML
 private Button btnKuerzen;
}
```

Hier sorgt die Annotation[1] **@FXML** dafür, dass über private Instanzvariablen die GUI-Komponenten angesprochen werden können, die im Scene Builder eine Kennung (**fx.id**) erhalten haben.

Wir definieren nun eine Methode namens `buttonListener()`, die nach einem Mausklick auf den Befehlsschalter unserer Anwendung ausgeführt werden soll und verzichten dabei auf jede Absicherung gegen fehlerhafte Eingaben:[2]

```
public void buttonListener() {
 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 über eine Felddeklaration mit Initialisierung bereitgestellt:

```
private Bruch b = new Bruch();
```

Um die erstellte Controller-Klasse im Scene Builder nutzen zu können, gehen wir folgendermaßen vor:

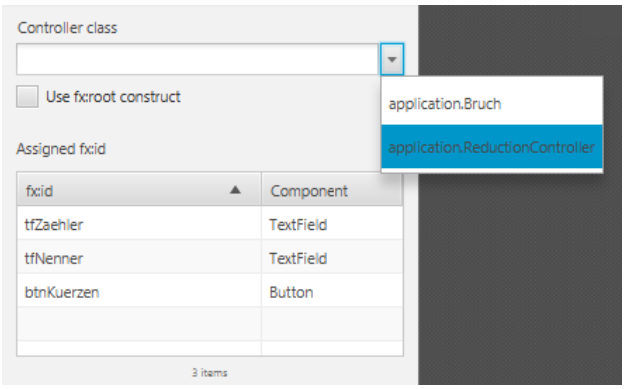
- Quellcode der Controller-Klasse speichern (z.B. über den Schalter )
- Scene Builder beenden und neu starten (wiederum über das Item **Open with Scene Builder** aus dem Kontextmenü zur FXML-Datei).

Anschließend kann im **Controller**-Segment der **Document**-Zone eine **Controller class** per Drop Down - Menü bestimmt werden, sofern sich die **class**-Datei zum Controller im selben Ordner befindet wie die FXML-Datei. Wir wählen die eben definierte Klasse `ReductionController` im Paket `application`:

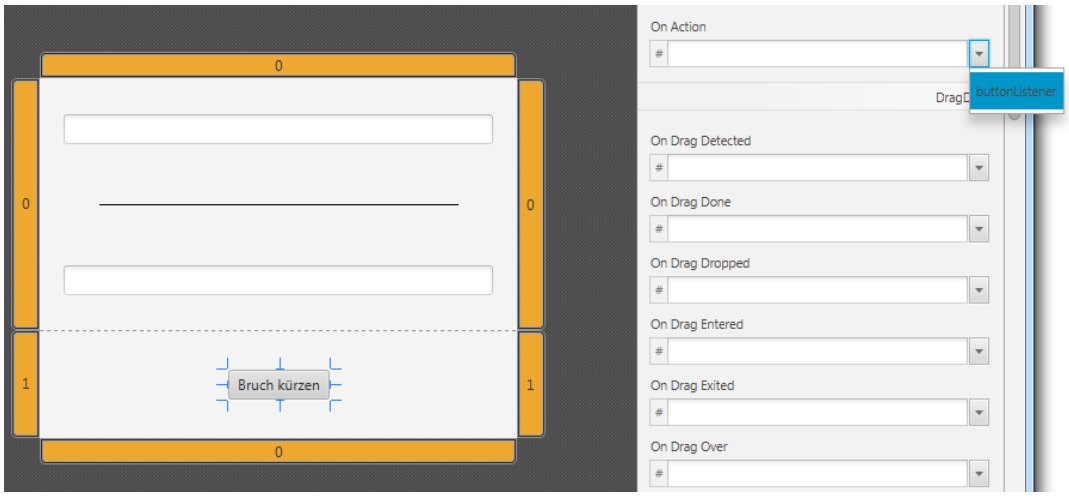
---

1 Annotationen werden später zusammen mit den Schnittstellen behandelt.

2 Die dazu sinnvollerweise zu verwendende Technik der Ausnahmebehandlung steht uns noch nicht zur Verfügung.



Um eine Ereignisbehandlungsmethode zu registrieren, markiert man die emittierende Komponente im **Hierarchy**-Segment oder in der Inhaltszone und wählt in der **Code**-Zone des **Inspectors** aus dem mit **On Action** betitelten Drop Down - Menü die gewünschte Methode der Controller-Klasse, z.B.:



Durch das Sichern der FXML-Datei per Scene Builder wird der neue Stand in Eclipse bekannt.

4.8.5 Hauptklasse der JavaFX-Anwendung korrigieren

Die Hauptklasse einer JavaFX-Anwendung hat bei der von uns gewählten GUI-Deklaration per FXML-Datei folgende Aufgaben:

- Die FXML-Datei laden
- Den Szenengraphen mit der Hierarchie der GUI-Elemente erstellen
- Die Bedienoberfläche anzeigen

Die vom Assistenten erstellte Klasse `Main` erfüllt diese Aufgaben unvollständig, so dass etwas Handarbeit erforderlich ist. Wir ersetzen die Anweisungen im **try**-Block der **start()** - Methode durch:[1]

```
Parent parent = FXMLLoader.load(getClass().getResource("Reduction.fxml"));
Scene scene = new Scene(parent);
primaryStage.setScene(scene);
primaryStage.setResizable(false);
primaryStage.setTitle("Brüche kürzen");
primaryStage.show();
```

[1] Mit **try**-Blöcken werden wir uns im Zusammenhang mit der Ausnahmebehandlung beschäftigen.

Über die unbedingt erforderlichen Anweisungen hinausgehend sorgen wir dafür, dass das Anwendungsfenster einen Titel und eine unveränderliche Größe erhält.

Außerdem importieren wir die verwendeten Klassen **Parent** und **FXMLLoader** mit der tatkräftigen Hilfe von Eclipse:

- Einfügemarke auf einen unterschlängelten Klassennamen positionieren
- den passenden Reparaturvorschlag wählen

Nun sollte das Programm einsatzbereit sein.

Das komplette Eclipse-Projekt `Brüche kuerzen` ist im folgenden Ordner zu finden

...\BspUeb\JavaFX\Brüche kuerzen

4.9 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 benötigt werden, erlaubt Java aber auch die Definition innerhalb einer umgebenden Klasse und sogar innerhalb einer Methode. So bleiben die zusammengehörigen Klassendefinitionen übersichtlich und wartungsfreundlich an einem Ort konzentriert. Außerdem lassen sich mehr Implementierungsdetails verstecken (erweitertes Information Hiding).

Allerdings sind die resultierenden Konstruktionen etwas kompliziert. Programmiereinsteiger sollten sich daher zunächst auf eine oberflächliche Lektüre von Abschnitt 4.9 beschränken und sich erst dann für Details interessieren, wenn diese später relevant werden.

Bei der Begriffsverwendung orientiert sich das Manuskript am Java-Tutorial (Oracle 2015).[1]

4.9.1 Mitgliedsklassen

Eine Mitgliedsklasse befindet sich im Quellcode einer umgebenden Klassendefinition, aber nicht in einer Methodendefinition, z.B.:

```
class Top {
  . . .
 class MemberClass {
   . . .
 }
  . . .
}
```

Man bezeichnet sie auch als *eingeschachtelte Klasse* (engl. *nested class*).

[1] Siehe http://docs.oracle.com/javase/tutorial/java/javaOO/whentouse.html

Einige Eigenschaften von Mitgliedsklassen:

- 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 4.9.1.1).
- Mitgliedsklassen dürfen geschachtelt werden.
- Der Compiler erzeugt auch für jede Mitgliedsklasse eine eigene **class**-Datei, in deren Namen die Bezeichner für die umgebende und die eingeschachtelte Klasse eingehen, z.B. **Top$Mitglied.class**.

4.9.1.1 Innere Klassen

Das folgende Programm enthält die äußere Klasse `Mantel` und darin die innere Klasse `Tasche`:

Quellcode	Ausgabe
`class Mantel {` `  private int anz = 3;` `  void tuWas() {` `   System.out.print("Mantel");` `  }` `  class Tasche {` `   void tuWas() {` `    System.out.print(anz+" Knöpfe in der ");` `    Mantel.this.tuWas();` `    System.out.println("tasche");` `   }` `  }` `}` `class InnerClassDemo {` `  public static void main(String[] args) {` `   Mantel mantel = new Mantel();` `   Mantel.Tasche tasche = mantel.new Tasche();` `   tasche.tuWas();` `  }` `}`	`3 Knöpfe in der Manteltasche`

Einige Eigenschaften von inneren Klassen (also von Mitgliedsklassen ohne Modifikator **static**):

- In einer inneren Klasse sind keine statischen Methoden erlaubt. Statische Variablen müssen finalisiert sein.
- Bei der Erstellung eines Objekts der inneren Klasse muss ein Objekt der äußeren Klasse als „Hülle“ beteiligt sein, indem ...

  - das äußere Objekt eine Instanzmethode ausführt und dort das innere Objekt kreiert,
  - in einem Konstruktor der umgebenden Klasse ein Objekt der inneren Klasse erstellt wird,
  - bei der Kreation des inneren Objekts explizit das äußere Objekt als Umgebung benannt wird, indem das Schlüsselwort **new** durch einen Punkt getrennt auf den Namen des äußeren Objekts folgt, z.B.:

```
Mantel.Tasche tasche = mantel.new Tasche();
```

- 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 explizit ansprechen (siehe obigen Quellcode der Klasse `Tasche`).
- Methoden und Konstruktoren der umgebenden Klasse können auf alle Felder, Methoden und Konstruktoren der inneren Klasse zugreifen (auch auf die privaten).

Das Beispiel sollte die Vorteile von inneren Klassen demonstriert haben:

- Die innere Klasse genießt Zugriffprivilegien (sozusagen als befreundete Klasse), während gegenüber anderen Klassen Datenkapselung bestehen kann.
- Wenn die innere Klasse ausschließlich im Kontext der äußeren Klasse benötigt wird, ist es vorteilhaft, den Quellcode beider Klassen zusammen zu halten.

4.9.1.2 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, muss aber einen Doppelnamen führen, z.B.:

Quellcode	Ausgabe
`class Mantel {` `  private static int kcal = 3000;` `  static class Motte {` `  void fressen() {` `  System.out.println("Mantel mit einem Nährwert\nvon "` `  + Mantel.kcal + " kcal verspeist.");` `  }` `  }` `}` `class StaticMemberClass {` `  public static void main(String[] args) {` `  Mantel.Motte motte = new Mantel.Motte();` `  motte.fressen();` `  }` `}`	`Mantel 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.

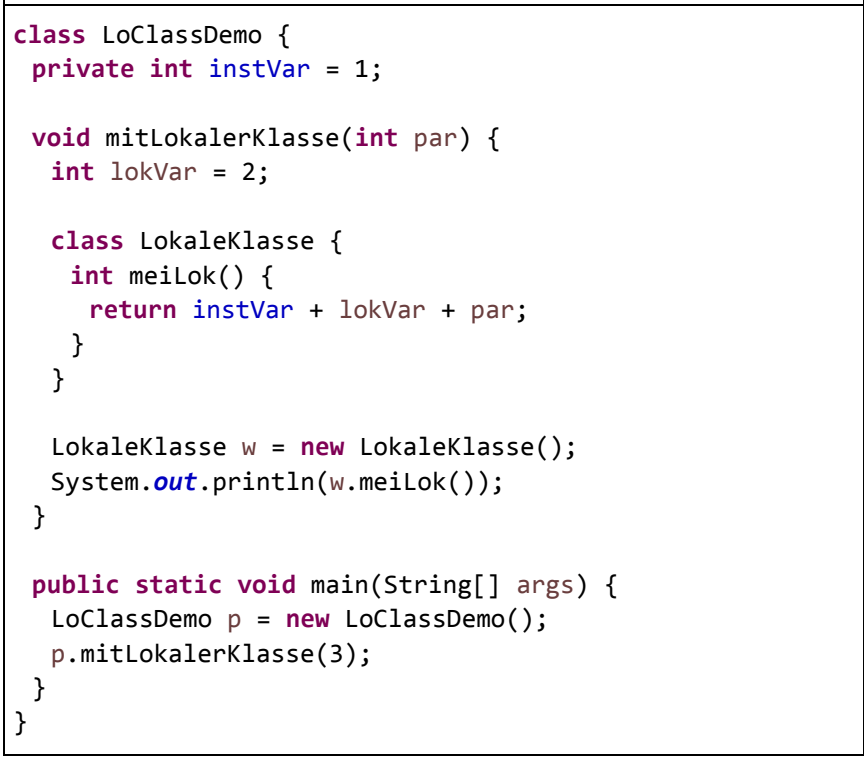
Die statische Mitgliedsklasse kann auf statische Mitglieder der äußeren Klasse zugreifen (auch auf die privaten). Wenn Bezeichner für statische Member der äußeren Klasse verdeckt worden sind, muss der Klassenname vorangestellt werden.

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 kommt dann in Frage, wenn zwei Klassen in einer engen Beziehung stehen, und der Quellcode beider Klassen zusammen gehalten werden soll.

4.9.2 Lokale Klassen

In einer Methode dürfen lokale Klassen definiert werden, z.B.:

Quellcode	Ausgabe
	6

```
class LoClassDemo {
 private int instVar = 1;

 void mitLokalerKlasse(int par) {
  int lokVar = 2;

  class LokaleKlasse {
   int meiLok() {
    return instVar + lokVar + par;
   }
  }

  LokaleKlasse w = new LokaleKlasse();
  System.out.println(w.meiLok());
 }

 public static void main(String[] args) {
  LoClassDemo p = new LoClassDemo();
  p.mitLokalerKlasse(3);
 }
}
```

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 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 finale Variablen bzw. Parameter beschränkt ist, sind natürlich nur *lesende* Zugriffe erlaubt.
- 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.

- Felder und lokale Variablen der lokalen Klasse 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 Klassenvariablen `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, so dass im Beispiel die folgende Datei entsteht: **LoClassDemo$1LokaleKlasse.class**.

Weitere Informationen über lokale Klassen bietet das Java-Tutorial (Oracle 2015).[1]

Später werden wir mit den *anonymen Klassen* noch eine wichtige Variante der lokalen Klassen kennen lernen.

4.10 Übungsaufgaben zu Kapitel 4

1) Welche von den 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 von den 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.

[1] Siehe: http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html

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) Entlarven Sie bitte die falschen Behauptungen:

1. Der Standardkonstruktor einer Klasse hat die Schutzstufe **public**.
2. Die Entsorgung überflüssig gewordener Objekte wird vom Garbage Collector der JVM automatisch erledigt.
3. Die in einer Methode erstellten Objekte sind nach Verlassen der Methode ungültig und werden (früher oder später) vom Garbage Collector aus dem Speicher entfernt.
4. Auf eine statische Methode können berechtigte Klassen nur „auf statischem Weg“ zugreifen, indem sie dem Methodennamen beim Aufruf einen Vorspann aus Klassennamen und Punktoperator voranstellen. In Methoden derselben Klasse darf der Klassenname entfallen.

7) Erstellen Sie die Klassen `Time` und `Duration` zur Verwaltung von Zeitpunkten (der Einfachheit halber nur innerhalb eines Tages) und Zeitintervallen (von beliebiger Länge).

Beide Klassen 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
  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) eine formatierte Ausgabe erlaubt. 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:

```
String.format("%02d:%02d:%02d %s", hours, minutes, seconds, "Uhr");
```

In der `Time`-Klasse sollen außerdem Methoden mit folgenden Leistungen vorhanden sein:

- Berechnung der Zeitdistanz zu einem anderen, als Parameter übergebenen Zeitpunkt am selben oder am folgenden Tag, z.B. mit dem Namen `getDistenceTo()`
- Addieren eines Zeitintervalls zu einem Zeitpunkt, z.B. mit dem Namen `addDuration()`

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.
```

8) 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();
     }
}
```

1 2 3 4 5 6 7 8 9

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. Die Idee zu dieser Übungsaufgabe stammt aus Mössenböck (2003).

9) 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.

10) 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 $\tilde{x}_1 := \dfrac{x_1}{\sqrt{x_1^2 + x_2^2}}$ und $\tilde{x}_2 := \dfrac{x_2}{\sqrt{x_1^2 + x_2^2}}$ 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:

  $$x + y := \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

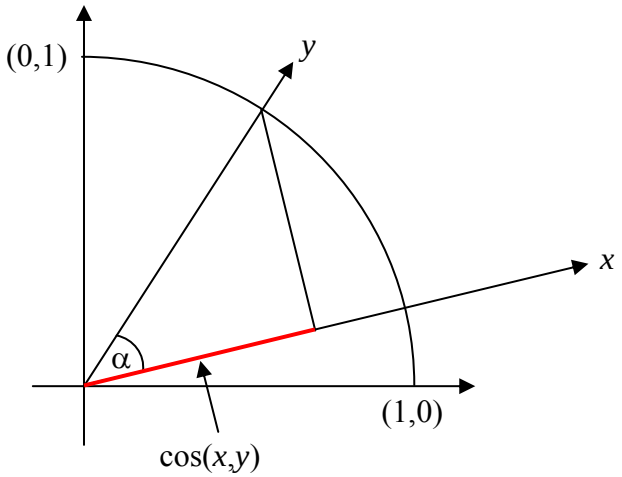
- **Skalarprodukt** zweier Vektoren ermitteln

  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 α 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:

---

[1] Dies folgt aus dem Additionstheorem für den Kosinus.

$$rad = \frac{deg}{360} \cdot 2\pi$$

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)
```

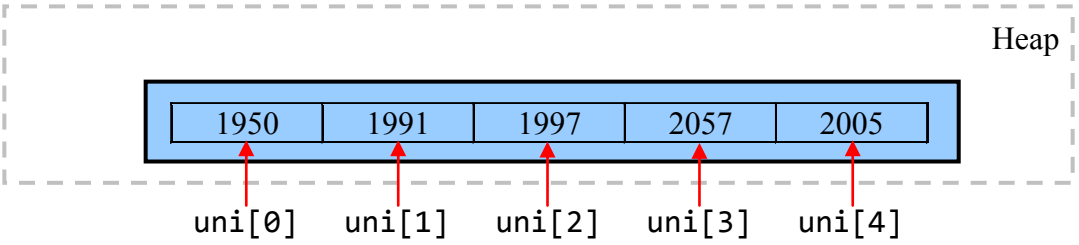
5 Elementare Klassen

In diesem Abschnitt wird gewissermaßen die objektorientierte Fortsetzung der elementaren Sprachelemente aus Kapitel 1 präsentiert. Es werden wichtige Bausteine für Programme behandelt, die in Java als *Klassen* realisiert sind (z.B. Arrays, Zeichenketten), und einige spezielle Datentypen vorgestellt. 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] Hier ist ein Array namens `uni` mit 5 Elementen vom Typ **int** zu sehen:

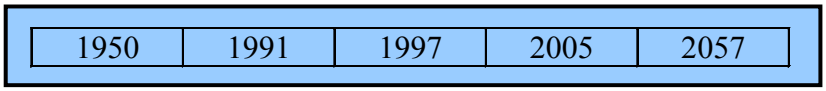


Beim Zugriff auf ein einzelnes Element gibt man nach dem Arraynamen den durch eckige Klammern begrenzten Index an, wobei die Nummerierung bei 0 beginnt und bei *n* Elementen folglich mit *n* - 1 endet.[2]

Man kann aber auch den kompletten Array ansprechen und z.B. als Aktualparameter an eine Methode übergeben. In der folgenden Anweisung

```
Arrays.sort(uni);
```

werden die Elemente des Arrays `uni` durch eine statische Methode der Klasse **Arrays** sortiert, was zum folgenden Ergebnis führt:



Neben den Elementen enthält das 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 erhebliche Vereinfachung der Programmierung:

- 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 oft die *gemeinsame* Verarbeitung *aller* Elemente per Methodenaufruf mit Array-Aktualparameter veranlassen.

---

[1] Arrays werden in vielen Programmiersprachen auch *Felder* genannt. In Java bezeichnet man jedoch recht einheitlich die Instanz- oder Klassenvariablen als Felder, so dass der Name hier nicht mehr zur Verfügung steht.

[2] Technisch gesehen liegt ein Array-Zugriffsausdruck mit dem Operator [] vor.

- Viele Algorithmen arbeiten mit Vektoren und Matrizen. Zur Modellierung dieser mathematischen Objekte sind Arrays unverzichtbar.

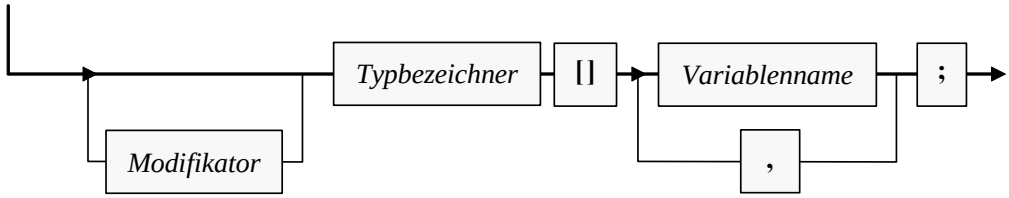
Wir beschäftigen uns erst *jetzt* mit den zur Grundausstattung praktisch jeder Programmiersprache gehörenden Arrays, weil diese Datentypen in Java als *Klassen* realisiert werden und folglich zunächst entsprechende Grundlagen zu erarbeiten waren.[1]

Wir befassen uns zunächst mit *eindimensionalen* Arrays, behandeln später aber auch den mehrdimensionalen Fall.

5.1.1 Array-Variablen deklarieren

Im Vergleich zu der bisher bekannten Variablendeklaration ist bei Array-Variablen hinter dem Typbezeichner zusätzlich ein Paar eckiger Klammern anzugeben:[2]

Deklaration einer Array-Variablen



Die Array-Variable `uni` aus dem einleitend beschriebenen Beispiel 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** 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 Größe auf dem Heap. In der folgenden Anweisung entsteht ein Array mit 5 **int**-Elementen, und seine Adresse landet in der Referenzvariablen `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 kann also die Länge eines Arrays zur Laufzeit festlegen, z.B. in Abhängigkeit von einer Benutzereingabe.

---

[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.

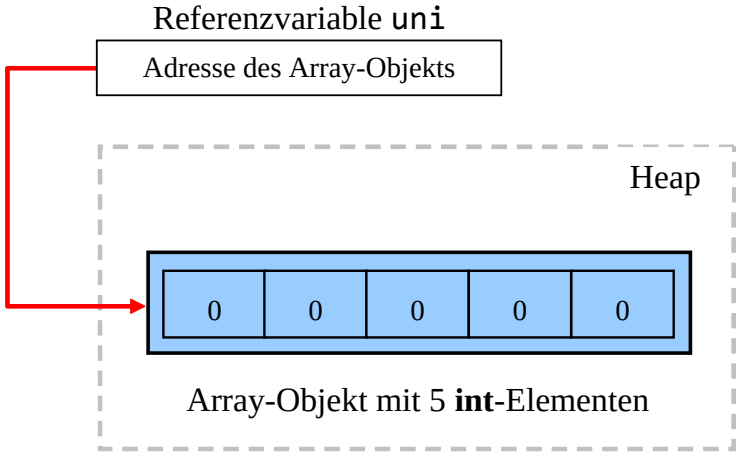
```
int uni[];
```

Hier wird eine Regel der älteren Programmiersprache C unterstützt, wobei die Lesbarkeit des Quellcodes aber leidet.

Die Deklaration einer Array-Referenzvariablen *und* die Erstellung des Array-Objekts kann man 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 Initialisierung nach den Regeln von Abschnitt 4.1.3. Die **int**-Elemente im Beispiel erhalten folglich den Startwert 0.

Als Objekt wird ein Array vom Garbage Collector entsorgt, wenn keine Referenz mehr vorliegt (vgl. Abschnitt 4.4.5). Um eine Referenzvariable aktiv von einem Array-Objekt zu „entkoppeln“, kann man ihr z.B. den Wert **null** (Zeiger auf nichts) 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 Arrays verwenden

Der Zugriff auf die Elemente 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, auf den die Referenzvariable `uni` gemäß obiger Deklaration und Initialisierung zeigt (siehe Abschnitt 5.1.2):

```
for (i = 0; i < drl; i++)
  uni[zzg.nextInt(5)]++;
```

Den Indexwert liefert die Zufallszahlenmethode **nextInt()** mit 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 **ArrayIndexOutOfBoundsException**, 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 unten), wird es vom Laufzeitsystem beendet. Man kann sich in Java generell darauf lassen, dass jede Überschreitung von Feldgrenzen verhindert wird, so dass 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.:

Quellcode	Eingabe (**fett**) und Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.print("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]);` `  }` `}`	`Länge des Vektors: 3` `Wert von Element 0: 7` `Wert von Element 1: 13` `Wert von Element 2: 4711` `7` `13` `4711`

5.1.4 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 Service-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*)

Hinzu gekommene Elemente werden mit dem typspezifischen Nullwert initialisiert.

Einige später vorzustellende Kollektionsklassen zur Verwaltung von Elementlisten gehen im Bedarfsfall analog vor, um die Kapazität zu erhöhen. Im Quellcode der API-Klasse **ArrayList**, die wir noch als „größendynamischen" Container mit Array-Innenleben kennen lernen werden, findet sich z.B. die folgende Anweisung

```
elementData = Arrays.copyOf(elementData, newCapacity);
```

in der privaten Methode **grow()**.

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.

5.1.5 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 von ihrem Startwert abhängt, kann sie in der Regel echte Zufallszahlen ersetzen. Manchmal ist es sogar von Vorteil, eine Serie über ihren *festen* Startwert reproduzieren zu können. In der Regel verwendet man aber *variable* Startwerte, z.B. abgeleitet aus einer Zeitangabe. Der Einfachheit halber redet man oft von *Zufallszahlen* und lässt den *Pseudo*-Zusatz weg.[1]

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 Startwert für den Pseudozufall.[2]

---

[1] Man kann übrigens mit moderner EDV-Technik unter Verwendung von physikalischen Prozessen auch *echte* Zufallszahlen produzieren, doch ist der Zeitaufwand im Vergleich zu Pseudozufallszahlen erheblich höher (siehe z.B. Lau 2009).

[2] 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());
```

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:[1]

```
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 guter Näherung ein:

```
Absolute Haeufigkeiten:
1950 1991 1997 2057 2005

Relative Haeufigkeiten:
0.195 0.1991 0.1997 0.2057 0.2005
```

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[a]
df	4
Asymptotische Signifikanz	**,569**

a. Bei 0 Zellen (,0%) werden weniger als 5 Häufigkeiten erwartet. Die kleinste erwartete Zellenhäufigkeit ist 2000,0.

Über die im Beispielprogramm verwendete Klasse **Random** aus dem Paket **java.util** können Sie sich z.B. mit Hilfe der API-Dokumentation informieren.

[1] In der Sprache der Wahrscheinlichkeitstheorie erfolgt die Ziehung „mit Zurücklegen“.

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)]++;
```

5.1.6 Initialisierungslisten

Bei Arrays mit wenigen Elementen ist die Möglichkeit von Interesse, beim Deklarieren der Referenzvariablen eine Initialisierungsliste mit Werten für die Elementvariablen anzugeben und das Array-Objekt dabei implizit (ohne Verwendung des **new**-Operators) zu erzeugen, z.B.:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  int[] wecktor = {1, 2, 3};` `  System.out.println(wecktor[2]);` `  }` `}`	`3`

Die Deklarations- und Initialisierungsanweisung

```
int[] wecktor = {1, 2, 3};
```

ist äquivalent zu:

```
int[] wecktor = new int[3];
wecktor[0] = 1;
wecktor[1] = 2;
wecktor[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};
```

5.1.7 Objekte als Array-Elemente

Für die Elemente eines Arrays sind natürlich auch Referenztypen erlaubt. In folgendem Beispiel wird ein Array mit `Bruch`-Objekten erzeugt:

Quellcode	Ausgabe
`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();` `  }` `}`	`  5` `b2 =  -----` `  6`

Im nächsten Abschnitt lernen wir einen wichtigen Spezialfall von Arrays mit Referenztyp-Elementen kennen. Dort zeigen die Elementvariablen wiederum auf Arrays, so dass mehrdimensionale Arrays entstehen.

5.1.8 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	Ausgabe
`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();` `  }` `  }` `}`	`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.

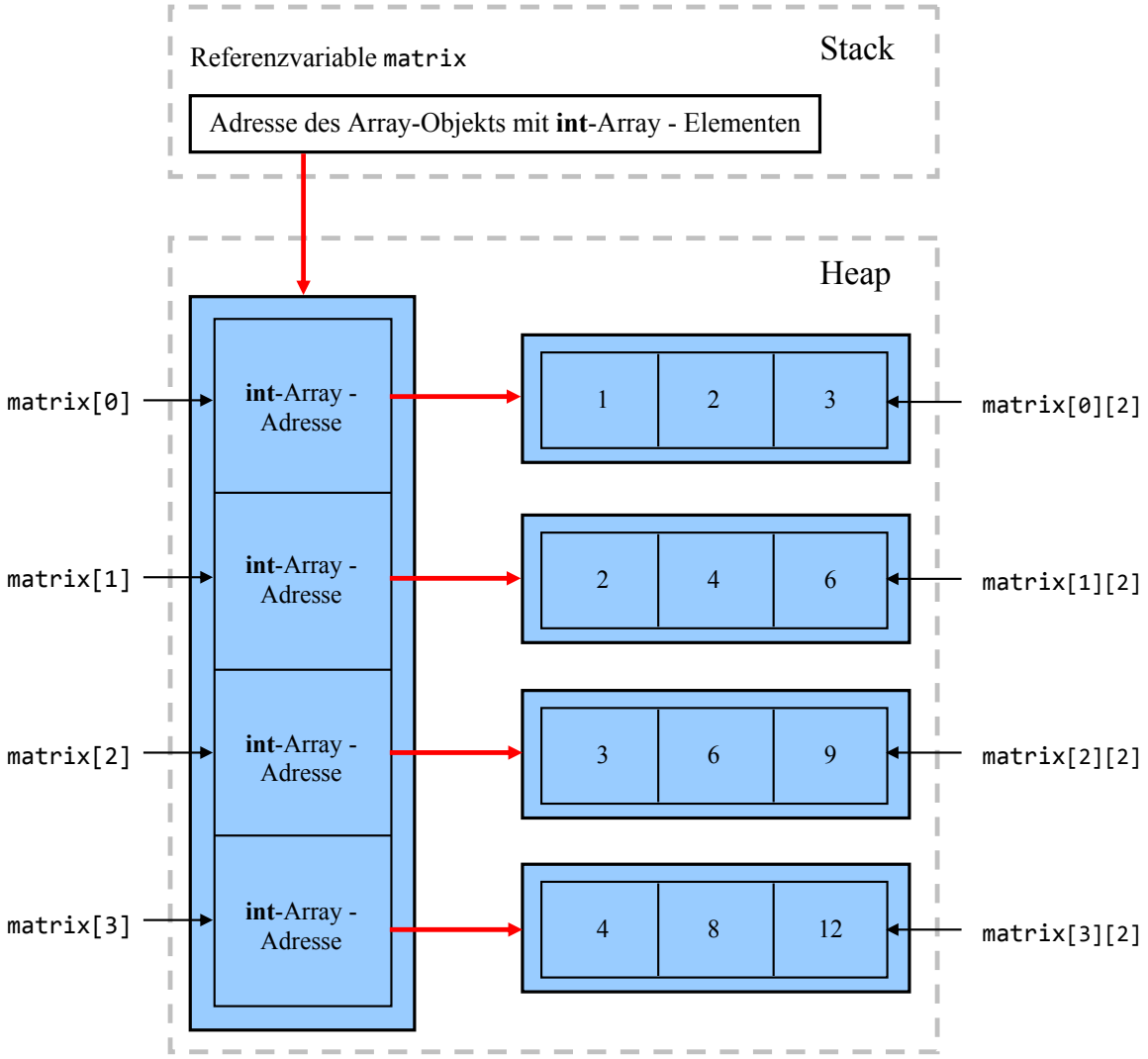
Im Beispiel wird ein Array-Objekt namens `matrix` mit den vier Elementen `matrix[0]` bis `matrix[3]` erzeugt, bei denen es sich jeweils um eine Referenz auf einen Array mit drei **int**-Elementen handelt. Wir haben damit eine zweidimensionale Matrix zur Verfügung, auf deren Zellen man per Doppelindizierung zugreifen kann, wobei sich die Syntax leicht von der mathematischen Notation unterscheidet, z.B.:

```
matrix[i][j] = (i+1)*(j+1);
```

Man kann aber auch mit einfacher Indizierung eine komplette Zeile ansprechen, was in obigem Programm geschieht, um die Länge der eindimensionalen Zeilen-Arrays zu ermitteln:

```
matrix[i].length
```

In der folgenden Abbildung wird die Situation im Hauptspeicher beschrieben:



Im nächsten Beispielprogramm wird die Möglichkeit demonstriert, mehrdimensionale Arrays mit unterschiedlich langen Elementen anzulegen, so dass z.B. eine ausgesägte (engl. *jagged*) Matrix entsteht:

Quellcode	Ausgabe
`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();` `  }` `  }` `}`	`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`

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.:

```
int[][] matrix = {{1}, {1,2}, {1, 2, 3}};
```

5.2 Klassen für Zeichenfolgen

Java bietet für den Umgang mit Zeichenfolgen mehrere Klassen an:

- **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 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.

5.2.1.1 Erzeugen von String-Objekten

In der folgenden Deklarations- und Initialisierungsanweisung

```
String s1 = "abcde";
```

wird:

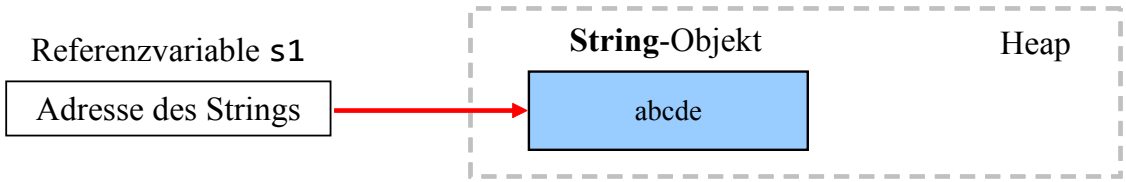
- eine **String**-Referenzvariable namens `s1` angelegt,
- ein neues **String**-Objekt auf dem Heap erzeugt,
- die Adresse des Heap-Objekts in der Referenzvariablen abgelegt

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, so dass z.B.

```
"abcde"
```

einen Ausdruck darstellt, der als Wert einen Verweis auf ein **String**-Objekt auf dem Heap liefert.

Die 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 gleich 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 Überschrift zu diesem Abschnitt wird für diesen Sachverhalt eine Abkürzung aus der Elektronik ausgeliehen: WORM (**W**rite **O**nce **R**ead **M**any). Eventuell werden Sie die Starrheit des **String**-Inhalts 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 Text „abc“), sondern erzeugt ein neues **String**-Objekt (mit dem Text „abcdef“) und schreibt dessen Adresse in die Referenzvariable `testr`.

5.2.1.3 Interner String-Pool und Identitätsvergleich

Erfolgt die Initialisierung einer **String**-Referenzvariablen über ein Literal oder einen anderen *konstanten* Ausdruck, so dass schon der Compiler die resultierende Zeichenfolge kennt, dann kommt der so genannte **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. 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.

Wie im folgenden Beispiel

```
String s4 = new String("abcde");
```

ein **String**-Literal 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 ein neues **String**-Objekt, 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).

Beim Vergleich von **String**-Variablen *per Identitätsoperator* haben obige Ausführungen wichtige Konsequenzen, wie das folgende Programm zeigt:

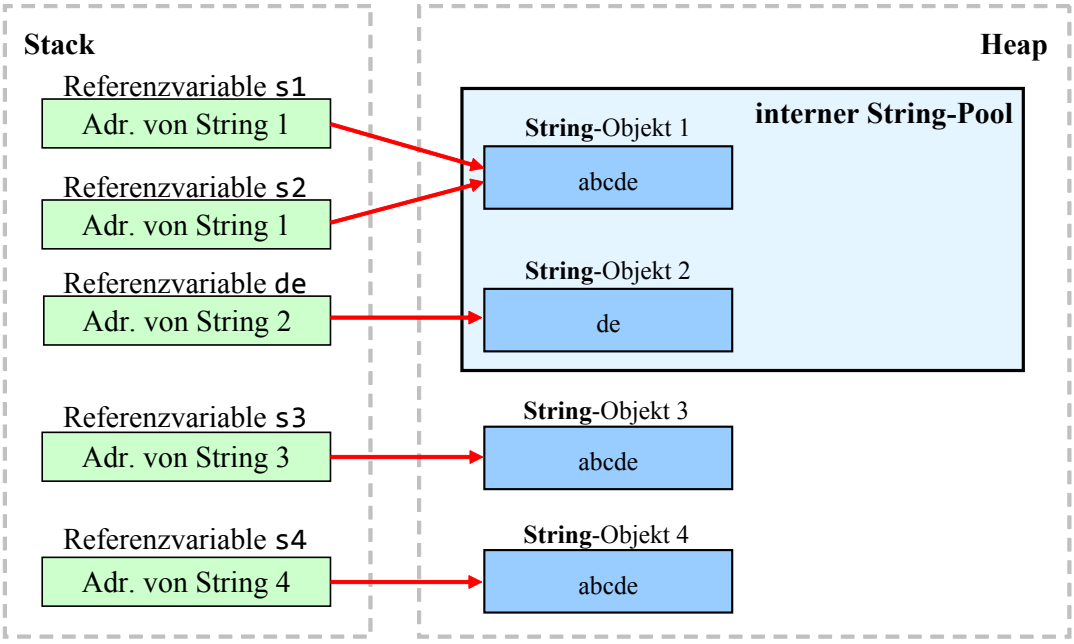
Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  String s1 = "abcde";` `  String s2 = "abc"+"de";` `  String de = "de";` `  String s3 = "abc"+de;` `  String s4 = new String("abcde");` `  System.out.print("(s1 == s2) = "+(s1==s2)+"\n"+` `  "(s1 == s3) = "+(s1==s3)+"\n"+` `  "(s1 == s4) = "+(s1==s4));` `  }` `}`	`(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 Pool unterstützt, so dass anschließend Referenz- und Inhaltsvergleich äquivalent sind (siehe Abschnitt 5.2.1.5). Die Initialisierung durch einen konstanten Ausdruck ist also *nicht* der einzige Anlass für die Aufnahme eines **String**-Objekts in den Pool.

5.2.1.4 Methoden für String-Objekte

Von den ca. 70 Methoden der Klasse der **String** werden in diesem Abschnitt nur die wichtigsten angesprochen. Für spezielle Anwendungen lohnt sich also ein Blick in die Dokumentation zum Java-API.

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. In folgendem Beispiel wird mit Klammern dafür gesorgt, dass der Compiler die „+“ - Operatoren jeweils sinnvoll interpretiert (Verketten von Strings bzw. Addieren von Zahlen):

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.println("4 + 3 = " + (4 + 3));` `  }` `}`	`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 in Abschnitt 5.2.1.3 erläuterten Tücken beim Vergleich von **String**-Referenzvariablen per Identitätsoperator aus dem Weg zu gehen. In folgendem Programm werden zwei **String**-Objekte zunächst nach ihren Speicheradressen verglichen, dann nach dem Inhalt:

Quellcode	Ausgabe
`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));` `  }` `}`	`false` `true`

5.2.1.4.3 Lexikographische Priorität

Zum Vergleich von Zeichenfolgen hinsichtlich der lexikographischen Ordnung (Sortierreihenfolge) kann die **String**-Methode

public int compareTo(String *vergl***)**

dienen, z.B.:

Quellcode	Ausgabe
`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));` `  }` `}`	`< : -10` `= : 0` `> : 10`

Die Methode **compareTo()** liefert folgende **int**-Rückgabewerte:

		compareTo()-Rückgabe
Die lexikographische Priorität des angesprochenen **String**-Objekts ist im Vergleich zum Parameterobjekt:	kleiner	negative Zahl
	gleich	0
	größer	positive Zahl

5.2.1.4.4 Länge einer Zeichenkette

Während bei Array-Objekten die Anzahl der Elemente in der Instanz*variablen* **length** zu finden ist (vgl. Abschnitt 5.1), wird die aktuelle Länge einer Zeichenkette über die Instanz*methode* **length()** ermittelt:

Quellcode	Ausgabe
`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());` `  }` `}`	3 3

5.2.1.4.5 Zeichen(folgen) extrahieren, suchen oder ersetzen

Im folgenden Programm werden einige anschließend beschriebene **String**-Methoden verwendet:

Quellcode	Ausgabe
`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));` `  }` `}`	rg 2 -1 false B

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 wiederum 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 neuer **char**-Array mit identischem Inhalt erzeugt werden, z.B.:

Quellcode	Ausgabe
`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]);` `  }` `}`	a b c

f) Zeichen oder Teilzeichenfolgen ersetzen

Mit der Methode

public String replace(char *oldChar*, **char** *newChar*)

erhält man einen neuen **String**, der aus dem angesprochenen Original durch Ersetzen eines alten Zeichens durch ein neues Zeichen hervorgeht, 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

Mit 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
`class Prog {` `  public static void main(String[] args) {` `  String a = "Otto", b = "otto";` `  System.out.println(a.toUpperCase().equals(b.toUpperCase()));` `  }` `}`	`true`

In der Anweisung mit dem **equals()** - Aufruf stoßen wir auf eine stattliche Anzahl von Punktoperatoren, so dass 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 in 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 in 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 so genannte *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 Collection Framework, siehe unten). 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 Collection 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 Rechenzeitaufwand
- reduziert aber den Speicheraufwand *und* beschleunigt vor allem 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]

[1] Die Ergebnisse stammen von einem PC mit der Intel-CPU Core i3 (3,2 GHz) unter Windows 7 (64 Bit).

	Laufzeit in Millisekunden	
	equals() - Vergleiche	Intern. plus Adress-Vergl.
`anz = 50000, len = 20, wdh = 5`	16	47
`anz = 50000, len = 20, wdh = 50`	125	78
`anz = 50000, len = 20, wdh = 500`	1485	329

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 zeitaufwendige 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 spezielle syntaktische Unterstützung durch den Compiler, z.B. durch den überladenen „+“ - Operator.

Der einzige Unterschied zwischen den Klassen **StringBuilder** und **StringBuffer** besteht darin, dass die Klasse **StringBuffer** Thread-sicher ist, so dass ein Objekt dieser Klasse gefahrlos von mehreren Threads (Ausführungsfäden, siehe unten) eines Programms genutzt werden kann. Diese Thread-Sicherheit ist aber mit Aufwand verbunden, so dass die Klasse **StringBuilder** zu bevorzugen ist, wenn eine variable Zeichenfolge nur von *einem* Thread genutzt wird. 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");`

In folgendem Programm wird eine Zeichenfolge 20.000-mal verlängert, zunächst mit Hilfe der Klasse **String**, dann mit Hilfe der Klasse **StringBuilder**:

```
class Prog {
 public static void main(String[] args) {
  final int drl = 20_000;
  String s = "*";
  long vorher = System.currentTimeMillis();
  for (int i = 0; i < drl; i++)
   s = s + "*";
  long diff = System.currentTimeMillis() - vorher;
  System.out.println("Zeit fuer die String-\"Verlaengerung\": \t\t"+diff);

  StringBuilder t = new StringBuilder("*");
  vorher = System.currentTimeMillis();
  for (int i = 0; i < drl; i++)
   t.append("*");
  s = t.toString();
  diff = System.currentTimeMillis() - vorher;
  System.out.println("Zeit fuer die StringBuilder-Verlaengerung: \t"+diff);
 }
}
```

Die Laufzeiten (gemessen in Millisekunden auf einem PC mit Intel-CPU Core i3 mit 3,2 GHz) unterscheiden sich erheblich:

```
Zeit fuer die String-"Verlaengerung":       224
Zeit fuer die StringBuilder-Verlaengerung:  1
```

Ein **StringBuilder**-Objekt beherrscht u.a. die folgenden **public**-Methoden:

Methode	Erläuterung
int length()	Diese Methode liefert die 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, in folgendem Beispiel also gerade 2 Zeichen, falls der **StringBuilder** mindestens 4 Zeichen enthält: `sb.delete(1, 3);`
replace()	Ein Bereich des **StringBuilder**-Objekts wird durch den Argument-**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())`

5.3 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 nur für Objekte verfügbar ist. Außerdem 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**:

- **public Integer**(**int** *value*)
  Beispiel: `Integer iw = new Integer(4711);`
- **public Integer**(**String** *str*)
  Beispiel: `Integer iw = new Integer(args[0]);`

Ferner ist die statische Fabrikmethode **valueOf()** in zwei analogen Überladungen vorhanden, z.B. bei der Klasse **Integer**:

- **public static Integer valueOf**(**int** *value*)
- **public static Integer valueOf**(**String** *str*)

Die **valueOf()** - Methoden verwenden Zeit und Speicherplatz sparend einen Cache mit bereits erzeugten Wrapper-Objekten vom eigenen Typ:

- Ist beim Aufruf bereits ein Wrapper-Objekt mit dem angeforderten Wert vorhanden, wird dessen Adresse zurückgeliefert.
- Anderenfalls wird ein neues Objekt erstellt und dessen Adresse geliefert.

Wenn nicht unbedingt ein neues Objekt benötigt wird, sollte an Stelle eines Konstruktors die Methode **valueOf()** verwendet werden.

Die Wrapper-Klassen im Java-API sind **unveränderlich** (engl.: *immutable*), so dass der Inhalt eines Objekts nach dem Erzeugen nicht mehr geändert werden kann. Daher besitzen die Wrapper-Klassen keinen parameterfreien Konstruktor.

Im nächsten Abschnitt geht es um das Erstellen von Wrapper-Objekten über einen Compiler-Automatismus.

5.3.2 Autoboxing

Seit der Version 5 kann der Java-Compiler Werte eines primitiven Typs automatisch in 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");
  al.add(13);
  al.add(23.77);
  al.add('x');
  System.out.println("Der ArrayList-Container enthält:");
  for(Object o : al)
   System.out.println(" " + o + "\t Typ: " + o.getClass().getName());
 }
}
```

Wie die folgende Programmausgabe zeigt, sind tatsächlich diverse Wrapper-Klassen im Spiel:

```
Der ArrayList-Container enthaelt:
 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 wird der eingepackte Wert entnommen und einer **int**-Variablen zugewiesen.

Dank Autoboxing sind die primitiven Typen zuweisungskompatibel zur Klasse **Object**, wobei zum Auspacken aber eine explizite Typumwandlung erforderlich ist, z.B.:

---

[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 so genannten *Rohtyp* der Klasse **ArrayList**, der sich aus didaktischen Gründen gut für den aktuellen Abschnitt eignet, ansonsten aber zu vermeiden ist.

```
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. 2015, 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 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 **Double**-Klassenmethode

  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 identischen Objekten (vgl. Abschnitt 5.3.1).

Wenn eine Konvertierung mit **parseDouble()** oder **valueOf()** scheitert, dann informieren die Methoden ihren Aufrufer durch das Werfen einer Ausnahme vom Typ **NumberFormatException**. Die später noch ausführlich zu behandelnde Ausnahmetechnik ist gleich in einem Beispiel zu sehen. Bisher blieb im Manuskript bei der Beschreibung einer Methode, die potentiell Ausnahmeobjekte wirft, diese Kommunikationstechnik aus didaktischen Gründen unerwähnt.

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 Werten 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);
```

5.3.4 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	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.println("Max. double-Zahl:\n"+` `  Double.MAX_VALUE);` `  }` `}`	`Max. double-Zahl:` `1.7976931348623157E308`

5.3.5 Character-Methoden zur Zeichen-Klassifikation

Die Wrapper-Klasse **Character** zum primitiven Typ **char** bietet einige 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 u.a.: • 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.

Java bietet mit den **Enumerationen** (**Aufzählungstypen)** eine Lösung, die folgende Vorteile bietet:

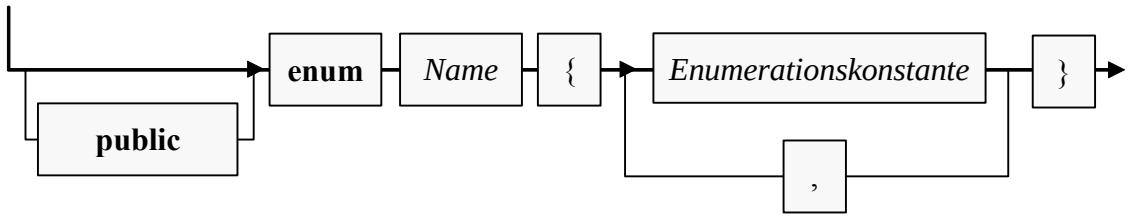
- Eine exakt definierte Menge gültiger Werte
  Damit ist die Einhaltung des Wertebereichs nicht mehr von der Sorgfalt des Klassendesigners abhängig, sondern wird vom Compiler sichergestellt.
- Gut lesbarer Quellcode
  Im obigen Beispiel kann z.B. der folgende logische Ausdruck verwendet werden:

```
if (otto.temp == Temperament.PHLEGMATISCH) ...
```

5.4.1 Einfache Enumerationstypen

Ein einfacher Aufzählungstyp besteht aus einer Anzahl von Konstanten. Bei seiner Definition folgt nach dem optionalen **public**-Modifikator (Sichtbarkeit des Typs in beliebigen Paketen) auf das Schlüsselwort **enum** und den Typbezeichner eine geschweift eingeklammerte Liste mit den Namen der Enumerationskonstanten:

Einfache Enumerationsdefinition



Weil Syntaxdiagramme zwar präzise, aber nicht unbedingt mit einem 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 (wie die Namen von finalisierten statischen Variablen).

Wie für eine Klassendefinition sollte auch für eine Enumerationsdefinition eine eigene Quellcodedatei verwendet werden. Bei einer Enumeration mit der Zugriffsstufe **public** ist dies obligatorisch.

Objekte der folgenden Klasse `Person` (der Einfachheit halber ohne Datenschutz) 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 stets mit dem Typnamen qualifiziert werden müssen, ist einige Tipparbeit erforderlich, die aber mit einem 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 entstehen. Nun ist klar, warum den Enumerationskonstanten der Typname vorangestellt werden muss.
- Es ist nicht möglich, weitere Objekte der Enumerationsklasse (per **new**-Operator oder auf andere Weise) zu erzeugen.
- Man kann eine Enumeration nicht beerben.
  In Abschnitt 7.1 werden wir solche Klassen als *finalisiert* bezeichnen.

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.:

[1] Im Kapitel über Pakete werden wir eine Möglichkeit kennen lernen, häufige Wiederholungen des Aufzählungstypnamens im Quellcode zu vermeiden: Mit der Direktive **import static** kann man alle statischen Variablen und Methoden eines Typs importieren, so dass 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.

Quellcode	Ausgabe
`class PersonTest {` `  public static void main(String[] args) {` `  Person otto = new Person("Otto", "Hummer",` `  35, Temperament.SANGUIN);` `  System.out.println(otto.temp.ordinal());` `  }` `}`	`3`

Bei jeder Enumerationsklasse kann man mit der statischen Methode **values()** einen Array mit ihren Objekten anfordern, z.B.:

Quellcode	Ausgabe
`class PersonTest {` `  public static void main(String[] args) {` `  for (Temperament t : Temperament.values())` `  System.out.println(t.name());` `  }` `}`	`MELANCHOLISCH` `CHOLERISCH` `PHLEGMATISCH` `SANGUIN`

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.

5.5 Übungsaufgaben zu Kapitel 5

Abschnitt 5.1 (Arrays)

1) Welche von den 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. In der **for**-Schleife für Kollektionen (siehe Abschnitt 3.7.3.2) sind auch Arrays als Kollektionsobjekte erlaubt.
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** (Selection Sort) benutzen:

- 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, 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.

[1] Der griechische Gelehrte Eratosthenes lebte laut Wikipedia ca. von 275 bis 194 v. Chr.

- 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·*b*, 3·*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, kann das Streichverfahren enden, ohne die Vielfachen der neuen Basis auch noch streichen zu müssen.

In der Kandidatenmenge befinden sich dann nur noch Primzahlen. Um dies einzusehen, nehmen wir an, es gäbe noch eine Zahl $n \le K$ mit echtem Teiler. Mit zwei positiven Zahlen *u*, *v* würde dann gelten:

$$n = u \cdot v \text{ und } u < b \text{ oder } v < b \text{ (wegen } b > \sqrt{K} \text{ und } n \le K)$$

Wir nehmen ohne Beschränkung der Allgemeinheit $u < b$ 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.

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

4) Definieren Sie eine Klasse für eine zweidimensionale Matrix mit Elementen vom Typ **double** zur Aufnahme von Beobachtungswerten aus einer empirischen Studie. Machen Sie die vereinfachende Annahme, dass jeder vorhandene Zeilenvektor vollständig ist, d.h. in jeder Zelle einen regulären **double**-Wert enthält. Implementieren Sie …

- eine Methode zum Transponieren der Matrix
- Methoden für elementare statistische Analysen mit den Spalten der Matrix:
  - Eine Methode sollte den 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 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 von den folgenden Aussagen sind richtig bzw. falsch?

1. Mit Hilfe der **for**-Schleife für 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 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 Zeichensatznummern addieren und die Summe als Startwert 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	Ausgabe

```
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));
 }
}
```

```
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.

In folgendem 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. In folgendem Programm wird demonstriert, wie ein **StringTokenizer** arbeitet:

Quellcode	Ausgabe
`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());` `  }` `}`	`Dies` `ist` `der` `Satz,` `der` `zerlegt` `werden` `soll.`

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.

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

In der Standardbibliothek und auch in jeder größeren Einzelanwendung sind zahlreiche Klassen anzutreffen. Der Ordnung und Funktionalität halber werden die Klassen in *Pakete* (engl.: *packages*) einsortiert. Im Java-8-SE - API gibt es über 200 Pakete mit zusammengehörigen Klassen.

Neben den Klassen spielen in der objektorientierten Programmierung die so genannten *Interfaces* eine wichtige Rolle, und in den meisten Paketen sind daher sowohl Klassen als auch Interfaces zu finden. Ein Interface taugt wie eine Klasse als Datentyp und enthält ebenfalls Methoden, doch fehlt bei den Interface-Methoden die Implementation (der Anweisungsblock). Ein Interface 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. Wir werden uns in Kapitel 9 mit den Interfaces und ihrer Rolle bei der objektorientierten Programmierung beschäftigen. Im Manuskript ist ab jetzt von *Typen* die Rede, wenn sowohl Klassen als auch Interfaces einbezogen werden sollen.

Pakete erfüllen wichtige Aufgaben:

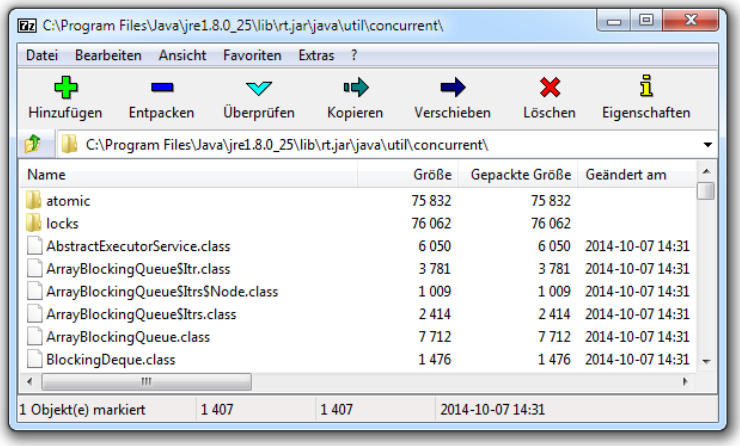
- **Große Projekte strukturieren**
  Wenn viele Typen vorhanden sind, sollte man diese nach funktionaler Verwandtschaft auf mehrere Pakete verteilen. Ist die Paketorganisation auf ein Dateisystem abgebildet, werden alle **class**-Dateien eines Pakets in einem Ordner abgelegt, dessen Name mit dem Paketnamen übereinstimmt.
  In jede Quellcodedatei, die zum Paket gehörige Typen (Klassen oder Interfaces) definiert, ist eine **package**-Deklaration mit der Paketbezeichnung (siehe Abschnitt 6.1) einzufügen, 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

  Bei Ablage in einem Dateisystem wird die Paketstruktur auf einen Dateiverzeichnisbaum abgebildet.
  Vor allem bei der Weitergabe von Programmen ist es nützlich, mehrere (eventuell hierarchisch organisierte) Pakete in eine **Java-Archivdatei** (mit Namenserweiterung **.jar**) verpacken zu können (siehe Abschnitt 6.4). So befinden sich z.B. wesentliche Teile der Java-SE - Standardbibliothek in der Datei **rt.jar**, die man z.B. mit dem kostenlosen Hilfsprogramm 7-Zip (vgl. Abschnitt 2.1.2) öffnen kann. Hier ist der Unterordner mit dem Paket **java.util.concurrent** zu sehen:



- **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 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. In Abschnitt 6.3 wird die Rolle der Pakete bei der Zugriffsverwaltung genauer erläutert.

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 unten).

Im Quellcode müssen fremde Typen prinzipiell über ein durch Punkt getrenntes Paar aus Paketnamen und Typnamen angesprochen werden, wie Sie es schon bei etlichen Beispielen kennen gelernt haben. 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, so dass 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 6.2.2), können seine Typen *ohne* Paketnamen angesprochen werden. Das Paket **java.lang** mit besonders wichtigen Klassen (z.B. **Object**, **System**, **String**) wird bei jeder Anwendung *automatisch* importiert.

6.1 Pakete erstellen

6.1.1 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.:

```
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);
   }
}
```

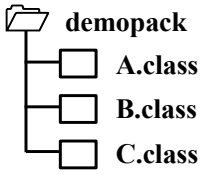
Vor der **package**-Anweisung dürfen höchstens Kommentar- oder Leerzeilen stehen.[1]

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.

Die Typen eines 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 und Felder einer Klasse explizit per Zugriffsmodifikator für fremde Pakete freigegeben werden. Steht z.B. in einer Klasse kein **public**-Konstruktor zur Verfügung, können fremde Pakete eine Klasse zwar „sehen“ und auf öffentliche statische Mitglieder zugreifen, aber keine Objekte dieses Typs erzeugen. Mit den Zugriffsrechten für Typen, Methoden, Konstruktoren und Felder werden wir uns in 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.[2] In unserem Beispiel mit den **public**-Klassen `A`, `B` und `C` im Paket `demopack` muss also folgende Situation hergestellt werden:

- Jede Klasse wird in einer eigenen Quellcodedatei implementiert. Wo diese Dateien 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 Eclipse verwendet per Voreinstellung im Ordner eines Projekts für die Quellcodedateien den Unterordner **src** und für die Bytecodedateien den Unterordner **bin**.
- Die drei Bytecodedateien **A.class**, **B.class** und **C.class** befinden sich in einem Ordner namens **demopack**:



- Der Ordner **demopack** ist über den Suchpfad für **class**-Dateien auffindbar (siehe Abschnitt 6.2.1), d.h. der übergeordnete Ordner von **demopack** ist im Suchpfad aufgelistet. Per Voreinstellung enthält der Suchpfad nur das aktuelle Verzeichnis.

In Abhängigkeit von der verwendeten Java-Entwicklungsumgebung geschieht das Erstellen des Paketordners und das Einsortieren der Bytecode-Dateien eventuell automatisch (siehe Abschnitt 6.1.4 für Eclipse).

---

[1] 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** auszuwertende) Dokumentationskommentare zum Paket unterzubringen (Gosling et al. 2015, Abschnitt 7.4.1).

[2] 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. 2015, Abschnitt 7.2).

6.1.2 Standardpaket

Ohne **package**-Definition 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. für andere Typen im Standardpaket sichtbar sind.

Um vom Compiler und von der JRE gefunden zu werden, müssen die **class**-Dateien mit den Typen des Standardpakets über den Suchpfad für Bytecode-Dateien erreichbar sein (siehe Abschnitt 6.2.1). Bei passender CLASSPATH-Definition dürfen sich die Dateien also in verschiedenen Ordnern oder auch Java-Archiven befinden. Wir haben z.B. im Kursverlauf die zum Standardpaket gehörige Klasse `Simput` in einem zentralen Ordner oder Java-Archiv abgelegt und für verschieden Projekte (d.h. die jeweiligen Typen im Standardpaket) nutzbar gemacht. Dazu wurde der Ordner oder das Archiv mit der Datei **Simput.class** per CLASSPATH-Definition oder eine äquivalente Technik unserer Entwicklungsumgebung Eclipse (vgl. Abschnitt 3.4.2) in den Suchpfad für **class**-Dateien aufgenommen.

6.1.3 Unterpakete

Mit Ausnahme des Standardpakets kann ein Paket **Unterpakete** enthalten, was bei den Java-API - Paketen in der Regel der Fall ist, z.B.:

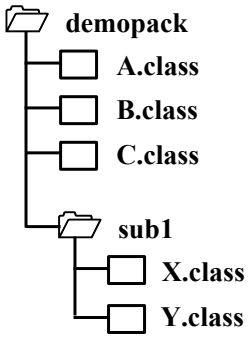
Auf jeder Stufe der Pakethierarchie sind sowohl Typen (Klassen, Interfaces) als auch Unterpakete erlaubt. So enthält z.B. das Paket **java.util** u.a.

- die Klassen **AbstractCollection<E>, Arrays**, **Random**, ...
- die Interfaces **Collection<E>**, **List<E>**, **Map<K,V>**, ...
- die Unterpakete **concurrent**, **function**, **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:



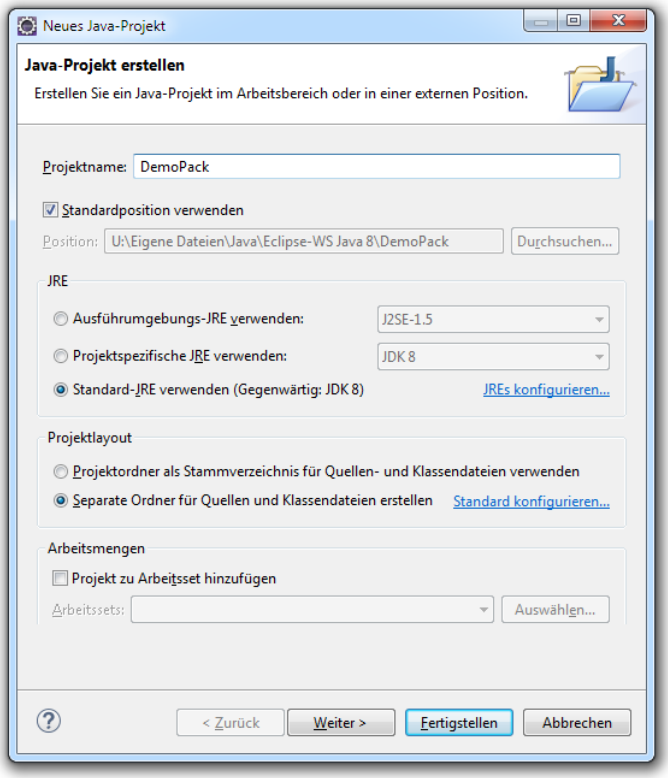
Typen eines Unterpakets gehören *nicht* zum übergeordneten Paket, was beim Importieren von Paketen (siehe Abschnitt 6.2.2) zu beachten ist. Außerdem haben gemeinsame Bestandteile im Paketnamen keine Relevanz für die wechselseitigen Zugriffsrechte (vgl. Abschnitt 6.3). Klassen im Paket `demopack.sub1` haben z.B. für Klassen im Paket `demopack` dieselben Rechte wie Klassen in beliebigen anderen Paketen.

6.1.4 Paketunterstützung in Eclipse

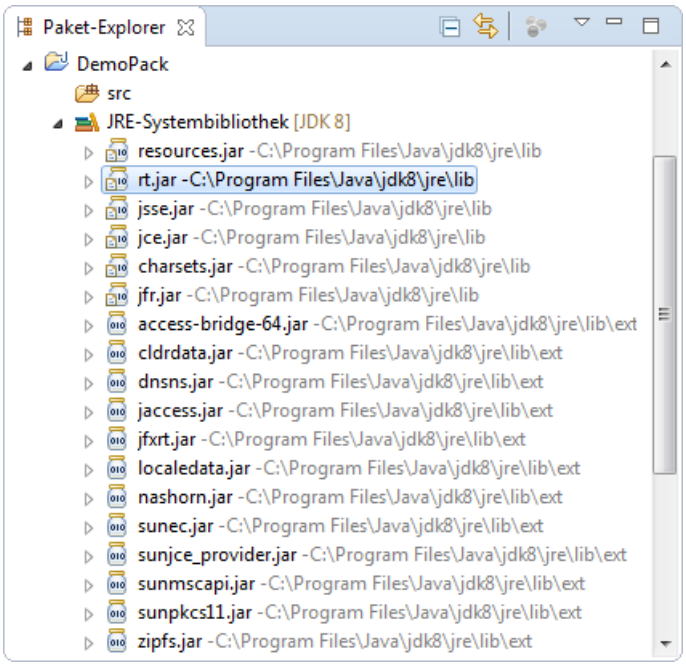
Nachdem schon einige Quellen aus dem Beispielpakt `demopack` zu sehen waren, geht es nun endlich an die Erstellung. Wir starten in Eclipse mit dem Symbolschalter oder dem Menübefehl

Datei > Neu > Java-Projekt

ein neues Java-Projekt mit dem Namen `DemoPack`:



Zunächst zeigt der Paket-Explorer zum neuen Projekt nur die **JRE-Systembibliothek** an:



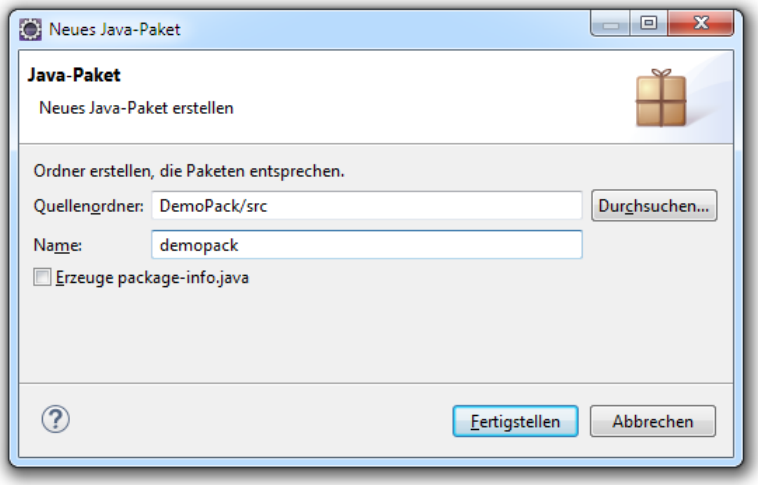
Um ein Paket anzulegen, öffnen wir über den Symbolschalter , den Hauptmenübefehl

Datei > Neu > Paket

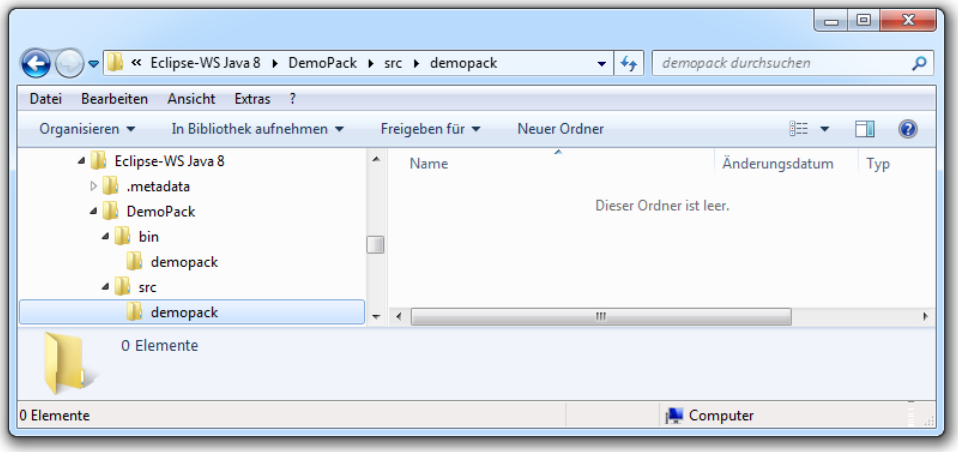
oder den Befehl

Neu > Paket

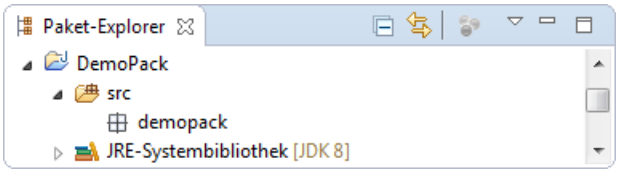
aus dem Kontextmenü zum Projekteintrag im Paket-Explorer den folgenden Dialog und tragen den gewünschten **Namen** für das neue Paket ein:[1]



Anschließend erzeugt Eclipse im **src**-Unterordner des Projekts (zur Aufnahme der Quellcodedateien) und im **bin**-Unterordner des Projekts (zur Aufnahme der Bytecode-Dateien) jeweils einen Unterordner namens **demopack**, z.B.:

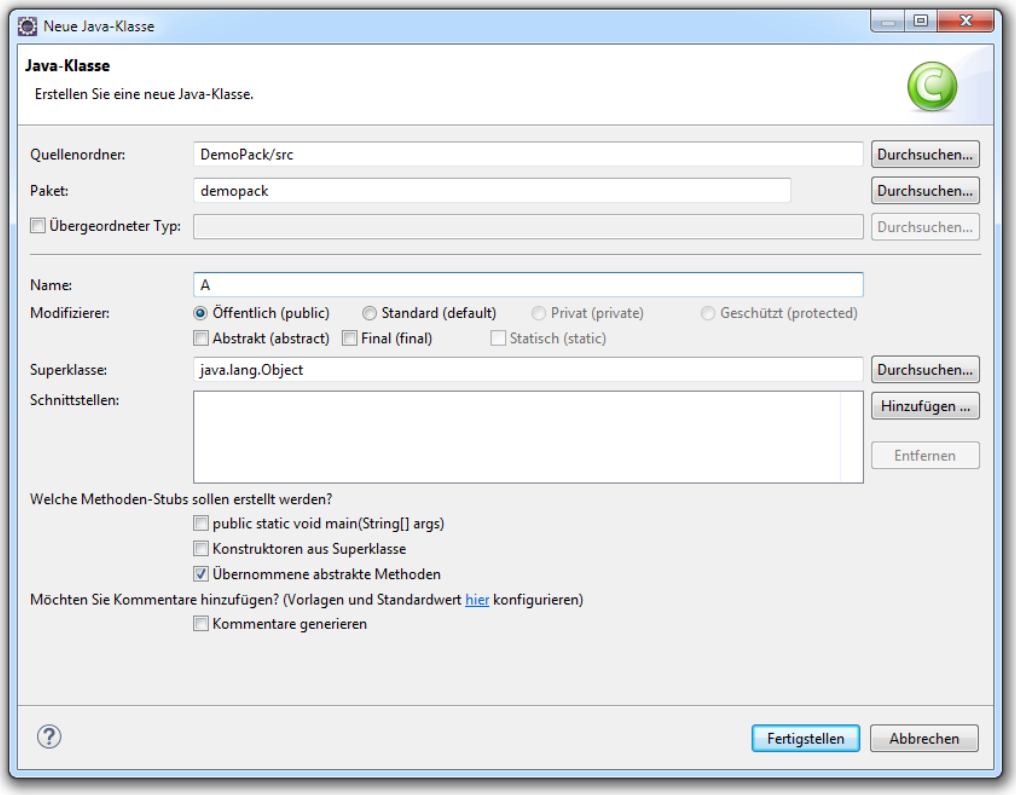


Im Paket-Explorer von Eclipse erscheint der **demopack**-Ordner zur Aufnahme der Quellcodedateien zum neuen Paket:

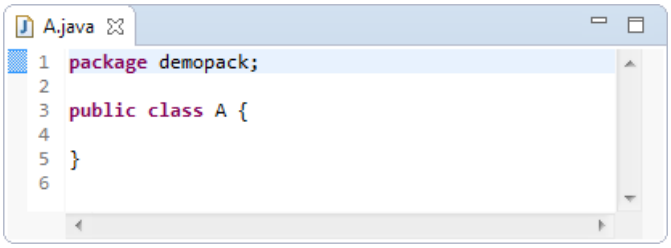


Nun legen wir im Paket `demopack` die Klasse `A` an, z.B. über den Befehl **Neu > Klasse** im Kontextmenü zum Paket:

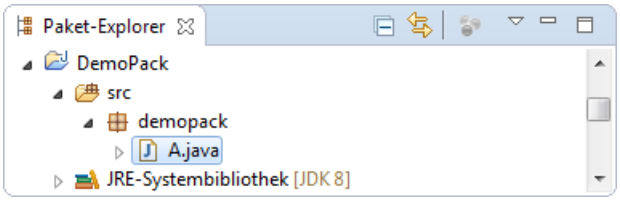
[1] Optional kann in diesem Dialog auch die Erstellung der Datei **package-info.java** zur Aufnahme eines Dokumentationskommentars und/oder einer Annotation zum Paket veranlasst werden (siehe Fußnote in Abschnitt 6.1.1).



Nach dem **Fertigstellen** startet Eclipse im Editor eine Klassendefinition mit **package**-Anweisung



und zeigt im Paket-Explorer den aktuellen Projekt-Entwicklungsstand:



Wir vervollständigen den Quellcode der Klasse `A` (siehe Abschnitt 6.1.1) und legen analog auch die Klassen `B` und `C` im Paket `demopack` an:

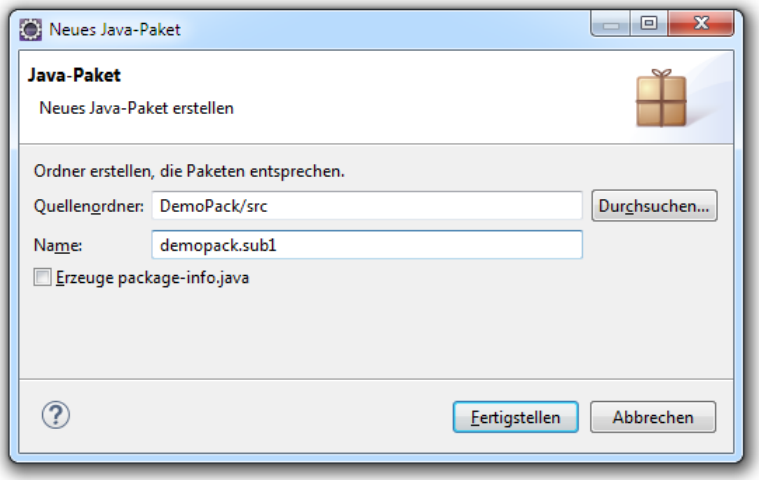
```
package demopack;

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);
  }
}
```

```
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);
  }
}
```

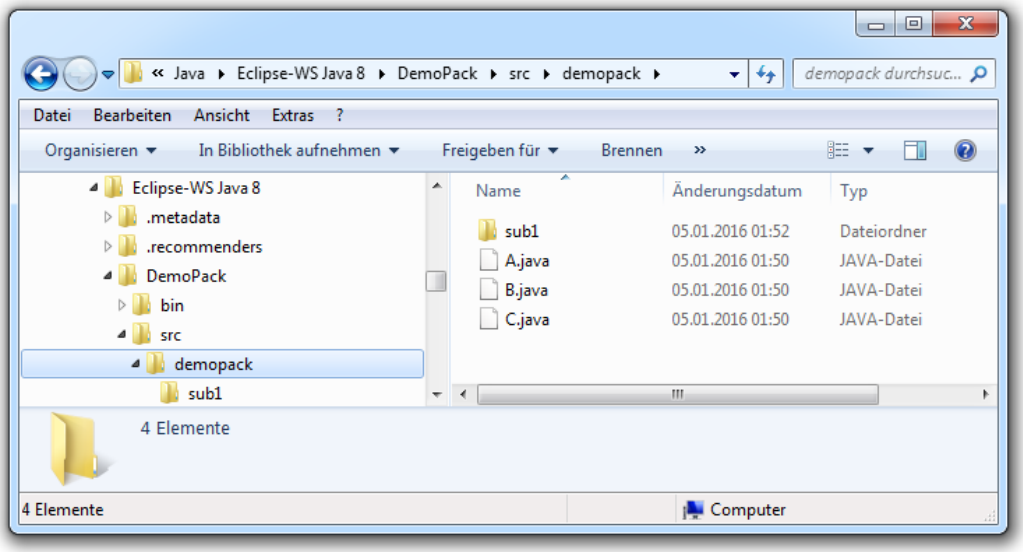
Um das Unterpaket `sub1` zu erzeugen, wählen wir im Paket-Explorer aus dem Kontextmenü zum Paket `demopack` den Befehl

Neu > Paket

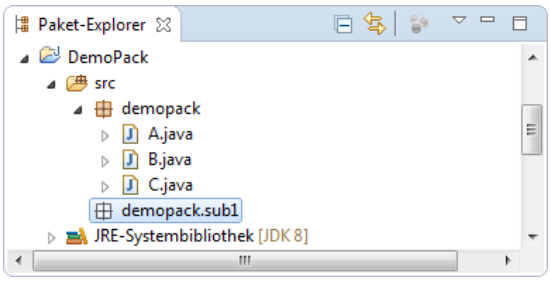
und geben in folgender Dialogbox den gewünschten Namen für das Unterpaket an:



Anschließend erzeugt Eclipse einen Ordner **demopack\sub1** im **src**- und im **bin**-Unterordner des Projekts, z.B.:

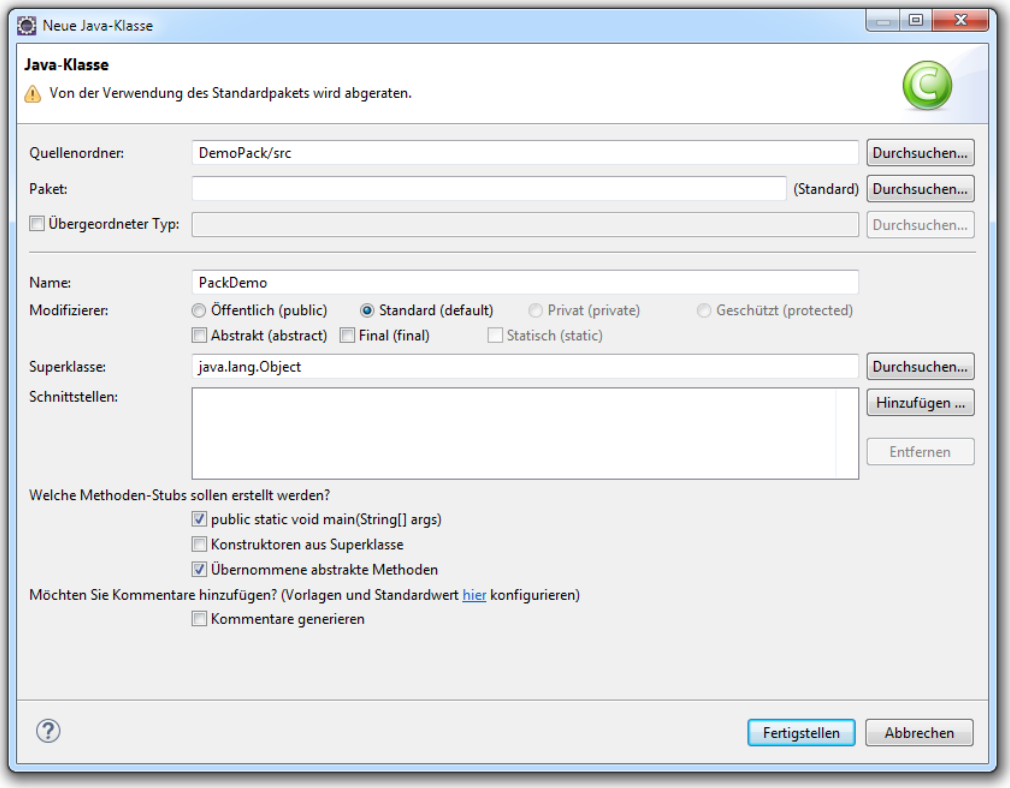


Im Paket-Explorer erscheint das Paket `demopack.sub1`:



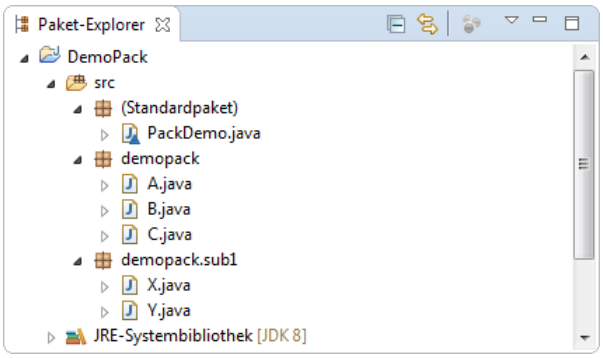
In diesem Unterpaket legen wir nun (z.B. über den Kontextmenübefehl **Neu > Klasse**) die Klasse `X` an, deren Quellcode schon in Abschnitt 6.1.3 zu sehen war, und danach die analog aufgebaute Klasse `Y`.

Schließlich erstellen wir noch eine Startklasse namens `PackDemo` (bitte die Reihenfolge der Namensbestandteile beachten) im *Standardpaket* zu Testzwecken (Quellcode folgt in Abschnitt 6.2.2). Das Standardpaket kann und muss nicht explizit erzeugt werden. Es entsteht automatisch, wenn im Projekt ein Typ ohne Paketzuordnung angelegt wird (leeres Feld **Paket**). Für die Startklasse ist kein **public**-Modifizierer erforderlich, aber eine **main()** - Methode unverzichtbar:



Schließlich entsteht im Paket-Explorer das folgende Bild:[1]

[1] Im Symbol zur Klasse `PackDemo` signalisiert das Dreieck in der rechten unteren Ecke, dass die Klasse nicht als **public** deklariert ist.



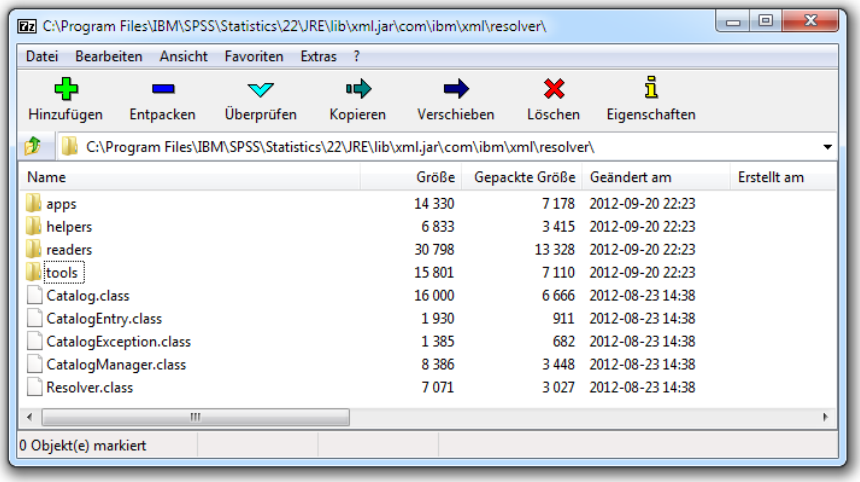
6.1.5 Konventionen für weltweit eindeutige Paketnamen

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. Professionelle Software-Hersteller sollten daher durch Beachtung der folgenden Regeln für weltweit eindeutige Paketnamen sorgen:[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 einer Domäne zu vermeiden. Erstellt z.B. die Firma **IBM** mit der Internet-Domäne **ibm.com** das Paket **xml.resolver**, dann kann die folgende Pakethierarchie verwendet werden:

  com.ibm.xml.resolver

  Unter Beachtung dieser Regel hat die Firma **IBM** zusammen mit dem Programm **SPSS Statistics** (Version 22) in der Java-Archivdatei **xml.jar** das als Beispiel verwendete Paket ausgeliefert:



- Bestandteile im Domänenamen, die zu einem ungültigen Java-Bezeichner führen würden, müssen ersetzt werden, z.B. der Bindestrich durch einen Unterstrich. 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`

[1] Quelle (abgerufen am 10.01.2015): http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html

6.2 Pakete verwenden

Damit man die in fremden Paketen vorhandenen Typen in einem eigenen Programm verwenden kann, …

- müssen die **class**-Dateien für die beteiligten Werkzeuge (z.B. Compiler) auffindbar sein
- und die Klassen bzw. Schnittstellen im eigenen Quellcode korrekt angesprochen werden.

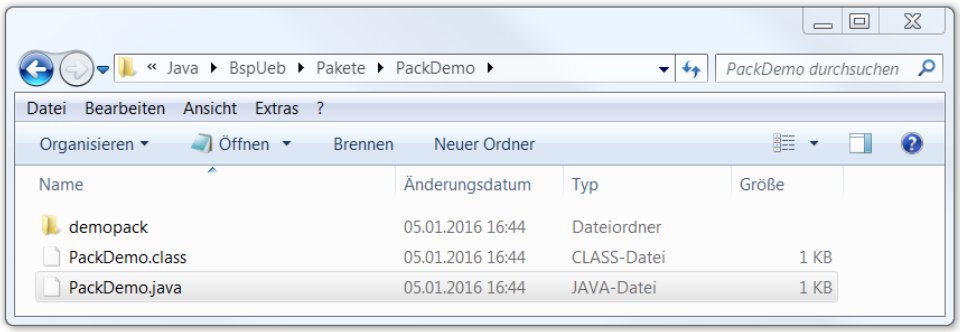
Mit den beiden Themen wurden wir schon konfrontiert, doch sollen die zugehörigen Lösungen gleich noch einmal zusammengestellt werden.

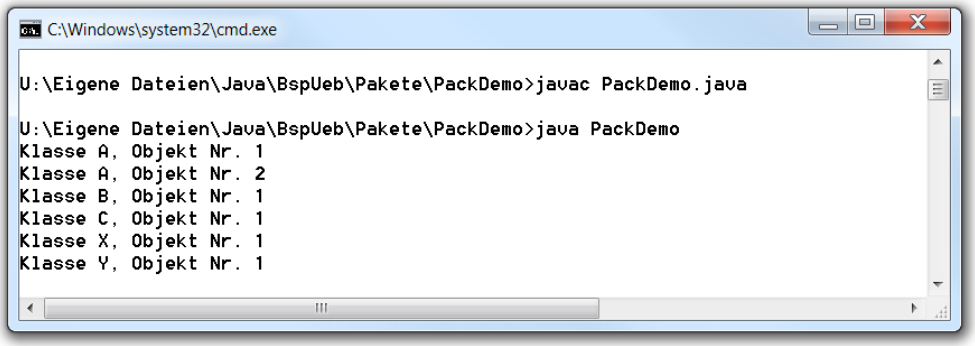
Mit einer weiteren Nutzungsvoraussetzung, den Zugriffsrechten, werden wir uns in Abschnitt 6.3 beschäftigen.

6.2.1 Verfügbarkeit der class-Dateien

Damit die in einem Paket vorhandenen Typen genutzt werden können, muss der Aufenthaltsort dem Compiler bzw. Interpreter 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 6.4) und beschränken uns passend zum Entwicklungsstand des `demopack`-Beispiels auf Pakete in Verzeichnissen. Zunächst behandeln wir das Übersetzen und die Ausführung eines Java-Programms auf einem Windows-Rechner, auf dem neben einer öffentlichen JRE auch ein JDK (mit dem Compiler **javac.exe**) installiert ist. Die Unterstützung bzw. spezielle Umgebung einer Entwicklungsumgebung wird dabei *nicht* berücksichtigt:

- Paketordner im **aktuellen Verzeichnis**
  Befindet sich das oberste Verzeichnis im Paketnamen (in unserem Beispiel: **demopack**) im selben Ordner wie eine zu übersetzende oder auszuführende Klasse (im Beispiel: `PackDemo`), dann werden die Paketklassen vom JDK-Compiler bzw. von der JRE gefunden, sofern keine ungeeignete CLASSPATH-Definition das aktuelle Verzeichnis ausschließt:



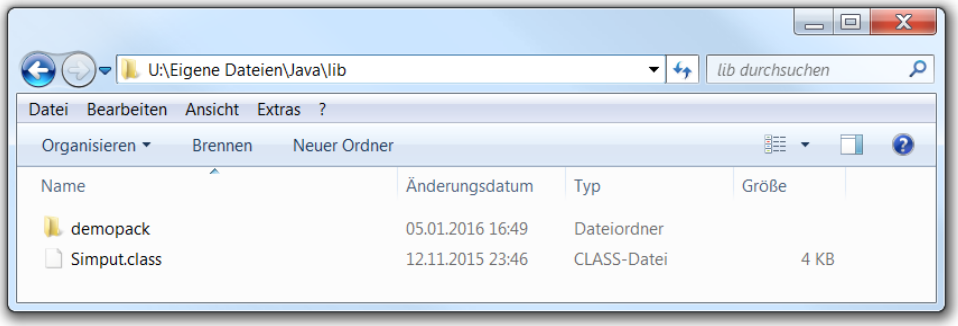


Dies ist allerdings keine sinnvolle Option, wenn ein Paket in mehreren Projekten eingesetzt werden soll.

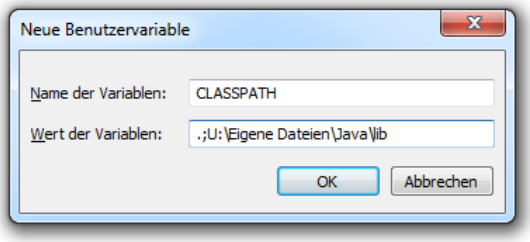
- Stammordner des Pakets in der **CLASSPATH-Umgebungsvariablen**
  Über die Betriebssystem-Umgebungsvariable CLASSPATH lässt sich der Suchpfad für **class**-Dateien anpassen. Im bisherigen Kursverlauf haben wir auf diese Weise den Ordner (oder die Archiv-Datei) mit der Standardpaket-Klasse **Simput.class** bekannt gegeben (vgl. Abschnitt 2.2.4). Analog lassen sich auch komplette Pakete (samt Unterpaketen) verfügbar machen. 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:

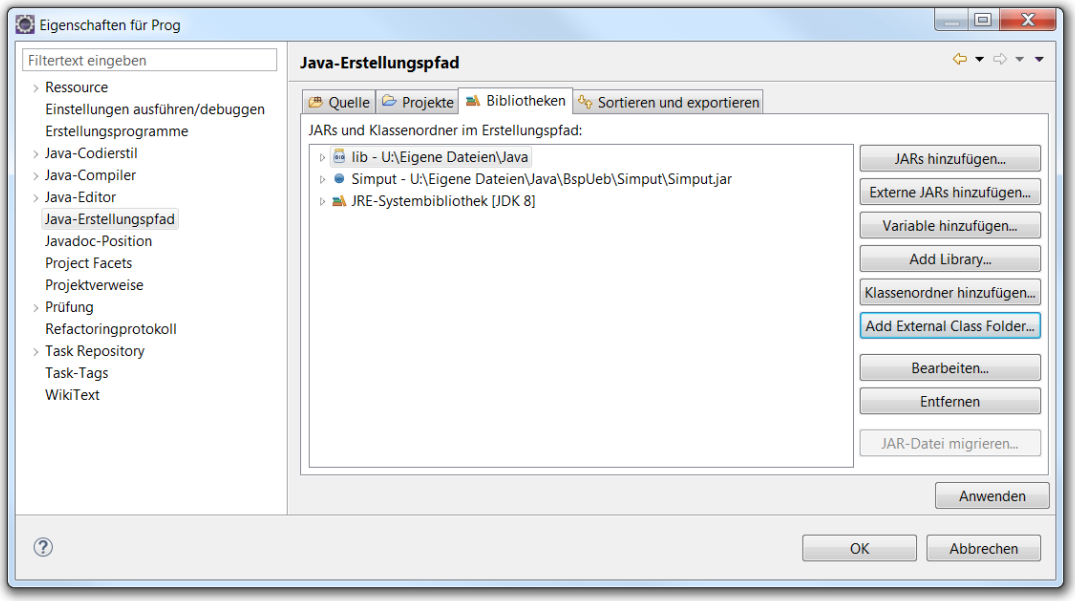


  Der JDK-Compiler und die JRE finden z.B. den Bytecode der Klasse `demopack.A`, indem sie jedes Verzeichnis in der CLASSPATH-Definition nach der Datei **...\demopack\A.class** durchsuchen.

- Stammordner des Pakets in der **classpath-Befehlszeilenoption**
  Bei Aufruf der JDK-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
```

Eclipse ignoriert die Umgebungsvariable CLASSPATH, bietet aber alternative Optionen, **class**-Dateien für ein Projekt verfügbar zu machen. Man kann einen nicht zum Projekt (und nicht zum Java-API) gehörigen Paketordner verfügbar machen, indem man per Projekt-Eigenschaftsdialog das übergeordnete Verzeichnis als externen Klassenordner (engl.: *external class folder*) in den **Java-Erstellungspfad** aufnimmt, z.B.:



Für einen in vielen Projekten benötigten externen Klassenordner sollte man auf der Arbeitsbereichsebene eine Klassenpfadvariable definieren und nach Bedarf in den Erstellungspfad von Projekten aufnehmen (vgl. Abschnitt 3.4.2).

6.2.2 Typen aus fremden Paketen ansprechen

Java bietet folgende Möglichkeiten, die Typen aus einem fremden und von **java.lang** verschiedenen Paket im eigenen Quellcode anzusprechen:

- **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** im 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). In folgendem 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.*;
  ```

  Beachten Sie bitte, dass *Unterpakete* dabei *nicht* einbezogen werden. Für sie ist bei Bedarf eine separate **import**-Anweisung 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.
  Zwei Pakete werden vom Compiler automatisch importiert:
  - Das API-Paket **java.lang** mit wichtigen Klassen wie **System**, **String**, **Math**
  - Das Paket, zu dem die aktuell übersetzte Quelle gehört

- **Import von statischen Methoden und Feldern**

  Seit Java 5 besteht die Möglichkeit, statische Methoden und Variablen fremder Typen so zu importieren, so 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:

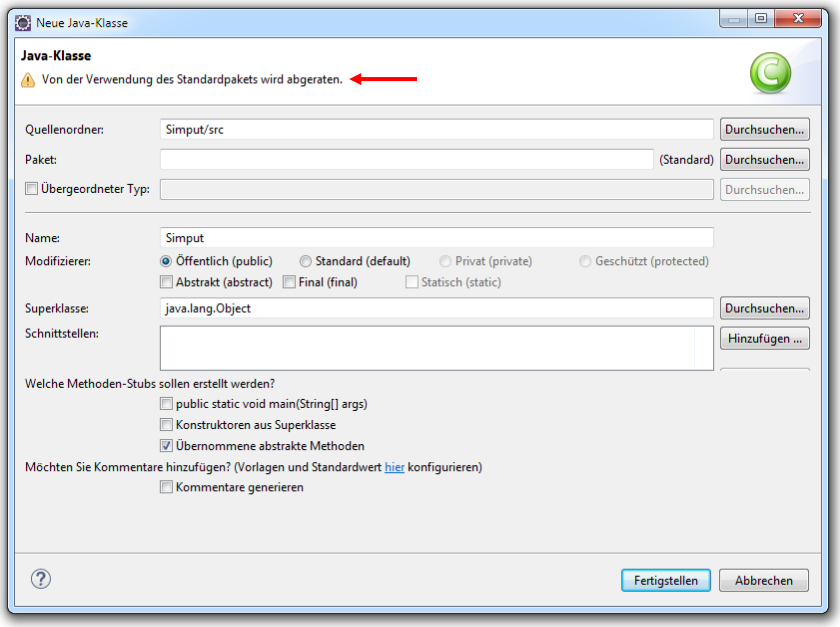
  ```
  import static java.lang.Math.*;
  class Prog {
    public static void main(String[] args) {
    System.out.println("Sin(Pi/2) = " + sin(PI/2));
    }
  }
  ```

  In der importierenden Quellcodedatei wird im Vergleich zum normalen Paketimport nicht nur der Paket- sondern auch der Klassenname eingespart.

In folgendem Programm, das unser `demopack`-Beispiel komplettiert, werden das Paket `demopack` und das Unterpaket `demopack.sub1` importiert:

Quellcode	Ausgabe
`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();` `  }` `}`	`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`

Die Typen im unbenannten Standardpaket sind in anderen Paketen generell (auch bei Zugriffsstufe **public**) *nicht* verfügbar. Diese gravierende Einschränkung haben wir bisher der Einfachheit halber meist in Kauf genommen (z.B. auch bei den häufig verwendeten Beispielklassen `Bruch` und `Simput`), obwohl unsere Entwicklungsumgebung Eclipse stets warnt, z.B.:



Obwohl uns die Paketierungstechnik nun bekannt ist, werden wir im weiteren Verlauf des Kurses bei kleineren Projekten der Einfachheit halber auf die Definition eines Pakets verzichten.

6.2.3 Startklassen in Paketen

Wir haben uns im bisherigen Kursverlauf nur wenig damit beschäftigt, wie ein Java-Programm außerhalb unserer Entwicklungsumgebung Eclipse 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 *Installationsverzeichnis* des zu startenden Programms 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, so dass sich das aktuelle Verzeichnis der Konsole (das Installationsverzeichnis) im impliziten Klassensuchpfad befand.

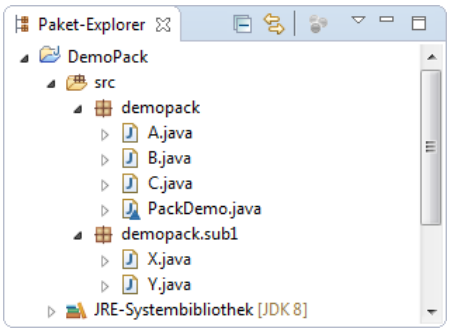
Unter diesen Bedingungen gelingt der Programmstart mit einem simplen Kommando, z.B.:

```
>java PackDemo
```

Wird für ein Programm ein eigenes Paket verwendet, sollte in der Regel aber auch die Startklasse dort untergebracht werden. Um dieses Ziel im `DemoPack`-Beispiel zu realisieren, ersetzen wir im `PackDemo`-Quellcode den `demopack`-Import durch eine **package**-Deklaration:

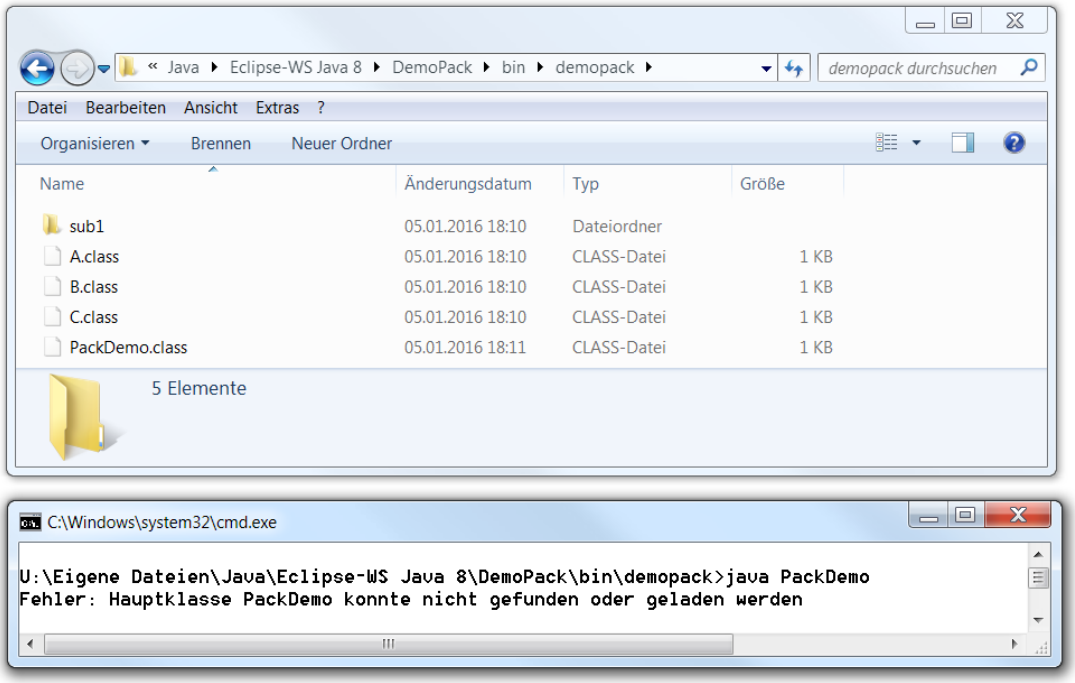
```
package demopack;
import demopack.sub1.*;

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();
 }
}
```

Im Paket-Explorer von Eclipse sieht die neue Projektstruktur so aus:[1]



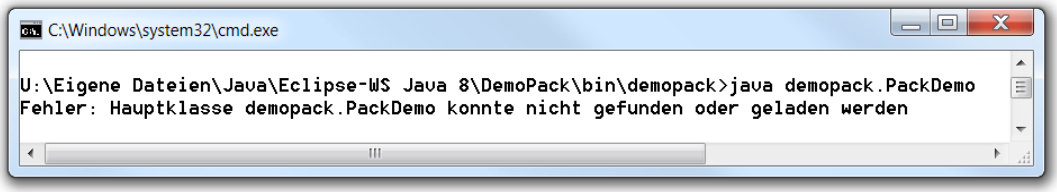
Nun scheitert aber der Versuch, mit dem obigen **java**-Aufruf die Klasse `PackDemo` zu starten (Datei **PackDemo.class** im aktuellen Ordner der Konsole, keine explizite CLASSPATH-Definition):

[1] Im Symbol zur Klasse `PackDemo` signalisiert das Dreieck in der rechten unteren Ecke, dass die Klasse nicht als **public** deklariert ist.



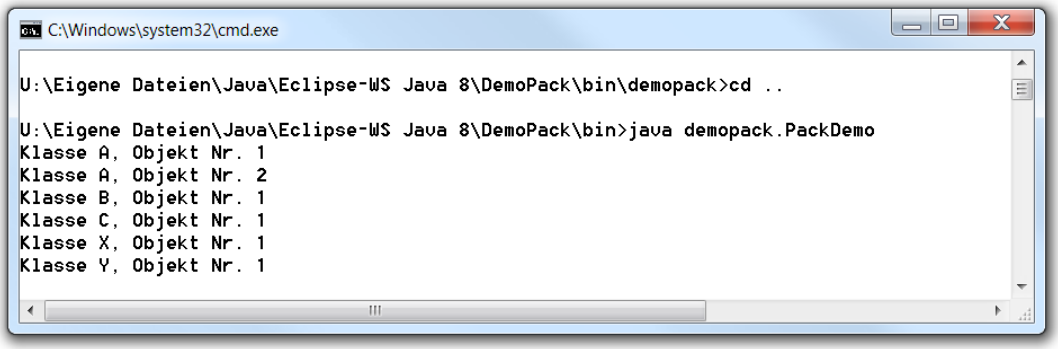
Wir erinnern uns daran, dass im **java**-Aufruf keine *Datei* anzugeben ist, sondern eine *Klasse*. Im aktuellen Beispiel hat die gewünschte Klasse den Namen `demopack.PackDemo`, sodass der Java-Starter zu Recht reklamiert, es sei keine Hauptklasse mit dem Namen `PackDemo` zu finden.

Den vollständigen Namen der Hauptklasse anzugeben, hilft aber nicht, solange das Konsolenfenster auf den Ordner `demopack` positioniert ist:



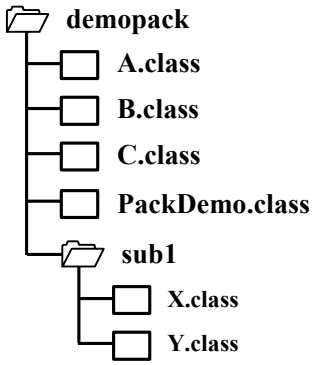
Nun sucht der Java-Starter nämlich ausgehend vom aktuellen Ordner vergeblich nach dem Paketordner **demopack**.

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 `DemoPack`-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 **PackDemo.class** dorthin kopieren,



- zu **demopack** einen Unterordner namens **sub1** anlegen und die Dateien **X.class** sowie **Y.class** dorthin kopieren.

Ist ein Konsolenfenster auf den Installationsordner positioniert und der voreingestellte Klassensuchpfad in Verwendung, dann kann die Hauptklasse über ihren vollständigen Namen gestartet werden:

```
>java demopack.PackDemo
```

Soll dieser Start aus einem Konsolenfenster mit einem *beliebigen* aktuellen Verzeichnis möglich sein, 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 e:\prog\mapp demopack.PackDemo
```

6.3 Zugriffsschutz

Nach der Beschäftigung mit Paketen lässt sich endlich präzise erläutern, wie in Java die Zugriffsrechte für Klassen, 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 Zugriffsschutz für Top-Level - Klassen

Bisher haben wir uns überwiegend mit *Top-Level - Klassen*) beschäftigt, die *nicht* innerhalb des Quellcodes anderer Klassen definiert werden. In diesem Abschnitt geht es um den Zugriffsschutz für solche Klassen.[1]

Bei den Top-Level - Klassen ist nur der Zugriffsmodifikator **public** erlaubt, so dass zwei Schutzstufen möglich sind:

---

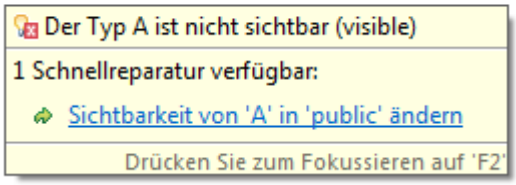
[1] Die *Mitgliedsklassen* (synonym: *eingeschachtelten Klassen*), die innerhalb einer Top-Level - Klasse, aber außerhalb von Methoden definiert werden (siehe Abschnitt 4.9.1), sind beim Zugriffsschutz wie andere Klassenmitglieder (z.B. Felder und Methoden) zu behandeln (siehe Abschnitt 6.3.2). Bei den innerhalb von Methoden definierten *lokalen Klassen* (siehe Abschnitt 4.9.2) und den *anonymen Klassen* sind Zugriffsmodifikatoren irrelevant und verboten.

- Ohne Zugriffsmodifikator ist die Klasse nur innerhalb des eigenen Pakts verwendbar.
- Durch den Zugriffsmodifikator **public** wird die Verwendung in beliebigen Paketen erlaubt, z.B.:

```
package demopack;
public class A {
  .  .  .
}
```

Wird im `demopack`-Paket die Klasse `A` ohne **public**-Zugriffsmodifikator definiert, scheitert die Übersetzung der in Abschnitt 6.2.2 vorgestellten Hauptklasse `PackDemo`, die sich im Standardpaket befindet. Der Eclipse-Compiler hilft durch das Markieren der Fehlerstellen und durch einen sinnvollen Vorschlag zur **Schnellreparatur**:

```
public static void main(String[] args) {
    A a1 = new A(), a2 = new A();
```



Auch der JDK-Compiler liefert eine hilfreiche Problembeschreibung:

```
> javac PackDemo.java
PackDemo.java:6: error: A is not public in demopack;
cannot be accessed from outside package
                A a1 = new A(), a2 = new A();
                ^
```

Pro Quellcodedatei darf nur *eine* Klasse als **public** deklariert werden. Eventuell in derselben Datei vorhandene zusätzliche Klassen sind also nur paketintern verwendbar.

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 JRE 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. (2015) lassen bei Hauptklassen den Modifikator **public** systematisch weg.

6.3.2 Zugriffsschutz für Klassenmitglieder

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 genutzt werden. Typischer-

weise 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 stehen.

Methoden einer Klasse `B` (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 einer Top-Level - Klasse beschrieben, die selbst als **public** definiert ist. Auch bei den „Zugriffsbewerbern“ soll es sich um Top-Level - Klassen handeln.

Modifikator	**Der Zugriff ist erlaubt für ...**			
	die eigene Klasse	**andere Klassen im eigenen Paket**	**abgeleitete Klassen in fremden Paketen**	**sonstige Klassen**
ohne	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 in Kapitel 7 beschäftigen.

Wird im `demopack`-Beispiel die Klasse `A` mit **public**-Zugriffsmodifikator versehen, ihre `prinr()`-Methode jedoch *nicht*, dann scheitert das Übersetzen der Klasse `PackDemo`, die sich im Standardpaket befindet, durch den JDK-Compiler mit folgender Meldung:

---

[1] Unter dem Begriff *Mitgliedstypen* sind hier Klassen und Interfaces zu verstehen, die innerhalb einer Top-Level-Klasse außerhalb von Methoden definiert werden, z.B.:

```
public class Top {
 . . .
 private class MemberClass {
   . . .
 }
 . . .
}
```

```
>javac PackDemo.java
PackDemo.java:8: error: prinr() is not public in A;
cannot be accessed from outside package
                a1.prinr();
                  ^
```

Für Konstruktoren gilt:

- Bei expliziten Konstruktoren sind wie bei Methoden die Modifikatoren **public**, **private** und **protected** erlaubt. Ein als **protected** deklarierter Konstruktor darf im eigenen Paket und von abgeleiteten Klassen in beliebigen Paketen genutzt werden.
- Der vom Compiler bereitgestellte Standardkonstruktor (vgl. Abschnitt 4.4.3) hat denselben Zugriffsschutz wie die Klasse.

Für die voreingestellte Schutzstufe (nur das eigene Paket darf zugreifen) wird gelegentlich die Bezeichnung *package* verwendet.

6.4 Java-Archivdateien

Wenn zu einem Programm 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.

6.4.1 Eigenschaften von Archivdateien

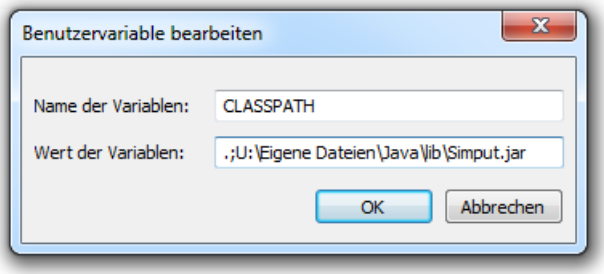
Java-Archivdateien bieten viele Vorteile, z.B.:

- **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.
- **Verkürzte Übertragungszeiten**
  Eine einzelne Archivdatei reduziert im Vergleich zu zahlreichen einzelnen Dateien die Wartezeit beim Laden von einer Festplatte oder über ein Netzwerk.
- **Kompression**
  Java-Archivdateien können komprimiert werden, was für Applets[1] wegen des beschleunigten Internet-Transports sinnvoll, bei lokal installierten Anwendungen jedoch wegen der erforderlichen Dekomprimierung eher nachteilig ist. Die Archivdateien mit den Java-API -Paketen (z.B. **rt.jar**) sind daher *nicht* komprimiert.
- **Sicherheit**
  Bei *signierten* JAR-Dateien kann sich der Anwender Gewissheit über den Urheber verschaffen und der Software entsprechende Rechte einräumen.
- **Versionsangaben für Pakete**
  In einem Archiv kann man Hersteller- und Versionsangaben zu den enthaltenen Paketen unterbringen.

[1] Mit Java-Applets werden wir uns in diesem Kurs nicht beschäftigen.

Mit den beiden zuletzt genannten Vorteilen können wir uns in diesem Manuskript aus Zeitgründen nicht beschäftigen.

Eine Archivdatei kann beliebig viele Pakete enthalten. Damit die dortigen **class**-Dateien vom Compiler und von der JRE 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 6.2.1), z.B. über die Umgebungsvariable CLASSPATH:

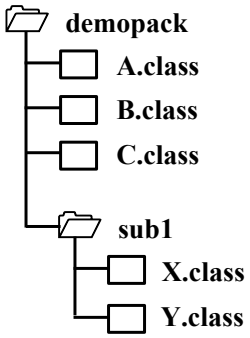


Bei den Archiven mit den Java-API - Paketen ist dies allerdings *nicht* erforderlich.

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 6.4.2 und 6.4.4) oder einer entsprechend ausgestatteten Entwicklungsumgebung überlassen (zum Verfahren in Eclipse siehe Abschnitt 6.4.5).

6.4.2 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` (auf dem Stand von Abschnitt 6.1.3) und im Unterpaket `sub1`



zu erzeugen.

Wir öffnen ein Konsolenfenster und wechseln zum Verzeichnis, das den Ordner **demopack** enthält (z.B. **U:\Eigene Dateien\Java\BspUeb\Pakete\DemoPack\bin**). Dann lassen wir mit folgendem **jar**-Aufruf das Archiv **demarc.jar** mit der gesamten Paket-Hierarchie erstellen:[1]

```
>jar cf0 demarc.jar demopack
```

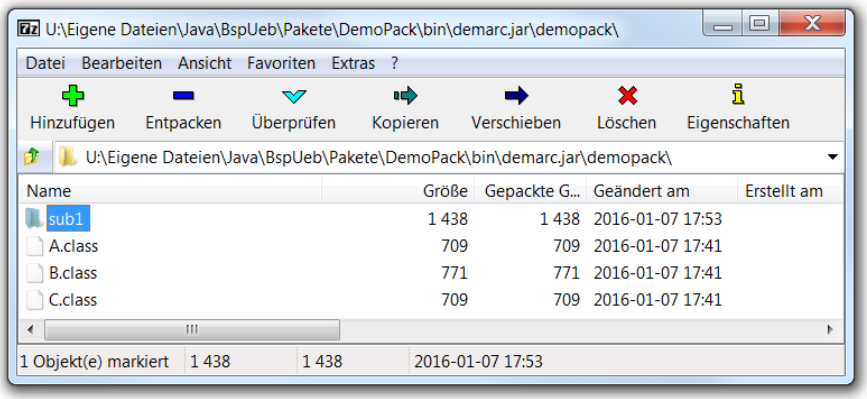
[1] Sollte der Aufruf nicht klappen, befindet sich vermutlich das JDK-Unterverzeichnis **bin** (z.B. **C:\Program Files\Java\jdk8\bin**) nicht im Suchpfad für ausführbare Programme. In diesem Fall muss das Programm mit kompletter Pfadangabe gestartet werden.

Darin bedeuten:

- 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. Selbstverständlich kann eine Archivdatei auch mehrere Pakete bzw. Ordner aufnehmen, was z.B. die ca. 33 MB große Datei **rt.jar** demonstriert, die praktisch alle Java-API - Pakete enthält. Mehrere Ordnernamen werden durch Leerzeichen getrennt aufgelistet.

Weitere Informationen über das Archivierungswerkzeug finden Sie z.B. in der JDK-Dokumentation über den Link **jar**.

Aus obigem **jar**-Aufruf resultiert die folgende JAR-Datei (hier angezeigt vom kostenlosen Hilfsprogramm **7-Zip**, vgl. Abschnitt 2.1.2):



Es ist übrigens erlaubt, dass sich die zu einem Paket gehörigen **class**-Dateien in verschiedenen JAR-Dateien befinden.

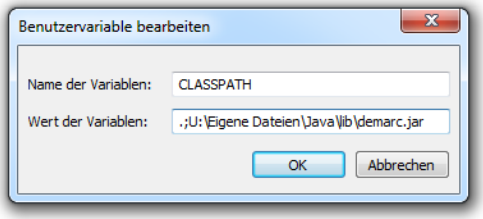
Die Quellcodedateien sind für die Verwendung eines Archivs (als Programm oder Klassenbibliothek) *nicht* erforderlich und können daher (z.B. aus urheberrechtlichen Gründen) durch die Ablage in einer separaten Ordnerstruktur aus dem Archiv herausgehalten werden. Unsere Entwicklungsumgebung Eclipse verwendet per Voreinstellung für Quell- und Bytecodedateien eines Projekts separate Ordner namens **src** und **bin** (siehe Option **Separate Ordner für Quellen und Klassendateien erstellen** im ersten Assistentendialog für neue Java-Projekte).

6.4.3 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** und die Quellcodedatei der Klasse `PackDemo`, die zum Standardpaket gehört und das Paket `demopack` importiert, im aktuellen Verzeichnis, dann kann das Übersetzen und Ausführen dieser Klasse mit folgenden Aufrufen der JDK-Werkzeuge **javac** und **java** erfolgen:

```
>javac -cp "U:\Eigene Dateien\Java\lib\demarc.jar" PackDemo.java
>java -cp ".;U:\Eigene Dateien\Java\lib\demarc.jar" PackDemo
```

Analog zur **classpath**-Option in den Werkzeugaufrufen kann eine Archivdatei in die CLASSPATH-Umgebungsvariable des Betriebssystems aufgenommen werden, z.B.:



Danach lassen sich die obigen Werkzeug-Aufrufe vereinfachen:

```
>javac PackDemo.java
>java PackDemo
```

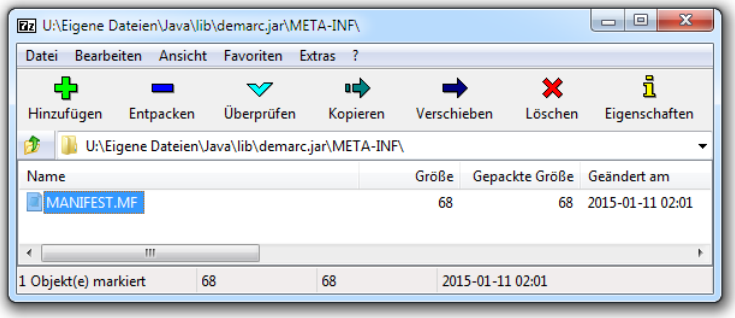
Für die Nutzung von Archivdateien in Eclipse ist das Setzen von Klassenpfadvariablen sehr zu empfehlen (siehe Abschnitt 3.4.2).

6.4.4 Ausführbare JAR-Dateien

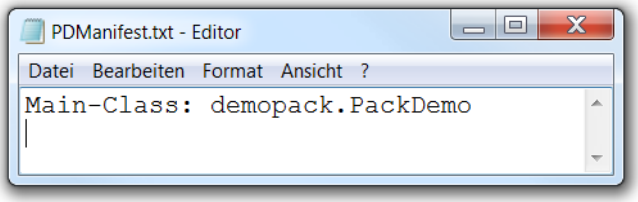
Um eine als Anwendung ausführbare JAR-Datei zu erstellen, nimmt man die gewünschte Startklasse in das Archiv auf. Diese Klasse muss bekanntlich eine Methode **main()** mit folgendem Definitionskopf besitzen:

```
public static void main(String[] args)
```

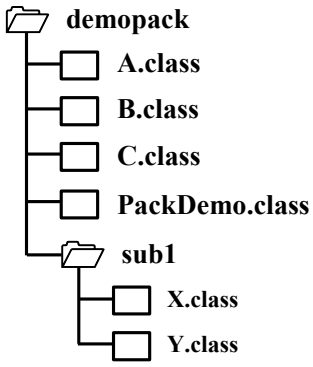
Außerdem muss die Klasse im so genannten **Manifest** des Archivs, dem wir bisher keine Beachtung geschenkt haben, als **Main-Class** eingetragen werden. Das Manifest befindet sich in der Datei **MANIFEST.MF**, die das **jar**-Werkzeug im Archiv-Ordner **META-INF** anlegt, z.B. bei **demarc.jar**:



Im **jar**-Aufruf kann man eine Textdatei mit Manifestinformationen übergeben. Um z.B. die Startklasse `PackDemo` im Paket `demopack` auszuzeichnen, legt man eine Textdatei an, die folgende Zeile und eine anschließende Leerzeile (!) enthält:



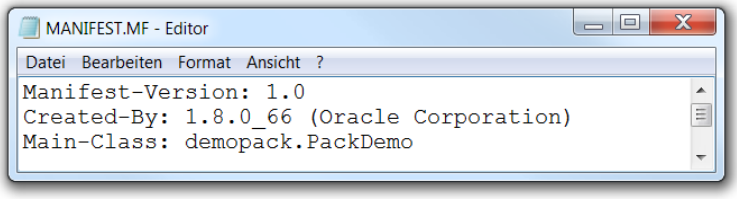
Das `DemoPack`-Projekt hat nun wieder den Entwicklungsstand aus Abschnitt 6.2.3, so dass sich die Startklasse `PackDemo` nicht im Standardpaket befindet, sondern im Paket `demopack`:



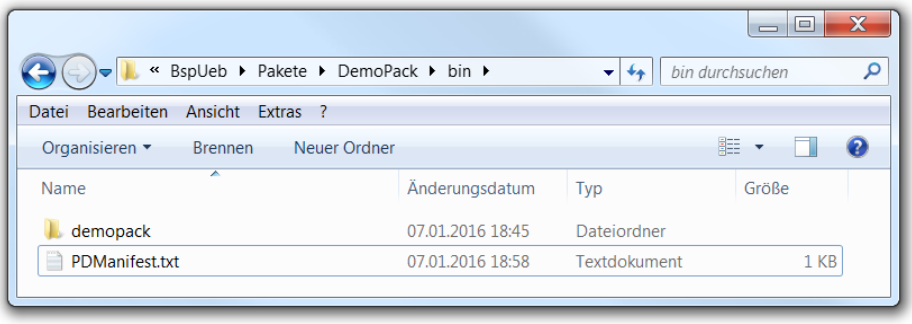
Im **jar**-Aufruf zum Erstellen des Archivs wird über die Option **m** eine Datei mit Manifestinformationen angekündigt, z.B. mit dem Namen **PDManifest.txt**:

```
>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 ohne CLASSPATH-Definition, 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.PackDemo PDApp.jar demopack
```

Unter Verwendung der Archivdatei **PDApp.jar** lässt sich das Programm `PackDemo` 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 JRE 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.

Wenn mit dem Betriebssystem die Behandlung von **jar**-Dateien passend vereinbart wird, klappt der Start sogar per Mausdoppelklick auf die Archivdatei. Unter Windows sind dazu folgende Registry-Einträge geeignet:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.jar]
@="jarfile"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\jarfile]
@="Executable Jar File"

[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\\Java\\jre1.8.0_66\\bin\\java.exe\" -jar \"%1\" %*"
```

An Stelle des für Konsolenprogramme erforderlichen Starters **java.exe** wird bei der JRE-Installation allerdings das für GUI-Anwendungen sinnvollere Startprogramm **javaw.exe** in die Windows-Registry eingetragen, z.B.:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\jarfile\shell\open\command]
@="\"C:\\Program Files\\Java\\jre1.8.0_66\\bin\\javaw.exe\" -jar \"%1\" %*"
```

Weil **javaw.exe** kein Konsolenfenster anzeigt, bleibt der Doppelklick auf **PDApp.jar** ohne sichtbare Folgen.

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.

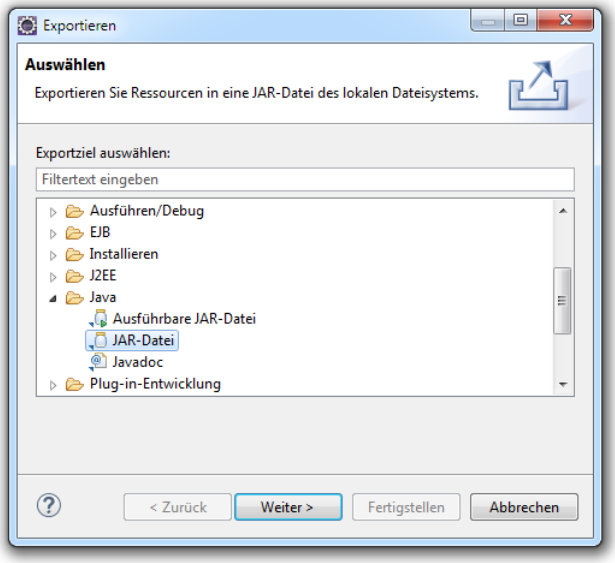
[1] Siehe: http://docs.oracle.com/javase/tutorial/deployment/jar/downman.html

6.4.5 Archivdateien in Eclipse erstellen

In Eclipse ist ein bequemer Assistent zum Erstellen von JAR-Dateien verfügbar. Wählen Sie z.B. aus dem Kontextmenü zum Projekt `DemoPack` das Item

Exportieren

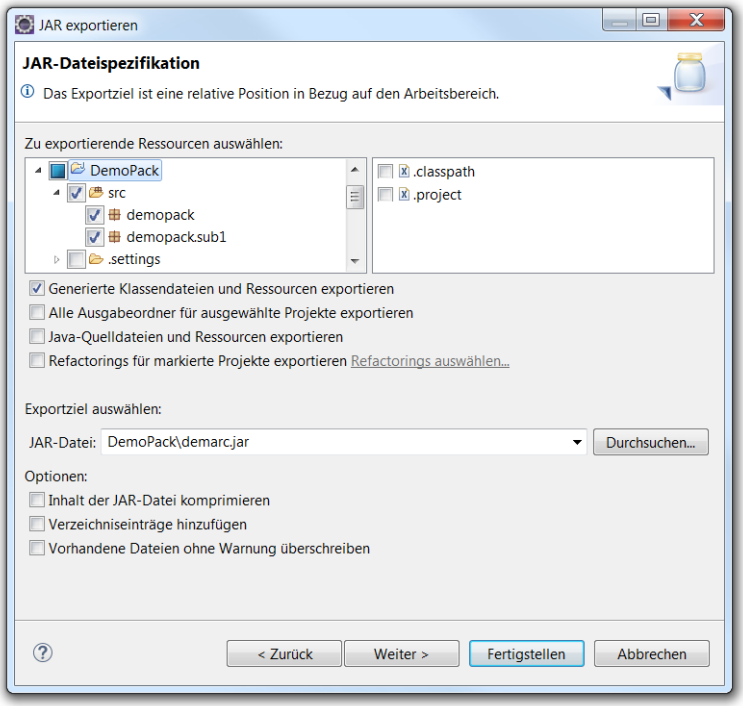
und entscheiden Sie sich im ersten Assistentendialog



für die Option

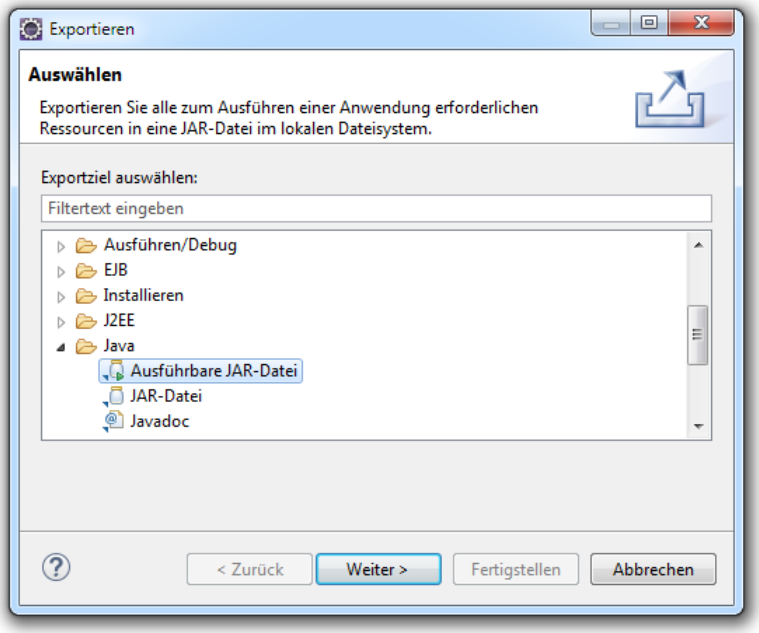
Java > JAR-Datei

Mit der folgenden Wahl von Exportumfang und -ziel:

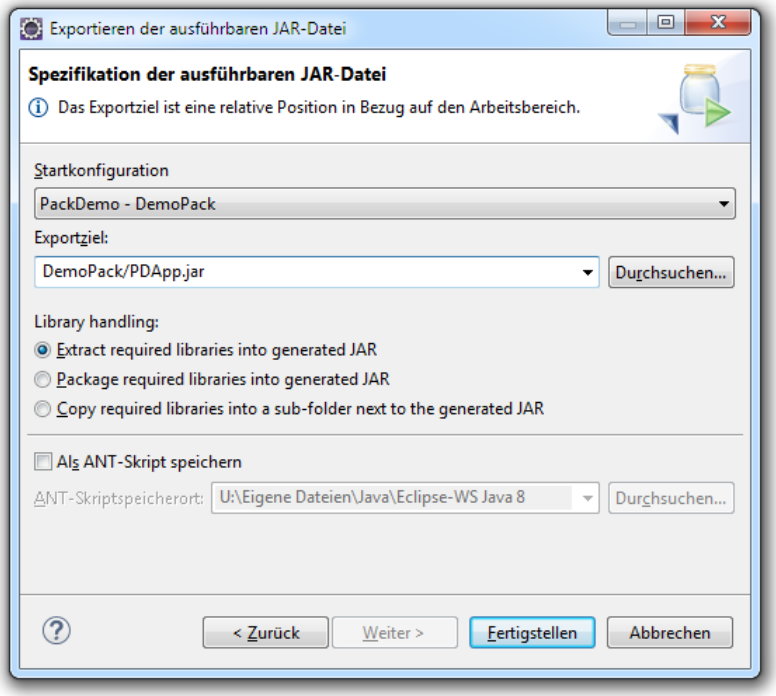


arbeitet der Assistent nach dem **Fertigstellen** analog zu dem in Abschnitt 6.4.2 vorgestellten **jar**-Kommando.

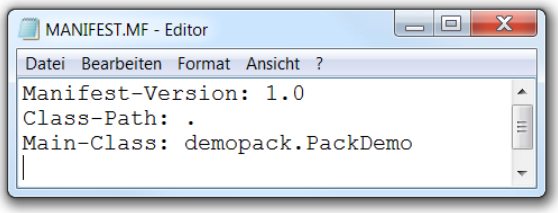
Wählt man im ersten Exportassistentendialog eine **ausführbare JAR-Datei**,



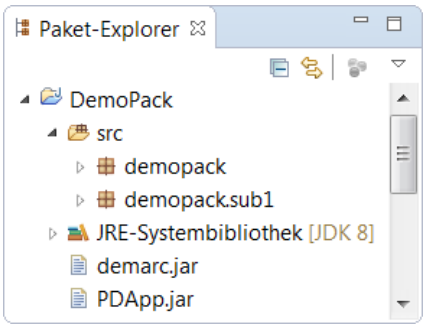
kann man im nächsten Schritt eine (nötigenfalls vorher angelegte) **Startkonfiguration** (siehe Abschnitt 3.7.2.4) wählen, das **Exportziel** nennen



und die Anwendung **fertigstellen**. Eclipse fügt selbständig die benötigten Paketdateien in das Archiv ein und erstellt eine geeignete Manifestdatei:



Die eben von Eclipse produzierten JAR-Dateien sind auch im Paket-Explorer zu sehen:



6.5 Das API der Java Standard Edition

Zur Java-Plattform gehören zahlreiche Pakete, die Klassen und Schnittstellen für wichtige Aufgaben der Programmentwicklung (z.B. Zeichenkettenverarbeitung, Netzwerkverbindungen, Datenbankzugriffe) enthalten. Die Zusammenfassung dieser Pakete wird oft als **Java-API** (***A**pplication **P**rogramming **I**nterface*) bezeichnet. Allerdings kann man nicht von *dem* Java-API sprechen, denn neben der **Java Standard Edition (JSE)**, auf die wir uns bisher beschränkt haben, bietet die Firma Oracle noch weitere Java-APIs an, z.B.:

- Java Enterprise Edition (JEE)
- Java Micro Edition (JME)

In der JDK-Dokumentation zur Standard Edition sind deren Pakete umfassend dokumentiert:[1]

- Klicken Sie nach dem Start auf den Link **Java SE API** (am rechten Rand).
- Im linken oberen Rahmen kann ein spezielles Paket oder die Option **All Classes** gewählt werden, und im linken unteren Rahmen erscheinen alle zur Auswahl gehörigen Typen (die Klassen in normaler und die Interfaces in kursiver Schrift).
- Nach dem Anklicken einer Klasse oder Schnittstelle wird diese im Hauptrahmen ausführlich erläutert (z.B. mit einer Beschreibung der öffentlichen Methoden).

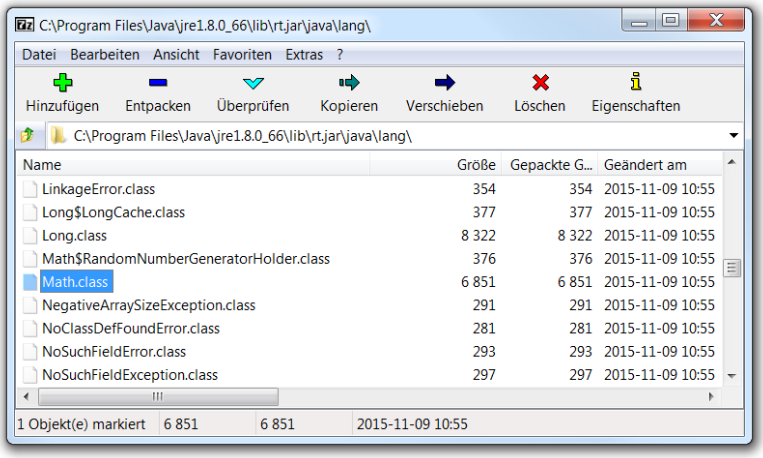
Vermutlich haben Sie schon mehrfach von diesem Informationsangebot Gebrauch gemacht.

Die zum Java-API gehörigen Bytecode-Dateien sind auf mehrere Java-Archivdateien (***.jar**) verteilt, die sich im **lib**-Unterordner der JRE befinden, z.B. in:

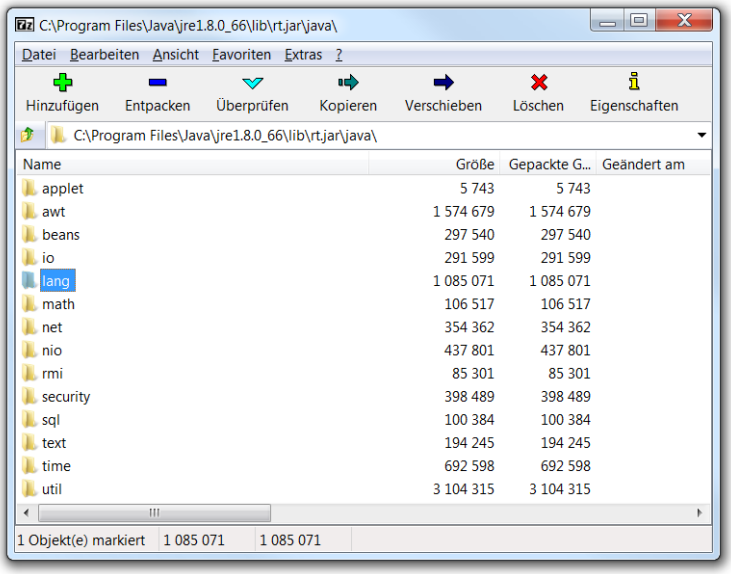
C:\Program Files\Java\jre1.8.0_66\lib

Den größten Brocken bildet die Datei **rt.jar**. Dort befindet sich z.B. das Paket **java.lang**, das u.a. die Bytecode-Datei zur Klasse **Math** enthält, deren Klassenmethoden wir schon mehrfach benutzt haben:

[1] Zur Installation der JDK-Dokumentation siehe Abschnitt 2.1.3



Hier ist das Paket **java** mit seinen Unterpaketen zu sehen:

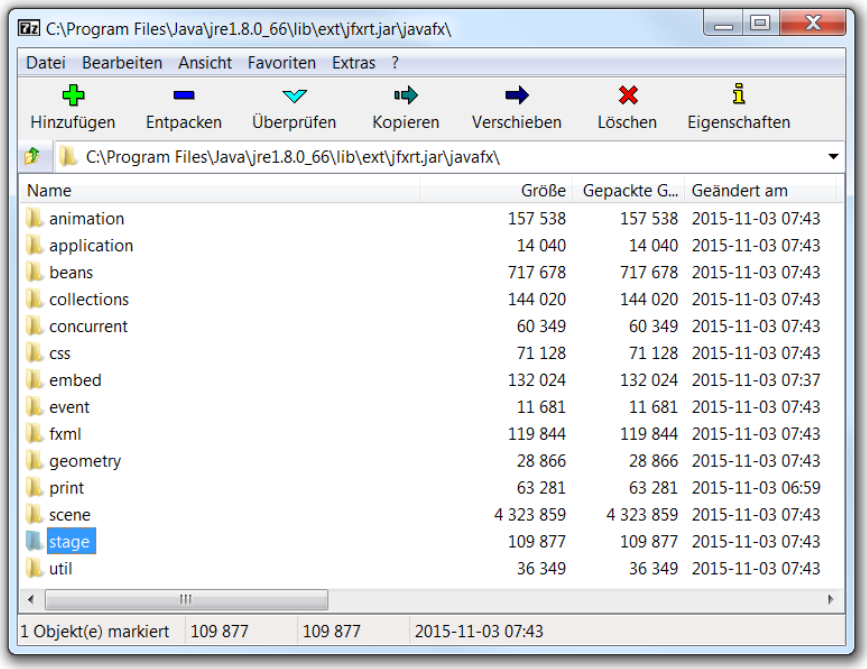


Es folgen kurze Beschreibungen wichtiger Pakete im API der Java Standard Edition:

- **java.awt**, **java.awt.event**
  Das Paket **java.awt** (***A**bstract **W**indowing **T**oolkit*) enthält traditionelle Typen zur Gestaltung von grafischen Bedienoberflächen. Diese wurden schon vor vielen Jahren durch die Komponenten im aktuelleren Swing-Paket abgelöst, das seit Java 8 seinerseits durch JavaFX ersetzt wird. Das Paket **java.awt.event** enthält Klassen zur Ereignisverwaltung, die für Komponenten aus den Paketen **java.awt** und **javax.swing** relevant sind.
- **java.beans**
  Dieses Paket enthält Typen zum Programmieren von Java-Komponenten (*beans* genannt).
- **java.io**, **java.nio**
  Mit Klassen aus diesen Paketen werden wir in Dateien schreiben und aus Dateien lesen.
- **java.lang**
  Dieses Paket mit fundamentalen Typen (z.B. **Object**, **System**, **String**) wird vom Compiler automatisch in jede Quellcodedatei importiert, so dass seine Typen generell ohne Paketnamen angesprochen werden können.

- **java.math**
  Das Paket enthält u.a. die Klassen **BigDecimal** und **BigInteger** für Berechnungen mit beliebiger Genauigkeit. Das *Paket* **java.math** darf nicht verwechselt werden mit der *Klasse* **Math** im Paket **java.lang**.
- **java.net**
  Dieses Paket enthält Klassen für die Netzwerkprogrammierung.
- **java.text**
  Hier geht es um die Formatierung von Textausgaben und die Internationalisierung von Programmen.
- **java.util**
  Dieses Paket enthält neben Typen aus dem Java Collection Framework (z.B. **List<E>**, **ArrayList<E>**) wichtige Hilfsmittel wie den Pseudozufallszahlengenerator **Random**.
- **javax.swing**
  Im Paket **javax.swing** sind GUI-Klassen enthalten, die sich im Unterschied zu den Klassen im Paket **java.awt** kaum auf die GUI-Komponenten des jeweiligen Betriebssystems stützen, sondern eigene Steuerelemente realisieren, was eine höhere Flexibilität und Portabilität mit sich bringt. Seit Java 8 ist Swing als bevorzugte GUI-Lösung von JavaFX abgelöst worden.

Im **lib**-Unterordner der JRE-Installation befinden sich neben **rt.jar** noch weitere Java-Archivdateien mit Paketen. Besonders wichtig ist die im Ordner **…\lib\ext** abgelegte Datei **jfxrt.jar** mit den Paketen der neuen GUI-Technologie JavaFX. Hier befinden sich z.B. die bereits erwähnten Pakte **javafx.stage** und **javafx.scene**:



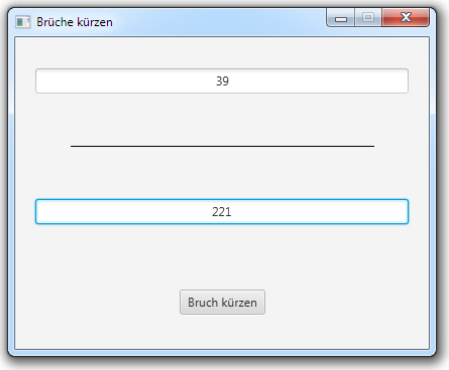
6.6 Übungsaufgaben zu Kapitel 6

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();
 }
}
```

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) Erstellen Sie eine ausführbare JAR-Datei mit dem in Abschnitt 4.8 erstellten Programm zum Kürzen von Brüchen:



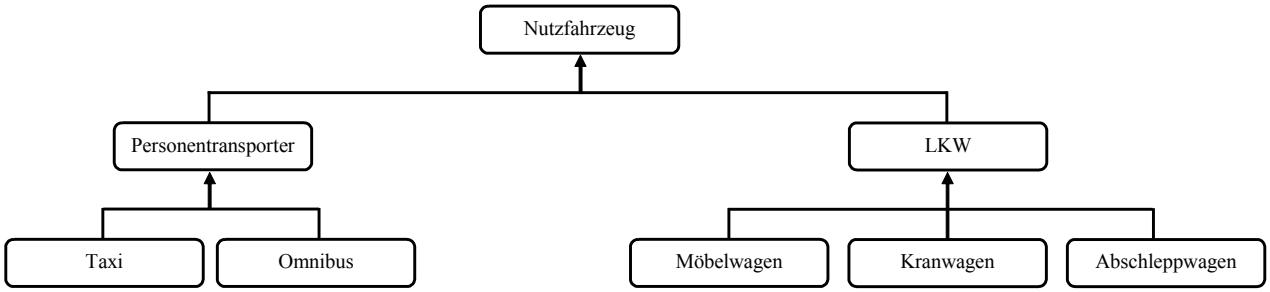
Verwenden Sie die bequeme Erstellung der JAR-Datei mit Eclipse (vgl. Abschnitt 6.4.5).

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 Softwareentwicklung im Detail.

Modellierung realer Klassenhierarchien

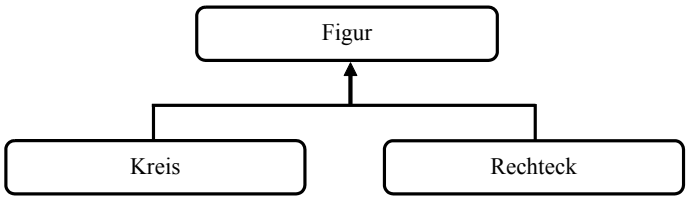
Beim Modellieren eines Gegenstandbereiches durch Klassen, die durch Eigenschaften (Instanz- und Klassenvariablen) sowie Handlungskompetenzen (Instanz- und Klassenmethoden) gekennzeichnet sind, müssen auch die Spezialisierungs- bzw. Generalisierungsbeziehungen zwischen real existierenden Klassen abgebildet 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 der Fahrgäste, maximale Anhängelast). Ebenso sind einige Handlungsmöglichkeiten bei allen Nutzfahrzeugen vorhanden (z.B. eigene Position melden, ein Ziel ansteuern), während andere speziellen Fahrzeugen vorbehalten sind (z.B. Fahrgäste befördern, Lasten transportieren). Ein Programm zur Einsatzplanung und Verwaltung des Fuhrparks sollte diese Klassenhierarchie abbilden.

Übungsbeispiel

Bei unseren Beispielprogrammen bewegen wir uns in einem bescheideneren Rahmen und betrachten eine einfache Hierarchie mit Klassen für geometrische Figuren:[1]



[1] 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 in Abschnitt 7.9 über das *Liskovsche Substitutionsprinzip* genau diese Ableitungen (von Kreis aus Ellipse bzw. von Quadrat aus Rechteck) kritisieren. Daher ist es akzeptabel, an Stelle der Ellipse den Kreis neben das Rechteck zu stellen, um das Erlernen der neuen Konzepte durch ein möglichst einfaches Beispiel zu erleichtern.

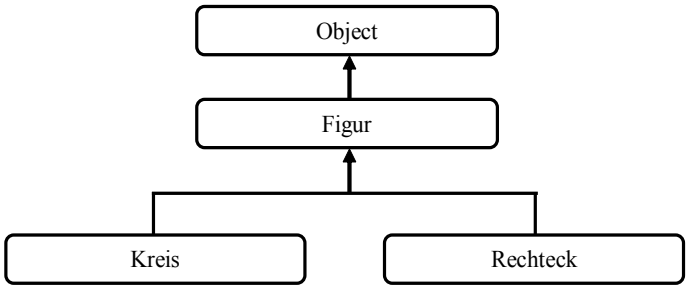
Die Vererbungstechnik der OOP

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 und Methoden 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 Programmiersprachen (wie z.B. C++) erlaubte **Mehrfachvererbung ausgeschlossen**, so dass jede Klasse (mit Ausnahme von **Object**) genau eine Basisklasse hat.

Software-Recycling

Mit ihrem Vererbungsmechanismus bietet die objektorientierte Programmierung ideale Voraussetzungen dafür, vorhandene Software auf rationelle Weise zur Lösung neuer Aufgaben zu verwenden. Dabei können allmählich umfangreiche Softwaresysteme entstehen, die gleichzeitig robust und wartungsfreundlich sind. Die (leider auch im Kurs einmal verwendete) Praxis, vorhandenen Code per *Copy & Paste* in neuen Projekten bzw. Klassen zu verwenden, hat gegenüber einer sorgfältig geplanten Klassenhierarchie offensichtliche Nachteile. Natürlich kann auch Java nicht garantieren, dass jede Klassenhierarchie exzellent entworfen ist und langfristig von einer stetig wachsenden Entwicklergemeinde eingesetzt wird.

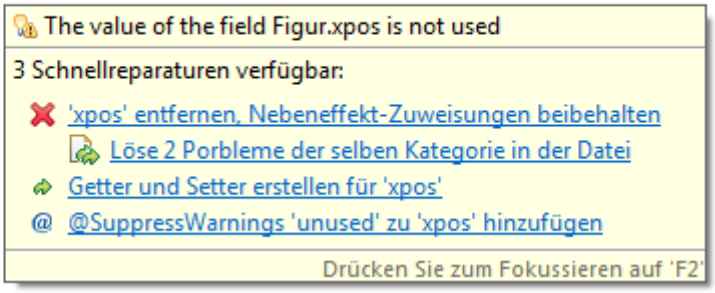
7.1 Definition einer abgeleiteten Klasse

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() {}
}
```

Eclipse erkennt, dass die privaten Instanzvariablen `xpos` und `ypos` aktuell nutzlos sind:

```
public class Figur {
    private double xpos = 100.0, ypos = 100.0;
```



Dieser Zustand wird sich aber bald ändern.

Mit Hilfe des Schlüsselwortes **extends** wird 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() {}
}
```

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 Abschnitt 7.4 werden wir uns mit einigen Regeln für **super**-Konstruktoren beschäftigen.

In der `Kreis`-Klasse wird (wie in der Basisklasse `Figur`) auch ein parameter*freier* Konstruktor definiert. Vielleicht hat jemand gehofft, die `Kreis`-Klasse könnte den parameterfreien Konstruktor ihrer Basisklasse (bei Anpassung des Namens) übernehmen. Konstruktoren werden jedoch grundsätzlich **nicht** vererbt.

Das folgende Programm erzeugt ein Objekt aus der Basisklasse und ein Objekt aus der abgeleiteten Klasse:

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`

Gelegentlich gibt es Gründe dafür, eine Klasse mit dem Modifikator **final** zu deklarieren, so dass sie zwar verwendet, aber nicht beerbt werden kann. Bei der Klasse **String** im API-Paket **java.lang**

```
public final class String
```

ist das Finalisieren erforderlich, damit keine abgeleitete Klasse die Unveränderlichkeit von **String**-Objekten (vgl. Abschnitt 5.2.1) unterlaufen kann.

7.2 Mehrfachvererbung

In Java ist **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 so genannten *Deadly Diamond of Death* bei der Mehrfachvererbung).

Einen gewissen Ersatz bieten die in 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 Methoden 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. Auf das `xpos`-Feld eines `Figur`-Objekts könnte eine Methode der `Kreis`-Klasse *nicht* direkt zugreifen.

Während Objekte aus abgeleiteten Klassen ihre geerbten **protected**-Elemente direkt ansprechen können, haben andere paketfremde fremde Klassen auf Elemente mit dieser Schutzstufe grundsätzlich *keinen* Zugriff:

```
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 super-Konstruktoren 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 Konstruk-

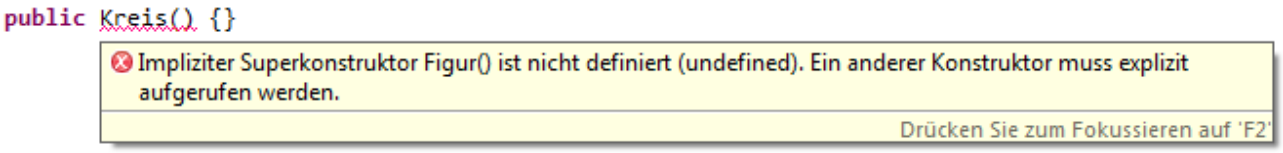
[1] Falls Sie sich über die Berechnung des Kreismittelpunkts wundern: In der Computergrafik ist die Position (0, 0) in der *oberen* linken Ecke des Bildschirms bzw. des aktuellen Fensters angesiedelt. Die X-Koordinaten wachsen (wie aus der Mathematik gewohnt) von links nach rechts, während die Y-Koordinaten von oben nach unten wachsen. Wir wollen uns im Hinblick auf die in absehbarer Zukunft anstehende Programmierung grafischer Bedienoberflächen schon jetzt daran gewöhnen.

tors 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 Compiler. Um die folgende Fehlermeldung des JDK-Compilers zu provozieren, wurde in der Klasse `Figur` der parameterfreie Konstruktor auskommentiert:

```
Kreis.java:11: error: constructor Figur in class Figur cannot be
applied to given types;
        public Kreis() {}
                       ^
```

Eclipse liefert eine bessere Problembeschreibung:



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 der abgeleiteten Klasse aufgerufen, so dass 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. 2015, 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 gemäß Deklaration.
  - Der Rumpf des Konstruktors wird ausgeführt.

Betrachten wir als Beispiel 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 `Figur(150.0, 200.0)` 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 angeleiteten 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 deklariert wird. In diesem Fall sind die beiden Signaturen verschieden, und es handelt sich um eine Über*ladung*.

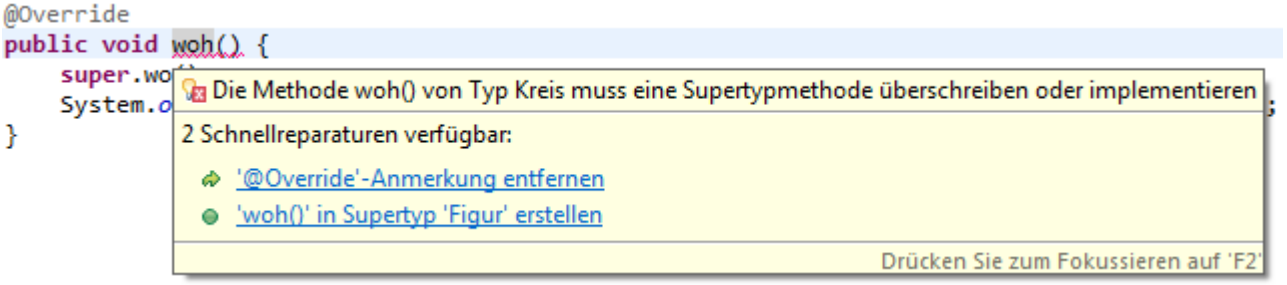
Um das Überschreiben demonstrieren zu können, erweitern wir die `Figur`-Basisklasse um eine Methode namens `wo()`, welche die Position der linken oberen Ecke ausgibt:

```
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 kann eine bessere Ortsangabenmethode realisiert werden, 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, warnt der Compiler:



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 Test {` `  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 Empfehlung wird in der folgenden Klasse `Mint` (ein **int**-Wrapper, siehe Übungsaufgabe zu Abschnitt 5.3) umgesetzt:

```
public class Mint {
  public int val;

  public Mint(int val_) {
    val = val_;
  }

  public Mint() {}

  @Override
  public String toString() {
    return String.valueOf(val);
  }
}
```

Ein `Mint`-Objekt antwortet auf die **toString**() - Botschaft mit der Zeichenfolgendarstellung des gekapselten **int**-Werts:

Quellcode	Ausgabe
`class Test {` `  public static void main(String[] args) {` `  Mint zahl = new Mint(4711);` `  System.out.println(zahl);` `  }` `}`	`4711`

Den Versuch, eine Instanzmethode der Basisklasse durch eine *statische* Methode der abgeleiteten Klasse zu überschreiben, verhindert der Compiler.

7.5.2 Überdecken von statischen Methoden

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.

Es ist selbstverständlich möglich, in einer abgeleiteten Klasse eine statische Methode zu definieren, welche die Signatur einer Basisklassenmethode besitzt (selber Name und selbe Parameterliste). Zur eben beschriebenen späten Entscheidung durch das Laufzeitsystem kann es aber nicht kommen, weil man sich beim Aufruf einer statischen Methode an eine *Klasse* richtet und prinzipiell dem Methodennamen den Klassennamen voranstellt. Die auszuführende *statische* Methode steht grundsätzlich schon zur Übersetzungszeit fest, und man spricht hier vom *Überdecken* oder *Verstecken* der Basisklassenmethode. Dabei bleibt die Basisklassenvariante durch Voranstellen des Klassennamens in den Methoden der abgeleiteten Klasse weiterhin ansprechbar, z.B.:

```
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() {
  Figur.sm();
  System.out.println("Statische Kreis-Methode");
 }
   .  .  .
}
```

Das Schlüsselwort **super** ist im statischen Kontext *nicht* erlaubt.

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 7.1 wissen, kann man für eine Klasse generell verbieten, abgeleitete Klassen zu definieren. Finalisiert man statt der gesamten Klasse eine Methode, darf zwar eine abgeleitete Klasse definiert, dabei aber das finalisierte Erbstück 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 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 den spezialisierten 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 Laufzeitsystem* 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	Ausgabe
`// Datei General.java` `class General {` `  String x = "x-Gen";` `  void gm() {` `    System.out.println("x in gm():\t\t "+x);` `  }` `}` `// Datei Spezial.java` `class Spezial extends General {` `  int x = 333;` `  void sm() {` `   System.out.println("x in sm().:\t\t "+x);` `   System.out.println("\nsuper-x in sm():\t "+` `                      super.x);` `  }` `}` `// Datei Test.java` `class Test {` `  public static void main(String[] args) {` `   Spezial sp = new Spezial();` `   sp.gm();` `   sp.sm();` `  }` `}`	`x in gm():  x-Gen` `x in sm():  333` `super-x in sm():  x-Gen`

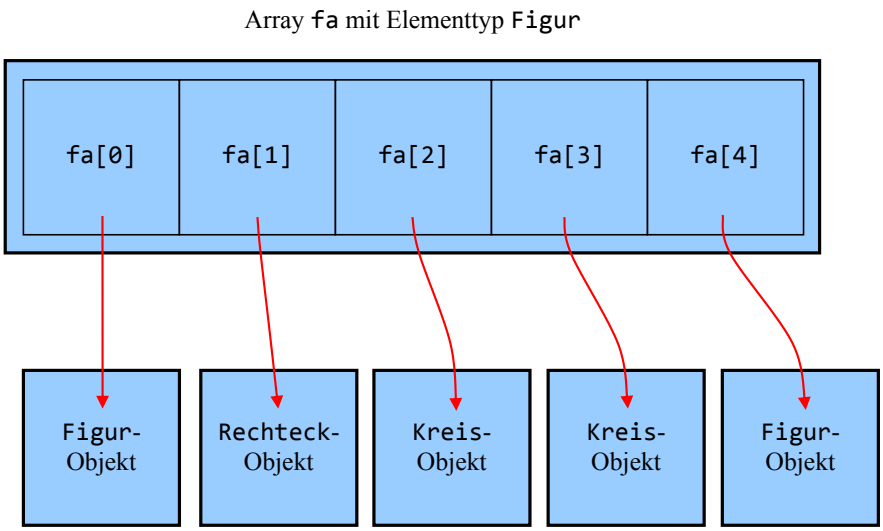
Während das Überschreiben von Methoden oft von entscheidender Bedeutung bei der Entwicklung guter Lösungen ist, finden sich für das potentiell verwirrende Überdecken von Feldern nur wenige sinnvolle Einsatzzwecke.

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. Angewendet auf das Beispiel in Abschnitt 7.1 ergibt sich die sehr plausible Feststellung, dass jeder Kreis auch eine Figur ist.

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 Grafik-Programms 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:



Das folgende Programm verwaltet Referenzen auf Figuren und Kreise in einem Array vom Typ `Figur`:

Quellcode	Ausgabe
`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");` `  }` `}`	`Figur 0: kein Kreis` `Figur 1: Radius = 25.0` `Figur 2: kein Kreis` `Figur 3: Radius = 10.0`

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 in Abschnitt 7.5.1 die zusätzliche Methode `gibRadius()` erhalten:

```
public int gibRadius() {
 return radius;
}
```

7.7 Polymorphie

Werden Objekte aus verschiedenen Klassen über Referenzvariablen eines gemeinsamen Basistyps verwaltet, 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:

```
import de.uni_trier.zimk.figuren.*;

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();
  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 breite Datentyp `Figur` genutzt, dann führen die zu diversen `Figur`-Unterklassen gehörigen Objekte bei einem Methodenaufruf ihr artspezifisches Verhalten aus. Später können neue `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 Unterklassen über Referenzvariablen von einem Basisklassentyp nutzen und dabei artgerecht realisierte Methodenaufrufe realisieren zu können, müssen die betroffenen Methoden in der Basisklasse vorhanden sein. Wenn es für die Basisklasse zu einer Methode keine sinnvolle Implementierung gibt, erstellt man dort eine **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 Rechtseck: (" + (xpos+breite) +
        ", " + (ypos+hoehe) + ")");
 }

 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.

Außerdem eignet sich eine abstrakte Klasse 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(), fa[i].meldeInhalt());
   ges += fa[i].meldeInhalt();
  }
  System.out.printf("\nGesamtflaeche:  %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

Gesamtflaeche:    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 Vertiefung: Das Liskovsche Substitutionsprinzip (LSP)

In diesem Abschnitt geht es um eine etwas formale, aber keinesfalls praxisfremde Vertiefung zum Thema *Vererbung*. 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 Perspektive 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 müssen. Um das Auftreten irregulärer 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 z.B. 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 Analyse und Programmierung, kann aber z.B. eine Verletzung des Substitutionsprinzips nicht verhindern.

7.10 Übungsaufgaben zu Kapitel 7

1) Warum kann der folgende Quellcode (mit zwei Klassen im Standardpaket) nicht übersetzt werden?

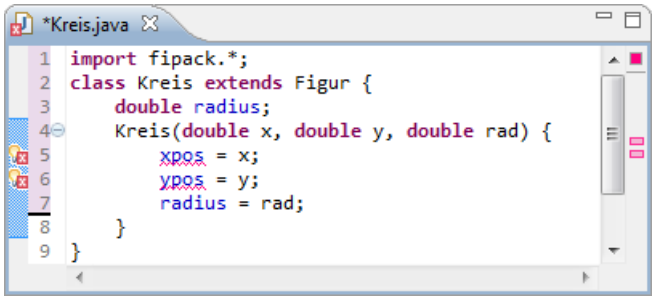
```
// Datei General.java
class General {
   int ig;
   General(int igp) {
      ig = igp;
   }
   void hallo() {
      System.out.println("hallo-Methode der Klasse General");
   }
}
```

```
// 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 von den 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 (z.B. per Klassensuchpfad) dafür zu sorgen, dass die aktualisierte Basisklasse von der JRE 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, so dass der Compiler für Typsicherheit sorgen, d.h. die Zuweisung ungeeigneter Werte bzw. Objekte verhindern kann. Andererseits 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 Definition 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 zur Verwaltung von Listen, Mengen oder Schlüssel-Wert - Tabellen (Abbildungen) im Java Collection Framework, das in 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] (2008, 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 Container mit ihrer weitgehend inhaltstypunabhängigen Verwaltungslogik beschränkt.

8.1.1 Vorzüge und Verwendung generischer Klassen

8.1.1.1 Veraltete Technik mit Risiken und Umständlichkeiten

In Abschnitt 5.3.2 haben Sie die Klasse **ArrayList** aus dem Paket **java.util** als Container für Objekte beliebigen Typs kennen gelernt, 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 in Java noch unterstützte, so genannte *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.

---

[1] Joshua Bloch hat nicht nur ein lesenswertes Buch über Java verfasst, sondern auch viele Klassen im Java-API programmiert und insbesondere das Java Collection Framework entworfen.

Im Unterschied zu einem Java-Array (siehe Abschnitt 5.1) bietet die Klasse **ArrayList** bei der eben vorgeführten Verwendungsart:

- eine automatische Größenanpassung
- Typflexibilität bzw. -beliebigkeit

In der Praxis ist oft ein Container mit automatischer Größenanpassung (ein dynamischer Array) für Objekte eines bestimmten, *identischen* Typs gefragt (z.B. zur Verwaltung von **String**-Objekten). Bei dieser Einsatzart stören zwei Nachteile der Typbeliebigkeit:

- Wenn 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 Benutzer) entdeckt.
- 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) {
  // Bitte nur String-Objekte einfüllen!
  ArrayList al = new ArrayList();
  al.add("Otto");
  al.add("Rempremerding");
  al.add('.');
  int i = 0;
  for (Object s: al)
   System.out.printf("Laenge von String %d: %d\n", ++i, ((String)s).length());
 }
}
```

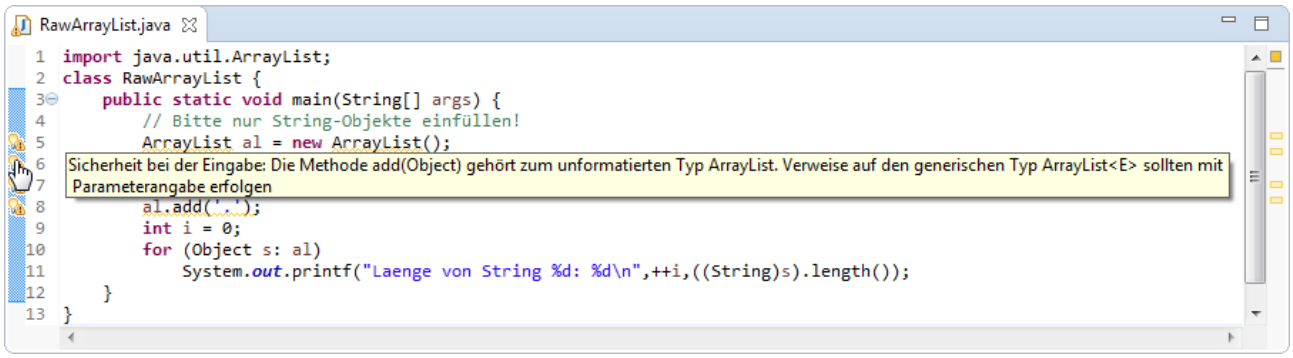
Bevor ein **String**-Element des Containers nach seiner Länge befragt werden kann, ist eine lästige Typanpassung fällig, weil der Compiler nur die Typzugehörigkeit **Object** kennt:

```
((String)s).length()
```

Beim dritten **add() -** Aufruf wird ein **Character**-Objekt (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 bemerken 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.lang.Character
cannot be cast to java.lang.String
 at RawArrayList.main(RawArrayList.java:11)
```

Unsere Entwicklungsumgebung Eclipse erkennt das sich anbahnende Unglück und warnt:



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 wird die gesamte Handlungskompetenz der Klasse typunabhängig formuliert. Bei jedem Instanzieren wird der Typ jedoch konkretisiert, so dass Typsicherheit und syntaktische Eleganz resultieren.

Wie ein Blick in die API-Dokumentation zeigt, ist die Klasse **ArrayList<E>** selbstverständlich generisch definiert und verwendet den Typformalparameter **E**:

java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

Der in 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 (Typaktualparameter) **String** verwendet,

```
import java.util.ArrayList;
class GenArrayList {
 public static void main(String[] args) {
  ArrayList<String> al = new ArrayList<>();
  al.add("Otto");
  al.add("Rempremerding");
  // al.add('.');  // führt zum Übersetzungsfehler
  int i = 0;
  for (String s: al)
   System.out.printf("Laenge von String %d: %d\n", ++i, s.length());
 }
}
```

dann verhindert der Compiler die Aufnahme eines Elements mit unpassendem Datentyp:

Die Elemente des auf **String**-Objekte eingestellten **ArrayList**-Containers beherrschen ohne Typanpassung die Methoden ihrer Klasse.

Generische Klassen ermöglichen robuste Programme (dank Typüberwachung durch den Compiler), die zudem leicht lesbar sind.

Bei der Verwendung eines generischen Typs durch Wahl konkreter Datentypen an Stelle der Typformalparameter entsteht ein so genannter **parametrisierter Typ**, z.B. **ArrayList<String>**.

Als Konkretisierung für einen Typformalparameter ist ein *Referenztyp* vorgeschrieben. Den Grund für diese Einschränkung erfahren Sie in 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, so dass 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.[1]

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 ausschließlich den so genannten **Rohtyp**. Hier sind Typformalparameter 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 JRE nichts von dieser Technik. Die damit fälligen expliziten Typanpassungen 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 die-

[1] Manche Autoren bezeichnen das Paar spitzer Klammern in laxer Redeweise als *diamond operator*, obwohl es sich *nicht* um einen Operator handelt.

sem Datentyp bestehen. Aus einem unrestringierten Typformalparameter resultiert im Bytecode der Typ **Object**, z.B. bei der Deklaration von Variablen. Solchen Variablen können aber keinen Wert mit primitivem Datentyp aufnehmen. Dies hat zur Folge, dass zur Konkretisierung von Typformalparametern nur Referenztypen erlaubt sind. Primitive Typen sind also durch die zugehörige Wrapper-Klasse zu ersetzen.

Auf ihre Klassenzugehörigkeit befragt, nennen Objekte eines parametrisierten Typs stets den zugehörigen Rohtyp, z.B.:

Quellcodefragment	Ausgabe
`public static void main(String[] args) {` `  ArrayList<String> al = new ArrayList<>();` `  System.out.println(al.getClass());` `}`	`class java.util.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, so dass z.B. die folgende Anweisung *nicht* übersetzt werden kann:

```
System.out.println(al instanceof ArrayList<String>);
```

Cannot perform instanceof check against parameterized type ArrayList<String>. Use the form ArrayList<?> instead since further generic type information will be erased at runtime

Drücken Sie zum Fokussieren auf 'F2'

Hier ist eine Ausnahme zu machen von der Regel, den Rohtyp im Quellcode zu vermeiden, z.B.:

Quellcodefragment	Ausgabe
`System.out.println(al instanceof ArrayList);`	`true`

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<?>);
```

Als die Generizität in Java eingeführt wurde, existierte die Programmiersprache bereits ca. 10 Jahre, sodass die Interoperabilität mit alten Java-Lösungen einen sehr hohen Stellenwert besaß und die aus heutiger Sicht suboptimale Designentscheidung mit Typlöschung und Rohtyp erzwungen hat.

Nun müssen Programmierer 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:

```
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());
}
```

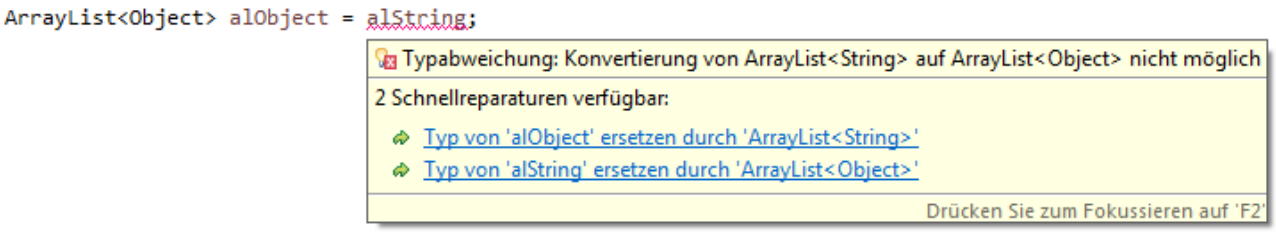
Ein Aufruf dieser **main()** - Methode führt zu einer **ClassCastException**, weil das eingeschmuggelte **Integer**-Objekt (Autoboxing!) keine **length()** - Methode beherrscht:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot
  be cast to java.lang.String t Prog.main(Prog.java:8)
```

Ist explizit ein „Gemischtwaren“ - Container gewünscht, sollte trotzdem kein Rohtyp verwendet werden, sondern eine Konkretisierung mit dem Elementtyp **Object**, z.B.:

```
ArrayList<Object> alObject = new ArrayList<Object>();
```

Bei einer Referenzvariablen vom Typ **ArrayList<Object>** kann der eben beschriebene Fehler nicht auftreten:



Weil man es nicht oft genug sagen kann, steht am Ende des Abschnitts noch einmal in den Worten von Joshua Bloch (2008, S. 109) der dringende Rat:

Don’t use raw types in new code.

8.1.2.2 Spezialisierungsbeziehungen bei parametrisierten Klassen und Arrays

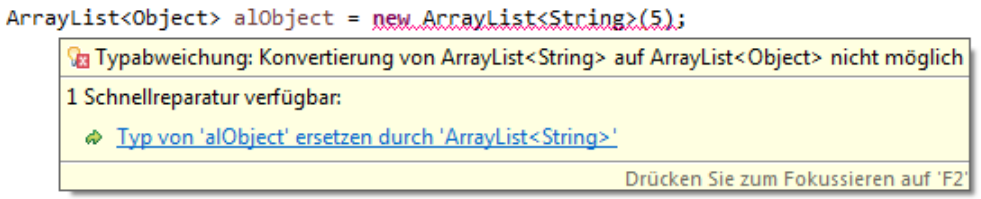
Bei der ersten Beschäftigung mit generischen Klassen könnte man z.B. den parametrisierten Datentyp

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> slObject = 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 für die Übertragung der Spezialisierungsrelation von der Element- auf die Container-Ebene haben sich die folgenden Begriffe eingebürgert (siehe z.B. Bloch 2008, S. 119):

- Generischen Typen sind **invariant**.
- Arrays sind **kovariant**.

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:10)
```

Der per

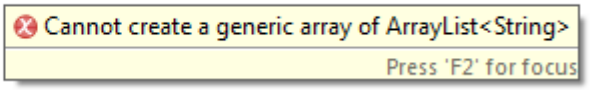
```
new String[5]
```

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 Verbot von Arrays mit einem parametrisierten Elementtyp

Wegen der Typlöschung bei generischen Klassen und der Kovarianz von Arrays passen in Java die Generizität und Arrays schlecht zusammen (Bloch 2008, S. 119). Z.B. lässt sich kein Array mit einer konkretisierten Variante des generischen Typs **ArrayList<E>** als Elementtyp erstellen:

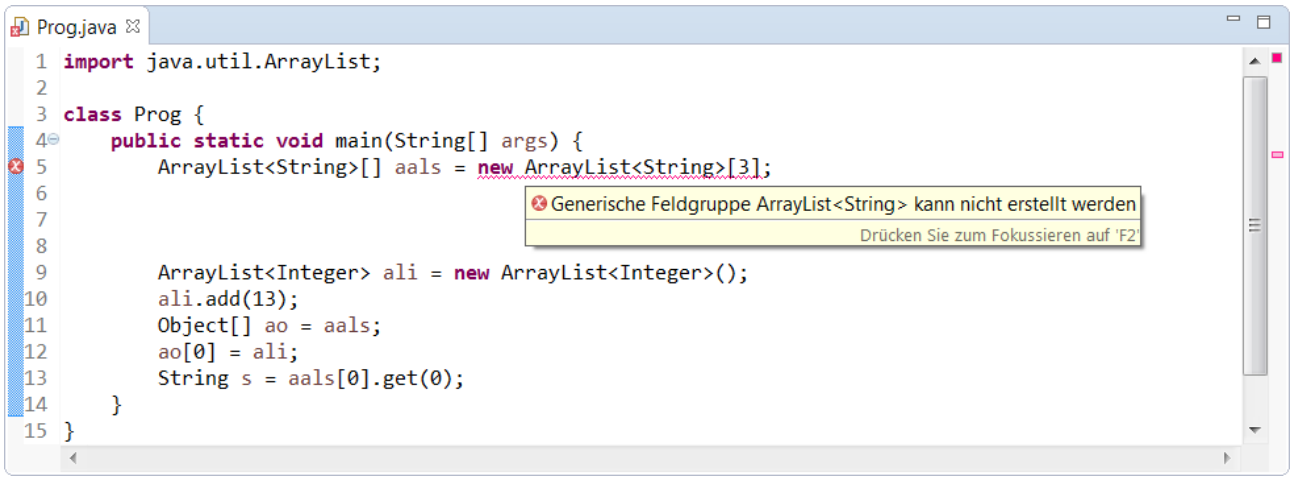
```
ArrayList<String>[] aals = new ArrayList<String>[100];
```



Wer an einer näheren Erläuterung der Ursachen für das Verbot der generischen Array-Kreation momentan kein Interesse hat, 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] Das Beispiel wurde übernommen von Bloch (2008, S. 120) bzw. Flanagan (2005, S. 166), wo es in weitgehend identischer Form zu finden ist.



Würde der Compiler die Anweisung in Zeile 5 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[]**, so dass der Compiler die Zeile 11 nicht beanstandet.
- In Zeile 12 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 12 kein Problem machen (keine **ArrayStoreException** verursachen), obwohl der Array `ao` sehr wohl wüsste, dass seine Elemente vom Rohtyp **ArrayList** sind. Schließlich hat das eingefügte Element `ali` ja genau diesen Rohtyp.
- In der Zeile 13 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 2008, S. 120; siehe auch Gosling et al. 2015, 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.

8.1.3 Definition von generischen Klassen

Wie Sie in 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.

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 als einfaches Beispiel eine generische Klasse namens `SimpleList<E>`, die hin-

sichtlich Einsatzzweck und Konstruktion den Listenverwaltungsklassen aus dem Java Collection Framework ähnelt (z.B. **ArrayList<E>**, siehe oben und Abschnitt 10.4), aber nicht annähernd denselben Funktionsumfang bietet:

```
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, z.B.:

T Type
E Element
R Return Type
K Key
V Value

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 in 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];
```

Generische Feldgruppe E kann nicht erstellt werden
Drücken Sie zum Fokussieren auf 'F2'

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 (Bloch 2008, S. 125).

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();
```

Der Typ E kann nicht instanziert werden
Drücken Sie zum Fokussieren auf 'F2'

Wir müssen bei der Array-Kreation als Elementtyp den generellsten zulässigen (nichtgenerischen) Datentyp für Konkretisierungen von **E** zu verwenden: den Urahntyp **Object**.

Beim Datentyp für die Referenzvariable `elements` haben wir zwei, letztlich äquivalente Alternativen, die an unterschiedlichen Stellen der Klassendefinition Typumwandlungen erfordern, deren Korrektheit der Compiler *nicht* sicherstellen kann:

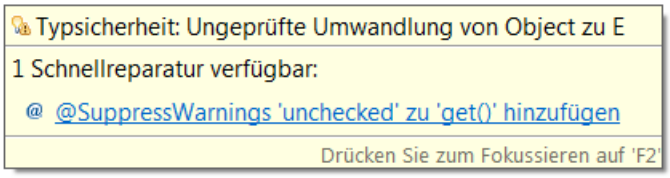
- **Object[]**
- **E[]**

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];
```

Der Compiler warnt vor einer ungeprüften Umwandlung, weil er die Typsicherheit nicht garantieren kann:

```
return (E) elements[index];
```



Bloch (2008, S. 126) beschreibt die Situation treffend so:

> The compiler may not be able to prove that your program is typesafe, but you can.

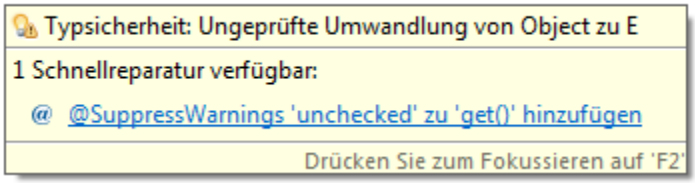
Im aktuellen Beispiel kann ausgeschlossen werden, dass ein Element in den privaten Array `elements` gelangt, das nicht vom Typ **E** ist. Folglich kann bei der Typwandlung in `get()` nichts schief gehen.

Nachweislich *irrelevante* Warnungen sollten abgeschaltet werden, damit wir uns nicht daran gewöhnen, Warnungen zu ignorieren, und damit in der Entwicklungsumgebung bei einem korrekten Projekt ein sauberer Status zu sehen ist. Um den Compiler anzuweisen, eine Warnung zu unterlassen, fügt man eine sogenannte **Annotation** vom Typ **SuppressWarnings** ein (siehe Abschnitt 9.5), die sich u.a. auf eine Variable, eine Methode oder eine Klasse beziehen kann, z.B.:

```
@SuppressWarnings("unchecked")
// Casting erforderlich, weil kein Array vom Typ E erstellt werden.
public E get(int index) {
   if (index >= 0 && index < size)
      return (E) elements[index];
   else
      return null;
}
```

Eclipse unterstützt uns beim Erstellen einer Annotation:

```
return (E) elements[index];
```



Das Unterdrücken von Warnungen sollte natürlich 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.
    @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:

```
elements = (E[]) new Object[DEF_INIT_SIZE];
```

Die eben beschriebene, letztlich gleichwertige Technik wird in Abschnitt 8.1.3.2 bei einem vergleichbaren Beispiel demonstriert.

Den Rohtyp zu `SimpleList<E>` kann man sich ungefähr so vorstellen:

```
import java.util.Arrays;

public class SimpleList {
  private Object[] elements;
  private final static 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 Problemen 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	Ausgabe
`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));` `}`	`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:

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 dem wichtigen Begriff *Interface*, mit dem wir uns in 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()** besitzen (hier beschrieben unter Verwendung des Typformalparameters **E**):

public int compareTo(E *vergl***)**

Das angesprochene Objekt vergleicht sich mit dem Parameterobjekt.

In 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 einem implementierenden Typ verlangt: Objekte des Typs 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:

```
import java.util.Arrays;

public class SimpleSortedList<E extends Comparable<E>> {
  private E[] elements;
  private static final int DEF_INIT_SIZE = 16;
  private int initSize;
  private int size;

  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];
    }
  }

  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 Sie 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.

Weil der Typformalparameter **E** in der Beschränkungsdefinition selbst auftaucht,

```
E extends Comparable<E>
```

spricht Bloch (2008, S. 132) hier 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 einer 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> {
 private Comparable[] elements;
   .  .  .
 public SimpleSortedList() {
  initSize = DEF_INIT_SIZE;
  elements = new Comparable[DEF_INIT_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 in Abschnitt 8.1.3.1 (bei der Klasse `SimpleList<E>`) die erste Technik zum Einsatz kam (allerdings 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:

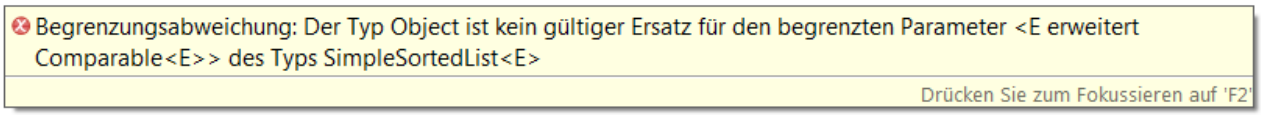
Quellcode	Ausgabe
`class SimpleSortedListTest {` `  public static void main(String[] args) {` `  SimpleSortedList<Integer> si =` `  new SimpleSortedList<>(3);` `  si.add(4); si.add(11); si.add(1); si.add(2);` `  System.out.println("Länge: "+si.size()+` `  " Kapazität: "+si.capacity());` `  for (int i=0; i < si.size();i++)` `  System.out.println(si.get(i));` `  }` `}`	`Länge: 4 Kapazität: 6` `1` `2` `4` `11`

Der Compiler stellt sicher, dass der Container sortenrein bleibt:

```
si.add("Verboten!");
```

Außerdem verhindert er das Konkretisieren des Typparameters durch eine Klasse, welche die Typrestriktion nicht erfüllt, z.B.:

```
SimpleSortedList<Object> so = new SimpleSortedList<>(3);
```



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, so dass 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:

Quellcode	Ausgabe

```
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));
 }
}
```

```
String-max:     def
int-max:        4711
```

In der Definition einer generischen Methode befindet sich vor dem Rückgabetyp zwischen spitzen Klammern mindestens ein Typformalparameter. Mehrere Typparameter werden durch Kommata getrennt. 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 in Abschnitt 8.1.2.1 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 Parameterobjekte 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 einfach im Bytecode. Die gelöschten Typkonkretisierungen werden vom Compiler durch explizite Typumwandlungen ersetzt.

Bei generischen Methoden sind Überladungen erlaubt, auch unter Beteiligung von gewöhnlichen Methoden, z.B.:

Quellcode	Ausgabe
`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));` `  }` `}`	`String-max:  def` `int-max:  4711`

Der Compiler ermittelt zu einem konkreten Aufruf die am besten passende Methode und beschwert sich bei 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 normalen 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):

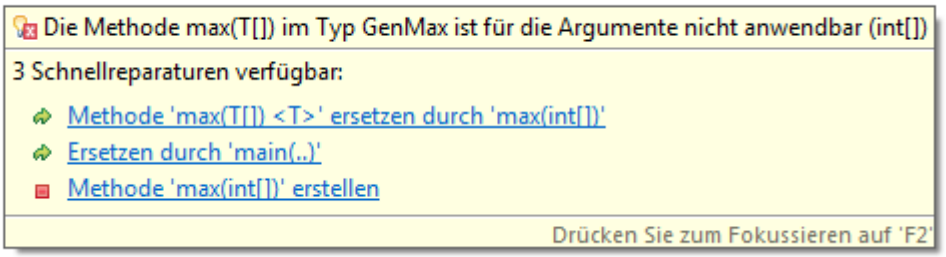
```
Zeit für 1000000 Aufrufe der generischen Methode:     24
Zeit für 1000000 Aufrufe der traditionellen Methode:  3
```

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[] aa) {
  . . .
}
```

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\t"+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 primitive Typen unterstützen, muss man entsprechende Überladungen erstellen.

Bei einer generischen Methode, die das maximale Element zu einer beliebig langen Serie von Argumenten zurückgibt,

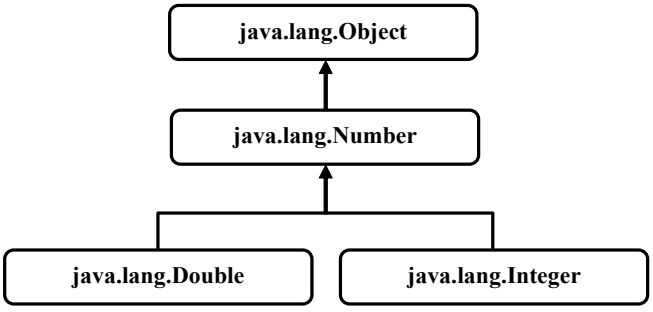
```
public static <T extends Comparable<T>> T max(T... aa) {
  . . .
}
```

klappt die Nutzung durch Argumente mit primitivem Typ hingegen, z.B.:

```
System.out.println("Max. von int-Serie\t"+max(4, 777, 11, 81));
```

8.3 Wildcard-Datentypen

Generische Klassen sind invariant (vgl. Abschnitt 8.1.2.2), so dass 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 bei einem Methodenformalparameter vom Typ `SimpleList<Number>` als Aktualparameter z.B. keine Referenz auf ein Objekt vom Typ `SimpleList<Integer>` zugelassen.[1]

[1] Dies ist auch gut so, weil über eine `SimpleList<Number>` - Referenz auch Fließkommazahlen in ein `SimpleList<Integer>` - Objekt geschmuggelt werden könnten.

Es ist jedoch oft wünschenswert, für einen Methodenparameter einen generischen Datentyp zu verwenden 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-Typ wird die *beschränkte* Variante benötigt, wobei z.B. als Konkretisierungen für einen Typformalparameter eine Basisklasse und deren Ableitungen (Spezialisierungen) 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, so dass die angesprochene Liste alle Elemente einer zweiten, typkompatiblen Liste übernehmen kann. Wir starten mit der folgenden Definition:

```
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<Number>(5);
```

Bei der Einzelelementaufnahme über die Methode

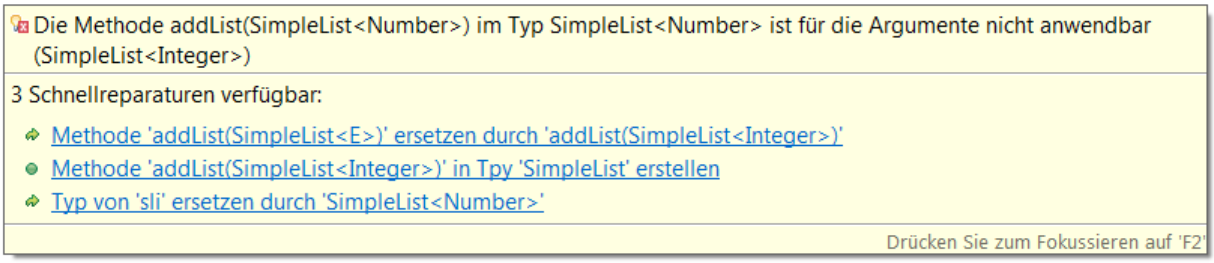
```
public void add(E element) {
  . . .
}
```

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<>(5);
SimpleList<Integer> sli = new SimpleList<>(2);
sli.add(1); sli.add(2);
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	Ausgabe
`class SimpleListWildcardTest {` `  public static void main(String[] args) {` `  SimpleList<Number> sln = new SimpleList<Number>(3);` `  sln.add(13); sln.add(1.13);` `  SimpleList<Integer> sli = new SimpleList<Integer>(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());` `  }` `}`	`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 Wildcard-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 (2008, S. 136) 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.

8.3.1.3 Kompetenzen von Wildcard-Parameterobjekten abrufen

Um die Kompetenzen von Objekten, die in einer Methode per Formalparameter mit Wildcard-Typ bekannt sind, nutzen zu können, muss man dem Compiler etwas auf die Sprünge helfen. Im folgenden Beispiel

```
public class SimpleSortedList<E extends Comparable<E>> {
  . . .
 public void wcTest(SimpleSortedList<? extends E> parlis) {
  E com = parlis.get(0);
  com.compareTo(parlis.get(1));
 }
  . . .
}
```

wird der Methode `wcTest()` aus der Klasse `SimpleSortedList<E extends Comparable<E>>` (vgl. Abschnitt 8.1.3.2) per Formalparameter vom nach oben beschränkten Wildcard-Typ `SimpleSortedList<? extends E>` das Listenobjekt `parlis` übergeben. Seine Elemente beherrschen die Methode **compareTo()**, weil …

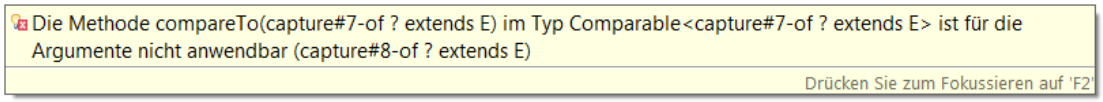
- ihr Typ eine Spezialisierung von E ist,
- und der Typ **E** das Interface **Comparable<E>** implementiert.

Der Compiler akzeptiert, dass die Elemente von `parlis` den Typ **E** erfüllen und fügt in der folgenden Zeile eine implizite Typwandlung ein, um die Typlöschung zu kompensieren:

```
E com = parlis.get(0);
```

Ohne den Umweg über eine lokale **E**-Referenzvariable führt der **compareTo()** - Aufruf an dasselbe Objekt zu einer kryptischen Fehlermeldung:

```
E com = parlis.get(0);
com.compareTo(parlis.get(1));
parlis.get(0).compareTo(parlis.get(1));
```



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 10.8), 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 bei der Methodenformalparameterdeklaration begegnet, und dort 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 10.8) 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 im Zusammenhang mit der Klasse **Collections** erklärt. 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 Kollektionsklasse möglich, welche die Methode **compareTo()** nicht selbst implementiert, sondern von einer Basisklasse erbt. Schließlich ist die Vergleichbarkeit mit Artgenossen auf diese Weise sichergestellt.

Zum Glück sind im Alltag der Softwareentwicklung komplexe Datentypen wie im letzten Beispiel sehr ungewöhnlich.

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 2008, S. 137)

8.4 Einschränkungen der Generizitätslösung in Java

Generische Klassen und Methoden sind für die Software-Entwicklung in Java unverzichtbare und unvermeidbare Bestandteile, doch enthält die Java-Generizitätslösung (z.B. bedingt durch das Bemühen um Kompatibilität) einige Einschränkungen, die abschließend noch einmal behandelt 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<>();
```

In 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 20 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.

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 eob = new E();
```

Eine solche Möglichkeit zur generischen Objektkreation wird allerdings nur sehr selten vermisst.

8.4.2.3 Keine Array-Kreation mit einem per Typformalparameter bestimmten Elementtyp

Wegen der Typlöschung bei generischen Klassen und der Kovarianz von Arrays lässt sich kein Array mit einem generisch bestimmten Elementtyp erstellen, was bei Kollektionsklassen mit Array-Datenablage zu Lücken in der Typsicherheit führt (siehe Abschnitt 8.1.2.3). Davon ist aber nur die *Definition* einer generischen Klasse betroffen, nicht ihre *Verwendung*. In Abschnitt 8.1.2.1 haben wir die generische Kollektionsklasse `SimpleList<E>` definiert und zur internen Verwaltung der Listenelemente einen **Object**-Array verwendet:

```
private Object[] elements;
private final 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, ist daher eine explizite Typumwandlung erforderlich:

```
public E get(int index) {
 if (index >= 0 && index < size)
  return (E) elements[index];
 else
  return null;
}
```

Weil im Rohtyp der Typformalparameter durch die obere Schranke (z.B. **Object**) ersetzt ist, liegt es in der Verantwortung des Klassendesigners, dass die Methode `get()` tatsächlich 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 (z.B. Datenkapselung) hergestellt und die somit irrelevante Warnung unterdrückt (siehe Abschnitt 8.1.2.1).

Die bei `SimpleList<E>` benutzte Lösung wird übrigens auch bei der API-Klasse **ArrayList<E>** verwendet, z.B.:

```
public class ArrayList<E> extends AbstractList<E> {
 transient Object[] elementData; // non-private to simplify nested class access

  . . .
 @SuppressWarnings("unchecked")
 E elementData(int index) {
     return (E) elementData[index];
 }

 @Override
 public E get(int index) {
     rangeCheck(index);
     return elementData(index);
 }

  . . .
}
```

Man beachte die Methode `elementData()` (mit der voreingestellten Schutzstufe *Paket*) die denselben Namen trägt wie das intern zum Speichern der Elemente verwendete **Object[]** - Array. In

dieser Methode 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.

8.4.2.4 Objekte generischer Klassen sind möglich

In einer generischen Klasse oder Methode kann zwar kein Objekt einer per Typformalparameter bestimmten Klasse erstellt werden, doch lässt sich durchaus ein Objekt einer generisch definierten Klasse erstellen. Zu dieser Leistung ist die JRE trotz Typlöschung fähig, weil sie ja nur ein Objekt des Rohtyps zu erstellen hat, den sie sehr wohl kennt. Ein Objekt einer per Typformalparameter bestimmten Klasse kann die JRE hingegen *nicht* erstellen, weil sie die Klasse nicht kennt und folglich nicht weiß, welcher Konstruktor aufzurufen ist.

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:

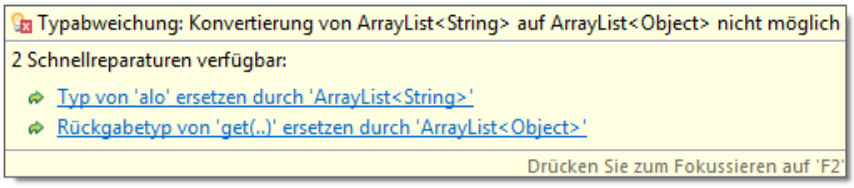
```
import java.util.ArrayList;

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;
 }
}
```

Wird ein Objekt vom parametrisierten Typ `Genni<String>` erstellt, kommt intern erwartungsgemäß ein Objekt vom parametrisierten Typ **ArrayList<String>** zum Einsatz, z.B.:

Quellcode	Ausgabe
`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());` `  }` `}`	`[Otto, Rempremerding]`

Der Compiler kann die Typsicherheit garantieren, z.B.:

```
java.util.ArrayList<Object> alo = gs.get();
```



8.5 Übungsaufgaben zu 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	Ausgabe
`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);` `  }` `}`	`Otto` `Ludwig` `Karl`

Erstellen Sie bitte eine funktionsgleiche generische Methode mit einem Typformalparameter.

2) Das folgende, bei Bloch (2008, S. 12) 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.lang.Integer cannot
 be cast to java.lang.String at Prog.main(Prog.java:10)
```

In der reklamierten Zeile

```
System.out.println(sl.get(0));
```

ist aber gar kein Casting-Operator zu sehen. Wie kommt die Fehlermeldung zu Stande?

3) Erstellen Sie zu der in Abschnitt 8.2 vorgestellten generischen Methode `max()` eine Überladung, die das maximale Element zu einer beliebig langen Serie von Argumenten zurückgibt. Beim Typ des Serienparameters soll nur vorausgesetzt werden, dass er das Interface **Comparable<T>** erfüllt (zum noch nicht offiziell behandelten Begriff Interface siehe Abschnitt 8.1.3.2).

4) Warum sollte man bei Referenzvariablen 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 Referenztyp 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 das Interface als Datentyp vorgeschrieben ist. Auf diese Weise können Interface-Datentypen eine wichtige Rolle bei der Polymorphie spielen.

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. bei der Wrapper-Klasse **java.lang.Double** (vgl. Abschnitt 5.3):

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 Abschnitt 15 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 = new Double[13];
    . . .
  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 Methode mit diesem Namen. Das Beispiel der Klasse **String** lehrt, dass eine „vernünftige“ **compareTo()** - Realisation keinen beliebigen **int**-Wert abliefern darf, sondern das Vergleichsergebnis folgendermaßen mitteilen muss:

- negativer Rückgabewert, wenn das angesprochene Objekt in der Anordnung vor dem Parameterobjekt steht (kleiner ist)
- Rückgabewert 0, wenn beide hinsichtlich der Anordnung gleich sind
- positiver Rückgabewert, wenn das angesprochene Objekt in der Anordnung hinter dem Parameterobjekt steht (größer ist)

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.

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 abstrakten Instanzmethoden implementieren (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.:

- Schnittstellen als Datentypen in eigenen Klassen- und Methodendefinitionen
  In einer Methodendefinition ist es oft sinnvoll, Parameterdatentypen über Schnittstellen zu definieren. Man vermeidet es, sich auf konkrete Klassen festzulegen. In den Anweisungen der Methode werden Verhaltenskompetenzen der Parameterobjekte genutzt, die durch Schnittstellenverpflichtungen garantiert sind. Damit wird 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` das Interface **Comparable<`Bruch`>** implementiert, könne 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 Collection 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

Neben Instanzmethoden (mit und ohne **default**-Implementierung) kann ein Interface noch folgende Bestandteile (Mitglieder) enthalten:

- 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.
- Konstanten (vgl. Abschnitt 9.2.4)
  Manche Schnittstellen dienen als zentraler Aufbewahrungsort für die in einem Programm benötigten Konstanten.
- Statische Mitgliedstypen (engl.: *member types*, vgl. Abschnitt 9.2.5)
  Die in einem Interface definierten Typen (z.B. Klassen, Enumerationen, Schnittstellen) sind implizit als **static** deklariert (vgl. Abschnitt 4.9.1.2). Sie werden also wie Top-Level-Typen behandelt, doch muss bei ihrer Verwendung durch fremde Typen ein „Doppelname“ verwendet werden, z.B. `WinterFace.SchneeNum.PULVER` bei dem im Interface `WinterFace` definierten Aufzählungstyp `SchneeNum` mit der Konstanten `PULVER`. Der Modifikator **static** kann weggelassen werden, ist aber erlaubt.

Diese Interface-Bestandteile sind ebenso wie die (abstrakten oder **default**-implementierten) Instanzmethoden 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.

9.2 Interfaces definieren

Wir behandelt zuerst das im Programmieralltag vergleichsweise seltene Definieren einer Schnittstelle, weil dabei Inhalt und Funktion gut zu erkennen sind. Allerdings verzichten wir zunächst 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, so dass für konkrete Typen jeweils eine eigene Interface-Definition entsteht (vgl. Kapitel 8 zu generischen Typen). In 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*) kennen lernen.

Im Schnittstellenrumpf werden in der Regel abstrakte Methoden aufgeführt, deren Rumpf durch ein Semikolon ersetzt ist. Dabei werden die Typformalparameter wie gewöhnliche Typbezeichner verwendet. Mit einer Schnittstelle wird festgelegt, dass Objekte eines implementieren Datentyps 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, so dass er z.B. auch die folgende **compareTo()** - Realisation der Klasse **Double** akzeptieren würde:

```
public int compareTo(Double a) {return 1.0;}
```

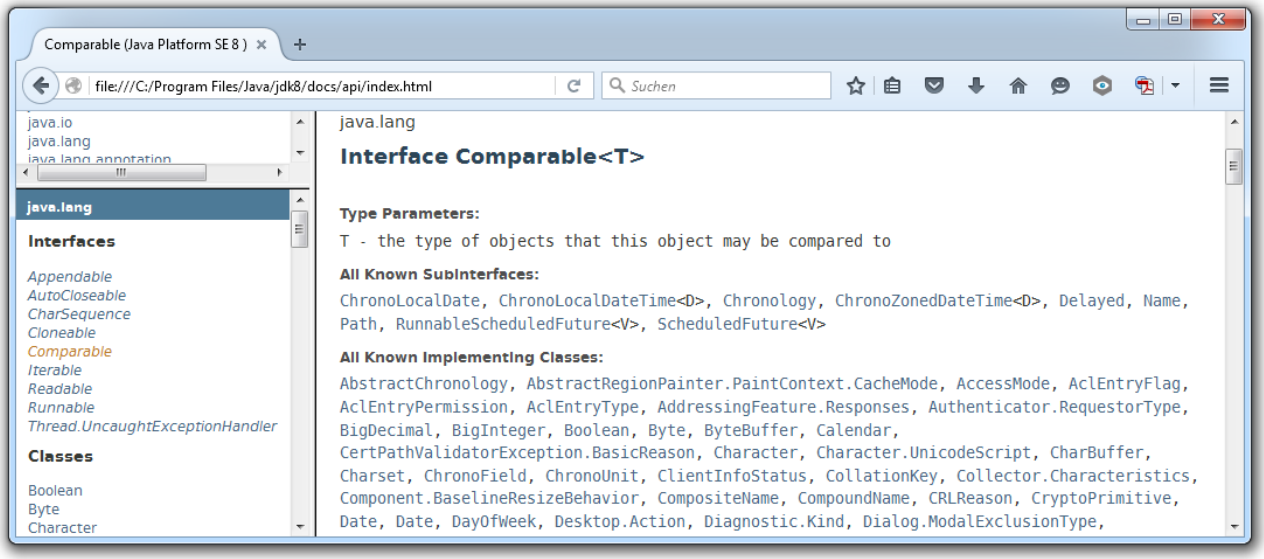
Hinsichtlich der Dateiverwaltung gilt analog zu Klassen, dass ein **public**-Interface in einer eigenen Datei gespeichert werden muss, wobei der Schnittstellenname übernommen und die Namenserweiterung **.java** angehängt wird. In der Regel wendet man diese Praxis bei *allen* Schnittstellen an, die nicht in andere Typen eingeschachtelt sind.

Analog zu Mitgliedsklassen (vgl. Abschnitt 4.9.1) können Schnittstellen nicht nur auf Paketebene (in einer eigenen Quellcodedatei) definiert werden, sondern auch innerhalb von Klassen oder anderen Schnittstellen.

In der Java-API - Dokumentation sind die Schnittstellen in der Paketübersicht (unten links) an den kursiv gesetzten Namen zu erkennen, z.B.:

---

[1] Sie finden diese Definition in der Datei **Comparable.java**, die wiederum im Archiv **src.zip** mit den API-Quelltexten steckt. Das Quelltextarchiv kann bei der JDK-Installation auf die Festplatte Ihres PCs befördert werden (siehe Abschnitt 2.1.3).



9.2.1 Kopf einer Schnittstellen-Definitionen

Regeln für den Kopf einer Schnittstellen-Definitionen:

- Erlaubte Modifikatoren für Schnittstellen
  - **public**
    Wird **public** nicht angegeben, ist die Schnittstelle nur innerhalb ihres Pakets verwendbar. Ein Interface mit Schutzstufe **public** muss prinzipiell in einer eigenen Datei (ohne weitere Top-Level - Typen) definiert werden.
  - **abstract**
    Weil Schnittstellen grundsätzlich **abstract** sind, muss der Modifikator nicht angegeben werden.
- 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 in Kapitel 8 beschriebenen generischen Klassen).

9.2.2 Vererbung bei Schnittstellen

Die bei Klassen äußert wichtige Vererbung über das Schlüsselwort **extends** (siehe Kapitel **7.1**) ist auch bei Schnittstellen erlaubt, z.B.:

```
public interface SortedSet<E> extends Set<E> {
  . . .
}
```

Während bei Java-Klassen die (z.B. von C++ bekannte) *Mehrfachvererbung nicht* unterstützt wird, 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 Methoden aller Schnittstellen aus der Ahnenreihe realisieren.

9.2.3 Schnittstellen-Methoden

In einer Schnittstelle sind alle Methoden grundsätzlich **public**. Das Schlüsselwort kann also weggelassen werden. In der *Java Language Specification* findet sich die folgende Empfehlung (Gosling et al. 2015, 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);
```

Ein dem Schlüsselwort **public** widersprechender Zugriffsmodifikator ist natürlich verboten.

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 Instanzmethoden mit **default**-Implementierung sowie statische Methoden möglich.

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 (siehe obiges Beispiel **compareTo()**). Der Modifikator **abstract** kann angegeben werden, ist aber nicht vorgeschrieben.

9.2.3.2 Instanzmethoden mit default-Implementierung

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 gelöst durch die Möglichkeit, neue Instanzmethoden *mit* Implementierung in eine Schnittstelle aufzunehmen, wobei der Modifikator **default** zu verwenden ist. Altklassen erfüllen auch das erweiterte Interface, weil ihnen die **default**-Implementierung zur Verfügung steht.

Wir betrachten ein einfaches Beispiel mit einem Interface `WinterFace1` und einer implementierenden Klasse `Impl1`:

```
interface WinterFace1 {
 void sagA();
}
```

```
class Impl1 implements WinterFace1 {
 public void sagA() {
  System.out.println("A");
 }

 static public void main(String[] args) {
  Impl1 dob = new Impl1();
  dob.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 einer kompletten Implementierung und mit dem Modifikator **default**:

```
interface WinterFace1 {
 void sagA();

 default void sagB() {
  sagA();
  System.out.println("B");
 }
}
```

In einer **default**-Methode dürfen abstrakte Schnittstellenmethoden verwendet werden.

In einer abgeleiteten (erweiternden) Schnittstelle kann eine geerbte **default**-Methode ...

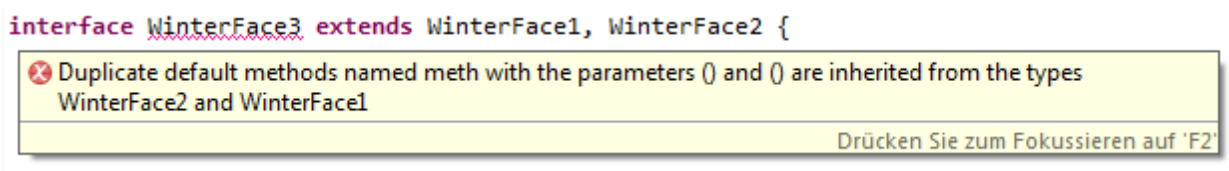
- durch eine eigene **default**-Implementierung überschrieben
- oder durch eine abstrakte Definition ersetzt werden.

Implementiert eine bestehende Klasse eine neuerdings um eine **default**-Methode erweiterte Schnittstelle, dann bleibt die Klasse binärkompatibel zum erweiterten Interface (siehe Abschnitt 9.2.3.2.1).

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:

Quellcode	Ausgabe
`class Impl2 implements WinterFace1 {` `  public void sagA() {` `  System.out.println("A");` `  }` `  static public void main(String[] args) {` `  Impl2 dob = new Impl2();` `  dob.sagB();` `  }` `}`	A B

Wenn eine Klasse mehrere Interfaces implementiert (siehe Abschnitt 9.3) und dabei ein Konflikt mit Signatur-gleichen **default**-Methoden auftritt, verweigert der Compiler die Übersetzung, z.B.:



Das Problem ist dadurch zu lösen, dass die betroffene Klasse die kritische Methode implementiert (siehe Abschnitt 9.2.2) oder als **abstract** definiert.

9.2.3.2.1 Vertiefung: Effekte von default-Methoden auf das Verhalten von bestehenden 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 auftritt, wenn eine vorhandene Klasse nach der Schnittstellenerweiterung neu übersetzt wird.

In der Klasse `AlteKlasse`, die das Interface `WinterFace` implementiert, existiert die Instanzmethode `tuWas()` mit einem Parameter vom Typ **long**. In der **main()** - Methode von `AlteKlasse` wird die Methode mit einem **int**-Argument aufgerufen, das der Compiler implizit erweiternd in den Typ **long** wandelt:

```
interface WinterFace {
 void sagA();
}

class AlteKlasse implements WinterFace {
 public void sagA() {
  System.out.println("A");
 }

 void tuWas(long par) {
  System.out.println("Methode in AlteKlasse: " + par);
 }

 static public void main(String[] args) {
  AlteKlasse dob = new AlteKlasse();
  dob.tuWas(3);
 }
}
```

Nun wird das implementierte Interface um eine **default**-Methode namens `tuWas()` mit einem Parameter vom Typ **int** erweitert:

```
interface WinterFace {
 void sagA();
 default void tuWas(int par) {
  System.out.println("default-Methode in WinterFace: " + par*par);
 }
}
```

Wird nur `WinterFace` neu übersetzt, `AlteKlasse` hingegen nicht, bleibt das Verhalten der Klasse unverändert. Insofern wird das folgende Versprechen aus der Sprachbeschreibung von Java 8 eingehalten (Gosling et al 2015, 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 `AlteKlasse` (aus welchem Grund auch immer) neu übersetzt wird, bevorzugt der Compiler die besser zum Aktualparameter passende **default**-Methode aus dem Interface an Stelle

der klasseneigenen Methode. Anschließend zeigt `AlteKlasse` 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.

9.2.3.3 Statische Methoden

Neu in Java 8 ist auch die Möglichkeit, *statische* Methoden in einem Interface zu realisieren. Die in 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();
  sagA();
  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. Nach Kreft & Langer (2014) soll auf diese Weise verhindert werden, dass sich durch die Aufnahme von statischen Methoden in ein Interface das Verhalten von Klassen ändert, welche das Interface implementieren.

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, z.B.:

```
public interface DiesUndDas {
   int KW = 4711;
   double PIHALBE = 1.5707963267948966;
}
```

Implementierende Klassen können auf die Konstanten ohne Angabe des Schnittstellennamens zugreifen. Auch nicht implementierende Klassen dürfen die Interface-Konstanten 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.

Die `Demo`-Schnittstelle in folgendem Beispiel enthält eine **int**-Konstante namens `ONE` und verlangt das Implementieren einer Methode namens `say1()`:

```
public interface Demo {
 int ONE = 1;
 int say1();
}
```

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. Statische Mitgliedstypen verhalten sich wie Top-Level - Typen, müssen jedoch über einen „Doppelnamen“ angesprochen werden, wobei auf den Namen des äußeren Typs ein Punkt und der Namen des inneren Typs 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:[1]

```
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 von **Map<K,V>** abstammt (siehe Abschnitt 10.6.3), definierte Methode **firstEntry()**:

public 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	Ausgabe
`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());` `  }` `}`	`AAA`

[1] Sie finden diese Definition in der Datei **Map.java**, die wiederum im Ordner **java\util** des Archivs **src.zip** mit den API-Quelltexten steckt. Das Quelltextarchiv kann bei der JDK-Installation auf die Festplatte Ihres PCs befördert werden (vgl. Abschnitt 2.1.3).

9.2.6 Optionale Operationen

Am Ende des Abschnitts über die Interface-Definition soll noch von einer Besonderheit bei manchen Schnittstellendefinitionen im Java Collection Framework (siehe Kapitel 10) berichtet werden. Wer z.B. 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 und sich nur scheinbar im Widerspruch zu den obigen Erläuterungen über Schnittstellen als *Verpflichtungserklärungen* befindet:

`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 nach folgendem Muster (aus der **AbstractCollection<E>** - Klassendefinition) 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 ist beim Entwurf des Java Collection Frameworks entstanden, weil man …

- die Zahl der Schnittstellen möglichst gering halten wollte,
- weil spezielle Kollektionsklassen einerseits z.B. die Schnittstelle **Collection<E>** erfüllen sollen, aber andererseits keine Strukturveränderungen (z.B. durch Aufnahme neuer Elemente) vornehmen sollen.

9.2.7 Zugriffsschutz bei Schnittstellen

Bei den Top-Level - Schnittstellen ist nur der Zugriffsmodifikator **public** erlaubt, so dass zwei Schutzstufen möglich sind:

- Ohne Zugriffsmodifikator ist das Interface nur innerhalb des eigenen Pakts verwendbar.
- Durch den Zugriffsmodifikator **public** wird die Verwendung in beliebigen Paketen erlaubt,

Bei den Mitgliedern von Schnittstellen benötigt man kein Regelwerk für den Zugriffsschutz, weil sie grundsätzlich **public** sind. Das gilt auch für 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.

9.2.8 Marker - Interfaces

Es sind auch Schnittstellen erlaubt, die weder Methoden noch Konstanten oder sonstige Bestandteile enthalten, also nur aus einem Namen bestehen und gelegentlich als *marker interfaces* bezeichnet werden. Ein besonders wichtiges Beispiel ist die beim Sichern (*Serialisieren*) kompletter Objekte (siehe Abschnitt 15.6) 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. Eine Verweigerung der Serialisierbarkeit kann z.B. durch die mit dieser Technik verbundene Einschränkung mit der Weiterentwicklung der Klasse begründet sein.

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`-Objekt eine Anordnung definiert ist:

```
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 in Abschnitt 9.2 wiedergegebenen **Comparable<T>** - Definition ist also im aktuellen Beispiel eine Methode mit dem folgenden Definitionskopf erforderlich:

```
public int compareTo(Figur fig)
```

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 eine -1 zurückgemeldet.
- Haben beide Figuren dieselbe X-Koordinate, lautet die Antwort 0.
- Ansonsten wird eine 1 gemeldet.

Damit wird eine Anordnung der `Figur`-Objekte definiert, und einem erfolgreichen Sortieren (z.B. per **java.util.Arrays.sort()**) steht nichts mehr im Weg.

Wenn eine implementierende Klasse eine Schnittstellenmethode weglässt (oder abstrakt implementiert), dann resultiert eine abstrakte Klasse, die auch als solche deklariert werden muss (vgl. Abschnitt 7.8).

Weil die 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.

Während eine Klasse nur *eine direkte Basisklasse* besitzt, kann sie *beliebig viele Schnittstellen* implementieren, z.B.:

```
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 in der Ahnenreihe realisieren. Weil z.B. die Klasse **TreeSet<E>** aus dem Java Collection Framework (siehe Abschnitt 10.5.3) 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>**:

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 maintained by this set

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 (und zwar nicht nur abstrakt).

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 **static** und **final** (siehe Abschnitt 9.2.4). Folglich können *Instanzvariablen* nur von der Basisklasse (also nur von *einer* Klasse) übernommen werden, und der so genannte *Deadly Diamond of Death* kommen ist ausgeschlossen (siehe Kreft & Langer 2014, vgl. Abschnitt 7.2).

Im Zusammenhang mit dem Thema *Vererbung* ist auch noch 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,

```
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
`class ImpleDemo {` `  public static void main(String[] args) {` `  Kreis[] ka = new Kreis[3];` `  ka[0] = new Kreis("C", 250, 50, 10);` `  ka[1] = new Kreis("B", 150, 50, 20);` `  ka[2] = new Kreis("A", 50, 50, 30);` `  for (Kreis ko : ka)` `  System.out.print(ko.name+" ");` `  java.util.Arrays.sort(ka);` `  System.out.println();` `  for (Kreis ko : ka)` `  System.out.print(ko.name+" ");` `  }` `}`	`C B A` `A B C`

Die Schnittstelle **Comparable<`Kreis`>** befindet sich weder im Erbe der `Kreis`-Klasse noch darf sie hier zusätzlich implementiert werden, wozu auch kein Anlass besteht. Es ist aber selbstverständlich erlaubt, in der Klasse `Kreis` die geerbte Methode **compareTo()** zu überschreiben.

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	Ausgabe
`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[] demintar = {new Ritter(), new Wolf()};` `  for (Quatsch di : demintar)` `  di.sagWas();` `  }` `}`	`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 bestimmte 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, Enumerationen, Annotationen), Methoden, Konstruktoren, Parameter und lokale Variablen lassen sich Annotationen anheften, um zusätzliche **Metainformationen** bereit zu stellen, die …

- zur Entwicklungs- bzw. Übersetzungszeit
- 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 generell die als *Reflexion* (engl.: *reflection*) bezeichnete Ermittlung von Informatio-

[1] Wer die Programmiersprache C# kennt, fühlt sich zu Recht an die dortigen *Attribute* erinnert.

nen über Typen zur Laufzeit eine zunehmende Rolle. Dabei leisten Annotationen einen wichtigen Beitrag. Man spricht in diesem Zusammenhang auch von *Meta-Programmierung*.

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.

Unsere Entwicklungsumgebung Eclipse bezeichnet Annotationen als *Anmerkungen*, und man kann die Unterstützung bei der Definition einer Annotation z.B. mit dem folgenden Menübefehl einleiten:

Datei > Neu > Anmerkung

Annotationen mit Sichtbarkeit **public** benötigen wie andere öffentliche Schnittstellen eine eigene Quellcode- und Bytecodedatei.

9.5.1 Definition

Wir starten mit der (im typischen Alltag 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 eben erwähnte API-Annotation **Deprecated** (Paket **java.lang**) zur Kennzeichnung veralteter Programmbestandteile (siehe Abschnitt 9.5.4). Sie enthält keine Annotationselemente (siehe unten) und gehört daher zu den *Marker-Annotationen*:

```
public @interface Deprecated {
}
```

Hinter dem optionalen Zugriffsmodifikator steht das Schlüsselwort **interface** mit dem Präfix „@“ zur Unterscheidung von gewöhnlichen Schnittstellendefinitionen. Dann folgen der Typname und der Definitionsrumpf.

Als **Annotationselemente** kann man (Name-Wert) - Paare mit Detailinformationen vereinbaren, die syntaktisch als Interface-Methoden mit Rückgabetyp und Name realisiert werden. Über die folgende selbstkreierte Annotation können Versionsinformationen an Programmbestandteile geheftet werden:

```
public @interface VersionInfo {
 String   version();
 int      build();
 String   date() default "unknown";
 String[] contributors() default {};
}
```

Als Rückgabetypen sind bei Annotationen erlaubt:

- Primitive Typen
- Die Klassen **String** und **Class**
- Aufzählungstypen
- Annotationstypen
- Arrays mit einem Elementtyp aus der vorgenannten Liste

Parameter sind *nicht* erlaubt.

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 eine leere Liste als Voreinstellung zu vereinbaren, setzt man hinter das Schlüsselwort **default** ein Paar geschweifter Klammern, z.B.:

```
String[] contributors() default {};
```

Hat eine Annotation nur ein einziges Element, sollte dieses den Namen **value()** erhalten, z.B.

```
public @interface Retention {
 RetentionPolicy value();
}
```

Dann genügt bei der Zuweisung (siehe Abschnitt 9.5.2) an Stelle einer (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 in Abschnitt 9.5.4 näher beschrieben.

Die Ableitung von einem Basistyp ist bei Annotationstypen *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 Mixen erlaubt. Eine Annotation besteht aus einem Namen samt Präfix „@“ und einer Elementenliste mit (Name = Wert) - Paaren. Im folgenden Beispiel wird einer Methode die Annotation `VersionInfo` zugewiesen, deren Definition in Abschnitt 9.5.1 zu sehen war:

```
@VersionInfo(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.:
  ```
  @Deprecated
  ```
- 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 `VersionInfo` ist der Verzicht auf eine Datumsangabe erlaubt, weil das zugehörige Annotationselement einen **default**-Wert besitzt.
- 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 Zeile eignet sich auch für die in Abschnitt 9.5.1 vorgestellten Annotation `VersionInfo`, die im folgenden Beispielprogramm bei einer Methode zum Einsatz kommt:

```
import java.lang.reflect.Method;

class AnnoReflection {
 @VersionInfo(version = "7.1.4", build = 3124, contributors = {"Häcker", "Kwikki"})
 public static void meth() {
  // Not yet implemented
 }

 public static void main(String[] args) {
  VersionInfo vi;
  for (Method meth : AnnoReflection.class.getMethods()) {
   System.out.println("\npublic method "+meth.getName()+"()");
   vi = meth.getAnnotation(VersionInfo.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 `VersionInfo`-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`:
  ```
  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:
  ```
  VersionInfo vi = meth.getAnnotation(VersionInfo.class);
  ```
- Nun lassen sich die Werte der Annotationselemente ermitteln, z.B.:
  ```
  vi.version()
  ```

Bei einem Lauf 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

Nun 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 16 über Multithreading wird z.B. zu erfahren sein, 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 Marker-Annotation **Deprecated** erhalten:
  ```
  @Deprecated
  public final void stop() {
    . . .
  }
  ```
  Die Vergabe dieser Annotation sollte nach den Empfehlungen der Java-Designer von einem Dokumentationskommentar (vgl. Abschnitt 3.1.5) mit dem Tag **@deprecated** (kleiner Anfangsbuchstabe!) begleitet werden. Im Beispiel:
  ```
  /**
   * 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.
   . . .
   */
  ```
  Im Editor unserer Entwicklungsumgebung Eclipse sind abgewertete Programmelemente an einer durchgestrichenen Bezeichnung zu erkennen (siehe obiges Beispiel **stop()**).

- **Override**
  Mit dieser Marker-Annotation kann man seine Absicht bekunden, bei einer Methodendefinition eine Basisklassenvariante 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. aufgrund eines Tippfehlers, warnt der Compiler.

- **SuppressWarnings**
  Mit dieser Annotation überredet man den Compiler, Warnungen aus bestimmtem Anlass zu unterdrücken. Sie kann auf sehr viele Programmbestandteile bezogen werden (auf Typen, Felder, Methoden, Konstruktoren, Parameter, lokale Variablen). Es ist anzustreben, den Gültigkeitsbereich der Unterdrückung so klein wie möglich zu halten. Die unterstützten Werte für den Zeichenfolgenparameter **value** hängen vom Compiler ab. Welche Zeichenfolgen der Compiler in Eclipse versteht, erfährt man im Hilfefenster (zu starten über den Menübefehl **Hilfe > Hilfeinhalte**) über eine Suche nach „**SuppressWarnings**“. Im folgenden Beispiel aus Abschnitt 8.1.3.1 werden für eine (kleine und übersichtliche!) Methode die Warnungen vor den vom Compiler nicht kontrollierbaren Typumwandlungen abgeschaltet, was stets kommentiert werden sollte:

```
@SuppressWarnings("unchecked")
// Casting erforderlich, weil kein Array vom Typ E erstellt werden.
public E get(int index) {
   if (index >= 0 && index < size)
      return (E) elements[index];
   else
      return null;
}
```

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 zum Träger 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 in 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. Eine folgendermaßen

  ```
  @Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
  ```

  dekorierte Annotation kann einer Methode oder einem Konstruktor angeheftet werden.

9.6 Übungsaufgaben zu Kapitel 9

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Eine Schnittstelle ist grundsätzlich (auch ohne Zugriffsmodifikator) als **public** definiert.
2. Die Methoden einer Schnittstelle sind grundsätzlich (auch 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 in Abschnitt 4 als zentrales Beispiel diente, eine Variante, welche die Schnittstelle **Comparable<Bruch>** implementiert, so dass z.B. ein `Bruch`-Array mit der statischen Methode **sort()** aus der Klasse **Arrays** sortiert werden kann.

3) Definieren Sie eine generische Methode mit einem Parameter, dessen Typ von einer bestimmten Klasse abstammen und zwei Interfaces implementieren muss.

10 Java Collection Framework

Die in diesem Kapitel vorgestellten Typen zur Verwaltung von Listen oder Mengen von Elementen oder (Schlüssel-Wert) - Paaren stammen aus dem **Java Collection Framework (JCF)**, dessen Weiterentwicklung seit der Einführung in Java 2 insbesondere von der seit Java 5 verfügbaren Generizität profitiert hat. In Abschnitt 8.1.1 haben Sie einen Eindruck davon erhalten, welchen Fortschritt eine generische Klasse gegenüber der auf unsicheren **Object**-Referenzen und expliziter Typumwandlung basierenden Vorgängerlösung bei der häufig benötigten Verwaltung von 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 erleichtern.

Für die Objekte der im aktuellen Kapitel vorzustellenden Klassen wird im Manuskript alternativ zur offiziellen Bezeichnung *Kollektionen* aus sprachlichen Gründen oft auch die Bezeichnung *Container* verwendet.

In diesem Kapitel beschäftigen wir uns *nicht* damit, eigene Kollektionstypen zu definieren, sondern konzentrieren und darauf, die im JCF zahlreich vorhandenen Typen effektiv zu nutzen. Als Basis für die Ableitung einer eigenen Kollektionsklasse eignen sich die abstrakten Klassen **AbstractCollection<E>**, **AbstractList<E>**, **AbstractSet<E>** und **AbstractMap<K,V>** aus dem JCF.

10.1 Arrays versus Kollektionen

Die in Abschnitt 5.1 vorgestellten Arrays taugen als Container für Elemente mit einem identischen, frei wählbaren Typ und bieten einen schnellen Indexzugriff auf die Elemente. Man kann sich fragen, wozu eigentlich noch weitere Datenstrukturen benötigt werden. Beginnen wir bei den sehr häufig auftretenden listenartigen Datenstrukturen. Dabei zeigen Arrays folgende Schwächen:

- Die Größe eines Arrays kann nach der Erzeugung nicht mehr geändert werden.
- Das Einfügen oder Entfernen von Elementen ist mit großem Aufwand verbunden.

Kollektionen zur Verwaltung von Listen bieten hingegen Größendynamik sowie performantes Einfügen und Löschen.

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 übereinstimmt. Kollektionen zur Verwaltung von Mengen bieten hingegen schnelle Detektionsmöglichkeiten und verhindern automatisch identische Elemente (Dubletten).

Oft sind Mengen von (Schlüssel-Wert) - Paaren 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 aufwändig aus einer Datenbank eingelesen 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 schon beschriebenen Probleme auf (feste Anzahl von Elementen, umständliches Einfügen und Löschen, aufwändige Suche nach den Schlüsseln).

Im zu modellierenden Aufgabenbereich einer Anwendung treten oft Datenstrukturen vom Typ Liste, Menge oder Abbildung auf, und im Java Collection Framework finden sich passende Typen, so dass im Vergleich zur Verwendung von Arrays eine bessere Modellierung und ein besser lesbarer Quellcode resultieren. Sicherlich sind Sie mittlerweile in der Lage, eine generische Kollektionsklasse zu definieren, die ihre Elemente in einem Array lagert und diesen bei Bedarf automatisch durch ein größeres Exemplar ersetzt (siehe die in Abschnitt 8.1.3 vorgestellte Klasse `SimpleList<E>`). Allerdings sind im JCF bereits exzellente Lösungen für derartige Standardaufgaben vorhanden (im Beispiel die Klasse **ArrayList<E>**), so dass wir uns auf andere Herausforderungen konzentrieren können.

Das JVF bietet oft für einen Aufgabentyp verschiedene Implementierungen, welche trotz unterschiedlicher Innenarchitekturen dieselbe Schnittstelle erfüllen, sodass problemadäquat ein einfacher Wechsel möglich ist. Manche Kollektionsklassen verwenden im Hintergrund einen Array zur Datenverwaltung (z.B. **ArrayList<E>)**, was in bestimmten Anwendungsfällen zu einer performanten Lösung führen kann (siehe z.B. Abschnitt 10.4.3 mit Einsatzempfehlungen für die Listenverwaltung.

Die Standardbibliothek hält ausgereifte Lösungen für typische Aufgaben bereit (z.B. Vereinigung von zwei Mengen ohne Entstehung von Dubletten), damit Programmierer möglichst selten „das Rad neu erfinden müssen“. Kollektionsmethoden wie **containsAll()**, **addAll()**, **removeAll()** etc. selbst zu kodieren, wäre hochgradig ineffektiv.

Schließlich hat sich in Abschnitt 8.1.2.2 herausgestellt, dass die so genannte *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;
```

Im Ergebnis drohen Laufzeitfehler vom Typ **ArrayStoreException**.

10.2 Zur Rolle von Interfaces beim JCF-Design

Wie bei jedem Framework spielen auch beim Java Collection Framework (JCF) Interfaces eine wichtige Rolle. In Kapitel 9 haben Sie erfahren, dass ein *Interface* meist aus einer Liste von abstrakten Methoden besteht. Anschließend ist das in Abschnitt 10.3 zu diskutierende Interface **Collection<E>** im Paket **java.util** zu sehen, das (quasi als Pflichtenheft) grundlegende Kompetenzen einer Kollektionsklasse vorschreibt:

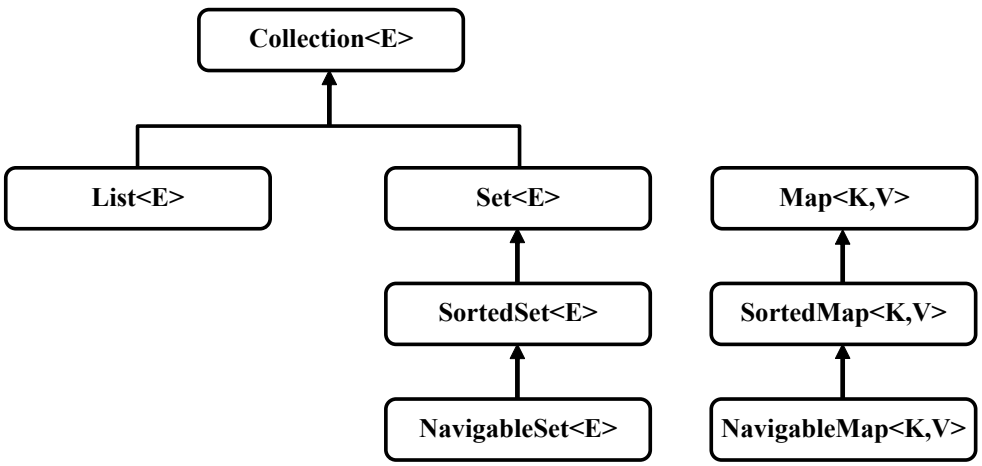
```
public interface Collection<E> extends Iterable<E> {
    boolean add(E e);
    boolean addAll(Collection<? extends E> c);
    void clear();
    boolean contains(Object o);
    boolean containsAll(Collection<?> c);
    boolean equals(Object o);
    int hashCode();
    boolean isEmpty();
    Iterator<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);
    int size();
    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, dieses Interface zu implementieren, muss sie alle abstrakten Interface-Methoden implementieren. Hinzu kommen sogar 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 und Methoden bewältigt werden, ohne dass die vorhandene Code-Basis geändert werden muss.

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 Collection Framework ist so eine Interface-Hierarchie entstanden, die einen guten Eindruck von den Kompetenzprofilen der im Framework natürlich auch enthaltenen Klassen vermittelt. In den folgenden Abschnitten werden die folgenden Interfaces (d.h. die jeweils geforderten Methoden) beschrieben und wichtige implementierende Klassen vorgestellt:



10.3 Das Interface Collection<E> mit Basiskompetenzen

Viele Klassen im Java Collection Framework implementieren direkt oder indirekt das generische Interface **Collection<E>** und beherrschen damit u.a. die folgenden Methoden:

- **public boolean add(E** *element***)**
  Wenn die Kollektion aufgrund der beantragten Neuaufnahme verändert wurde, 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, wirft die Methode eine Ausnahme (z.B. **ClassCastException**).[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***)**
  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 ein anderer Thread die Kollektion verändert. Zum Thema *Thread-Sicherheit* folgen im weiteren Verlauf von Kapitel 10 noch einige Hinweise.
- **public void clear()**
  Mit dieser Methode fordert man eine Kollektion auf, alle Elemente. Dabei werden die im Kollektionsobjekt vorhandenen Objektreferenzen gelöscht, nicht die Objekte selbst.
- **public boolean contains(Object** *object***)**
  Diese Methode informiert darüber, ob ein Element der Kollektion im Sinne der **equals()** - Methode mit dem Aktualparameter übereinstimmt.
- **public boolean containsAll(Collection<?>** *collection***)**
  Diese Methode informiert darüber, ob die angesprochene Kollektion im Sinne der **equals()** - Methode 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 unterstützt (siehe Abschnitt 10.7). Sie gehört zum Interface **Iterable<E>**, das vom Interface **Collection<E>** erweitert wird.
- **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***)**
  Diese Methode entfernt ggf. *ein* Element aus der Kollektion, das sich vom Parameterobjekt gemäß **equals()** - Methode nicht unterscheidet. Mit dem Rückgabewert informiert die Methode darüber, ob die Kollektion tatsächlich geändert worden ist.

---

[1] Mit der Ausnahmebehandlung werden wir uns bald beschäftigen.

- **public boolean removeAll(Collection<?> Object** *collection***)**
  Diese Methode entfernt ggf. *alle* Elemente aus der angesprochenen Kollektion, die mit einem Element der Parameterkollektion gemäß **equals()** - Methode identisch sind. Mit dem Rückgabewert informiert die Methode darüber, ob die Kollektion geändert worden ist. 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<?> Object** *collection***)**
  Diese Methode entfernt ggf. *alle* Elemente aus der angesprochenen Kollektion, die *nicht* mit einem Element der Parameterkollektion gemäß **equals()** - Methode identisch sind. Mit dem Rückgabewert informiert die Methode darüber, ob die Kollektion geändert worden ist.
- **public int size()**
  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).

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 9.2). 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 offen zu legen, für welche Methoden nur eine Pseudo-Implementation vorhanden 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 Elementklassendefinition für die von **Object** geerbte **equals()** - Methode eine sinnvolle Überschreibung erwartet.

10.4 Listen

Eine Liste enthält eine Sequenz von Elementen (Objektreferenzen) mit einer festen Reihenfolge, auf die man sequentiell sowie wahlfrei über einen nullbasierten Index zugreifen kann, und passt ihre Größe (im Unterschied zu einem Array) automatisch an die Aufgabenstellung an. Wir haben also einen größendynamischen Container zur Verfügung, der dank Typgenerizität Elemente von einem wählbaren Referenztyp sortenrein (mit Compiler-Typsicherheit) verwaltet.

Damit haben Listen sehr viele Einsatzmöglichkeiten bei der Software-Entwicklung. Man verwendet sie z.B. für ...

- die in einem Vektorgrafikdokument übereinander gestapelten Zeichnungselemente
- die Wörter in einem Text
- die in Abhängigkeit von Alter und Priorität geordneten Einträge in einem Blog

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.
- Mehrere Referenzziele können im Sinn der **equals()** - Methode inhaltsgleich sein.

Neben den anschließend vorgestellten Typen zur Listenverwaltung enthält das Java Collection Framework noch Lösungen für wichtige Spezialfälle, die im Manuskript nicht behandelt werden können, z.B.:

- Stapel
  Die Klasse **Stack<E>** unterstützt eine nach dem LIFO-Prinzip (Last In First Out) arbeitende Liste.
- Warteschlangen
  Die das Interface **Queue<E>** unterstützenden Klassen verwalten eine Liste nach dem FIFO-Prinzip (First In First Out).

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 und modifiziert 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 13.4.2.3).

10.4.1 Das Interface List<E>

Zur Realisation von Listen enthält das Java Collection 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 bieten folglich über die **Collection<E>** - Methoden (siehe Abschnitt 10.3) hinaus u.a. die folgenden Kompetenzen:

- **public void add(int** *index***, E** *element***)**
  Es wird ein neues Element an der gewünschten Indexposition eingefügt.
- **public boolean addAll(int** *index***, Collection<? extends E>** *c***)**
  Diese Methode fügt die Elemente der Parameterkollektion an der gewünschten Position in die Liste ein und verhält sich ansonsten wie die eben beschriebene Überladung.
- **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()** mit dem Parameterobjekt übereinstimmen, wird der kleinste Index unter diesen Elementen geliefert, ansonsten der Wert -1. Zeigt der Parameter auf **null**, liefert die Methode ggf. den kleinsten Index zu einem Element, das gleich **null** ist.
- **public E remove(int** *index***)**
  Diese Methode entfernt das Element an der Position *index* aus der Liste und liefert dessen Adresse zurück.

- **public E set(int** *index*, **E** *element***)**
  Das Element mit der im ersten Parameter genannten Position wird durch das Objekt im zweiten Parameter ersetzt.

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.

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 Collection Framework erhebliche Unterschiede, die nun behandelt werden.

10.4.2 Listenarchitekturen

Die Klasse **ArrayList<E>** arbeitet intern mit einem Array zum Speichern der Elemente und bietet daher einen schnellen wahlfreien Zugriff. 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.

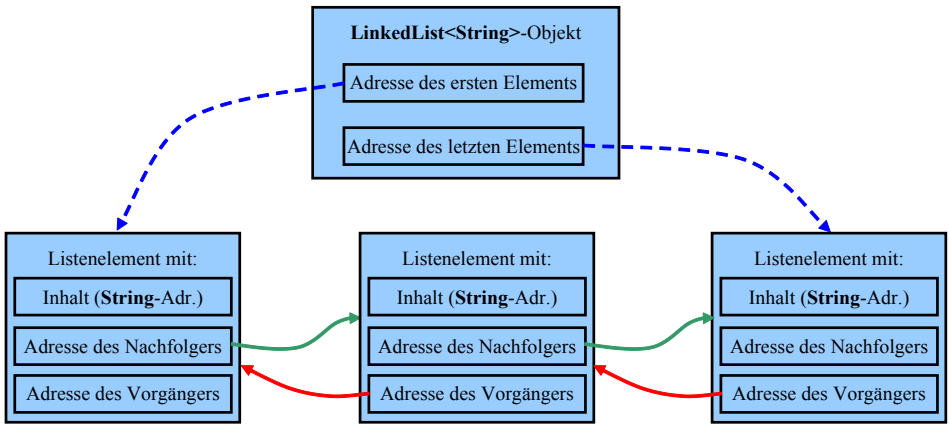
Die ebenfalls Array-basierte Klasse **Vector<E>** ist von Beginn an im Java-API enthalten, wurde zwar an das Java Collection Framework angepasst, steht aber trotzdem mittlerweile nicht mehr im besten Ruf. Sie enthält neben den empfohlenen Methoden aus dem Interface **List<E>** auch noch veraltete Methoden, 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 implementiert ist, so dass ein Container-Objekt ohne Risiko simultan durch mehrere Threads benutzt werden kann. Was das genau bedeutet, werden Sie im Kapitel über Threads (nebenläufige Programmierung) erfahren. Allerdings ist die Sicherheit nicht kostenlos zu haben, so dass die Klasse **ArrayList<E>** performanter arbeitet und zu bevorzugen ist, wenn *kein* simultaner Container-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 eine Synchronisationsmöglichkeit erwähnt werden, die dem momentanen Kursentwicklungsstand weit vorgreift. Die Service-Klasse **Collections** (mit einem *s* am Ende des Namens, siehe Abschnitt 10.8) 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.[1]

Die Klasse **LinkedList<E>** arbeitet im Unterschied zu den bisher beschriebenen **List<E>** - Implementationen intern mit einer **doppelt verlinkten 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 und Löschen von Elementen müssen keine anderen Elemente verschoben werden. Stattdessen wird ein Listenelement erzeugt bzw. gelöscht, und die Adressketten werden neu verknüpft.
- Die Länge der Liste ist zu keinem Zeitpunkt festgelegt, so dass keine aufwändigen Maßnahmen zur Kapazitätsanpassung erforderlich werden.

Um ein Listenelement mit bestimmtem Indexwert aufzufinden, muss die Liste allerdings ausgehend vom ersten oder letzten Element durchlaufen werden. Folglich ist die verkettete Liste beim wahlfreien Zugriff auf vorhandene Elemente einem Array deutlich unterlegen, weil dessen Elemente im Speicher hintereinander liegen und nach einer einfachen Adressberechnung direkt angesprochen werden können. Außerdem benötigt eine verkettete Liste mehr Speicher.

Insgesamt sind verlinkte Listen besonders geeignet für Algorithmen, die …

- häufig Elemente einfügen oder entfernen und sich dabei nicht auf das Listenende beschränken,
- Elemente überwiegend sequentiell aufsuchen.

Zum sequentiellen Aufsuchen der Listenelemente muss bei der Klasse **LinkedList<E>** aus Performanzgründen an Stelle des Indexzugriffs unbedingt ein Iterator-Objekt verwendet werden (siehe Abschnitt 10.7).

Wie die Klasse **ArrayList<E>** bietet auch die Klasse **LinkedList<E>** aus Performanzgründen keine Thread-Sicherheit, so dass Sie ggf. selbst für die Synchronisation von Threads sorgen müssen (siehe den obigen Hinweis auf die **Collections**-Methode **synchronizedList()**).

---

[1] Quelle: http://docs.oracle.com/javase/tutorial/collections/implementations/list.html

10.4.3 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 aufwendige 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 10.7). 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 Array-Element eingefügt oder entfernt werden muss.

Man kann je nach Einsatzschwerpunkt und benötigter Thread-Sicherheit zwischen den Listenverwaltungsklassen aus dem Java Collection Framework wählen und sogar 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

```
import java.util.*;

class Listen {
 static final int ANZ = 20_000;

 static void testList(List<String> plis) {
  List<String> liste = plis;
  StringBuilder sb = new StringBuilder();
  Random ran = new Random();

  // Füllen
  System.out.println("Kollektionsklasse:\t" + liste.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)));
   liste.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++)
   liste.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++)
     liste.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++)
     liste.remove(0);
    System.out.println(" Zeit zum Löschen:\t" +
       (System.currentTimeMillis()-start) + "\n");
   }

   public static void main(String[] args) {
    testList(new ArrayList<String>());
    testList(new Vector<String>());
    testList(new LinkedList<String>());
   }
}
```

mit den Aufgaben

- eine Liste mit 20.000 Zeichenketten füllen
- aus der Liste 20.000 Elemente mit zufällig bestimmter Indexposition abrufen
- 20.000 neue Elemente einzeln am Anfang der Liste einfügen
- 40.000 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:    16
 Zeit zum Abrufen:   2
 Zeit zum Einfügen:  125
 Zeit zum Löschen:   210

Kollektionsklasse:   class java.util.Vector
 Zeit zum Füllen:    8
 Zeit zum Abrufen:   2
 Zeit zum Einfügen:  151
 Zeit zum Löschen:   180

Kollektionsklasse:   class java.util.LinkedList
 Zeit zum Füllen:    11
 Zeit zum Abrufen:   400
 Zeit zum Einfügen:  4
 Zeit zum Löschen:   2
```

[1] Die Zeiten stammen von einem PC unter Windows 7 (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 Container.
- 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]

10.5 Mengen

Zur Verwaltung einer *Menge* von Elementen, die im Unterschied zu einer Liste *keine Dubletten* (im Sinne der **equals()** - Methode) aufweisen darf, enthält das Java Collection Framework u.a. die generischen Klassen **HashSet<E>**, **LinkedHashSet<E>** und **TreeSet<E>**, die das Interface **Set<E>** implementieren. 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 bilden). 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 oder auf einer Ordnung des Elementtyps basiert.

10.5.1 Das Interface Set<E>

Alle Mengenverwaltungsklassen im Java Collection Framework implementieren das von **Collection<E>** abstammende Interface **Set<E>** und beherrschen daher u.a. die folgenden Instanzmethoden:

- **public boolean add(E** *element***)**
  Das Parameterelement wird in die Menge aufgenommen, falls es dort noch nicht existiert.
- **public boolean addAll(Collection<? extends E>** *collection***)**
  Die Elemente der übergebenen Kollektion werden in die Menge aufgenommen, falls sie dort noch nicht 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.

---

[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).

- **public boolean contains(Object** *object***)**
  Diese Methode informiert darüber, ob das fragliche Element in der Kollektion vorhanden ist und arbeitete bei den Mengenverwaltungsklassen erheblich flotter als bei den Listenverwaltungsklassen (siehe Abschnitt 10.4).
- **public boolean remove(Object** *element***)**
  Das angegebene Element wird aus der Menge entfernt, falls es dort vorhanden ist.
- **public boolean removeAll(Collection<?>** *collection***)**
  Die Elemente der übergebenen Kollektion werden ggf. aus der angesprochenen Kollektion entfernt, so dass man nach einem erfolgreichen Methodenaufruf die **Differenz** der beiden Mengen erhält.
- **public boolean retainAll(Collection<?>** *collection***)**
  Aus der angesprochenen Kollektion werden alle Elemente entfernt, die nicht zur übergebenen Kollektion gehören, so dass man nach einem erfolgreichen Methodenaufruf den **Durchschnitt** der beiden Mengen erhält.

Die Methoden **add()**, **addAll()**, **remove()**, **removeAll()** und **retainAll()** informieren mit ihrem **boolean**-Rückgabewert darüber, ob die 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 Elementklassendefinition für die von **Object** geerbte **equals()** - Methode eine sinnvolle Überschreibung erwartet.

Einen Indexzugriff auf ihre Elemente bieten die Kollektionsklassen zur Mengenverwaltung nicht, ein Iterator (vgl. Abschnitt 10.7) ist jedoch verfügbar.

Während in eine Liste auch ein Element mit **null**-Referenz eingefügt werden kann, führt der Versuch bei einer Menge zu einer **NullPointerException**.

Eine **Set<E>** - Implementierung kann Dubletten *nicht* verhindern, wenn die 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). 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:

Quellcode	Ausgabe
`import java.util.*;` `class Dubletten {` `  public static void main(String[] args) {` `   HashSet<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.:

```
HashSet<String > shs = Collections.synchronizedSet(new HashSet<String>());
```

In einem Testprogramm mit den Aufgaben

- eine Menge mit 20.000 **String**-Objekten füllen
- für 20.000 neue **String**-Objekte prüfen, ob sie bereits in der Menge vorhanden sind

zeigen die Klassen **ArrayList<String>**, **LinkedList<String>**, **HashSet<String>** und **TreeSet<String>**folgende Leistungen:[1]

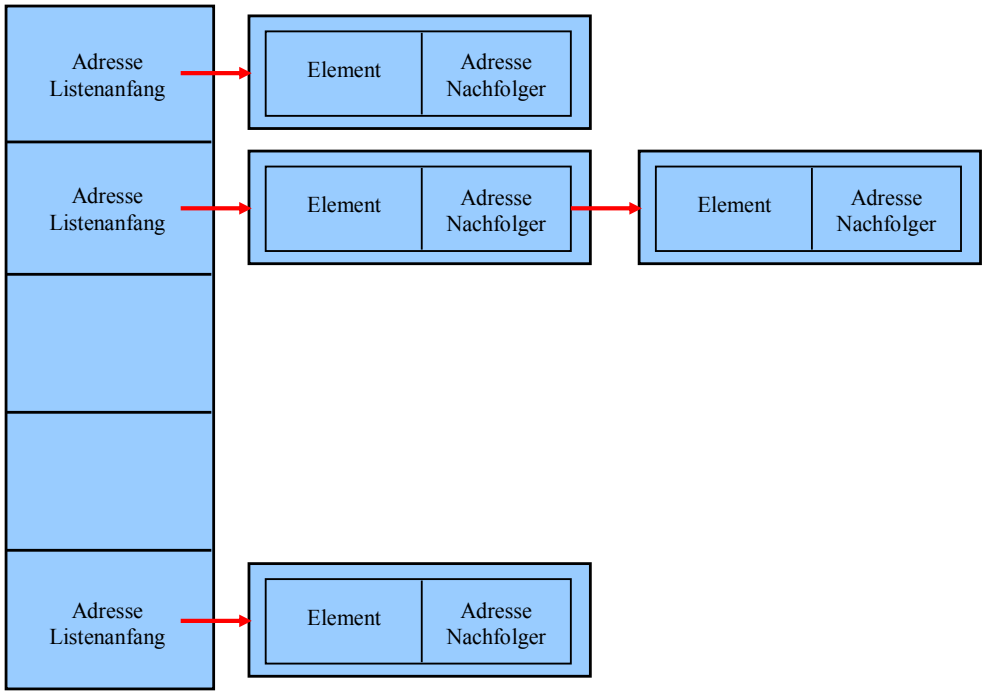
```
Kollektionsklasse:                class java.util.ArrayList
 Zeit zum Füllen:                 30
 Zeit für die Existenzprüfungen:  1704

Kollektionsklasse:                class java.util.LinkedList
 Zeit zum Füllen:                 13
 Zeit für die Existenzprüfungen:  2258

Kollektionsklasse:                class java.util.HashSet
 Zeit zum Füllen:                 26
 Zeit für die Existenzprüfungen:  16

Kollektionsklasse:                class java.util.TreeSet
 Zeit zum Füllen:                 42
 Zeit für die Existenzprüfungen:  29
```

Besonders auffällig ist die Überlegenheit der **Set<E>** - Implementierungen bei der Mengenzugehörigkeitsprüfung.

[1] Die Zeiten stammen von einem PC unter Windows 7 (64 Bit) mit Intel-CPU Core i3 (3,2 GHz).

10.5.2 Hashtabellen

Benötigt ein Algorithmus zahlreiche Mengenzugehörigkeitsprüfungen, sind Kollektionen mit Listenbauform wenig geeignet, weil ein fragliches Element potentiell mit jedem vorhandenen über einen Aufruf der **equals()** - Methode verglichen werden muss. Um diese Aufgabe schneller lösen zu können, kommt bei der Klasse **HashSet<E>** eine so genannte *Hashtabelle* zum Einsatz. Dies ist ein Array mit einfach verketteten Listen als Einträgen:



Bei der Aufnahme eines neuen Elements entscheidet die typspezifische Implementierung der bereits in der Urahnklasse **Object** definierten Instanzmethode **hashCode()**

public int hashCode()

über den Array-Index der zu verwendenden Liste. Im günstigsten Fall ist die Liste noch leer. Anderenfalls spricht man von einer *Hash-Kollision*. 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 ersetzen:

- 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.
- Die **hashCode()** - Werte sollten dazu taugen, Array-Elemente zu indizieren.
- Die **hashCode()** - Werte sollten möglichst gleichmäßig über den möglichen Wertebereich 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 die folgende Hash-Funktion zum Einsatz:

$$u(0)\ 31^{n-1} + u(1)\ 31^{n-2} + ... + u(n-1)$$

Dabei steht ***u(i)*** für die Unicode-Nummer des Zeichens an der (nullbasierten) Position ***i*** und ***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()** - Aufruf mit allen Insassen verglichen werden. Stattdessen wird sein Hashcode berechnet und sein Array-Index ermittelt. Befindet sich hier noch kein Listenanfang, ist die Existenzfrage geklärt (**contains()** - Rückmeldung **false**). Anderenfalls ist nur für die Objekte der im Array-Element startenden verketteten 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).

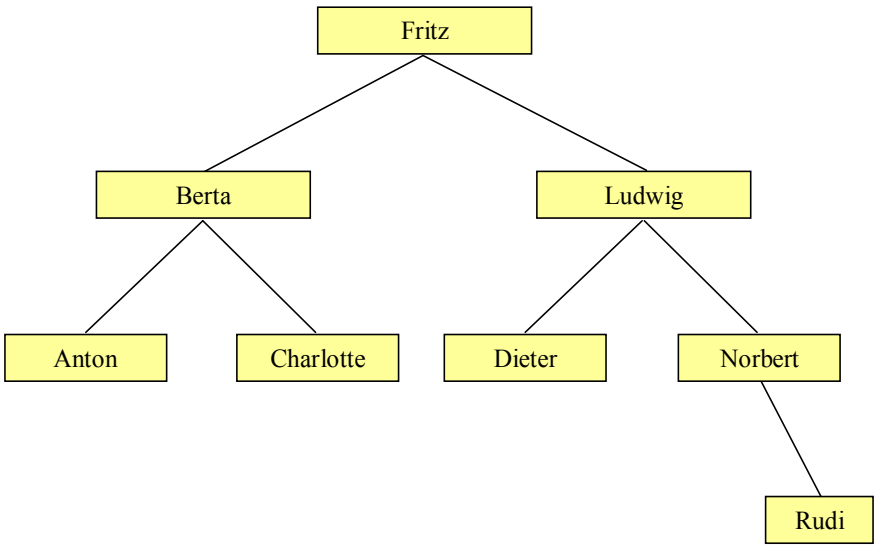
Weil die Klasse **HashSet<E>** das Interface **Collection<E>** (siehe Abschnitt 10.3) implementiert, kann sie (als Rückgabe der Methode **iterator()**) einen Iterator (siehe Abschnitt 10.7) zur Verfügung stellen, der sukzessive alle Elemente aufsucht und dabei erwartungsgemäß eine zufällig wirkende Reihenfolge 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, und im Ergebnis erhalten wir einen Iterator, der die Einfügereihenfolge verwendet.

10.5.3 Balancierte Binärbäume

Existiert über den Elementen einer Menge eine **vollständige Ordnung** (z.B. Zeichenketten mit der lexikografischen Ordnung), kann man über einen Binärbaum die Elemente im sortierten Zustand halten, ohne den Aufwand bei den zentralen Mengenverwaltungsmethoden (z.B. **add()**, **contains()** und **remove()**) im Vergleich zur Hashtabelle wesentlich steigern zu müssen.

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:

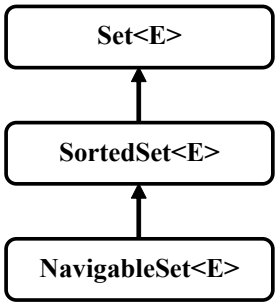


Bei einem **balancierten** Binärbaum kommen Forderungen zum maximal erlaubten Unterschied zwischen der kürzesten und der längsten Entfernung zwischen der Wurzel und einem Endknoten hinzu, um den Aufwand beim Suchen und Einfügen von Elementen zu begrenzen.

Im Java Collection Framework nutzt u.a. die Klasse **TreeSet<E>** das Prinzip des balancierten Binärbaums, wobei durch die so genannte **Rot-Schwarz** -Architektur sicher gestellt wird, dass der längste Pfad höchstens doppelt so lang ist wie der kürzeste.

10.5.4 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 hinzu gekommene Interface **NavigableSet<E>**. Hier sind die drei Interfaces und Ihre Beziehungen zu sehen:



Das Interface **SortedSet<E>** fordert von implementierenden Klassen u.a. die folgenden Methoden:

- **public Comparator<? super E> comparator()**
  Die Methode liefert ein Objekt das die generische Schnittstelle **Comparator<? super E>** erfüllt, also eine für den Typ **E** oder für eine Basisklasse **B** geeignete Vergleichsmethode mit folgendem Definitionskopf bietet:

  public int compare(E *e1***, E** *e2***)** bzw. **public int compare(B** *e1***, B** *e2***)**

  Diese muss einen Wert kleiner, gleich oder größer Null liefern, wenn der Rang von *e1* im Vergleich zum Rang von *e2* kleiner, gleich oder größer ist. In der Regel sollte die Rückgabe Null genau dann erfolgen, wenn die beiden Objekte im Sinne der **equals()** - Methode gleich sind.

- **public E first()**
  Liefert das erste (kleinste) Element in der sortierten Menge
- **public E last()**
  Liefert das letzte (größte) Element in der sortierten Menge
- **public SortedSet<E> headSet(E** *obereSchranke***)**
  Man erhält als so genannte *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, so dass man z.B. mit der Methode **clear()** die komplette **headSet()** - Teilmenge löschen kann:

Quellcode	Ausgabe

```
import java.util.*;
class Prog {
 public static void main(String[] args) {
  TreeSet<String> tss = new TreeSet<>();
  tss.add("a");tss.add("b");tss.add("c");tss.add("d");
  System.out.println(tss);
  SortedSet<String> soSet = tss.headSet("c");
  System.out.println(soSet);
  soSet.clear();
  System.out.println(tss);
 }
}
```

```
[a, b, c, d]

[a, b]

[c, d]
```

- **public SortedSet<E> tailSet(E** *untereSchranke***)**
  Man erhält ein View-Objekt, dessen Methoden sich auf die Teilmenge mit allen Elementen der angesprochenen Kollektion auswirken, die mindestens so groß sind wie die untere Schranke.
- **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.

Es gibt zwei Möglichkeiten, die Ordnung der von einem **TreeSet<E>** - Objekt verwalteten Elemente zu begründen:

- Der Elementtyp **E** erfüllt das Interface **Comparable<E>**, besitzt also eine Instanzmethode **compareTo()**.
- Man übergibt dem **TreeSet<E>** - Konstruktor ein Objekt, das die Schnittstelle **Comparator<? super E>** erfüllt und folglich eine für den Typ **E** oder für eine Basisklasse **B** geeignete Vergleichsmethode bietet (siehe oben).

Im Definitionskopf einer das Interface **SortedSet<E>** implementierenden Klasse (z.B. **TreeSet<E>**) kann und sollte darauf verzichtet werden, den Typformalparameter 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
`import java.util.*;` `class CompaS implements Comparator<String> {` `  public int compare(String s1, String s2) {` `   if (s1.equals(s2))` `    return 0;` `   if (s1.equals("Otto"))` `    return -1;` `   if (s2.equals("Otto"))` `    return 1;` `   return s1.compareTo(s2);` `  }` `}` `class ComparatorTest {` `  public static void main(String[] args) {` `   TreeSet<String> tss = new TreeSet<>();` `   tss.add("Otto"); tss.add("Werner"); tss.add("Ludwig");` `   System.out.println(tss);` `   TreeSet<String> tssc = new TreeSet<>(new CompaS());` `   tssc.add("Otto"); tssc.add("Werner"); tssc.add("Ludwig");` `   System.out.println(tssc);` `  }` `}`	`[Ludwig, Otto, Werner]` `[Otto, Ludwig, Werner]`

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 10.7).

Existiert kein passendes Element, liefern die Methoden **pollFirst()**, **pollLast()**, **ceiling()**, **floor()**, **higher()** und **lower()** den Wert **null**. Im folgenden Programm sind die genannten Methoden bei der Arbeit zu beobachten:

Quellcode	Ausgabe

```
import java.util.*;
class Prog {
 public static void main(String[] args) {
  TreeSet<String> tss = new TreeSet<>();
  tss.add("a"); tss.add("c");
  System.out.println(tss.ceiling("c"));
  System.out.println(tss.floor("b"));
  System.out.println(tss.higher("a"));
  System.out.println(tss.lower("b"));
 }
}
```

```
c
a
c
a
```

Eine geordnete Menge kann wie eine Liste als *Sequenz* bezeichnet werden, doch es bestehen viele Unterschiede zwischen den beiden geordneten Kollektionen:

- In einer Menge sind keine Dubletten erlaubt, während die Eindeutigkeitsgarantie bzw. -restriktion (ja nach Aufgabenstellung) 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.
- 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 Mengen von Schlüssel-Wert - Paaren (Abbildungen)

Zur Verwaltung einer Menge von Schlüssel-Wert - Paaren stellt das Java Collection Framework Klassen zur Verfügung, die das Interface **Map<K,V>** erfüllen. Die Schlüssel (mit einer Konkretisierung des Typformalparameters **K** als Datentyp) werden wie eine Menge verwaltet, so dass also Eindeutigkeit herrscht (ohne Dubletten). Über einen Schlüssel ist sein Wert ansprechbar (mit einer Konkretisierung des Typformalparameters **V** als Datentyp), wobei es sich um ein (eventuell komplexes) Objekt handelt. Man könnte z.B. eine Personalverwaltungsdatenbank realisieren mit ...

- einer eindeutigen Personalnummer (Typ **Integer** als **K**-Konkretisierung)
- und einer geeigneten Klasse `Personal` (mit Instanzvariablen für den Namen, die Telefonnummer 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 10.5.2), sind also sehr schnell auffindbar, aber unsortiert.
- **TreeMap<K,V>**
  Die Schlüssel werden in einen Binärbaum verwaltet (vgl. Abschnitt 10.5.3), sind 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*!), die mittlerweile ebenfalls das generische 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.:

```
HashMap<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. Wer Abbildungs-Kollektionen in einem Multithreading - Programm (siehe Kapitel 16) einsetzen möchte, sollte sich z.B. bei Goetz (2006a) über die Optionen informieren.

Wie die Klasse **Vector<E>** (siehe Abschnitt 10.4.2) steht auch die Klasse **Hashtable<K,V>** trotz Anpassung an das Java Collection 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>** und seinen Methoden 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. Das Interface **Map<K,V>** verlangt von einer implementierenden Klasse u.a. die folgenden Instanzmethoden:

- **public V put(K** *key*, **V** *value***)**
  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***)**
  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 void clear()**
  Mit dieser Methode fordert man eine Abbildung auf, alle (Schlüssel-Wert) - Paare zu löschen.
- **public boolean isEmpty()**
  Mit dieser Methode erfährt man, ob die angesprochene Abbildung leer ist.
- **public V get(Object** *key***)**
  Man erhält den zum angegebenen Schlüssel gehörigen Wert oder **null**, falls der Schlüssel nicht vorhanden ist.
- **public V remove(Object** *key***)**
  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**.
- **public int size()**
  Liefert die Anzahl der Elemente in der Abbildung
- **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 zeitaufwendig nacheinander alle Elemente bis zum ersten Treffer inspizieren.
- **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 aus der angesprochenen Abbildung operiert. Man kann z.B. mit der **Set<K>** - Methode **clear()** sämtliche Schlüssel und damit sämtliche Elemente der Abbildung, löschen:

Quellcode-Fragment	Ausgabe
`Map<Integer,String> c = new TreeMap<>();` `c.put(1, "A");` `c.put(3, "C");` `c.put(2, "B");` `System.out.println(c);` `c.keySet().clear();` `System.out.println(c);`	`{1=A, 2=B, 3=C}` `{}`

  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 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 *internes* Interface (vgl. Abschnitt 9.2) innerhalb von **Map<K,V>** definiert wird:

```
public interface Map<K,V> {
  .  .  .
     interface Entry<K,V> {
        K getKey();
        V getValue();
        V setValue(V value);
        boolean equals(Object o);
        int hashCode();
     }
  .  .  .
}
```

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 offen zu legen, 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.

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:

```
e --> 1
t --> 5
s --> 1
p --> 1
L --> 1
o --> 3
l --> 1
O --> 1
i --> 1
```

Durch die Pfeilnotation wird betont, dass es sich tatsächlich um eine Abbildung im mathematischen Sinn (von einer Teilmenge der Buchstaben in die Menge der natürlichen Zahlen) handelt.

Die folgende 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 in Abschnitt 5.5):

```
public static HashMap<Character, Mint> countLetters(String text) {
 HashMap<Character, Mint> fred = new HashMap<>();
 Mint temp;
 for (int i = 0; i < text.length(); i++)
  if (Character.isLetter(text.charAt(i))) {
   Character c = new Character(text.charAt(i));
   if (fred.containsKey(c)) {
    temp = fred.get(c);
    temp.val++;
    fred.put(c, temp);
   } else
    fred.put(c, new Mint(1));
  }
 return fred;
}
```

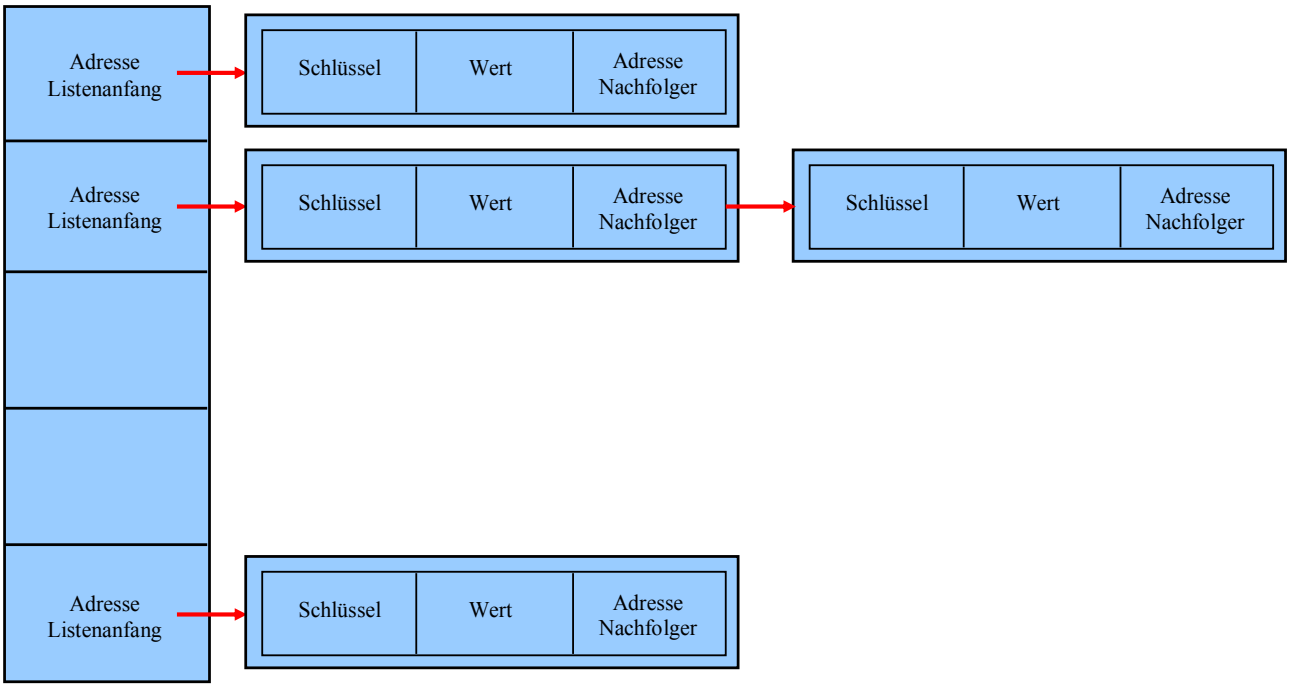
Wie die in Abschnitt 10.5.2 beschriebene Klasse **HashSet<E>** arbeitet auch die Klasse **HashMap<K,V>** mit einer Hashtabelle, verwendet also ein Array mit einfach verketteten Listen als Einträgen. Folglich muss die **K**-Konkretisierungsklasse eine **hashCode()** - Implementierung besitzen, welche die in Abschnitt 10.5.2 angegebenen Bedingungen erfüllt.[1]

Ein **HashMap<K,V>** - Objekt kann so skizziert werden:

---

[1] Ein Blick in den API-Quellcode zeigt übrigens, dass die Klasse **HashSet<E>** intern ein **HashMap<E,V>** - Objekt zur Datenablage verwendet, als **V**-Typ **Object** angibt und alle Elemente mit einem Dummy-Objekt als **V**-Wert anlegt:

```
// 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 <tt>HashMap</tt> instance has
 * default initial capacity (16) and load factor (0.75).
*/
public HashSet() {
  map = new HashMap<E, Object>();
}
```



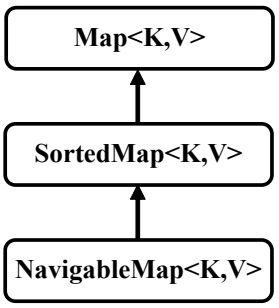
Um die (Schlüssel-Wert) - Paare in einem **HashMap<K,V>** - Container sukzessive anzusprechen, kann man über die Methode **entrySet()** ein Mengenverwaltungsobjekt mit den (Schlüssel-Wert) - Paaren als Elementen anfordern und dessen Iterator (siehe Abschnitt 10.7) benutzen. Dabei zeigt sich erwartungsgemäß 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, und im Ergebnis erhält man einen Iterator, der die Einfügereihenfolge verwendet.

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:



Das Interface **SortedMap<K,V>** fordert von implementierenden Klassen u.a. die folgenden Methoden:

- **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, so dass 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.

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. Zum Interface-Datentyp **Map.Entry<K,V>** siehe die Beschreibung der **Map<K,V>** - Methode **entrySet()** in Abschnitt 10.6.1.
- **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.

Existiert kein passendes Element, liefern die Methoden **firstEntry()**, **lastEntry()**, **pollFirstEntry()**, **pollLastEntry()**, **ceilingKey()**, **floorKey()**, **higherKey()**, **lowerKey()**, **ceilingEntry()**, **floorEntry()**, **higherEntry()** und **lowerEntry()** den Wert **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.

Analog zur Klasse **TreeSet<E>** (siehe Abschnitt 10.5.3) gibt es auch bei der Klasse **TreeMap<K,V>** zwei Möglichkeiten, die Ordnung der Elemente zu begründen:

- Der Schlüsseltyp **K** erfüllt das Interface **Comparable<K>**, besitzt also eine Instanzmethode **compareTo()**.
- Man übergibt dem **TreeMap<K,V>** - Konstruktor ein Objekt, das die Schnittstelle **Comparator<K>** erfüllt und folglich für den Typ **K** eine geeignete Vergleichsmethode

  public int compare(K *k1***, K** *k2***)**

  bietet.

Ersetzt man in der Buchstabenfrequenzen - Methode `countLetters()` aus Abschnitt 10.6.2 das **HashMap<Character,** `Mint`**>** - Objekt durch ein **TreeMap<Character,** `Mint`**>** - Objekt,

```
public static TreeMap<Character, Mint> countLetters(String text) {
 TreeMap<Character, Mint> fred = new TreeMap<>();
 Mint temp;
 for (int i = 0; i < text.length(); i++)
  if (Character.isLetter(text.charAt(i))) {
   Character c = new Character(text.charAt(i));
   if (fred.containsKey(c)) {
    temp = fred.get(c);
    temp.val++;
    fred.put(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 sind einige Methoden aus dem Interface **NavigableMap<Integer,String>** (vgl. Abschnitt 10.6.3) bei der Arbeit zu beobachten:

Quellcode	Ausgabe
`import java.util.*;` `class Prog {` `  public static void main(String[] args) {` `   TreeMap<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());` `   fi = 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));` `  }` `}`	`{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 Iteratoren

Die **Collection<E>** - Methode **iterator()** liefert ein Objekt, das die generische Schnittstelle **Iterator<E>** erfüllt und folglich u.a. die folgenden Methoden beherrscht:

- **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**

- **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. Dies ist die einzige zulässige Modifikation der Kollektion während einer Iteration. Bei sonstigen Änderungen ist das Verhalten des Iterators unspezifiziert.
  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 offen zu legen, wenn nur eine Pseudo-Implementation vorhanden ist.

Das folgende Programm demonstriert die Verwendung des Iterators zu einem **LinkedList<String>** - Objekt:

Quellcode	Ausgabe

```
import java.util.*;
class Prog {
 public static void main(String[] args) {
  LinkedList<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);
 }
}
```

```
Otto
Luise
Rainer

Rest der Liste:
Otto
Luise
```

Iteratoren haben einen Einsatzschwerpunkt bei verketteten Listen (siehe Abschnitt 10.4.2), wo sie im Vergleich zum zeitaufwendigen 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). Bei einem **ArrayList<E>** - Objekt bietet die Verwendung eines Iterators den Vorteil, problemlos auf eine verkettete Liste umsteigen zu können.

Dank der **for**-Schleife für Kollektionen (vgl. Abschnitt 3.7.3.2) ist der Iterator-Einsatz oft ohne nennenswerten Aufwand zu realisieren:

for (*Elementtyp Iterationsvariable* **:** *Kollektion***)**
    Anweisung

Diese Schleife verlangt vom Kollektionsobjekt das Interface **Iterator<E>** und verwendet im Hintergrund den somit verfügbaren Iterator.

Das vom Interface **Iterator<E>** abstammende Interface **ListIterator<E>** enthält zur Unterstützung von bidirektionalen Listenpassagen zusätzlich die Methoden **hasPrevious()** und **previous()** und außerdem die Methode **set()** zum Ersetzen von Listenelementen:

- **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 void set(E** *element***)**
  Das zuletzt von **next()** oder **previous()** gelieferte Element wird durch das Parameterobjekt ersetzt.

In den Klassen **ArrayList<E>**, **LinkedList<E>** und **Vector<E>** ist die Instanzmethode **listIterator()** vorhanden, die ein Objekt aus einer Klasse liefert, welche das Interface **ListIterator<E>** implementiert.

10.8 Die Service-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 wurde:

- **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 in 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 sind. 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).
- **public static <T extends Comparable<? super T>> void sort(List<T>** *liste***)**
  Eine Liste wird unter Verwendung der **compareTo()** - Methode ihres Elementtyps sortiert.
- **public static <T> void sort(List<T>** *liste***, Comparator<? super T>** *comp***)**
  Eine Liste wird unter Verwendung der **compare()** - Methode sortiert, die das vom zweiten Parameter referenzierte Objekt beherrscht.
- **public static void reverse(List<?>** *liste***)**
  Die Elemente einer Liste erhalten eine umgekehrte Reihenfolge.
- **public static void shuffle(List<?>** *liste***)**
  Diese Methode bringt die Elemente einer Liste in eine neue, zufällige Reihenfolge.
- **public <E> Collection<E> synchronizedCollection(Collection<E>** *coll***)**
  public <E> List<E> synchronizedList(List<E> *list***)**
  public <E> Set<E> synchronizedSet(Set<E> *set***)**
  public <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 über Multithreading behandelt.
- **public <E> Collection<E> unmodifiableCollection(Collection<? extends E>** *coll***)**
  public <E> List<E> unmodifiableList (List<? extends E> *list***)**
  public <E> Set<E> unmodifiableSet(Set<? extends E> *set***)**
  public <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.

Im folgenden Programm werden einige **Collections**-Methoden demonstriert:

---

[1] Diese Erklärung stammt von der (am 14.02.2016 abgerufenen) Webseite http://www.angelikalanger.com/GenericsFAQ/FAQSections/ProgrammingIdioms.html#FAQ104.

Quellcode	Ausgabe
`import java.util.*;` `class Prog {` `  public static void main(String[] args) {` `  LinkedList<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));` `  }` `}`	`Original:  [Otto, Luise, Rainer]` `Sortiert:  [Luise, Otto, Rainer]` `Invertiert:  [Rainer, Otto, Luise]` `Verwirbelt:  [Otto, Luise, Rainer]` `Minimum:  Luise` `Maximum:  Rainer`

10.9 Übungsaufgaben zu 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 drastisch reduziert.

2) Erweitern Sie die als Aufgabe zu Abschnitt 5.2 in Auftrage 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 TreeMap<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.TreeMap;
import de.uni_trier.zimk.strutil.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));

  TreeMap<Character,Integer> freqs = StringUtil.getStartCharFreqs(s);
  System.out.println("Anfangsbuchstabe  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 dazu gelernt 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 Kontrollausgaben 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, unterbrochene Netzverbindungen) kann die reguläre Ausführung einer Methode scheitern. Java bietet ein modernes Verfahren zur Meldung und Behandlung von Problemen: 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 wird somit über das Problem informiert und mit Daten für die Behandlung versorgt.

Die Initiative beim Auslösen einer Ausnahme kann ausgehen …

- vom Laufzeitsystem
  Entdeckt es einen Fehler, der nicht zu schwerwiegend ist und vom Benutzerprogramm prinzipiell behoben werden kann, wirft sie ein Ausnahmeobjekt, z.B. ein Objekt aus der Klasse **ArithmeticException** bei einer versuchten Ganzzahldivision durch 0.
- vom Programm, wozu auch die verwendeten Bibliotheksklassen gehören
  In jeder Methode kann mit der **throw**-Anweisung (siehe Abschnitt 11.6) eine Ausnahme generiert werden.

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 so genannten *Exception-Handler*) enthält. Gegebenenfalls wird dieser Exception-Handler angesprungen und erhält quasi als Aktualparameter das Ausnahmeobjekt mit Informationen über das Problem. Entscheidet sich eine Methode nach der Ausnahmebehandlung gegen die Fortführung des ursprünglichen Handlungsplans, dann sollte erneut ein Ausnahmeobjekt geworfen werden, entweder das ursprüngliche oder ein informativeres.
- 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):

Whatever can go wrong will go wrong.

11.1 Unbehandelte Ausnahmen

Findet die JRE zu einer Ausnahme entlang der Aufrufersequenz bis hinauf zur **main()** - Methode keine Behandlungsroutine, 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(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at Fakul.convertInput(Fakul.java:3)
        at Fakul.main(Fakul.java:14)
```

Wird ein Programm im Rahmen unsere Entwicklungsumgebung Eclipse ausgeführt, besitzt der Unfallbericht eine auffällige Färbung und klickbare Verknüpfungen zu den Quellcodezeilen der betroffenen Methoden in der Aufrufsequenz, 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)
```

Wenn Sie zu den API-Methoden in der Aufrufsequenz statt einer Ortsangabe (aus Dateinamen und Zeilennummer) nur *Unknown Source* sehen, muss in Eclipse ein JDK (*mit* begleitendem Quellcode) als Laufzeitumgebung eingestellt werden (siehe Abschnitt 2.5.8.3).

Das folgende Programm soll die Fakultät zu einer nichtnegativen ganzen Zahl berechnen, 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 Klasse **Integer**:

```
class Fakul {
 static int convertInput(String instr) {
  int arg = Integer.parseInt(instr);
  if (arg >= 0  &&  arg <= 170)
   return arg;
  else
   return -1;
 }
```

[1] Ist keine Konsole vorhanden (z.B. nach dem Start eines Programms mit grafischer Bedienoberfläche über **javaw.exe**), bleibt dem Benutzer der Unfallbericht allerdings verborgen (siehe Abschnitt 1.2.4).

```
  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]);
  }
}
```

Das Programm kümmert sich durchaus um einige kritische Fälle. Die Methode **main()** überprüft, 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 Fähigkeiten zum Befragen des Benutzers besitzt, belehrt es den Benutzer und beendet sich durch Aufruf der Methode **System.exit()**, der als Aktualparameter ein **Exitcode** übergeben wird. Dieser landet beim Betriebssystem und steht unter Windows 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

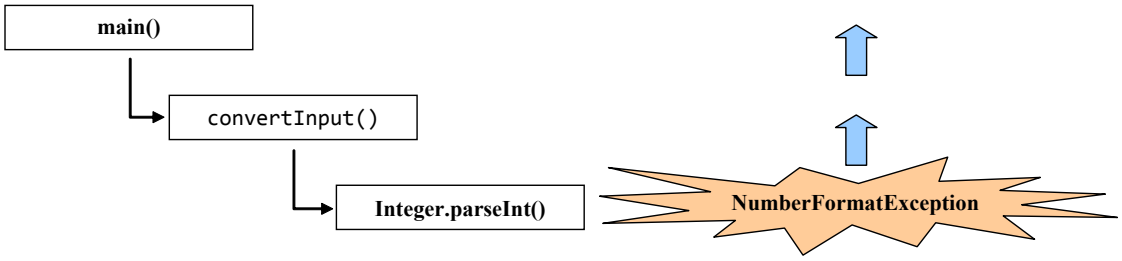
```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at Fakul.main(Fakul.java:26)
```

erscheint eine verwertbare Information.

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()** erkennt die spezielle Bedeutung des Rückgabewerts -1, so dass z.B. unsinnige Fakultätsberechnungen für negative Argumente vermieden werden. Diese traditionelle Fehlerbehandlung per Rückgabewert (*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 keine Behandlungsroutine bereithalten, bringt die JRE den im Ausnahmeobjekt enthaltenen Unfallbericht auf die Konsole (siehe oben) und beendet das Programm.[3]

---

[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 Java-Klassenbibliothek ignorieren wir an dieser Stelle. In 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 ruft das per Aktualparameter übergebene **Exception**-Objekt auf, die Methode **printStackTrace()** auszuführen.

11.2 Ausnahmen abfangen

11.2.1 Die try-catch-finally - Anweisung

In Java wird die Behandlung von Ausnahmen über die **try-catch-finally** - Anweisung unterstützt:

```
try {
  Überwachter Block mit Anweisungen für den regulären Ablauf
}
catch (Ausnahmeklassenliste1 parameter1) {
  Anweisungen für die Behandlung einer Ausnahme aus einer aufgelisteten Klasse
}
// Optional können weitere Ausnahmen abgefangen werden:
catch (Ausnahmeklassenliste2 parameter2) {
  Anweisungen für die Behandlung einer Ausnahme aus einer aufgelisteten 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
}
```

Die Anweisungen für den ungestörten Ablauf setzt man in den **try**-Block. Treten bei der Ausführung dieses geschützten bzw. überwachten Blocks *keine* Fehler auf, wird das Programm hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**-Block ausgeführt wird.

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 nach einem **catch**-Block, welcher eine Ausnahme der betroffenen Klasse behandeln kann.

Ein **catch**-Block, den man oft 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 einheitlich Behandlung vereinbart werden soll, z.B.:

```
catch (NumberFormatException | ArithmeticException e) {
  . . .
}
```

Bei einem Multi-Catch - Block sind folgende Regeln zu beachten:

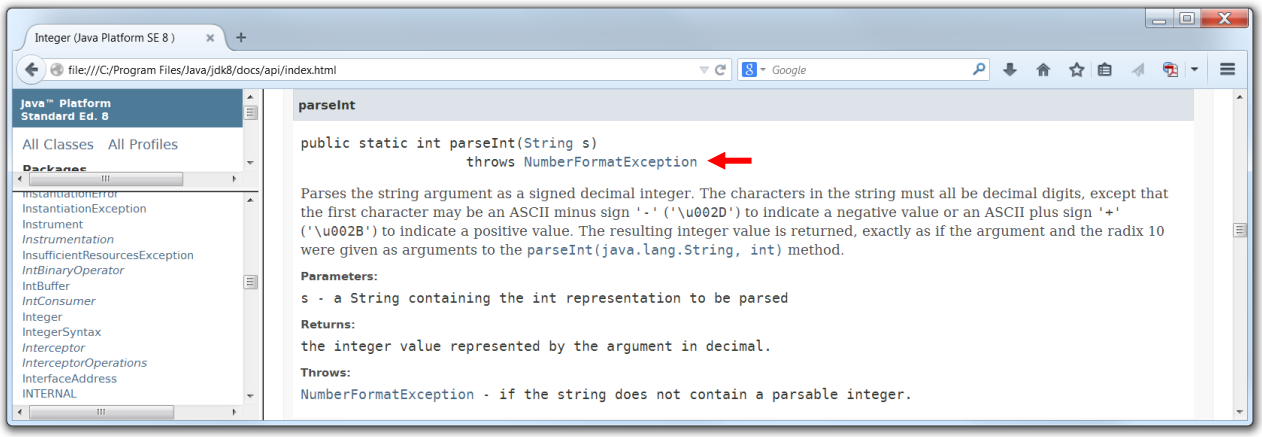
- Die Namen der Ausnahmeklassen werden durch einen senkrechten Strich | getrennt, der bekanntlich (zwischen zwei logischen Ausdrücken) auch für die logische ODER-Operation steht (vgl. Abschnitt 3.5.5). Das ist eine gute Wahl, denn im obigen Beispiel wird der Exception-Handler aktiv, wenn eine **NumberFormatException** *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 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 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.printf("Keine ganze Zahl im Intervall [0, 170]: " + args[0]);
           break;
  case -2: System.out.printf("Fehler beim Konvertieren von: " + 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 JRE, z.B.:

```
Fehler beim Konvertieren von: vier
```

Wenn ein **catch**-Block erfolgreich (d.h. ohne weiteres Ausnahmeereignis) ausgeführt worden ist, und die Methode dabei nicht per **return**-Anweisung verlassen wurde, dann wird die Methode hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**-Block ausgeführt wird. Die eventuell im überwachten **try**-Block auf die Anweisung, die zur Ausnahme geführt hat, noch folgenden Anweisungen werden *nicht* ausgeführt. Damit ein Algorithmus nach einer erfolgreichen Störungsbeseitigung hinter dem betroffenen Teilschritt fortgesetzt werden kann, ist für den (aus wenigen Anweisungen bestehenden) Teilschritt eine separate **try-catch-finally** - Anweisung erforderlich.

11.2.1.2 finally

In einen **finally**-Block gehören Anweisungen, die auf jeden Fall ausgeführt werden sollen:

- nach der ungestörten Ausführung des **try**-Blocks
  Auch ein vorzeitiges Verlassen der Methode durch eine **return**-Anweisung im **try**-Block verhindert *nicht* die Ausführung des **finally**-Blocks.
- nach einer Ausnahmebehandlung in einem **catch**-Block
- nach dem Auftreten einer unbehandelten Ausnahme im **try**-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 16.2.3). Die Methode zum Abheben aktiviert die Sperre, erfragt dann beim Benutzer den gewünschten Betrag und gibt die Sperre schließlich frei, damit der Einzahlungs-Thread wieder Zugang zum Konto erhält. Für die Nachfrage beim Benutzer kommt ausnahmsweise *nicht* die Methode `gint()` der Klasse `Simput` zum Einsatz, weil für den aktuellen Demonstrationszweck eine Methode benötigt wird, von der Ausnahmen zu erwarten sind. Daher 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 eingelesen 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**. Daher wird sie im Rahmen einer **try**-Anweisung mit **catch**-Block für die **NumberFormatException** aufgerufen:

```
void abheben() {
 Scanner input = new Scanner(System.in);
 int amount;
 while (true) {
  try {Thread.sleep(3000);} catch (Exception e) {}
  lock.lock();
  try {
   System.out.print("\n\nWelcher Betrag soll abgehoben werden: ");
   amount = Integer.parseInt(input.nextLine());
   if (amount < 0) {
    stopp = true;
    break;
   }
   konto -= amount;
   System.out.println("Neuer Kontostand:  " + konto);
  } catch(NumberFormatException e) {
   System.out.println("Kein gültiger Betrag!");
  } finally {
   lock.unlock();
  }
 }
}
```

Es ist unbedingt sicherzustellen, dass die ggf. vom 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: vier
Kein gültiger Betrag!

Kontostand erhöht auf: 84
Kontostand erhöht auf: 91
Kontostand erhöht auf: 112

Welcher Betrag soll abgehoben werden: 4
Neuer Kontostand:  108

Kontostand erhöht auf: 112
Kontostand erhöht auf: 113
Kontostand erhöht auf: 126
```

11.2.2 Programmablauf bei der Ausnahmebehandlung

Findet das Laufzeitsystem für eine Ausnahme in der aktuellen Methode keinen zuständigen **catch**-Block, dann sucht es entlang der Aufrufersequenz weiter. Dies macht es leicht, die Behandlung einer Ausnahme der bestgerüsteten Methode zu überlassen. In folgendem Beispiel dürfen Sie allerdings keine optimierte Einsatzplanung erwarten. Es soll demonstrieren, welche Programmabläufe sich bei Ausnahmen ergeben können, die auf verschiedenen Stufen einer Aufrufhierarchie behandelt werden. Um das Beispiel einfach zu halten, wird auf Nützlichkeit und 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 erg = 0;
  try {
   System.out.println("try-Block von calc()");
   erg = Integer.parseInt(instr);
   erg = 10 % erg;
  }
  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 erg;
 }
```

```
    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

```
erg = 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. (siehe 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 wenig überzeugenden Ausgabe

```
10 % acht = 0
```

kommt es, weil die **NumberFormatException** in `calc()` *nicht sinnvoll* behandelt wird. Das aktuelle Beispiel soll ausschließlich 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. Bevor die Methode verlassen wird, um entlang der Aufrufsequenz nach einem geeigneten Handler zu suchen, wird noch ihr **finally**-Block ausgeführt.

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 gar 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:

```
try-Block von main()
finally-Block von main()
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
 at Sequenzen.main(Sequenzen.java:25)
```

In einer komplexen Methode 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:

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

Das Ergebnis enthält den Namen der Ausnahmeklasse und eventuell eine situationsspezifische Information, falls eine solche beim Erstellen des Ausnahmeobjekts via Konstruktor erzeugt 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 Methode **printStackTrace()** befördert man diese Ausgabe zu dem per Parameter benannten **PrintStream**-Objekt, z.B. zur Standardausgabe:

```
catch (NumberFormatException e) {
 e.printStackTrace(System.out);
    .  .  .
}
```

Im Beispiel erscheint:

```
java.lang.NumberFormatException: For input string: "acht"
 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 Sequenzen.calc(Sequenzen.java:6)
 at Sequenzen.main(Sequenzen.java:25)
```

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 Aufrufersequenz gleich *zwei* **Integer**-Methoden **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 traditionellen Fehlerbehandlung

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 **Return Code** mit Datentyp **int** verwendet, wobei die Null einen erfolgreichen Ablauf meldet, während andere Zahlen für einen bestimmten Fehlertyp 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 (siehe Beispielprogramm in Abschnitt 11.2.1.1)

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 länglichen und recht unübersichtlichen Konstruktion (nach Mössenböck 2005, S. 254):

---

[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 ruft das per Aktualparameter übergebene **Exception**-Objekt auf, die Methode **printStackTrace()** auszuführen.

```
public static void main(String[] args) {
 int returncode;
 returncode = m1();
 if (returncode == 0) {
  returncode = m2();
  if (returncode == 0) {
   returncode = m3();
   // Behandlung für diverse m3() - Fehler
   if (returncode == 1) {
     // . . .
   }
     // . . .
  } else {
   // Behandlung für diverse m2() - Fehler}
  }
 } else {
  // Behandlung für diverse m1() - Fehler
 }
}
```

Mit Hilfe der Ausnahmetechnik bleibt 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. Damit ein Algorithmus nach einer erfolgreichen Störungsbeseitigung hinter dem betroffenen Teilschritt fortgesetzt werden kann, ist für den Teilschritt eine separate **try-catch** - Anweisung erforderlich. In der Regel enthält ein **try**-Block nicht allzu viele Anweisungen.

Beim traditionellen Verfahren nutzt ein gut gesetzter Rückgabewert natürlich nichts, wenn sich der Aufrufer nicht darum kümmert.

Neben dem unübersichtlichen Quellcode und der ungesicherten Beachtung eines Rückgabewerts ist am klassischen Verfahren zu bemängeln, dass eine Fehlerinformation aufwändig entlang der Aufrufersequenz nach oben gemeldet werden muss, wenn sie nicht an Ort und Stelle behandelt werden soll.

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 Fehlerbehandlung 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("Zaehler: ");
  setzeZaehler(Simput.gint());
} while (Simput.checkError());
```

Auch die Methoden der zur Ausgabe in Textdateien geeigneten API-Klasse **PrintWriter** (siehe Abschnitt 15.4.1.5) werfen *keine* **IOException**, sondern setzen ein Fehlersignal, das mit der Methode **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 vom Laufzeitsystem beendet.
- Obligatorische Vorbereitung auf Ausnahmen
  In Java wird zwischen der obligatorischen und der freiwilligen Ausnahmebehandlung unterschieden (siehe Abschnitt 11.5). 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 zu verhindern
- Automatische Weitermeldung bis zur bestgerüsteten Methode
  Oft ist der unmittelbare Verursacher nicht gut gerüstet zur Behandlung einer Ausnahme, z.B. nach dem vergeblichen Öffnen einer Datei. Dann soll eine „höhere“ Methode über das weitere Vorgehen entscheiden.
- Bessere Lesbarkeit des Quellcodes
  Mit Hilfe einer **try-catch-finally** - Konstruktion erreicht man eine bessere Trennung zwischen den Anweisungen für den normalen Programmablauf und den diversen Ausnahmebehandlungen, so dass 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 klassischen 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. Es folgende einige (keinesfalls vollständige) Einsatzempfehlungen für die verschiedenen Techniken der Fehlerbehandlung.

- Wenn ein **Problem mit erheblicher Wahrscheinlichkeit** auftritt, sollte eine routinemäßige, aktive Kontrolle stattfinden. Eine auf das Problem stoßende Methode sollte 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.
- Bei **Fehlern mit geringer Wahrscheinlichkeit** 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.

Manche Klassen bieten für kritische Aktionen eine Methode zur Prüfung der Realisierbarkeit an, so dass gescheiterte Aufrufe vermieden werden können. In der Klasse **Scanner**, die sich auch dazu eignet, aus einer Textdatei Werte primitiver Datentypen zu lesen (vgl. Abschnitt 15.5), finden sich z.B. die beiden folgenden Methoden:

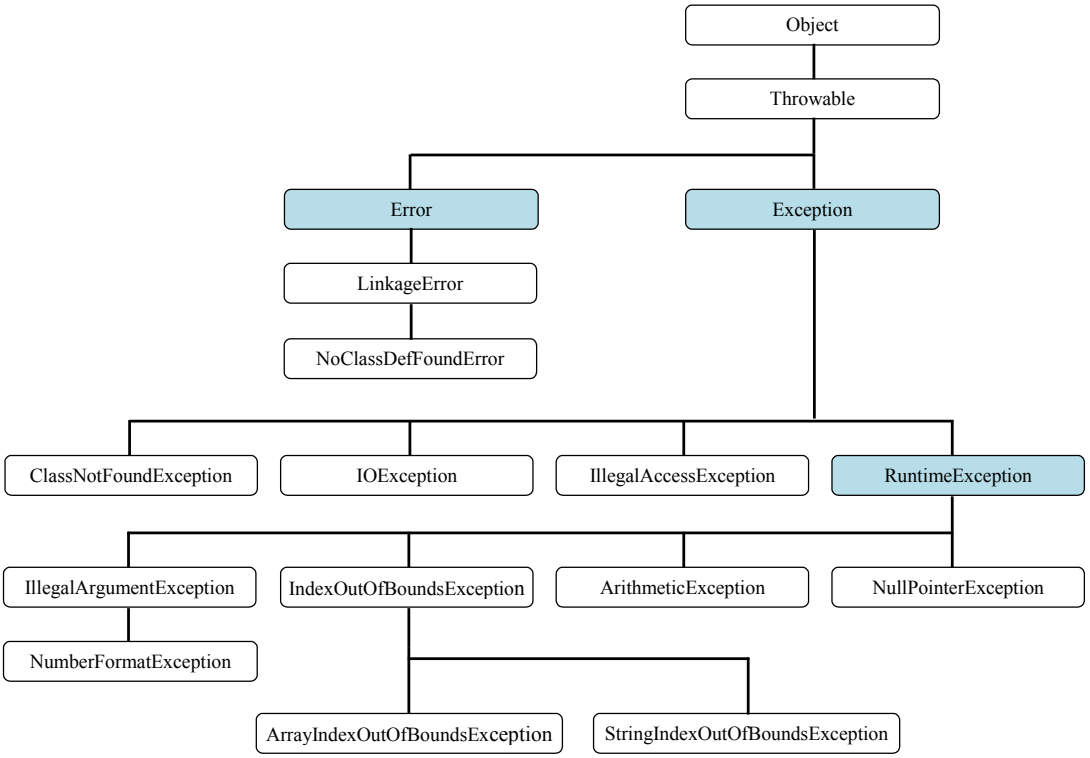
- **public double nextDouble()**
  Es wird versucht, aus der Eingabedatei eine abgegrenzte Zeichenfolge zu ermitteln und als **double**-Zahl zu interpretieren. Wenn dies misslingt, wirft die Methode eine Ausnahme.
- **public boolean hasNextDouble()**
  Es wird überprüft, ob das eben beschriebene Unterfangen 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++;
}
```

11.4 Ausnahmeklassen in Java

In der folgenden Abbildung sind wichtige, teilweise im weiteren Textverlauf noch anzusprechende Ausnahmeklassen mit ihren Vererbungsbeziehungen zu sehen:



In **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.

Wie die obige Klassenhierarchie zeigt, gilt neben der **Exception** auch der **Error** als **Throwable**. Allerdings geht es hier um kapitale Pannen, die auf jeden Fall einen regulären Programmablauf verhindern. Daher ist ein Abfangen von **Error**-Objekten nicht sinnvoll und nicht vorgesehen. Kann die JVM z.B. eine für den Programmablauf benötigte Klasse nicht finden, meldet sie einen **NoClassDefFoundError** und beendet das Programm:[1]

```
>java PackDemo
Exception in thread "main" java.lang.NoClassDefFoundError: demopack/A
        at PackDemo.main(packdemo.java:7)
```

11.5 Obligatorische und freiwillige Vorbereitung auf eine Ausnahme

Bei Ausnahmeobjekten aus der Klasse **RuntimeException** und aus daraus abgeleiteten Klassen (siehe Klassenhierarchie in Abschnitt 11.4) ist es dem Programmierer *freigestellt*, ob er sich auf

---

[1] 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 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 eine Klasse mit dem angegebenen Namen existiert.

eine Behandlung vorbereiten möchte. Weil der Compiler *nicht* prüft, ob eine Behandlung erfolgt, spricht man von *unkontrollierten 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 *kontrollierten Ausnahmen* (engl.: *checked exceptions*. Bei Verwendung einer Methode, die Probleme über obligatorische Ausnahmen meldet, muss der Aufrufer …

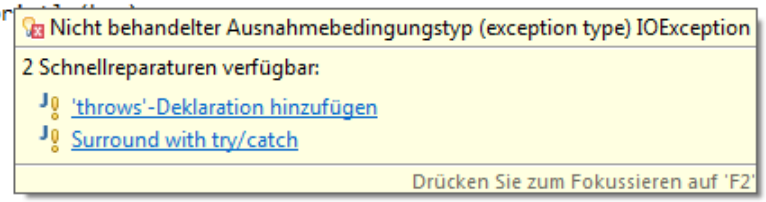
- entweder den Aufruf in einer **try**-Anweisung mit geeignetem **catch**-Block vornehmen (vgl. Abschnitt 11.2)
- oder im eigenen Definitionskopf das Durchreichen der Ausnahmen ankündigen (vgl. Abschnitt 11.6).

Bei einer Ausnahmeklasse *ohne* Behandlungszwang ist eine solche Vorbereitung nicht erforderlich. Allerdings muss *jede geworfene* Ausnahme (unabhängig von der Klassenzugehörigkeit) behandelt werden, um die Beendigung des Programms durch das Laufzeitsystem zu verhindern.

Ausnahmeobjekte werden auch in vielen anderen Programmiersprachen unterstützt, wobei aber nur Java zwischen kontrollierten und unkontrollierten Ausnahmen unterscheidet. In den anderen Sprachen (z.B. C#, C++) sind *alle* Ausnahmen vom unkontrollierten Typ.

In folgendem 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 **IOException** (siehe API-Dokumentation) behandlungspflichtig ist, besteht der Compiler auf einer Behandlung oder expliziten Weitergabe:

```
class ChEx {
    public static void main(String[] args)  {
        int key = 0;
        System.out.print("Beliebige Taste + Return: ");
        key = System.in.read();
        System.out.pr
    }
}
```



Da wir mittlerweile die **try**-Anweisung beherrschen, ist das Problem leicht zu lösen:

```
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 (Exception e) {}
```

Grundsätzlich ist es riskant, eine Ausnahme auf diese Weise zu eliminieren statt sie zu behandeln. Im letzten Beispiel ist das Verhalten ausnahmsweise akzeptabel, weil es in der Regel nicht interessiert, welche Ausnahme die statische Methode **sleep()** der Klasse **Thread** unterbrochen und damit das geplante Schläfchen abgekürzt hat (siehe Kapitel über Multithreading).

Zur Frage nach den Kriterien für die Klassifikation einer Ausnahme als (un)checked gibt Ullenboom (2012a, Abschnitt 6.5.5) einige Hinweise. Danach passt eine unchecked exception (mit der Basisklasse **RuntimeException**) zu folgenden Situationen:

- Ursache ist ein Programmierfehler (z.B. bei einer **ArrayIndexOutOfBoundsException**). Man muss sich nicht darauf vorbereiten, im laufenden Programm auf einen solchen Fehler angemessen zu reagieren. Stattdessen muss der Fehler schleunigst beseitigt werden.
- Das Problem kann vom laufenden Programm kaum behoben bzw. kompensiert werden. Folglich wird auf den Behandlungszwang verzichtet.

In den folgenden Fällen ist eine checked exception angemessen:

- Externe, vom Programmierer nicht zu kontrollierende Bedingungen haben das Problem verursacht (z.B. eine unterbrochene Netzverbindung).
- Das laufende Programm sollte in der Lage sein, das Problem zu kompensieren (mit einem Plan B).

Gleich lernen Sie eine Möglichkeit kennen, auf die Behandlung einer obligatorischen Ausnahme zu verzichten und dem Vorgänger in der Aufrufersequenz das Problem zu überlassen.

11.6 Ausnahmen in einer eigenen Methode auslösen und ankündigen

11.6.1 Ausnahmen auslösen (throw)

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. 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:

```
static int convertInput(String instr) throws IllegalArgumentException {
 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).

Viele **catch**-Blöcke betätigen sich als Informationsvermittler und werfen selbst eine Ausnahme, um 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. 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.

11.6.2 Ausnahmen ankündigen (throws)

Für die Benutzung einer Methode (durch andere Programmierer) ist es von Vorteil, wenn die von dieser Methode zu erwartenden Ausnahmen im Definitionskopf dokumentiert werden, was in der **throws**-Klausel zu geschehen hat, z.B.:

```
static int convertInput(String instr) throws IllegalArgumentException
```

Durch Kommata getrennt können nach dem Schlüsselwort **throws** auch *mehrere* Ausnahmeklassen angekündigt werden.

Bei Unchecked Exceptions (**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. Alle übrigen Ausnahmen (z.B. **IOException**) müssen entweder behandelt oder deklariert werden. In der aktuellen `convertInput()` - Variante erfolgt die Deklaration freiwillig, weil **IllegalArgumentException** von **RuntimeException** abstammt.

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.:

```
try {
 argument = convertInput(args[0]);
} catch (IllegalArgumentException iae) {
 System.out.println(iae.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());
 System.exit(1);
}
```

11.6.3 Pflicht zur Ausnahmebehandlung abschieben

In Abschnitt 11.5 haben Sie erfahren, dass man beim Aufruf einer Methode, die potentiell *obligatorische* 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 einer **catch**-Klausel 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 11.5 kann sich die Methode **main()**, welche den potentiellen **IOException**-Absender **read()** ruft, der Pflicht zur Ausnahmebehandlung auf folgende Weise entziehen:

```
class ChEx {
 public static void main(String[] args) throws java.io.IOException {
  int key = 0;
  System.out.print("Beliebige Taste + Return: ");
  key = System.in.read();
  System.out.println(key);
 }
}
```

Man kann mit **throws** also nicht nur selbst erzeugte Ausnahmen anmelden, sondern auch von aufgerufenen Methoden stammende Ausnahmen weiterleiten. Im Falle von Checked Exceptions kann man sich so der Behandlungspflicht entledigen.

11.6.4 Compiler-Intelligenz beim erneuten Werfen von abgefangenen Ausnahmen

Seit Java 7 bietet der Compiler beim erneuten Werfen einer obligatorischen 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 { }
  .  .  .
public 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, so dass 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) {
 throw e;
}
```

[1] Am 16.01.2015 übernommen von:
http://docs.oracle.com/javase/7/docs/technotes/guides/language/catch-multiple.html

11.7 Ausnahmen definieren

Mit Hilfe von Ausnahmeobjekten kann eine Methode beim Auftreten von Fehlern die aufrufende Methode präzise über Ursachen und Begleitumstände informieren. Dabei müssen sich Software-Entwickler keinesfalls auf die im Java-API vorhandenen Ausnahmeklassen beschränken, sondern können auch eigene Ausnahmentypen definieren, z.B.:

```
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, so dass 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 **Exception** 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
- in `error` einen numerischen Indikator für die Fehlerart:
  - -1: unbekannter Fehler
  - 1: kein Argument vorhanden
  - 2: Zeichenfolge kann nicht konvertiert werden
  - 3: konvertierter Wert außerhalb des erlaubten Bereichs
- in `value` das Konvertierungsergebnis (falls vorhanden, sonst -1)

Durch die Wahl der Basisklasse **Exception** ist eine *checked exception* entstanden, die im `convertInput()` - Methodenkopf deklariert werden *muss*:

```
static int convertInput(String instr) throws BadFactorialArgException {
 int arg;
 try {
  arg = Integer.parseInt(instr);
  if (arg < 0  || arg > 170)
   throw new BadFactorialArgException(
    "Unzulässiges Argument (erlaubt: 0 bis 170)", instr, 3, arg);
  else
   return arg;
 } catch (NumberFormatException e) {
  throw new BadFactorialArgException("Fehler beim Konvertieren", instr, 2, -1);
 }
}
```

Ebenso sind `convertInput()` - Aufrufer gezwungen, entweder die `BadFactorialArgException` in einem **catch**-Block zu behandeln oder im eigenen Methodenkopf die potentielle Ausnahme zu deklarieren:

```
public static void main(String[] args) {
 int argument = -1;

 try {
  if (args.length == 0)
   throw new BadFactorialArgException("Kein Argument angegeben", "", 1, -1);
  argument = convertInput(args[0]);
  double fakul = 1.0;
  for (int i = 1; i <= argument; i++)
   fakul = fakul * i;
  System.out.println("Fakultät: " + fakul);
 } catch (BadFactorialArgException e) {
  System.out.println("Fehler: " + e.getError() + " " + e.getMessage());
  switch (e.getError()) {
   case 2 : System.out.println("Zeichenfolge: \"" + e.getInstr() + "\"");
            break;
   case 3 : System.out.println("Wert: " + e.getValue());
            break;
  }
 }
}
```

Um eine *Unchecked Exception* zu erzeugen, wählt man eine Basisklasse aus der **RuntimeException**-Hierarchie. Am Ende von Abschnitt 11.5 wurden einige Kriterien für die Entscheidung zwischen einer kontrollierten und einer unkontrollierten Ausnahme genannt. Oft fällt diese Entscheidung schwer, und viele Entwickler entziehen sich der Mühe, indem sie generell unkontrollierte Ausnahmen verwenden (siehe Ullenboom 2012a, Abschnitt 6.5.5). Das kann so falsch nicht sein, weil andere Programmiersprachen (z.B. C#, C++) ausschließlich unkontrollierte Ausnahmen kennen.

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 sehr oft anzutreffende Dateifreigabe per **finally**-Block mit **close()** - Aufruf. Im folgenden Beispiel wird (teilweise dem Kapitel 15 vorgreifend) zur Demonstration der traditionellen Dateifreigabe eine statische Methode namens `mean()` definiert, die mit Hilfe eines **DataInputStream**-Objekts aus einer Binärdatei 100 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;
   for (int i = 1; i <= 100; i++)
    sum += dis.readDouble();
   System.out.println("Mittelwert zur Datei " + eingabe + ":  " + sum/100);
  } catch (IOException ioe) {
   System.out.println(ioe);
  } finally {
   if (dis != null)
    try {dis.close();} catch (IOException ioc) {
     System.out.println(ioc);
    };
  }
 }

 public static void main(String args[]) {
  mean("eingabe.dat");
 }
}
```

Bei Beendigung einer Anwendung werden alle von ihr geöffneten Dateien automatisch geschlossen, so dass 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 **IOException**-Objekte kommunizieren, auf die sich ein Aufrufer obligatorisch vorbereiten muss (vgl. Abschnitt 11.5). 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 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 eben geschilderten 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 (siehe Abschnitt 11.5), 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 nichts von der primären Ursache des Problems.

Neben dem nicht ganz unerheblichen Aufwand besteht ein weiterer Nachteil der traditionellen Lösung besteht 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 AutomaticallyClose {
 static void mean(String eingabe) {
  try (DataInputStream dis = new DataInputStream(new FileInputStream(eingabe))) {
   double sum = 0.0;
   for (int i = 1; i <= 100; i++)
    sum += dis.readDouble();
   System.out.println("Mittelwert zur Datei " + eingabe + ":  " + sum/100);
  } catch (IOException ioe) {
   System.out.println(ioe);
  }
 }

 public static void main(String args[]) {
  mean("eingabe.dat");
 }
}
```

[1] In der Klasse **FileInputStream** ist eine **finalize()** - Methode definiert, die ggf. vom Garbage Collector aufgerufen wird und für das Schließen der Datei sorgt.

Die **finally**-Klausel mit der **close()** - Anweisung ist überflüssig geworden, und die **DataInputStream**-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))) {
  .  .  .
}
```

11.9 Übungsaufgaben zu Kapitel 11

1) Welche von den 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 in 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. In folgendem Beispiel wird eine Methode namens `dualLog()` 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]

[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).

Quellcode	Ausgabe
`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);` `  }` `}`	`NaN`

Erstellen Sie eine Variante, die bei ungeeigneten Argumenten eine **IllegalArgumentException** wirft.

12 Funktionales Programmieren

Nach Horstmann (2014b) besteht der wesentliche Fortschritt von Java 8 in der Unterstützung der *funktionalen Programmierung*:

> The principal enhancement in Java 8 is the addition of functional programming constructs to its object-oriented roots.

Als zentrale Begriffe der funktionalen Programmierung in Java 8 sind zu nennen:

- **Lambda-Ausdrücke** (alias: *closures*)
  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.[1] Das folgende Beispiel empfängt einen Parameter vom Typ **String** und liefert eine Rückgabe vom Typ **boolean**, die genau dann **true** ist, wenn die Parameterzeichenfolge mindestens die Länge 5 besitzt:
  ```
  (String s) -> s.length() >= 5
  ```
  Lambda-Ausdrücke sind dazu vorgesehen, von einer anderen Methode ausgeführt zu werden, um deren Verhalten zu komplettieren oder zu konfigurieren. Ein Lambda-Ausdruck wird als Aktualparameter oder in einer Wertzuweisung akzeptiert, wenn dort eine bestimmte Interface-Implementation erwartet wird.
- **Ströme**
  Ein Strom ist eine Sequenz von Elementen aus einer Quelle (z.B. einer Kollektion) und unterstützt Operationen zur sequentiellen oder parallelen Massenabfertigung der Elemente (engl.: *bulk operations* oder *aggregate operations*). Beim Design hatte die bequeme (und damit *tatsächlich genutzte*) Parallelisierung von Standardoperationen bei Kollektionsobjekten (z.B. Listen) mit dem Ergebnis guter Leistungswerte auf Multi-Core - Systemen hohe Priorität.

12.1 Lambda-Ausdrücke

Wer zu einem Buch über Java 8 greift (z.B. Inden 2015; Sharan 2015), wird nicht selten mit einer bis Java 7 völlig ungewohnten Syntax wie im folgenden Aufruf der Methode **setOnAction()** konfrontiert:

```
Button button = new Button();
  .  .  .
button.setOnAction(event -> label.setText("Hallo JavaFX"));
```

Hier wird die Klickereignisbehandlung für einen Befehlsschalter mit Hilfe eines Lambda-Ausdrucks realisiert. Auch für andere Aufgaben sind Lambda-Ausdrücke so attraktiv und mittlerweile so verbreitet, dass wir uns mit dieser neuen Option beschäftigen müssen.

12.1.1 Sinn und Syntax von Lambda-Ausdrücken

Ein weiteres Zitat von Horstmann (2014b) ist dazu geeignet, den Nutzen und die Bedeutung vom Lambda-Ausdrücken zu umreißen:

---

[1] Dass tatsächlich ein Objekt im Spiel ist, das die Methode ausführt, wird sich noch zeigen (siehe Abschnitt 12.1.1.3). Man kann den Lambda-Ausdruck aber auch als *Funktion* bezeichnen.

The single most important change in Java 8 enables faster, clearer coding and opens the door to functional programming.

Wir beschreiben anschließend 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 an einer anderen Stelle des Programms zu bestimmten Gelegenheiten aufgerufen werden soll, 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 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 **run()** besitzt, erzeugt ein Objekt 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, T>** erfüllt, also eine entsprechende Methode **compare()** besitzt, und übergibt ein Objekt dieser Klasse z.B. an eine Methode zum Sortieren eines Arrays mit Elementen des fraglichen Typs.

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);
}
```

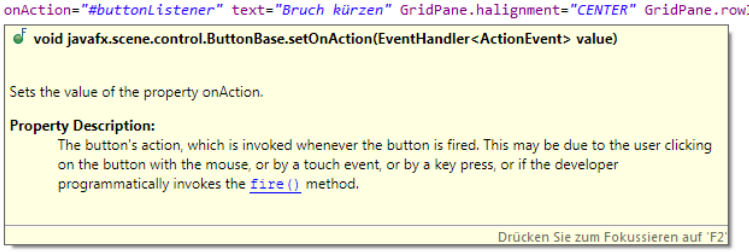
Seit Java 8 spricht man hier von einem *funktionalen Interface*, weil es bei der funktionalen Programmierung eine zentrale Rolle spielt. Neben der einen abstrakten Methode können beliebig viele Instanzmethoden mit **default**-Implementierung sowie statische Methoden vorhanden sein (vgl. Abschnitt 9.2.3).

In Java 8 wurde die Standardbibliothek um das Paket **java.util.function** erweitert, das über 40 funktionale Schnittstellen enthält.

---

[1] Wird die JavaFX-Bedienoberfläche deklarativ per FXML-Datei gestaltet, passiert einiges im Verborgenen. Wie die Eclipse-Erläuterung zum Attribut **onAction** im Element **Button** aus der FXML-Datei zu unserem Beispielprogramm in Abschnitt 4.8 zeigt, ändert sich nichts an der Grundlogik der Ereignisbehandlung:

```
<Button fx:id="btnKuerzen" mnemonicParsing="false" onAction="#buttonListener" text="Bruch kürzen" GridPane.halignment="CENTER" GridPane.rowIndex="1" />
```



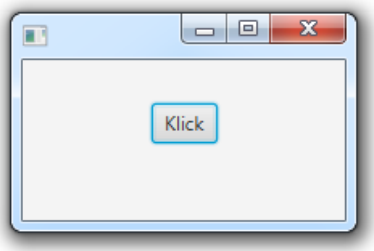
Um dem Compiler für ein als funktional konzipiertes Interface die Kontrolle der eben beschriebenen Regel (genau eine abstrakte Methode) zu ermöglichen, kann man der Definition die Marker-Annotation **@FunctionalInterface** voranstellen, was in der Java SE - Standardbibliothek regelmäßig geschieht, z.B. beim generischen Interface **Predicate<T>** im Paket **java.util.function**, das eine abstrakte Methode namens **test(T t)** mit einem Parameter vom Typ **T** und einer Rückgabe vom Typ **boolean** verlangt:

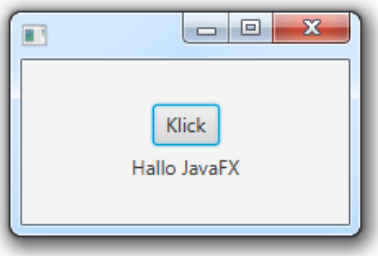
```
@FunctionalInterface
public interface Predicate<T> {

    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);
    }

    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.

12.1.1.2 Anonyme Klassen

Das in allen Beispielen von Abschnitt 12.1.1.1 vorhandene Muster zur Übergabe von Funktionalität an andere Programmbestandteile wird in Java bis zur Version 7 häufig mit Hilfe von sogenannten *anonymen Klassen* realisiert. Besonders oft kommt diese Technik bei der Ereignisbehandlung in GUI-Programmen zum Einsatz. Das gilt auch für JavaFX, was beim Verzicht auf die deklarativen Oberflächengestaltung per FXML-Datei klar zu Tage tritt. Im folgenden Programm wird per Mausklick ein Text zur Anzeige gebracht:





Im folgenden Quelltext wird die Bedienoberfläche komplett durch Anweisungen erzeugt, was in JavaFX nach wie möglich und bei anderen GUI-Techniken (z.B. Swing) alternativlos ist:

```
public class Main extends Application {
 @Override
 public void start(Stage primaryStage) {
  try {
   Label label = new Label();
   Button button = new Button("Klick");

   button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
     label.setText("Hallo JavaFX");
    }
   });

   VBox root = new VBox(5);
   root.setAlignment(Pos.CENTER);
   root.getChildren().addAll(button, label);
   Scene scene = new Scene(root, 200, 100);
   primaryStage.setScene(scene);
   primaryStage.show();
  } catch(Exception e) {
   e.printStackTrace();
  }
 }

 public static void main(String[] args) {
  launch(args);
 }
}
```

Mit dem Aufbau dieses Quellcodes werden wir uns im JavaFX-Kapitel 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 wird. Einen Namen erhält die nur lokal benötigte Klasse nicht, und es resultiert mit der so genannten **anonymen Klasse** eine Variante der lokalen Klassen (vgl. Abschnitt 4.9.2).

Einige 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 Beispiel 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, Mitgliedklasse oder reguläre Klasse).

- Weil der Klassenname fehlt, sind keine Konstruktoren möglich. Über die Instanzinitialisierer (vgl. Abschnitt 4.4.4) ist jedoch eine Ersatzlösung verfügbar (siehe Beispiel in Abschnitt 13.4.2.2.2).
- 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* vorausgesetzt. Diese besteht, wenn nach der Initialisierung keine Wertveränderung stattfindet. Der Modifikator **final** ist nicht mehr erforderlich. Seit Java 8 kann eine anonyme Klasse auch auf effektiv finale *Parameter* der umgebenden Methode zugreifen. Weil der Zugriff auf finale Variablen bzw. Parameter beschränkt ist, sind natürlich nur *lesende* Zugriffe erlaubt.
- Außerdem kann eine anonyme Klasse auf die statischen Variablen und Methoden der Klasse zugreifen, zu der die umgebende Methode gehört. 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 einer **private**-Deklaration möglich.
- Felder und lokale Variablen der anonymen Klasse überdecken gleichnamige Variablen der umgebenden Klasse. 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 `statEnc`:

  ```
  Aussen.statEnc
  ```

  Überdeckte Instanzvariablen der umgebenden Klasse 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 Klassenvariablen `instEnc`:

  ```
  Aussen.this.instEnc
  ```
- 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.

12.1.1.3 Compiler-Magie statt Zeremonie

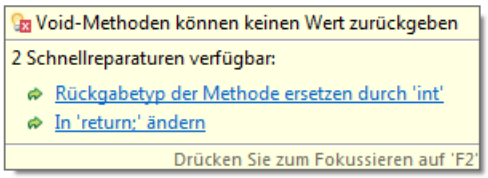
Die seit Java 8 mögliche Realisierung einer Ereignisbehandlungsmethode durch einen Lambda-Ausdruck enthält im Vergleich zur traditionellen Lösung durch eine anonyme Klasse deutlich weniger „Zeremonie“, z.B.:

```
button.setOnAction(event -> label.setText("Hallo JavaFX"));
```

Hinter den Kulissen bleibt alles beim Alten, wobei der Compiler viele Routinearbeiten übernimmt:

- Er kennt den Parametertyp **EventHandler<ActionEvent>** von **setOnAction()** und akzeptiert einen Lambda-Ausdruck, der die erforderliche Methode **handle()** realisiert, so dass sich eine passende anonyme Klasse erstellen lässt.
- Die im Lambda-Ausdruck vor dem Pfeil (->) angegebenen Parameter müssen vom passenden Typ sein. Man kann jedoch auf eine Typangabe verzichten, wobei der Compiler die Typen 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;});
```



- Weil die Typen **EventHandler** und **ActionEvent** im Quellcode nicht mehr explizit 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.

In 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

```
(String 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","Kurt","Frank");
Stream<String> str = als.stream();
str = str.filter((String s) -> s.length() >= 5);
```

Um praxisrelevante Anwendungsfälle für Lambda-Ausdrücke zu erhalten, verwenden wir im aktuellen Abschnitt bereits (in möglichst einfacher Form) die in Abschnitt 12.2 vorzustellenden **Stream**-Typen. Im Beispiel wird der **Stream<String>** - Methode **filter()** ein Lambda-Ausdruck übergeben, der auf jedes Element im Strom angewandt werden soll. 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**. Man kann den Lambda-Ausdruck einer Referenzvariablen von diesem Typ zuweisen

```
Predicate<String> ps = (String 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 sehr 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((String s) -> s.length() >= 5);
```

eine anonyme Klasse synthetisiert:

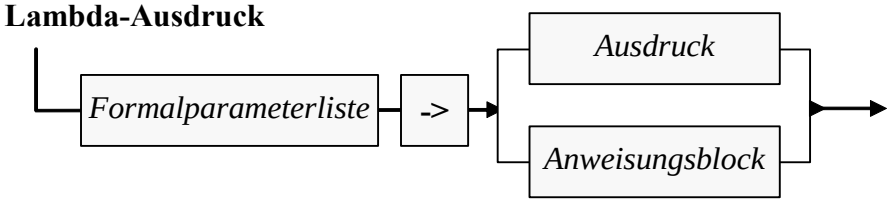
```
str = str.filter(new Predicate<String>() {
  public boolean test(String s) {
   return s.length() >= 5;
  }
 });
```

Aus den im Vergleich zur anonymen Klasse eingeschränkten syntaktischen Möglichkeiten eines Lambda-Ausdrucks ergibt sich, dass hier keine eigenen Instanzvariablen möglich sind. Wenn z.B. ein **ActionEvent**-Handler zu einem GUI-Bedienelement über die einzelnen Aufrufe hinweg Daten speichern muss, ist ein „vollwertiges" Objekt erforderlich.

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.[1]

12.1.1.4 Definition von Lambda-Ausdrücken

Um mit der Syntax des Lambda-Ausdrucks vertraut zu werden, verschaffen wir uns zunächst einen Überblick mit Hilfe eines Syntaxdiagramms:



Vorweg halten wir fest, dass kein Rückgabetyp anzugeben ist. Dieser wird generell vom Compiler aus der vom Lambda-Ausdruck implementierten Schnittstellenmethode ermittelt.

12.1.1.4.1 Formalparameterliste

Grundsätzlich gilt für die Formalparameterliste eines Lambda-Ausdrucks wie für die Formalparameterliste einer Methode (vgl. Abschnitt 4.3.1.3):

- 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.
- Ein Parameter kann als **final** deklariert werden.

---

[1] Nähere Erläuterungen finden sich auf der folgenden, am 31.01.2016 abgerufenen Web-Seite der Firma IBM: https://www.ibm.com/developerworks/library/j-java8lambdas/

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** 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.[1]

- Bei einem *einzelnen*, implizit typisierten Parameter kann man die runden Klammern weglassen. Das zu Beginn des aktuellen Kapitels vorgestellte Beispiel

  ```
  (String s) -> s.length() >= 5
  ```

  kann also (noch) einfacher notiert werden:

  ```
  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.

12.1.1.4.2 Rumpf

Der Lambda-Rumpf kann aus einem Anweisungsblock bestehen

[1] Ausführliche Erläuterungen zu **reduce()** und anderen Stromoperationen folgen in Abschnitt 12.2.5.

```
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 Checked Exception (vgl. Abschnitt 11.5) 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

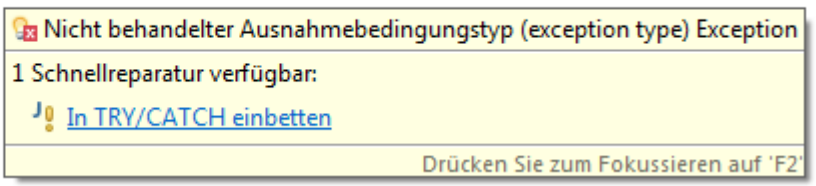
```
interface ExceptionThrower<T> {
 boolean thrower(T t) throws Exception;
}

class Prog {
 public static void main(String[] args) throws Exception {
  java.util.Random zg = new java.util.Random();

  java.util.function.Predicate<String> psr = s ->
   {if (zg.nextInt(5) <2) throw new RuntimeException(); return s.length() >= 5;};

  ExceptionThrower<String> et = s ->
   {if (zg.nextInt(5) <2) 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 Unchecked Exception geworfen, was generell erlaubt ist.
- Im Lambda-Ausdruck vom Typ `ExceptionThrower<String>` wird eine Checked Exception geworfen, die im implementierten Interface angemeldet wird (siehe Definition der Methode `thrower()`).

Einem das Interface **Predicate<String>** implementierenden Lambda-Ausdruck ist es hingegen *nicht* erlaubt, eine Unchecked Exception zu werfen:

```
java.util.function.Predicate<String> pse = s ->
    {if (zg.nextInt(5) <2) throw new Exception(); return s.length() >= 5;};
```



12.1.1.4.3 Zugriff auf Variablen in der Umgebung

Ein Lambda-Ausdruck hat Zugriff auf die Variablen in seiner Umgebung:

- Auf effektiv finale lokale Variablen einer umgebenden Methode
  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 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).
- Auf Instanzvariablen des umgebenden Objekts
  Ein umgebendes Objekt existiert, wenn der Lambda-Ausdruck einer Instanzvariablen zugewiesen oder in einer Instanzmethode definiert wird.
- Auf statische Variablen der umgebenden Klasse

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 des Programms ausgeführt werden kann.

Im folgenden Beispielprogramm werden die in einem Lambda-Ausdruck erlaubten Zugriffe auf Umgebungsvariablen demonstriert:

Quellcode	Ausgabe
`import java.util.function.Supplier;` `class LambdaScoping {` `  static int statEnc = 0;` `  int instEnc = 10;` `  Supplier<String> sups;` `  LambdaScoping() {` `  int locEnc = 13;` `  int[] locArrEnc = {100};` `  sups = () -> {` `  // int locEnc = 14; Verboten` `  return String.valueOf(++statEnc)+" "+` `  String.valueOf(++instEnc)+" "+` `  String.valueOf(locEnc)+" "+` `  String.valueOf(++locArrEnc[0]);` `  };` `  }` `  void clientWork() {` `  System.out.println(sups.get());` `  }` `  public static void main(String[] args) {` `  LambdaScoping ls = new LambdaScoping();` `  ls.clientWork(); ls.clientWork(); ls.clientWork();` `  }` `}`	`1 11 13 101` `2 12 13 102` `3 13 13 103`

Die Zusammenfassung eines Lambda-Ausdrucks mit den „eingefangenen“ Variablen aus der Umgebung wird als *Abschluss* (engl. *closure*) bezeichnet.

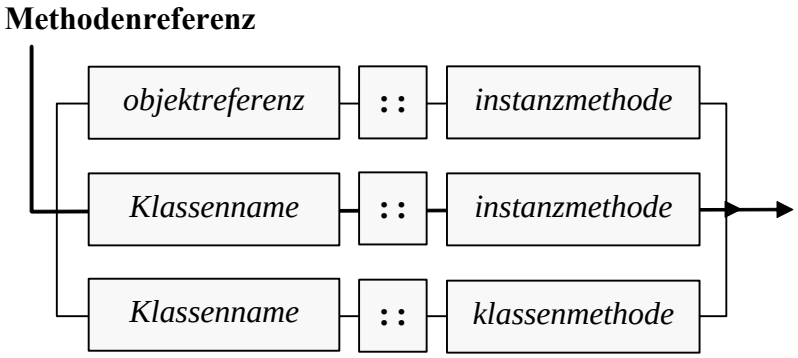
Beim Zugriff auf Umgebungsvariablen gelten für anonyme Klassen und Lambda-Ausdruck weitgehend identische Regeln mit einer Ausnahme: Eine anonyme Klasse begründet einen eigenen Gültigkeitsbereich, und in ihren Methoden dürfen lokale Variablen mit dem einem Namen angelegt werden, den lokale Variablen einer umgebenden Methode verwenden. Dabei werden die Umgebungsvariablen überdeckt. Ein Lambda-Ausdruck gehört hingegen wie ein gewöhnlicher eingeschachtelter Block zum Gültigkeitsbereich einer umgebenden Methode, so dass die dortigen lokalen Variablennamen im Lambda-Ausdruck *nicht* verwendet werden dürfen. In der englischsprachigen Literatur wird dafür die Bezeichnung *lexical scoping* verwendet.

12.1.2 Methoden- und Konstruktor-Referenzen

Man glaubt es kaum, doch in vielen Situationen gibt es zu den bisher beschriebenen Lambda-Varianten noch einfachere Alternativen. Dadurch gibt es allerdings für Einsteiger noch mehr Syntaxvarianten kennen zu lernen.

12.1.2.1 Methodenreferenzen

Gelegentlich existiert zu einem geplanten Lambda-Ausdruck eine Methode, bei der die Formalparameterliste und der Rückgabetyp exakt passen. Dann kann der Lambda-Ausdruck durch eine so genannte *Methodenreferenz* ersetzt werden:



Bei einer Instanzmethode wird der Auftragsnehmer entweder durch eine konkrete Objektreferenz (z.B. **System.out**) oder eine Klasse angegeben. Bei einer statischen Methode ist der Klassenname anzugeben. Hinter den Auftragsnehmer ist der **::** - Operator zu setzen. Schließlich folgt der Methodenname *ohne* Parameterliste.

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.

Im folgenden Beispiel wird für die **String**-Objekte in einer Liste die mittlere Länge berechnet, wobei ein Strom-Objekt 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((String s) -> s.length())
  .average();
```

Von **mapToInt()** wird die Schnittstellenmethode **applyAsInt(String** *value***)** mit jedem Listenelement als Aktualparameter aufgerufen. Ein Lambda-Ausdruck verwendet dieselbe Formalparameterliste wie die implementierte Schnittstellenmethode. Im Beispiel wird der erste (und einzige) Parameter (Typ **String**) explizit auf den Ansprechpartner für den Aufruf der Instanzmethode **length()** abgebildet. Genau diese Abbildung geschieht implizit bei einer Instanzmethodenreferenz gemäß Fall 2 aus dem obigen Syntaxdiagramm. 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 die folgende 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. Somit ist z.B. der Lambda-Ausdruck

```
s -> System.out.println(s);
```

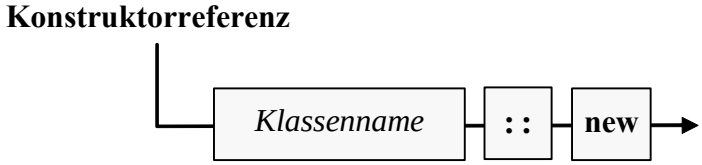
äquivalent zur Methodenreferenz

```
System.out::println
```

Weitere Details zu Methodenreferenzen sind z.B. bei Horstmann (2014b) zu finden.

12.1.2.2 Konstruktorreferenzen

Wenn ein Lambda-Ausdruck nichts anderes tut, als ein Objekt per Konstruktoraufruf zu instanziieren, dann kann der Lambda-Ausdruck durch eine so genannte *Konstruktorreferenz* ersetzt werden:



Eine Konstruktorreferenz unterscheidet sich von einer Methodenreferenz (siehe Abschnitt 12.1.2.1) 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 die in einer Liste befindlichen **String**-Objekte in Objekte der Klasse **BigDecimal** gewandelt werden. Wir erstellen aus der Liste ein Objekt vom Typ **Stream<String>** und verwenden seine Methode **map()**, um daraus einen **Stream<BigDecimal>** zu erzeugen. Der Methode **map()** übergeben wir ein Objekt vom Interface-Typ **Function<String, BigDecimal>**, der die abstrakte Methode **apply()** vorschreibt:

BigDecimal apply(String s);

Der **BigDecimal** - Konstruktor

public BigDecimal(String *val***)**

erfüllt den Job und kann daher per Konstruktorreferenz an **map()** übergeben werden:

Quellcode	Ausgabe
`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);` `  }` `}`	`3.14` `9.99` `47.11`

Die Konstruktorreferenz

```
BigDecimal::new
```

ist äquivalent zum Lambda-Ausdruck

```
s -> new BigDecimal(s)
```

12.2 Ströme

Neben den Lambda-Ausdrücken stellen die **Stream<T>** - Typen wohl die bedeutsamste Neuerung in Java 8 dar. Stromobjekte erlauben Abfrage- und/oder Verarbeitungsoperationen für eine Datenquelle (z.B. eine Kollektion).

Dabei ist eine *deklarative Programmierung* möglich, und das explizite Iterieren bei ständiger Aktualisierung von Variablen mit Zwischenergebnissen wird in die Tiefen der Standardbibliothek verlagert. Man kann z.B. analog zu einer Datenbankabfrage 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.

Von herausragender Bedeutung ist die Option, von der seriellen Bearbeitung mit Leichtigkeit auf die *parallele, mehrere Prozessorkerne nutzende Bearbeitung* 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.[1]

12.2.1 Beispiel

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 5 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 (in Abschnitt 10.3 erwähnte) **Collection<T>** - Methode **stream()** ein Objekt vom Typ **Stream<String>** erstellt.

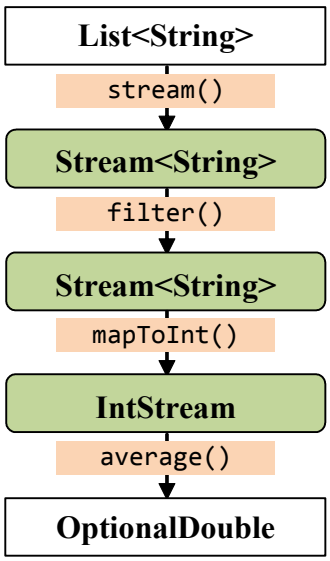
Daraus entsteht durch Anwendung der Operation **filter()** ein neues Stromobjekt vom selben Typ, das nur noch die **String**-Objekte mit mindestens 5 Zeichen enthält. Zur Bewertung der Zeichenfolgen im Ausgangsstrom dient ein Objekt einer anonymen, das Interface **Predicate<String>** implementierenden Klasse, die per Lambda-Ausdruck definiert wird und die Instanzmethode **test**(**String** *s*) mit **boolean**-Rückgabe besitzt.

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 Objekt einer anonymen, das Interface **ToIntFunction<String>** implementierenden Klasse zuständig, die per Lambda-Ausdruck definiert wird und die Instanzmethode **applyAsInt**(**String** *s*) mit **int**-Rückgabe 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:

---

[1] Im Hintergrund kommt das *Fork-Join* - Framework zum Einsatz, das im Abschnitt 16.6.1 behandelt wird.



Insgesamt wird das sogenannte **Filter-Map-Reduce** - Muster realisiert.

Ein Stromobjekt ist *kein* Container, sondern eine Station in einer Verarbeitungskette für die Daten aus einer Quelle.[1] Als Datenquellen kommen z.B. in Frage:

- eine Kollektion
- ein Array
- eine Methode zum Generieren von (potentiell unendlich vielen) Werten

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, so dass 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:

```
Exception in thread "main" java.lang.IllegalStateException: stream has already been
operated upon or closed
 at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
 at java.util.stream.ReferencePipeline.reduce(ReferencePipeline.java:479)
 at Prog.main(Prog.java:21)
```

[1] Die in Java 8 eingeführten **Stream**-Typen dürfen nicht mit den viel älteren I/O - Stream - Klassen verwechselt (z.B. **InputStream** and **OutputStream**, siehe Kapitel 15).

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 der Zeitaufwand und die Fehlergefahr groß sind,
- ist die oft wünschenswerte Parallelisierung nur schwer zu realisieren, so dass 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).

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

```
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);
 }
}
```

```
37
37
```

Im modernen Quellcode ist vom Initialisieren und 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 dazu lediglich einen weiteren Methodenaufruf benötigt, der aus dem seriell arbeitenden Strom einen parallel arbeitenden erstellt:

```
int summe = Arrays.stream(daten)
                  .parallel()
                  .sum();
```

Im konkreten Beispiel (mit der Summe 37) wird 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** etliche 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 stehen bequeme Methoden bereit, die für einen Strom statistische Kennwerte wie die Summe oder den Mittelwert durch einen einfachen Aufruf liefern (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 verschiedenen Threads ausgeführt werden können, um später die Ergebnisse zusammen zu führen.

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 größtem Interesse, um leistungsfähige Anwendungen entwickeln zu können. Die für Multithreading typische Komplexität mit dem Risiko von Programmierfehlern (siehe Kapitel 16) wird bei den Stromoperationen vermieden, weil die kritischen Aufgaben von der Standardbibliothek übernommen werden.

Für den Programmierer verbleibt die Ermessensentscheidung für oder gegen den Einsatz der Parallelisierung. Mehrere Threads zu starten, zu koordinieren und deren Ergebnisse zusammen zu führen, verursacht trotz ausgefeilter Bibliotheksroutinen unvermeidlichen Aufwand, der sich bei kleinen Problemen nicht lohnt.

Bei parallelen Stromoperationen kommt im Hintergrund das *Fork-Join* - Framework zum Einsatz, das Sie im Kapitel 16 über Multithreading kennen lernen werden.

12.2.4 Erstellung von Strom-Objekten

In diesem Abschnitt werden verschiedene Optionen zur Erstellung von Strom-Objekten vorgestellt.

12.2.4.1 Strom-Objekt aus einer Kollektion erstellen

Im Interface **Collection<E>** (siehe Abschnitt 10.3) sind zum Erstellen eines (parallelen) Stroms die Instanzmethoden **stream()** und **parallelStream()** vorhanden (mit **default**-Implementation), z.B.:

```
List<String> als = Arrays.asList("Rudolf","Emma","Otto","Agnes","Kurt");
Stream<String> sos = als.stream();
Stream<String> psos = als.parallelStream();
```

In diesem Code-Segment wird die statische und generische Methode **asList()** der Klasse **Arrays** dazu verwendet, ein **List<String>** - Objekt zu erstellen:

public static <T> List<T> asList(T... a)

12.2.4.2 Strom-Objekt aus einem Array erstellen

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 kleinen Arrays ist die in allen **Stream**-Interfaces vorhandene statische Methode **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 gewinnen, z.B.:

```
IntStream paris = Arrays.stream(new int[]{1,4,14,39})
                        .parallel();
```

12.2.4.3 Strom-Objekte aus gleichabständigen ganzen Zahlen

In den Schnittstellen **IntStream** und **LongStream** ermöglichen die statischen Methoden **range()** und **rangeClosed()** das bequeme Erstellen von Strömen bestehend aus einer Sequenz mit gleichabständigen 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	Ausgabe
`import java.util.stream.IntStream;` `class Range {` `  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);` `  }` `}`	`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).

12.2.4.4 Unendliche serielle Ströme

Die statischen Methoden **iterate()** und **generate()** in den Strom-Schnittstellen können einen endlosen seriellen Strom produzieren.

An **iterate()** 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 is = 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.

Eine einfache Anwendung von **generate()** besteht darin, einen konstanten Strom zu produzieren, z.B.:

```
IntStream is = IntStream.generate(() -> 1).limit(100);
```

12.2.4.5 Sonstige Erstellungsmethoden

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 einer Textdatei erleichtert. Im folgenden Code-Segment werden mit der Stromoperation **count()** die Zeilen in der Datei gezählt:

```
Stream<String> sol = Files.lines(Paths.get("U:/Eigene Dateien/Java/test.txt"));
System.out.println("Anzahl der Zeilen: " + sol.count());
```

12.2.5 Stromoperationen

Java 8 bietet 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 Datenbankartige 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 2 Kategorien einteilen:

- **Intermediäre Operationen**
  Intermediäre Operationen liefern ein neues Stromobjekt als Rückgabe, so dass 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 12.2.5.3.2).
  - **sorted()**
    Der neue Strom entsteht aus dem alten durch das Sortieren der Elemente (siehe Abschnitt 12.2.5.3.3).
  - **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 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	Ausgabe
`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> nge5 = 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(nge5);` `  }` `}`	`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 die Operationen **filter()** und **map()** gemeinsam (bei *einer* Datenpassage) ausgeführt worden sind,
- dass für die Fälle mit negativem Filterergebnis keine Abbildung vorgenommen wurde,
- dass nach dem Vorliegen von zwei positiven Fällen keine weiteren (überflüssigen) Operationen mehr ausgeführt wurden.

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 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 Objekt, 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 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	Ausgabe
`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");` `  long n4 = als.stream()` `  .filter(s -> s.length() == 4)` `  .count();` `  System.out.println(n4);` `  }` `}`	`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 Elementweise Abbildung

Mit der Methode **map()** aus dem Interface **Stream<T>** gewinnt man einen neuen Strom mit Elementen, die aus den Gegenstücken im alten Strom durch eine elementweise Abbildung entstehen:

<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 im Wesentlichen 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. Für die Ausgabe der **String**-Längen sorgt die terminale Operation **forEach()** (siehe Abschnitt 12.2.5.4.1).

Quellcode	Ausgabe
`import java.util.Arrays;` `class Mapping {` `  public static void main(String[] args) {` `  Arrays.asList("Rudolf","Emma","Otto","Agnes","Kurt")` `  .stream()` `  .map(s -> s.length())` `  .forEach(i -> System.out.print(i+" "));` `  }` `}`	`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
`import java.util.Arrays;` `class Mapping {` `  public static void main(String[] args) {` `  int n = Arrays.asList("Rudolf","Emma","Otto","Agnes","Kurt")` `  .stream()` `  .mapToInt(s -> s.length())` `  .sum();` `  System.out.println("Summe: "+n);` `  }` `}`	`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

12.2.5.3.3 Sortieren

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
`import java.util.Arrays;` `class Sorted {` `  public static void main(String[] args) {` `  Arrays.asList("Rudolf","Emma","Otto","Agnes","Kurt")` `  .stream()` `  .sorted()` `  .forEach(i -> System.out.println(i+" "));` `  }` `}`	`Agnes` `Emma` `Kurt` `Otto` `Rudolf`

Über die **sorted()** - Überladung mit einem Parameter vom Typ **Comparator<? super T>** kann man ein Sortierkriterium definieren:

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
`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(i -> System.out.println(i+" "));` `  }` `}`	`Rudolf` `Otto` `Kurt` `Emma` `Agnes`

Die Schnittstellen **IntStream**, **LongStream** und **DoubleStream** für primitive Elementtypen enthalten eine analoge arbeitende **sorted()** - Methode.

12.2.5.3.4 Zwischenberichte

Veranlasst man eine Ausgabe der Stromelemente über die in Abschnitt 12.2.5.3.2 beschriebene terminale Stromoperation **forEach()**, ist die gesamte Pipeline anschließend verbraucht und inoperabel, was die Fehlersuche umständlich macht. 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 Stromverarbeitungslogik werden die ungeraden Zahlen nur einmal, die geraden Zahlen hingegen zweimal nacheinander protokolliert.

Die Strommethode **peek()** ist wie **forEach()** (vgl. Abschnitt 12.2.5.3.2) ein 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

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, wobei eine Methodenreferenz als **forEach()** - Parameter dient:

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`

Laut API-Dokumentation produzieren die terminalen Operationen entweder einen Wert als spezielle Zusammenfassung des Stroms (z.B. durch seine Summe) oder Nebeneffekte, wobei **forEach()** ein Nebeneffekt-Produzent ist.[1]

Mit der zur Fehlersuche konzipierten Stromoperation **peek()** lassen sich ebenfalls elementweise Nebeneffekte produzieren, wobei jedoch die Pipeline *nicht* terminiert wird (siehe Abschnitt 12.2.5.3.4).

12.2.5.4.2 Reduktion eines Stroms auf einen Wert durch eine assoziative Funktion

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:

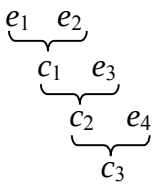
$$f(f(a, b), c) = f(a, f(b, c))$$

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.

---

[1] Siehe z.B. http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

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>**. Es ist also ein Objekt zu übergeben, das die folgende Methode beherrscht:

public T apply(T *first***, T** *second***)**

Die folgende **reduce()** - Überladung

public Optional<T> reduce(BinaryOperator<T> ***accumulator*****)**

liefert 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()**.

Im folgenden Beispiel wird ein Strom vom Typ **IntStream** mit Elementen vom primitiven Typ **int** erzeugt durch die statische **IntStream**-Methode **range()**, die einen inklusiven Startwert und einen exklusiven Endwert vom Typ **int** erwartet und die zugehörige Sequenz von ganzen Zahlen produziert:

Quellcode	Ausgabe
`import java.util.stream.IntStream;` `import java.util.OptionalInt;` `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("IntStream: " + (sq.isPresent() ? sq2.getAsInt():"Fehler"));` `  }` `}`	`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> *accumulator*)

Diesmal erhalten wir eine Rückgabe vom Typ **T**, die bei einem leeren Strom mit dem ersten **reduce()** - Parameter identisch ist, so dass auf jeden Fall eine Rückgabe vom Typ **T** resultiert, und der Aufwand mit einer **Optional<T>** - Rückgabe entfällt.

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):

```
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()** mit einem **int**-Parameter und einer Rückgabe vom Typ **BigDecimal**, die von einem Konstruktor der Klasse **BigDecimal** produziert wird. Per Konstruktorreferenz (vgl. Abschnitt 12.1.2.2) lässt sich der Aktualparameterausdruck drastisch vereinfachen, wobei dank Typinferenz auch auf die explizite Typkonkretisierung im Methodennamen verzichtet werden kann:

```
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***)**

---

[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).

Auf die recht lange Beschreibung folgt ein angenehm kurzes Programm:

Quellcode	Ausgabe
`import java.math.BigDecimal;` `class Reduce {` `  public static void main(String[] args) {` `  BigDecimal fak = IntStream` `  .rangeClosed(1,500)` `  .mapToObj(BigDecimal::new)` `  .reduce(BigDecimal.ONE, (c,x) -> c.multiply(x));` `  System.out.printf("%e", fak);` `  }` `}`	`1,220137e+1134`

Im Beispiel kann gut demonstriert werden, wie leicht die serielle Strombearbeitung auf eine parallele, mehrere Prozessorkerne nutzende, umgestellt werden kann. Dazu ist lediglich mit der **IntStream**-Methode **parallel()** aus dem seriellen Strom ein paralleler zu erstellen. Wir erweitern das letzte Beispielprogramm außerdem um eine Zeitmessung und erhalten:

Quellcode	Ausgabe
`import java.math.BigDecimal;` `class Reduce {` `  public static void main(String[] args) {` `  long start = System.currentTimeMillis();` `  BigDecimal fak = IntStream.rangeClosed(1,50_000)` `  .parallel()` `  .mapToObj(BigDecimal::new)` `  .reduce(BigDecimal.ONE, (c,x) -> c.multiply(x));` `  System.out.printf("\nFakultät: %e\nZeit in Millisekunden: %d",` `  fak, System.currentTimeMillis()-start);` `  }` `}`	`Fakultät: 3,347321e+213236` `Zeit in Millisekunden: 94`

Die Berechnung der Fakultät von 500 dauert nach der Parallelisierung *länger* (10 statt 5 Millisekunden). Offenbar wiegt bei dieser Problemgröße der durch Multithreading bedingte organisatorische Zusatzaufwand den Gewinn durch Verwendung mehrerer Kerne mehr als auf. Zur Berechnung der Fakultät von 50.000 benötigt der parallele Strom mit 210 Millisekunden allerdings deutlich weniger Zeit als der serielle, der nach 1247 Millisekunden zum Ergebnis kommt (gemessen mit dem Prozessor Intel Core i3 550 mit 3,2 GHz, 2 Kerne plus Hyperthreading). Wir stellen fest:

- Multithreading ist bei der funktionalen Strombearbeitung in Java 8 sehr leicht zu realisieren.
- Weil die Erstellung und Koordination von mehreren Threads Aufwand verursacht, entscheidet u.a. die Problemgröße darüber, ob ein Nutzen zu erzielen ist.

12.2.5.4.3 Spezielle Reduktionsoperationen

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(1,4,14,39);` `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

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** (also ein Container) 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 (einen Container) 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	Ausgabe

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

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);
 }
}
```

```
Anette
Anton
Ben
Berta
Charly
Clara
```

Als **collect()** - Parameter fungieren Konstruktor- bzw. Methodenreferenzen. 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()**:

```
collect(Collectors.toList())
```

Schlussendlich erhält man bei deutlich reduziertem Aufwand als Rückgabe von **collect()** einen Container vom Typ ArrayList<String>.

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 Typ **Map<Character, List<String>>**, das eine Gruppierung der Vornamen nach dem Anfangsbuchstaben leistet:

Quellcode	Ausgabe
`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));` `  }` `}`	`A [Anette, Anton]` `B [Ben, Berta]` `C [Charly, Clara]`

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
`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);` `  }` `}`	`Charly, Anton, Berta`

12.2.5.4.5 Stromelemente in einem Array sammeln

Im folgenden Beispiel wird ein **IntStream**-Objekt von Dubletten befreit und anschließend mit der **IntStream**-Methode **toArray()** in einen **int**-Array gewandelt:

Quellcode	Ausgabe
`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)` `  .distinct()` `  .toArray();` `  for(int i : istar)` `  System.out.println(i);` `  }` `}`	1 2 3 4 5

12.2.5.4.6 Strombezogene Bedingungen

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, so dass 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	Ausgabe
`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);` `  }` `}`	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

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

12.1.1.4.1), 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 Namen 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
`import java.util.Arrays;` `import java.util.List;` `import java.util.Optional;` `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: \t\t"+einTreffer);` `   Optional<String> ersterTreffer=crFStream().findFirst();` `   System.out.println("Erster Treffer: \t"+ersterTreffer);` `  }` `}`	`Ein Treffer:  Optional[Emma]` `Erster Treffer: Optional[Emma]`

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 8 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 Namen 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 8 *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 mlgt4 = als.stream()
                         .filter(s -> s.length() >= 5)
                         .mapToInt(s -> s.length())
                         .average();
System.out.println("Mittlere Länge der Namen mit > 4 Zeichen: " +
  (mlgt4.isPresent() ? mlgt4.getAsDouble() : "nicht vorhanden"));
```

Die Details der Ausführung bleiben den beteiligten Bibliotheksobjekten überlassen:

- Es werden keine Hilfsvariablen (wie `summe` und `n` in der traditionellen Lösung) deklariert, initialisiert und aktualisiert.
- Die Iterationen laufen intern (gekapselt in Bibliotheksobjekten) ab.

Mit dem Aufwand entfallen viele Fehlermöglichkeiten.

Im Beispiel werden die beiden zu Beginn von Kapitel 12 erwähnten Kerntechniken der funktionalen Programmierung mit Java 8 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 syntaktisch einfacher und eleganter möglich, die benötigte Funktionalität 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 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 in 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 mlgt4 =
 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()**, die hier als Funktion höherer Ordnung aufzufassen ist, 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.

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 Parameter 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 abstrakte Methode implementiert:

public boolean test(T *t***)**

Der obige Lambda-Ausdruck passt zu dieser Methode, was der Compiler per Typinferenz erkennt:

- Aus der Parameterliste vor dem Pfeil ergibt sich, dass *ein* Parameter vom Elementtyp des Stroms vorhanden ist.
- Es wird ein Rückgabewert vom Typ **boolean** geliefert.

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.Arrays;
import java.util.List;
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 zu Kapitel 12

1) Erstellen Sie eine Anweisung zur Berechnung der Summe aus den ersten 100 geraden ganzen Zahlen (beginnende mit 2). Vermutlich werden Sie eine elementweise abbildende Stromoperation verwenden (vgl. Abschnitt 12.2.5.3.2). Definieren Sie bitte 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 des Eclipse-Compilers zeigt:

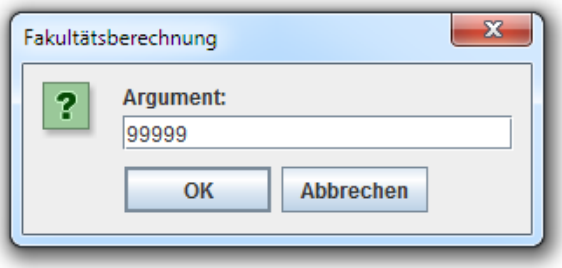
```
Object obj = s -> s.length() >= 5;
```

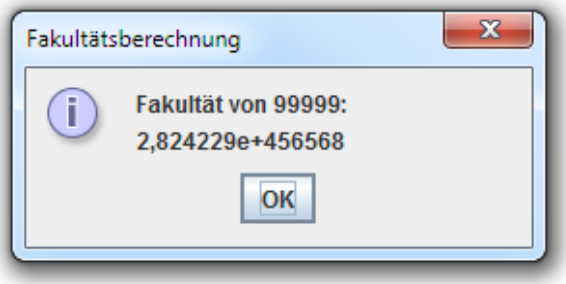
The target type of this expression must be a functional interface

Drücken Sie zum Fokussieren auf 'F2'

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 via **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 in Kapitel 11 (über die Ausnahmebehandlung) erlernten Techniken eine Möglichkeit, eine falsche Eingabe zu korrigieren.

13 GUI-Programmierung mit JavaFX

Eine Anwendung mit grafischer Bedienoberfläche (engl.: ***G**raphical **U**ser **I**nterface*) präsentiert dem Anwender ein oder mehrere Fenster, die neben Bereichen zur Bearbeitung von programmspezifischen Dokumenten (z.B. Texten oder Grafiken) in der Regel mehrere Bedienelemente zur Benutzerinteraktion besitzen (z.B. Menüs, Befehlsschalter, Kontrollkästchen, Textfelder, Auswahllisten). Die von einer Plattform zur Verfügung gestellten Bedienelemente bezeichnet man oft als *Komponenten*, *controls*, *Steuerelemente* oder *widgets*.[1] In Java bezeichnet man die Komponenten auch als *Beans*, wobei die Erfinder des Namens wohl an Kaffee gedacht haben.

Von der mehr oder weniger umfangreichen Ausstattung einer Plattform mit 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 in den Java-Entwicklungsumgebungen (z.B. der Scene Builder zu Java FX, den wir schon in Abschnitt 4.8 kennen gelernt haben).
- Weil die Bedienelemente intuitiv und in verschiedenen Programmen weitgehend konsistent zu bedienen sind, erleichtern sie dem Anwender den Umgang mit moderner Software.

13.1 Vorbemerkungen und 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. Für den Ablauf eines Programms mit grafischer Bedienoberfläche ist hingegen ein **ereignisorientiertes und benutzergesteuertes Paradigma** wesentlich, wobei das Laufzeitsystem als Vermittler 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, Touchscreen 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 so genannte *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*".

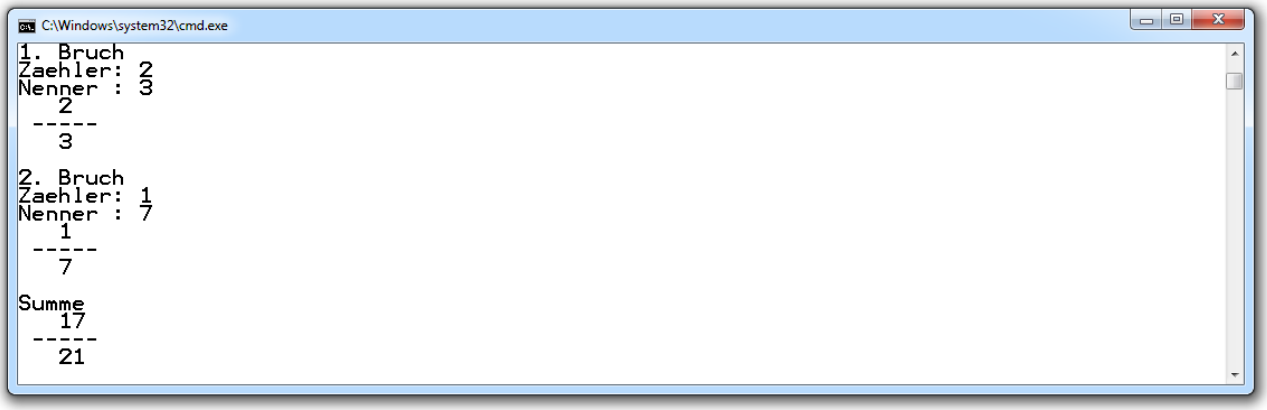
---

[1] Diese Wortkombination aus *window* und *gadgets* steht für ein *praktisches Fenstergerät*.

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 stellt eine Sammlung von Ereignisbehandlungsmethoden dar, 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)

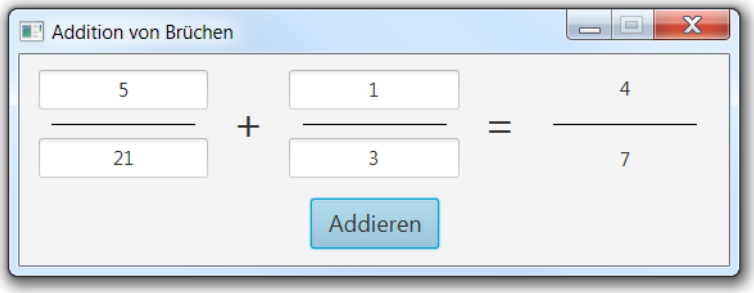


wird der gesamte Ablauf vom Programm diktiert:

- 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**).

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 Gestaltungs- bzw. Interaktionsmöglichkeiten aus.

Grundsätzlich ist das Erstellen einer GUI-Anwendung mit erheblichem Aufwand verbunden. Allerdings enthält das Java-API leistungsfähige Klassen (Komponenten, Beans) zur GUI-Programmierung, deren Verwendung durch Hilfsmittel der Entwicklungsumgebungen (z.B. Fensterdesigner) zusätzlich erleichtert wird.

13.1.2 GUI-Lösungen in Java

In der Java SE - Standardbibliothek sind leistungsfähige Klassen zur **plattformunabhängigen GUI-Programmierung** mit Hilfe vorgefertigter Steuerelemente enthalten. Die ursprüngliche, als ***A**bstract **W**indowing **T**oolkit* (**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-Anwendungen in Java war, wurde diese Rolle 2014 mit der Java-Version 8 von *JavaFX* übernommen.[1] 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 hatte, um die Entwicklung von graphischen 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 Hardware-Beschleunigung).
  - Es wurde eine XML-basierte GUI-Deklaration ermöglicht.
- **JavaFX 8**
  Als JavaFX im Jahr 2014 zum Standard für graphische 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.

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 schon nützlichen Kapitel noch weitere JavaFX-Techniken, z.B.:

- Erscheinungsbild einer Anwendung mit CSS (***C**ascading **S**tyle **S**heets*) individualisieren
- Multimedia-Inhalte darstellen (Audio, Video)

---

[1] Unter der Adresse http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6 wird von der Firma Oracle die folgende Frage eindeutig bejaht:

Is JavaFX replacing Swing as the new client UI library for Java SE?

Die meisten GUI-Programme kommen allerdings ohne die in diesem Kapitel noch fehlenden JavaFX-Techniken aus.

Den enorm wichtigen Multithreading-Bezügen von JavaFX gehen wir im aktuellen Kapitel aus dem Weg, holen deren Behandlung aber später nach (siehe Abschnitt 16.8).

An Literatur zu JavaFX herrscht kein Mangel. Eine kompakte Beschreibung bietet Inden (2015), eine sehr ausführliche ist bei Sharan (2015) zu finden.

Neben den GUI-Toolkits der 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

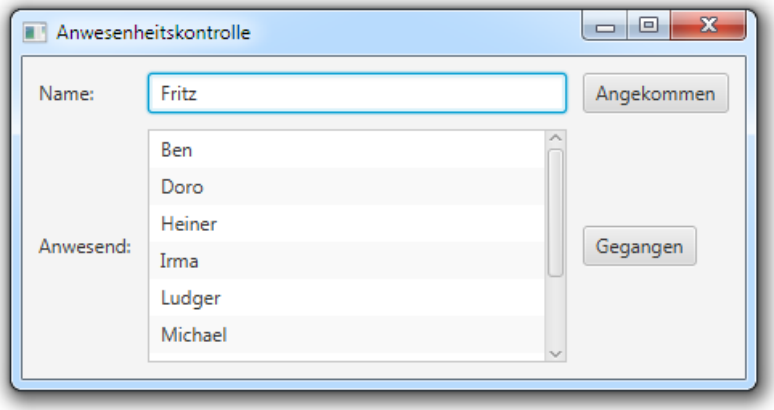
Während in Swing die Bedienoberfläche einer Anwendung grundsätzlich programmgesteuert erstellt wird, bietet JavaFX neben dieser traditionellen Vorgehensweise auch die Möglichkeit zur deklarativen GUI-Gestaltung über eine FXML-Datei. Die in JavaFX und auch in anderen modernen Programmiersprachen bzw. Software-Entwicklungsumgebungen (z.B. C#, Android) bevorzugte deklarative Technik bietet u.a. die folgenden Vorteile:

- Bessere Trennung von Design und Funktion
  Durch diese Trennung lassen sich beide Bestandteile besser pflegen. Außerdem wird eine arbeitsteilige Tätigkeit von Programmierern und GUI-Designern erleichtert.
- Bessere Übersichtlichkeit bei komplexen Bedienoberflächen

Während wir beim ersten JavaFX-Einsatz in Abschnitt 4.8 die deklarative Technik genutzt haben, verwenden wir im aktuellen Kapitel meist die programmatische Vorgehensweise, weil sie weitgehend auf Framework-Magie verzichtet und somit einen unverstellten Blick auf grundlegende Strukturen bietet.

13.2.1 Beispiel Anwesenheitsliste

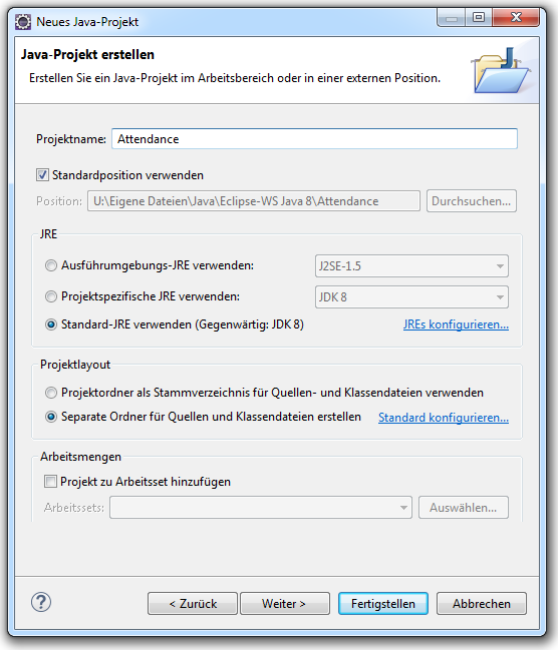
Wir verzichten auf ein vollends triviales Beispiel vom Typ *Hallo Welt* und erstellen ein nützliches Programm, das z.B. zur Anwesenheitsüberwachung durch einen Blockwart mit freiem Blick auf den einzigen Hauseingang dienen kann:



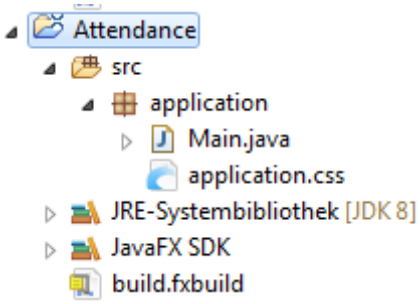
Um in Eclipse 4.5 bei installiertem e(fx)clipse-Plugin ein Projekt für dieses Programm anzulegen, wählen wir den Menübefehl

Datei > Neu > Andere > JavaFX > JavaFX Project

Als Projektname eignet sich z.B. *Attendance* (dt.: *Aufsicht*):

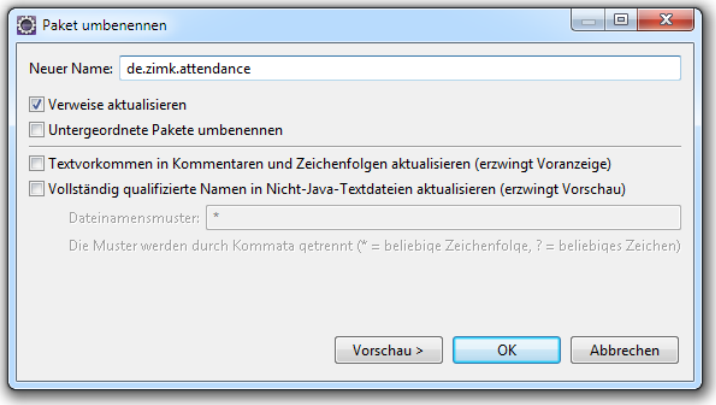


Die vom Assistenten angelegte Datei mit CSS-Formatdefinitionen (*Cascading Style Sheets*) kann gelöscht werden:



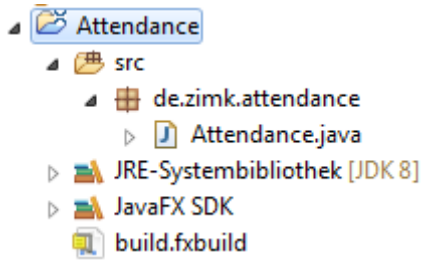
Das vom Assistenten angelegte Paket namens `application` sollte umbenannt werden, z.B. in `de.zimk.attendance`. Weil der Paketname vermutlich an mehreren Stellen im Projekt auftaucht, nutzen wir die Refaktorisierungs-Kompetenz von Eclipse:

- Aus dem Kontextmenü zum Paket wählen: **Refactoring > Umbenennen**
- Neuen Namen eintragen und quittieren:



- Eventuelle Selbstzweifel von Eclipse an der Qualität seiner Arbeit kann man in der Regel übergehen.

Auf analoge Weise kann die Quellcodedatei zur Hauptklasse (alias: Anwendungsklasse) statt `Main.java` einen individuellen Namen erhalten, z.B. `Attendane.java`:



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:

```
public class Attendance extends Application {
  .  .  .
}
```

Damit ein Objekt der Anwendungsklasse erzeugt werden kann, muss ein parameterfreier, öffentlich verfügbarer Konstruktor vorhanden sein.

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 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. Nachdem wir im Beispielprogramm auf eine CSS-Datei verzichten, ist die Anweisung zum Laden dieser Datei aus der Methode **start()** zu entfernen, und wir können bei den restlichen Anweisungen auf eine Ausnahmebehandlung per **try-catch** verzichten. Die resultierende **start()** - Methode

```
public class Attendance extends Application {
 @Override
 public void start(Stage primaryStage) {
  final BorderPane root = new BorderPane();
  final Scene scene = new Scene(root, 400, 400);
  primaryStage.setScene(scene);
  primaryStage.show();
 }
 public static void main(String[] args) {
  launch(args);
 }
}
```

erstellt unter Verwendung eines **BorderPane**-Objekts als Wurzel-Container (vgl. Abschnitt 13.2.3) ein **Scene**-Objekt

```
final BorderPane root = new BorderPane();
final Scene scene = new Scene(root, 400, 400);
```

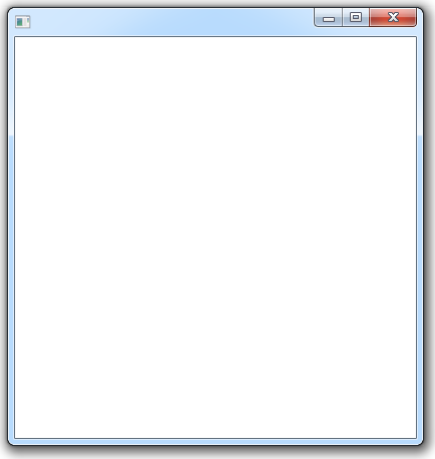
und bringt es mit der folgenden Anweisung

```
primaryStage.setScene(scene);
```

auf die Bühne, die schließlich per **show()** - Aufruf

```
primaryStage.show();
```

sichtbar gemacht wird:

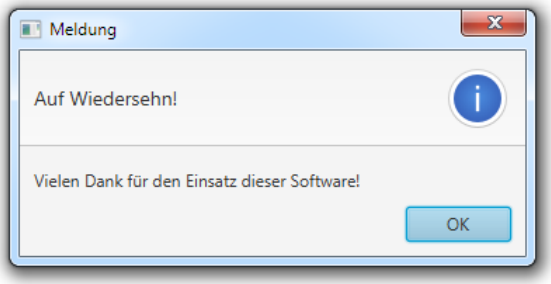


Im Vergleich zur Swing-Technik ist anzumerken, dass für einen Start ohne Multithreading-Probleme keine Vorsichtsmaßnahmen auf Seiten des Anwendungsprogrammierers erforderlich sind.

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() {
 final Alert alert = new Alert(AlertType.INFORMATION,
                         "Vielen Dank für den Einsatz dieser Software!");
 alert.setHeaderText("Auf Wiedersehn!");
 alert.showAndWait();
}
```

wird noch ein Standarddialog angezeigt:



Im Unterschied zu JavaFX muss in Swing durch spezielle Vorkehrungen (z.B. durch einen **WindowListener**) verhindert werden, dass nach dem Schließen aller Anwendungsfenster im Hintergrund ein unsichtbares Programm weiterläuft. Wird in JavaFX mit der folgenden Anweisung

```
Platform.setImplicitExit(false);
```

das automatische Programmende nach dem Schließen der Fenster verhindert, muss analog zu Swing für das Programmende gesorgt werden.

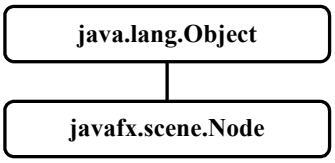
13.2.3 Grundbegriffe

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 per Parameter.

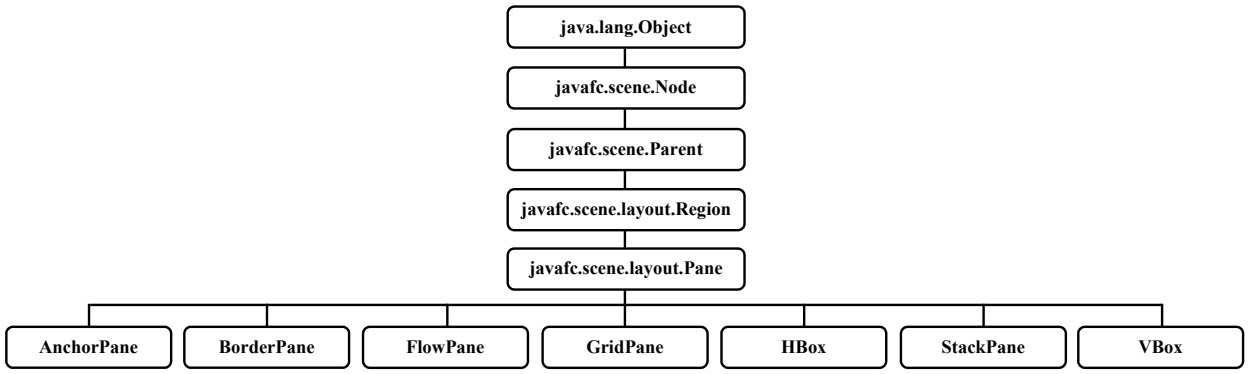
Um Leben auf eine Bühne zu bringen, muss dort ein Objekt der Klasse **Scene** (aus dem Paket **javafx.scene**) eingesetzt werden. Ein **Stage**-Objekt enthält (zu einem Zeitpunkt) genau ein **Scene**-Objekt, das seinerseits als Container für beliebig viele visuelle Komponenten (Bedienelemente und/oder Grafikelemente) dient.

Bei den Komponenten in der Szene kommen Endknoten (z.B. Bedienelemente wie Textfelder, Befehlsschalter etc.) und Container-Knoten (z.B. vom Typ **GridPane**) in Frage, die baumartig angeordnet sind mit einem Wurzelknoten an der Basis. Häufig wird der Komponentenbaum als **Szenengraph** (engl. *scene graph*) bezeichnet.

Alle Komponenten im Szenengraphen stammen von der Klasse **Node** im Paket **javafc.scene** ab:



Alle Komponenten im Szenengraphen, die Kinder (eingeschachtelte Knoten) haben können, stammen von der Klasse **Parent** im Paket **javafc.scene** ab:



Kurze Hinweise zu den Containern:

- **AnchorPane**
  Enthaltene Komponenten können an den Seiten in einem bestimmten Abstand verankert werden.
- **BorderPane**
  Eignet 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 (z.B. für einen zu bearbeitenden Text), der optional noch von Bedienelementen am linken und rechten Fensterrand flankiert wird.
- **FlowPane**
  Je nach Orientierung des Containers werden die Knoten neben bzw. untereinander angeordnet und nach Erreichen des Randes umgebrochen.
- **GridPane**
  Die Bedienelemente werden in einer Matrix angeordnet und dürfen sich bei Bedarf auch über mehrere Zellen erstrecken. 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);
  ```
- **HBox**, **VBox**
  Die Bedienelemente werden neben- bzw. untereinander angeordnet.
- **StackPane**
  Die Elemente werden übereinander geschichtet.

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));
```

13.2.4 Programmatische Layoutdefinition

Für das Beispielprogramm zur Anwesenheitsüberwachung eignet sich als Wurzel-Container ein **GridPane**-Objekt besser geeignet als ein **BorderPane**-Objekt, so dass wir in der **start()** - Methode einen entsprechenden Austausch vornehmen.

```
final GridPane root = new GridPane();
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**:

```
final Label lblName = new Label("Name:");
final Label lblPresent = new Label("Anwesend:");
```

Zur Aufnahme der Namen von Neuankömmlingen ist ein Objekt der Klasse **TextField** aus dem Paket **javafx.scene.control** zuständig:

```
final TextField tfName = new TextField();
```

Über zwei Befehlsschalter aus der Klasse **Button** sollen Personen in die Anwesenheitsliste aufgenommen bzw. aus dieser Liste gestrichen werden:

```
final Button btnAdd = new Button("Angekommen");
final 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 bei der Zusammensetzung der beobachtbaren Liste reagiert:[1]

```
final ObservableList<String> persons = FXCollections.observableArrayList();
final SortedList<String> perSorted = new SortedList<>(persons,
                                                      Comparator.naturalOrder());
final 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 13.4.2.3). 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.

Beim Aufnahmeschalter wird durch einen **setOnAction()** - Aufruf eine per Lambda-Ausdruck realisierte Klickereignisbehandlungsmethode registriert:

```
btnAdd.setOnAction(event -> {
 final String s = tfName.getText();
 if (s.length() > 0)
  persons.add(s);
});
```

Ist im Textfeld ein Eintrag (mit Länge > 0) vorhanden, wird dieser per **add()** - Aufruf in die beobachtbare Liste aufgenommen.

---

[1] Laut API-Dokumentation sollte die explizite Angabe eines Komparators im **SortedList<String>** - Konstruktor überflüssig sein. Aufgrund eines von Inden (2015, S. 198) beschriebenen Fehlers, der auch noch in der Java-Version 1.8.0_66 vorhanden ist, muss man jedoch die Konstruktor-Überladung *mit* Komparator verwenden, um eine sortierte Liste zu erhalten.

Analog erhält der Entlassungsschalter eine Klickbehandlungsmethode, die das im **ListView<String>** gewählte Element ermittelt und es per **remove()** - Methode aus der beobachtbaren Liste entfernt:

```
btnRemove.setOnAction(event ->
 persons.remove(lvPersons.getSelectionModel().getSelectedItem()));
```

Bei jeder Änderung der beobachtbaren Liste aktualisiert das **ListView<String>** - Objekt sofort die Anzeige.

Über die **GridPane**-Methode **add()** mit Parametern für die nullbasierte Spalten- bzw. Zeilennummer verfrachtet man die Bedienelemente in die passende Matrixzelle:

```
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);
```

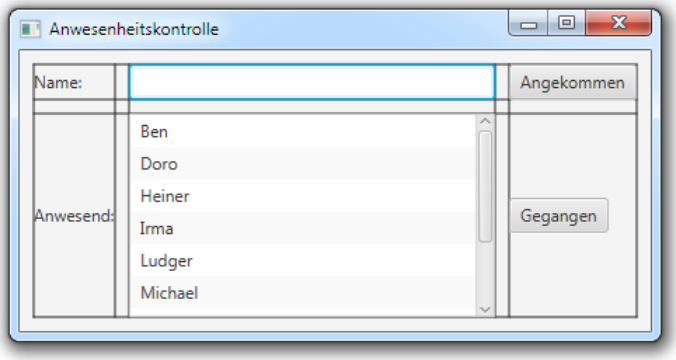
Abschließend sorgen wir noch dafür, dass die **GridPane**-Komponente den verfügbaren Platz im Fenster stets vollständig nutzt, wobei in horizontaler Richtung das Texteingabefeld und in vertikaler Richtung die Liste von einem wachsenden Platzangebot profitieren sollen:

```
GridPane.setVgrow(lvPersons, Priority.ALWAYS);
GridPane.setHgrow(tfName, Priority.ALWAYS);
```

Bei der Gestaltung eines **GridPane**-Containers kann man vorübergehend Gitterlinien aktivieren, um das Design zu erleichtern:

```
root.setGridLinesVisible(true);
```

Das Ergebnis im Beispielprogramm:



Unsere Anwendungsklasse ist noch relativ gut zu überblicken:

```
package de.zimk.attendance;

import javafx.application.Application;
  .  .  .
import javafx.scene.layout.Priority;

public class Attendance extends Application {
 final private int DIST = 10;

 @Override
 public void start(Stage primaryStage) {
  final ObservableList<String> persons =
   FXCollections.observableArrayList(anwesend);
  final SortedList<String> perSorted = new SortedList<>(persons,
                                                  Comparator.naturalOrder());
  final ListView<String> lvPersons = new ListView<>(perSorted);

  final GridPane root = new GridPane();
  root.setPadding(new Insets(DIST, DIST, DIST, DIST));
  root.setHgap(DIST); root.setVgap(DIST);

  final Label lblName = new Label("Name:");
  final Label lblPresent = new Label("Anwesend:");
  final TextField tfName = new TextField();

  final Button btnAdd = new Button("Angekommen");
  btnAdd.setOnAction(event -> {
   String s = tfName.getText();
   if (s.length() > 0)
    persons.add(s);
  });
  final Button btnRemove = new Button("Gegangen");
  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.setVgrow(lvPersons, Priority.ALWAYS);
  GridPane.setHgrow(tfName, Priority.ALWAYS);

  final Scene scene = new Scene(root, 450, 200);
  primaryStage.setScene(scene);
  primaryStage.setTitle("Anwesenheitskontrolle");
  primaryStage.show();
 }

 public static void main(String[] args) {
  launch(args);
 }
}
```

Bei ernsthaften Anwendungen ist die deklarative FXML-Alternative gegenüber der programmatischen GUI-Definition zu bevorzugen. Wir haben die FXML-basierte Technik schon in Abschnitt 4.8 verwendet und werden gleich in Abschnitt 13.3 weitere Erfahrungen damit sammeln.

13.3 JavaFX-Anwendungen mit Model-View-Controller - Architektur (MVC)

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 (persistent) verwalten,
- mit Benutzern interagieren, die meist mit Hilfe einer grafischen Bedienoberfläche Daten einsehen und verändern können.

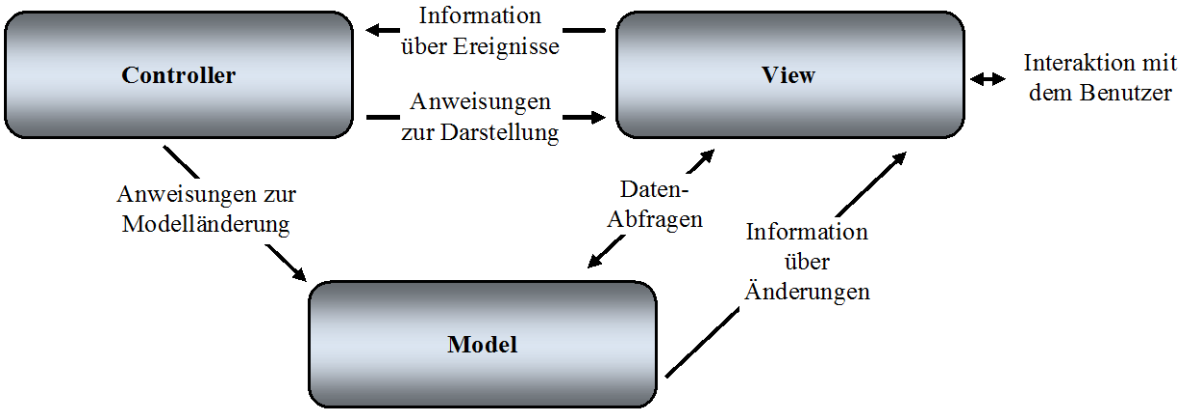
Es ist allgemeiner Konsens, dass in einer objektorientierten Software die Verwaltung und Transformation der Daten nach den Regeln der Geschäftslogik getrennt werden sollte von der Präsentation der Daten und der Benutzerinteraktion (siehe z.B. Sharan, S. 419ff). Seit ca. 30 Jahren werden Vorschläge zur 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 wollen uns nicht an der Diskussion über Software-Architekturmodelle beteiligen und übernehmen von Hommel (2014) folgende Begriffsexplikationen:

- **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 über Properties und Datenbindung (siehe Abschnitt 13.4) und erlauben eine kontrollierte Datenmodifikation durch Controller-Objekte (siehe unten).
- **View**
  Die View-Komponente einer JavaFX-Anwendung besteht aus den per FXML deklarierten Fenstern (Szenen). Als Knoten in den Szenegraphen der Fenster sind Bedienelemente vorhanden, die eigenständig mit dem Benutzer interagieren (z.B. Texteingabefelder, Befehlsschalter) und registrierte Controller-Objekte über eingetretene Ereignisse informieren (z.B. Enter-Tastendruck während der Texteingabe).
- **Controller**
  Jedem Fenster (jeder Szene) wird (z.B. per Scene Builder) 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 das Model kommen auch direkte Modifikationen in Frage (z.B. das (De)aktivieren von View-Knoten).

Wie sich in Abschnitt 13.3.5 herausstellen wird, befinden nach einer typischen JavaFX-Anwendungserstellung unter Verwertung von FXML-Deklarationsdateien die Bedienelemente in derselben Klasse wie die zugehörigen Ereignisbehandlungsmethoden. Für die View- und die Controller-Rollen ist also dieselbe Klasse zuständig.

In der folgenden Abbildung sind die Anwendungsbestandteile und ihre Kommunikationsbeziehungen dargestellt:



Wir erstellen nun das im bisherigen Verlauf das JavaFX-Kapitels erstellte Beispiel zur Anwesenheitskontrolle neu ...

- unter Verwendung der View-Deklaration per FXML
- und mit Beachtung der MVC-Anwendungsarchitektur.

Die ersten Schritte beginnend mit dem Menübefehl

Datei > Neu > Andere > JavaFX > JavaFX Project

laufen analog zu Abschnitt 13.2.1 ab:

- Das Projekt erhält den Namen `AttendanceMVC`.
- Das vom Assistenten angelegte Paket `application` wird per Refaktorisierung umbenannt in `de.zimk.attendance`.

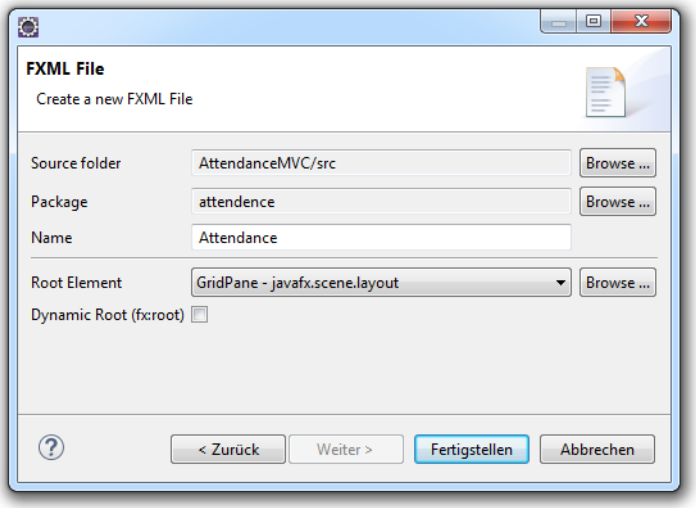
Anschließend löschen wir die im Paket vorhandenen Dateien (**Main.java** sowie **application.css**) und legen die JavaFX-Hauptklasse zum Projekt mit dem Namen `Attendance` neu an über den Menübefehl

Neu > Andere > JavaFX > Classes > JavaFX Main Class

aus dem Kontextmenü zum Paket. Schließlich entsteht über den Menübefehl

Neu > Andere > JavaFX > New FXML Document

eine FXML-View-Deklarationsdatei mit dem Namen **Attendance.fxml** und einem **Root Element** (Wurzelknoten) aus der Klasse **GridPane**:



13.3.1 Model

Im Beispiel soll das Model-Objekt ...

- eine beobachtbare Liste (siehe Abschnitt 13.4.2.3) mit Elementen vom Typ **String** verwalten,
- eine sortierte, unveränderliche Variante dieser Liste als Objekt vom Typ **SortedList<String>** anbieten,
- über öffentliche Methoden namens `add()` und `remove()` das Einfügen bzw. Entfernen von Listenelementen erlauben, wobei das Einfügen nicht zu Dubletten führen darf.

Die folgende Implementation der Klasse `AttendanceModel` kommt im Wesentlichen mit den in Abschnitt 13.2.4 erläuterten Techniken aus:

```
package de.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() {
  final String[] anwesend = new String[] {"Willi", "Otto", "Theo", "Irma", "Doro",
                                          "Heiner", "Michael", "Ludger", "Ben"};
  persons = FXCollections.observableArrayList(anwesend);
  perSorted = new SortedList<>(persons, Comparator.naturalOrder());
 }

 public SortedList<String> getSortedList() {
  return perSorted;
 }

 public void add(String s) {
  if (!persons.contains(s))
   persons.add(s);
 }

 public void remove(String s) {
  persons.remove(s);
 }
}
```

13.3.2 GUI-Gestaltung per Scene Builder

Aus dem Kontextmenü zur FXML-Datei **Attendance.fxml** wählen wir das Item **Open with Scene Builder**, um die GUI-Deklaration mit dem Scene Builder vornehmen zu können. Wir markieren im **Hierarchy**-Segment den vorhandenen **GridPane**-Wurzel-Container und setzen über das **Layout**-Segment der **Inspector**-Zone seine bevorzugte Breite bzw. Höhe (**Pref Width** bzw. **Pref Height**) auf 450 bzw. 200.

Über das Kontextmenü zum Wurzelknoten fügen wir 2 Zeilen und 3 Spalten ein:

GridPane > Add Row Below
Grid Pane > Add Row After

Aus der **Library** befördern wir unter Zuhilfenahme der Suche am oberen Fensterrand zwei **Label**-Komponenten, eine **TextField**-Komponente, eine **ListView**-Komponente und zwei **Button**-Komponenten in die gewünschte Gitterzelle.

Die **Label**- und **Button**-Komponenten werden 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 fern zu halten, 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 daneben stehend 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.

Nach dem Menübefehl

Preview > Show Preview in Window

kann man sich vom gelungenen Design überzeugen.

Um ein Steuerelement aus dem Szenegraphen später zum Zwecke 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:

TextView-Komponente	`tfName`
ListView-Komponente	`lvPersons`
Button „Angekommen“	`btnAdd`
Button „Gegangen“	`btnRemove`

13.3.3 FXML

Wir speichern die grafisch gestaltete FXML-Datei mit dem Menübefehl

File > Save

des Scene Builders, beenden das Programm anschließend und wechseln zu Eclipse. Dessen Editor zeigt den aktuellen Stand der FXML-Datei:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.GridPane?>

<GridPane prefHeight="200.0" prefWidth="450.0" hgap="10.0" vgap="10.0"
          xmlns="http://javafx.com/javafx/8.0.40"
          xmlns:fx="http://javafx.com/fxml/1">
   <padding>
      <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
   </padding>
   <children>
      <Label text="Name:" />
      <Label text="Anwesend:" GridPane.rowIndex="1" />
      <TextField fx:id="tfName" GridPane.columnIndex="1" GridPane.hgrow="ALWAYS" />
      <ListView fx:id="lvPersons" 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>
</GridPane>
```

Die Datei startet mit so genannten **XML-Verarbeitungsinstruktionen**:

```
<? ... ?>
```

Auf eine Zeile mit der XML-Versions- und Kodierungsangabe folgen Importdeklarationen für Java-Pakete oder -Klassen analog zu Abschnitt 3.1.7.

Das FXML-Wurzelelement namens **GridPane** enthält neben den im Scene Builder festgelegten Layout-Attributen noch XML-Namensraumvereinbarungen:

```
<GridPane prefHeight="200.0" prefWidth="450.0" hgap="10.0" vgap="10.0"
          xmlns="http://javafx.com/javafx/8.0.40"
          xmlns:fx="http://javafx.com/fxml/1">
</GridPane>
```

Weil die Vereinbarung eines umlaufenden Innenrandes für den **GridPane**-Container nicht per Attribut-Syntax bewältigt werden kann, erscheint das eingeschachtelte Element **padding**:

```
   <padding>
      <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
   </padding>
```

Das **children**-Element enthält die Knoten im Szenegraphen, wobei die hierarchische Struktur deutlich besser zu erkennen ist als bei der programmatischen Layout-Deklaration (vgl. Abschnitt 13.2.4).

```
   <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 ohne und mit Klassenpräfix auf:

```
<Button fx:id="btnRemove" mnemonicParsing="false" text="Gegangen"
        GridPane.columnIndex="2" GridPane.rowIndex="1" />
```

Bei Attributen ohne Klassenpräfix wird eine Instanzmethode zum Setzen einer Eigenschaftsausprägung aufgerufen (ein Property-Setter, vgl. Abschnitt 13.4). Bei Attributen mit Klassenpräfix wird eine statische Methode aufgerufen (Horstmann 2014, S. 87), z.B.

```
GridPane.setColumnIndex(btnRemove, 2);
```

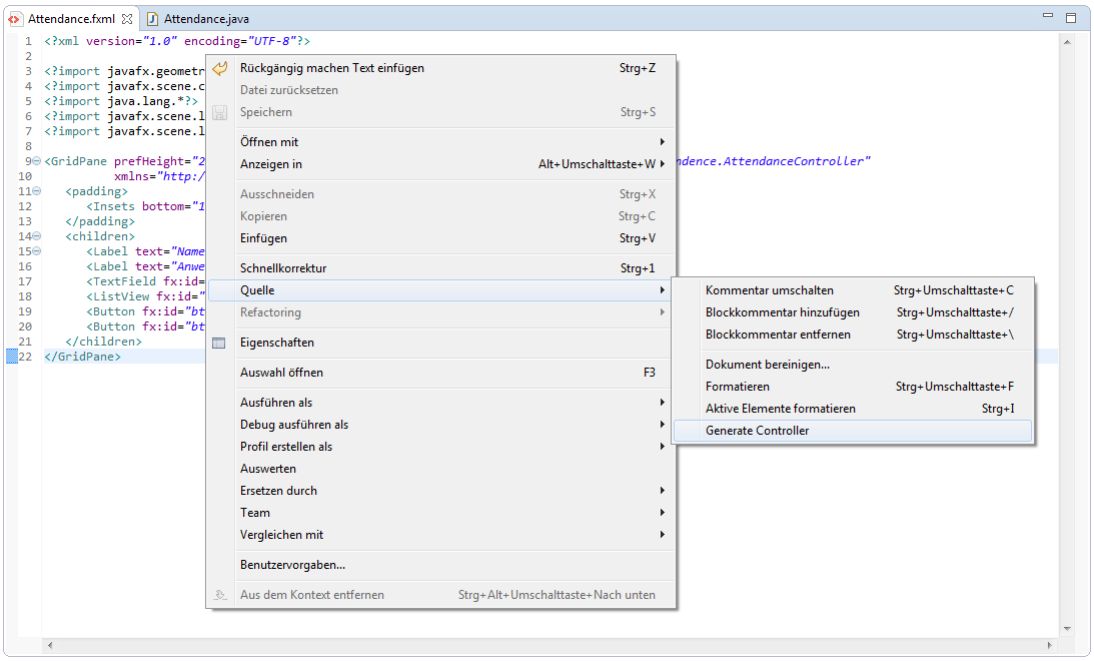
Weitere Hinweise zum FXML-Format finden sich z.B. bei Weaver et al (2014). Wie eine FXML-Datei beim Programmstart geladen wird, erfahren Sie in Abschnitt 13.3.5.

13.3.4 Controller

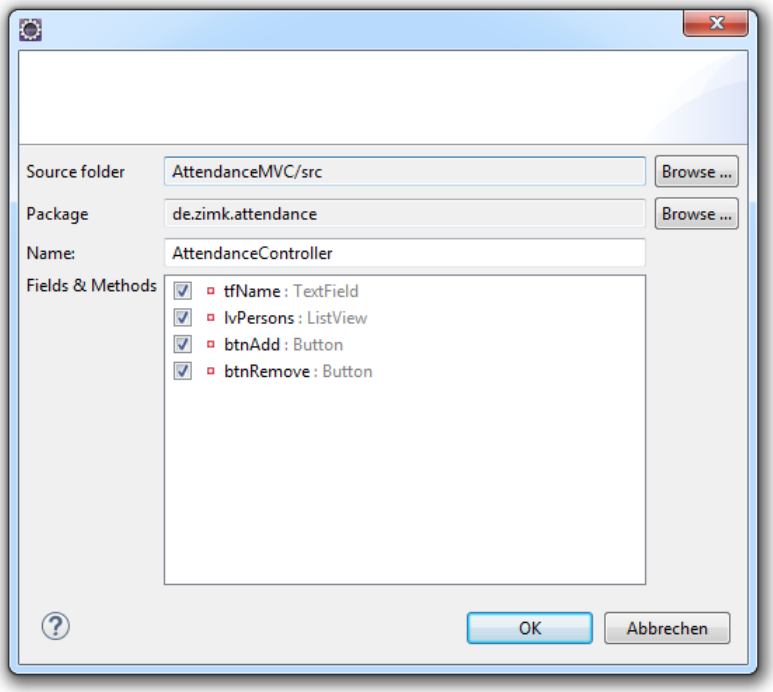
Die Ereignisbehandlung in unserem Projekt soll durch ein Objekt der Controller-Klasse erledigt werden. Dank **e(fx)clipse** - Plugin lässt sich ein Rohling für die Controller-Klasse über den Befehl

Quelle > Generate Controller

aus dem Kontextmenü zum Editor-Fenster mit der geöffneten FXML-Datei erstellen:



Es wird ein sinnvoller Name vorgeschlagen, und die im Scene Builder vereinbaren Steuerelementkennungen sind berücksichtigt:



Schließlich erhalten wir den folgenden Rohling für die Controller-Klasse:

```
package de.zimk.attendance;

import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.control.ListView;

public class AttendanceController {
 @FXML
 private TextField tfName;
 @FXML
 private ListView lvPersons;
 @FXML
 private Button btnAdd;
 @FXML
 private Button btnRemove;
}
```

Eclipse kritisiert zu Recht die Verwendung des Rohtyps der generischen Klasse **ListView<E>**, und wir korrigieren die betroffene Zeile:

```
private ListView<String> lvPersons;
```

Mit der Annotation **@FXML** wird dafür gesorgt, dass über private Instanzvariablen der Controller-Klasse die GUI-Komponenten angesprochen werden können, die im FXML-Code eine passende Kennung (**fx.id**) und einen passendem Typ erhalten haben. Beim Laden der FXML-Datei (siehe Abschnitt 13.3.5) wird ein Objekt der Controller-Klasse angelegt und 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).

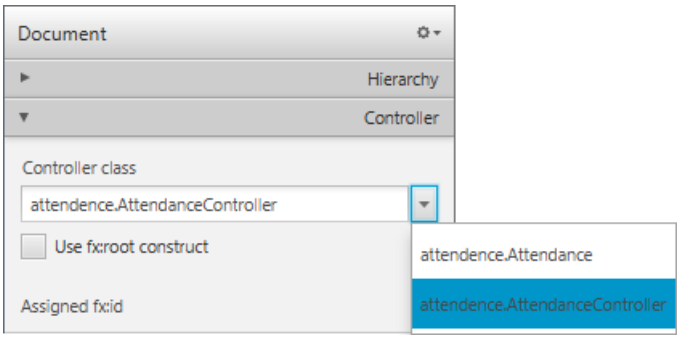
Wir ergänzen eine Instanzvariable vom Typ `AttendanceModel` (vgl. Abschnitt 13.3.1), weil der Controller mit dem Model kommunizieren soll:

```
private AttendanceModel am;
```

Zur Initialisierung des Controller-Objekts (nach der vollständigen Verarbeitung des Szenengraph-Wurzelelements) wird die parameterfreie Methode **initialize()** implementiert und mit **@FXML** annotiert, sodass sie beim Laden der FXML-Datei automatisch ausgeführt wird.[1] In **initialize()** ermittelt der Controller mit der 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:

```
@FXML
public void initialize() {
  am = Attendance.getModel();

  lvPersons.setItems(am.getSortedList());

  btnAdd.setOnAction(event -> {
    final String s = tfName.getText();
    if (s.length() != 0)
      am.add(s);
  });

  btnRemove.setOnAction(event ->
                        am.remove(lvPersons.getSelectionModel().getSelectedItem()));
}
```

Wir öffnen den Scene Builder erneut zum Editieren der Datei **Attendance.fxml**, wobei das Programm die neu erstellte Controller-Klasse zur Kenntnis nimmt, die sich dazu im selben Ordner wie die FXML-Datei befinden muss. Über das **Controller**-Segment der **Document**-Zone lässt sich die Controller-Klasse zur FXML-Datei wählen:



Im **GridPane**-Wurzelelement der FXML-Datei wird daraufhin die Controller-Klasse eingetragen:

```
<GridPane prefHeight="200.0" prefWidth="450.0" hgap="10.0" vgap="10.0"
          fx:controller="de.zimk.attendance.AttendanceController"
          xmlns="http://javafx.com/javafx/8.0.40"
          xmlns:fx="http://javafx.com/fxml/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 Oracle-Dokumentation (https://docs.oracle.com/javase/8/javafx/api/javafx/fxml/Initializable.html, abgerufen am 08.02.2016) heißt es dazu:

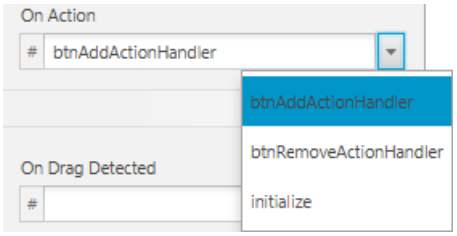
> This interface has been superseded by automatic injection of `location` and `resources` properties into the controller.

Ab JavaFX 8 sollte 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).

In Abschnitt 4.8.4 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 **Document**-Zone des Scene Builders eine **ActionEvent**-Behandlungsmethode der Controller-Klasse zuweisen, z.B.:



Nach dem Speichern der FXML-Deklaration im Scene Builder würde der FXML-Editor in Eclipse zeigen, wie die Vereinbarung der Ereignisbehandlungsmethoden umgesetzt wurde:

```
<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 (siehe oben).

13.3.5 Anwendungsklasse

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 bereits 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 in Abschnitt 13.2.3) auf das Wurzelelement des Szenegraphen:
  ```
  Parent parent = null;
  try {
      parent = FXMLLoader.load(getClass().getResource("Attendance.fxml"));
  } catch (Exception e) {
      System.out.println("Fehler beim Lesen der Layout-Spezifikation");
      System.exit(1);
  }
  ```
  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:

```
final Scene scene = new Scene(parent);
primaryStage.setScene(scene);
primaryStage.setTitle("Anwesenheitskontrolle");
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 eine statische Methode, mit der sich das Controller-Objekt 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.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();

  final Parent parent = null;
  try {
   parent = FXMLLoader.load(getClass().getResource("Attendance.fxml"));
  } catch (Exception e) {
   System.out.println("Fehler beim Lesen der Layout-Spezifikation");
   System.exit(1);
  }

  final Scene scene = new Scene(parent);
  primaryStage.setScene(scene);
  primaryStage.setTitle("Anwesenheitskontrolle");
  primaryStage.show();
 }
```

```
    static public AttendanceModel getModel() {
     return am;
    }

    public static void main(String[] args) {
     launch(args);
    }
}
```

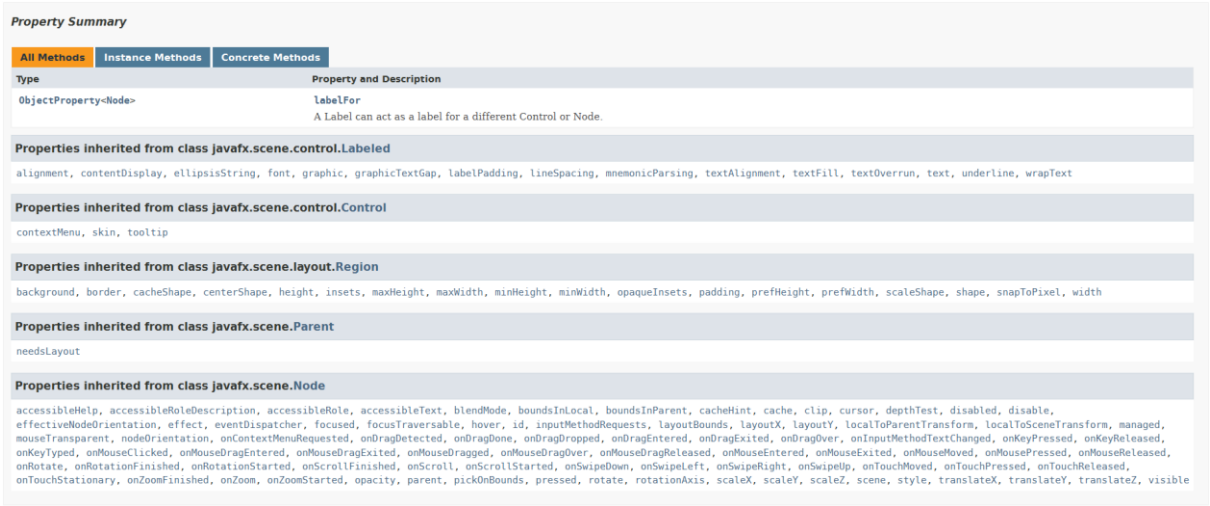
Das Programm sollte nun einsatzfähig sein. Es besteht aus vier Dateien:

- **AttendanceModel.java** mit der Model-Klasse
- **Attendance.fxml** mit der View-Deklaration
- **AttendanceController.java** mit der Controller-Klasse
- **Attendance.java** mit der Anwendungs-Klasse

Es existiert zwar eine eigene Datei für die View-Konfiguration, doch ist in der Präsentationsschicht der Anwendung *eine* Klasse (**AttendanceController**) mit View-und Controller-Aufgaben befasst. Eine Aufspaltung ist möglich, bringt aber wegen der engen Verzahnung von View und Controller weniger Vorteile als Umstände.

13.4 Properties und automatische Synchronisation

JavaFX-Komponenten (z.B. für die Steuerelemente und Zeichnungselemente in einer grafischen Bedienoberfläche) verwenden für ihre öffentlich zugänglichen Eigenschaften (z.B. Hintergrundfarbe, bevorzugte Höhe und Breite, Beschriftung) Objekte aus speziellen Property-Klassen, die wir uns nun ansehen werden. 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 13.5.1):



Property Summary

All Methods | Instance Methods | Concrete Methods

Type	Property and 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, visible

Die Property-Klassen von JavaFX gehen in ihrer Funktionalität weit über die traditionelle Realisation der Eigenschaften von Komponenten hinaus. Insbesondere lassen sich JavaFX-Properties auf elegante Weise miteinander verknüpfen, so dass eine automatische Synchronisation ihrer Werte stattfindet.

13.4.1 Properties

Unter dem Namen *Property* hat JavaFX beobachtbare Werte eingeführt. Sie werden realisiert durch Klassen, deren Objekte einen Wert kapseln und registrierte Interessenten informieren, wenn sich der Wert geändert hat oder ungültig geworden ist.

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 alle Eigenschaften Zugriffsmethoden mit genormten Namen bieten:

- 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()`).

Es resultieren *Komponenten*, die sich besonders gut für die Wiederverwendung und für die Unterstützung durch Entwicklungswerkzeuge eignen. Neben den Namenskonventionen besteht die JavaBeans-Komponententechnologie aus einem API mit etlichen Klassen und Schnittstellen im Paket **java.beans**. Zur Orientierung eignet sich z.B. das Java Tutorial (Oracle 2015).[1]

Auch bei den JavaFX-Komponenten werden Zugriffsmethoden unter Beachtung der traditionellen Namenskonventionen für JavaBeans verwendet, doch die Eigenschaften werden nicht in einfachen Instanzvariablen gespeichert, sondern in Objekten von speziellen Property-Klassen.

Neben den auch von einer JavaBeans-Komponente angebotenen Zugriffsmethoden für eine Eigenschaft bietet eine JavaFX-Komponente zu einem Property-Objekt eine zusätzliche Zugriffsmethode mit einer Referenz auf dieses Objekt als Rückgabe (z.B. mit dem Namen `numeratorProperty()`). Diese Rückgabe wird z.B. benötigt, um beim Property-Objekt einen Beobachter für Wertveränderungen zu registrieren.

13.4.1.1 Namensregeln

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. `accountBalance`).
- In den Namen der Methoden für den lesenden bzw. schreibenden Zugriff folgt (wie bei JavaBeans) auf *get* bzw. *set* der Eigenschaftsname mit groß geschriebenen Anfangsbuchstaben (z.B. `getAccountBalance` bzw. `setAccountBalance`).
- Im Namen der Methode zum Erfragen der Property-Referenz folgt auf den Eigenschaftsnamen der Zusatz *Property* (z.B. `accountBalanceProperty`).

---

[1] Die Behandlung der JavaBeans ist hier zu finden: http://download.oracle.com/javase/tutorial/javabeans.

13.4.1.2 Property-Klassen in JavaFX

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**:

- **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, stellt also 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.

Für Zugriffe auf den gekapselten Wert und die Selbstdarstellung beherrscht ein **IntegerProperty**-Objekt 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 Bestandteil einer Komponente oder eigenständig sein. Von **getBean()** erhält man ggf. die Adresse der Komponente, zu der die angesprochene Eigenschaft gehört, oder die Rückgabe **null**.

Wir betrachten die Verwendung von JavaFX-Properties im Rahmen einer simplen Komponentenklasse 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

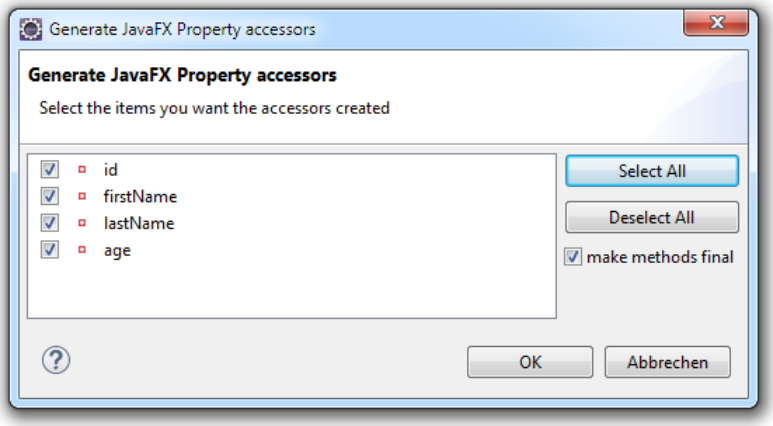
```
public class Person {
  private static int count = 0;
  private ReadOnlyIntegerWrapper id = new ReadOnlyIntegerWrapper(this, "id", ++count);
  private StringProperty firstName = new SimpleStringProperty(this, "firstName", "");
  private StringProperty lastName = new SimpleStringProperty(this, "lastName", "");
  private IntegerProperty age = new SimpleIntegerProperty(this, "age", -99);

  public Person(String first, String last, int age) {
    firstName.setValue(first);
    lastName.setValue(last);
    this.age.setValue(age);
  }
   .  .  .
}
```

kommen Property-Konstruktoren mit drei Parametern zum Einsatz:

- Im ersten Parameter (**Object** *bean*) wird festgelegt, zu welcher Komponente die Property gehört. Im Beispiel wird jeweils das gerade erzeugte Person-Objekt eingetragen.
- Im zweiten Parameter (**String** *name*) erhält die Property einen Namen (vgl. Abschnitt 13.4.1.1.
- Im dritten Parameter wird der initiale Wert festgelegt.

Bei der Erstellung der Getter- und Setter-Methoden zu den Eigenschaften greift uns Eclipse dank e(fx)clipse - Plugin kräftig unter die Arme:

- Wir wählen aus dem Kontextmenü zum Editorfenster mit der `Person`-Klassendefinition den Befehl **Quelltext > Generate JavaFX Getters and Setters**.
- Anschließend erlaubt der folgende Dialog eine Auswahl der zu versorgenden Properties:



Über die Checkbox **make methods final** mit voreingestellter Markierung wird für finalisierte Methoden gesorgt, die also in abgeleiteten Klassen nicht überschrieben werden können.

Der Assistent erstellt die auch bei JavaBean-Eigenschaften üblichen Zugriffsmethoden unter Beachtung der in Abschnitt 13.4.1.1 beschriebenen Namensregeln, z.B. für die Property `firstName`:

```
public final java.lang.String getFirstName() {
  return this.firstNameProperty().get();
}

public final void setFirstName(final java.lang.String firstName) {
  this.firstNameProperty().set(firstName);
}
```

Darüber hinaus erhalten wir auch Methoden, welche eine Referenz auf das Property-Objekt liefern, z.B.:

```
public final StringProperty firstNameProperty() {
 return this.firstName;
}
```

Eine Property-Referenz wird benötigt, um ...

- Veränderungs- bzw. Invalidierungs-Listener zu registrieren
- Bindungen vorzunehmen.

Bei der Read-Only - Property `id` fehlt die Setter-Methode, 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 javafx.beans.property.ReadOnlyIntegerProperty idProperty() {
 return this.id.getReadOnlyProperty();
}

public final int getId() {
 return this.idProperty().get();
}
```

Das folgende Programm zeigt die Verwendung der `Person`-Properties, kann aber noch nicht demonstrieren, wozu der im Vergleich zu gewöhnlichen gekapselten Instanzvariablen erheblich höherer Property-Aufwand taugt:

```
public class PersonDemo {
 public static void main(String[] args) {
  final 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.

13.4.1.3 Invalidierungs- und Veränderungsereignisse

Im Vergleich zu einer JavaBean-Eigenschaft besitzt eine JavaFX-Property zusätzliche Kompetenzen:

- Alle JavaFX-Eigenschaften 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.

- 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.

Nach der Kreation ist ein Property-Objekt im gültigen Zustand. Bei einem Property-Objekt in gültigen Zustand hat eine Wertveränderung folgende Effekte:

- Das Objekt wechselt in den ungültigen Zustand
- und feuert ein Invalidierungsereignis.

Bei einem ungültigen Property-Objekt haben weitere Wertveränderungen keine Invalidierungsereignisse zur Folge. Eine Wertabfrage per **get()** - Aufruf bringt ein ungültiges Objekt in den gültigen Zustand.

Das beschriebene Verhalten ist im folgenden Programm zu beobachten:

Quellcode	Ausgabe
`import javafx.beans.property.*;` `public class InvalidationListener {` `  public static void main(String[] args) {` `   final 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);` `  }` `}`	`Wert nach Konstruktor: 0` `Invalidated` `Wert = 2` `Invalidated`

Wie das folgende Programm zeigt, informiert ein Property-Objekt registrierte Change-Listener über *jede* Wertveränderung:

Quellcode	Ausgabe
`import javafx.beans.property.*;` `public class ChangeListener {` `  public static void main(String[] args) {` `   final 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);` `  }` `}`	`Wert nach Konstruktor: 0` `Changed to 1` `Changed to 2`

13.4.1.4 Vermeidung von überflüssigen Objektkreationen

Für die Komfortfunktionen der JavaFX-Properties im Vergleich zu JavaBean-Eigenschaften muss man mit zahlreichen Objekt-Kreationen bezahlen. Mit einem simplen Trick lässt sich eine Objektkreation hinauszögern und vielfach vermeiden. Man wickelt 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 in Abschnitt 13.4.1.2 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:

Quellcode	Ausgabe
`public class LacyPropertyCreation {` `  public static void main(String[] args) {` `  final Person p = new Person();` `  p.setFirstName("Otto");` `  System.out.printf("My Name is %s\n\n", p.getFirstName());` `  p.firstNameProperty().addListener((obs, old, nev) ->` `  System.out.println("Name changed to " + nev));` `  p.setFirstName("Otti");` `  }` `}`	`My Name is Otto` `Property created` `Name changed to Otti`

13.4.2 Automatische Synchronisation von Property-Objekten

Ein Property-Objekt kann ...

- uni- oder bidirektional mit einem anderen Property-Objekt vom gleichen Typ verbunden werden,
- unidirektional mit einem **ObservableValue<T>** - Quellobjekt verbunden werden, hinter dem auch um ein Ausdruck (z.B. eine Summe) stehen kann.

13.4.2.1 Uni- und bidirektionale Synchronisation von Property-Objekten

JavaFX-Properties können aneinander gebunden werden, so dass eine (uni- oder bidirektionale) Synchronisation ihrer Werte stattfindet, ohne dass sich Programmierer abmühen müssen. Alle JavaFX-Eigenschaften implementieren das von **ReadOnlyProperty<T>** und **WritableValue<T>** abstammende 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 mit einem Objekt vom Typ **ObservableValue<? extends T>** verbunden (z.B. mit einem anderen Property-Objekt).
- **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.
- **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.*;

public class PropBiSync {
 public static void main(String[] args) {
  final IntegerProperty i1 = new SimpleIntegerProperty(1);
  final IntegerProperty i2 = new SimpleIntegerProperty(2);

  i1.addListener((obs, old, nev) -> System.out.println("i1 nun gleich "+i1));
  i2.addListener((obs, old, nev) -> System.out.println("i2 nun gleich "+i2));

  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 Change-Listener kann die (ausbleibende) Fortpflanzung von Änderungen beobachtet werden:

```
i1 nun gleich IntegerProperty [value: 2]

Änderungen nach bidirektionaler Verbindung:
i1 nun gleich IntegerProperty [value: 11]
i2 nun gleich IntegerProperty [value: 11]
i2 nun gleich IntegerProperty [value: 22]
i1 nun gleich IntegerProperty [value: 22]

Bidirektionale Verbindung aufgehoben:
i1 nun gleich IntegerProperty [value: 111]
```

In der nächsten Programmvariante sorgt eine unidirektionale Verbindung

```
i1.bind(i2);
```

dafür, dass nur Wertveränderungen (`i2` → `i1`) propagiert werden:

```
import javafx.beans.property.*;

public class PropUniSync {
 public static void main(String[] args) {
   final IntegerProperty i1 = new SimpleIntegerProperty(1);
   final IntegerProperty i2 = new SimpleIntegerProperty(2);

   i1.addListener((obs, old, nev) -> System.out.println("i1 nun gleich "+i1));
   i2.addListener((obs, old, nev) -> System.out.println("i2 nun gleich "+i2));

   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:

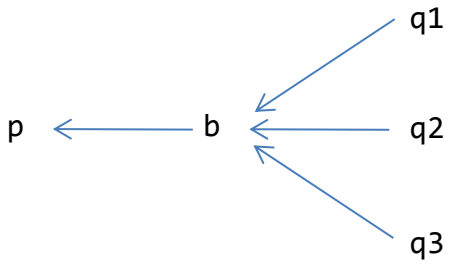
```
i1 nun gleich IntegerProperty [bound, value: 2]

Änderung nach unidirektionaler Verbindung:
i1 nun gleich IntegerProperty [bound, value: 22]
i2 nun gleich IntegerProperty [value: 22]

Unidirektionale Verbindung aufgehoben:
i2 nun gleich IntegerProperty [value: 222]
```

Direkte Schreibzugriffe auf die gebundene Property `i1` führen zu einem Ausnahmefehler.

13.4.2.2 Property-Objekt an einen Ausdruck binden

Es ist möglich, den Wert eines Property-Objekts an das Ergebnis eines Ausdrucks zu binden, zu dem mehrere Objekte vom Typ **ObservableValue<T>** (z.B. mehrere Property-Objekte) beitragen. Es ist kein Ausdruck im Sinne von Abschnitt 3.5 gemeint, sondern 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 Binding-Klassen **BooleanBinding**, **DoubleBinding**, usw.

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<T>** - 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).

13.4.2.2.1 High-Level Binding-API

Das High-Level Binding-API in JavaFX deckt häufige Anwendungsfälle auf bequeme Weise ab. Zur Demonstration greifen wir das Beispiel der `Person`-Komponente aus Abschnitt 13.4.1.2 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 zahlreiche 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 (z.B. durch Verketten von Aufrufen) erlauben (Sharan 2015, S. 62). 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**, welche das Interface **Binding<Number>** implementiert. An dieses Objekt wird die Property `age` gebunden:
  ```
  age.bind(ib);
  ```

[1] Der Einfachheit halber wird nur das Geburtsjahr erfasst, so dass die Altersberechnung nicht ganz korrekt ist.

- Statische Methoden der Klasse **Bindings**
  Ein Ausdruck vom Typ **Binding<T>**, an den eine Property gebunden werden kann, lässt sich auch über statische Methoden der Klasse **Bindings** erstellen, z.B.:

```
age.bind(Bindings.subtract(Year.now().getValue(), yearOfBirth));
```

Beide Techniken führen zur gewünschten automatischen Altersberechnung, was die erste Ausgabe des folgenden Programms

```
public class PersonDemo {
 public static void main(String[] args) {
  final Person p = new Person("Otto", "Rempremerding", 1990);

  System.out.println(p.getId()+", "+p.getFirstName()+" "+p.getLastName()+", "+p.getAge());

  p.setYearOfBirth(1990);
  p.ageProperty().addListener(obs -> System.out.println("\nAlter unbekannt"));

  p.ageProperty().addListener((obs, old, nev) ->
                              System.out.println("\nAlter geändert auf " + nev));
  p.setYearOfBirth(1995);
 }
}
```

demonstriert:

```
1, Otto Rempremerding, 26

Alter unbekannt

Alter geändert auf 21
```

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).

13.4.2.2.2 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. Dem in der Literatur üblichen 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, die Invalidierungs-Listener für alle Abhängigkeiten registriert:

protected final void bind(Observable... *dependencies***)**

Zur Initialisierung neuer Objekte können bei einer anonymen Klasse aber 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:

Quellcode	Ausgabe

```
import javafx.beans.binding.IntegerBinding;
import javafx.beans.property.*;

public class LowLevelBinding {
  public static void main(String[] args) {
    final IntegerProperty i1 = new SimpleIntegerProperty(1);
    final IntegerProperty i2 = new SimpleIntegerProperty(8);

    final 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);
    ggt.addListener((obs, old, nev) -> System.out.println("Neuer ggt: " + nev));
    i1.set(22);
    i1.set(12);
    i1.set(24);
    i1.set(32);
    i1.set(33);
  }
}
```

```
Neuer ggt: 2
Neuer ggt: 4
Neuer ggt: 8
Neuer ggt: 1
```

Die anonyme Klasse überschreibt die geerbte Methode **computeValue()** und realisiert dabei die Modulo-Variante des Euklidischen Verfahrens zur Berechnung des größten gemeinsamen Teilers (siehe Übungsaufgabe auf Seite 178). 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.

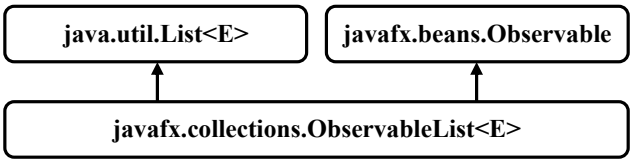
Weil das anonyme Objekt vom Typ **ObservableValue<? extends Number>** ist, kann im Beispielprogramm das **SimpleIntegerProperty**-Objekt `ggt` daran gebunden werden. Das Property-Objekt `ggt` erhält einen (per Lambda-Ausdruck realisierten) Change-Listener, so dass die Effekte der anschließend durchgeführten 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.

13.4.2.3 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 bei Sharan (2015, S. 83ff).

Während es z.B. sehr leicht fällt, ein **ListView<E>** - Steuerelement mit einem **ObservableList<E>** - Objekt zu verbinden, ist der Zugriff auf beliebige Ereignisse einer beobachtbaren Liste durch etliche technische Details belastet. Wer aktuell keine Informationen zur flexiblen Verarbeitung von Listenereignissen benötigt (z.B. die Anzahl der Listenelemente übersteigt eine Schwelle, irgendein Element wird auf einen bestimmten Wert gesetzt), sollte sich die technischen Details im weiteren Verlauf von Abschnitt 13.4.2.3 besser vorläufig ersparen.

Das Interface **ObservableList<E>** erweitert die Schnittstellen **List<E>** und **Observable**, so dass eine implementierende Klasse neben den Funktionalitäten einer Liste auch die Methoden **addListener()** und **removeListener()** anbieten muss, um Interessenten für Invalidierungsereignisse zu verwalten:



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***);**

Im Paket **javafx.collections** finden sich keine implementierenden Klassen zu den Schnittstellen für beobachtbare Kollektionen. Stattdessen ist die Klasse **FXCollections** mit statischen Fabrikmethoden vorhanden. Von der 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();
```

13.4.2.3.1 Zusammensetzung und Anordnung der Listenelemente beobachten

Um von der eben erstellten Liste 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("Veränderung " + change + ", entfernt: " + c.getRemoved());
   if (c.wasAdded())
    System.out.println("Veränderung " + change +
      ", betroffen: von " + c.getFrom() + " bis " + (c.getTo()-1) +
      ", ergänzt: " + c.getAddedSubList());
  }
 }
});
```

Ein **ListChangeListener<E>** muss die Methode **onChanged()** implementieren, die bei einer Listenveränderungen aufgerufen wird und dabei ein Parameterobjekt vom Typ der innerhalb von **ListChangeListener<E>** definierten statischen Mitgliedsklasse **Change** erhält. Der im **Change**-Objekt enthaltene Bericht kann mehrere Teile enthalten, 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**-Objekt keine weiteren Teilberichte vorhanden sind.

Über die (bei allen Teilberichten 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 13.4.2.3.2).

Welche Listenelemente von einer Veränderung betroffen sind, ist über die folgenden **Change**-Methoden zu ermitteln:

- **public int getFrom()**
  Liefert der Startindex (inklusive)
- **public int getTo()**
  Liefert der Endindex (exklusive)

Weitere **Change**-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 13.4.2.3 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("\nElemente 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:
Veränderung 1, betroffen: von 0 bis 5, ergänzt: [A, B, C, D, E, F]

Hintereinander liegende Elemente entfernen:
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:
Veränderung 1, entfernt: [A]
Veränderung 1, betroffen: von 0 bis 0, ergänzt: [A1]
```

Werden Listenelemente gelöscht, die *nicht* hintereinander liegen, dann enthält das **Change**-Objekt einen zweiteiligen Veränderungsbericht:

```
Nicht hintereinander liegende Elemente entfernen:
Veränderung 1, entfernt: [A1]
Veränderung 2, entfernt: [C]
```

13.4.2.3.2 Wertveränderungen bei Listenelementen beobachten

Sollen auch Wertveränderungen bei einzelnen Listenelementen beobachtet werden, dann muss für jedes Element ein Array mit Quellobjekten vom Typ **Observable** benannt werden, die auf Veränderungen beim Element abgehört werden sollen. Diese generelle und abstrakte Anforderung wird übersichtlich, wenn die Listenelemente selbst beobachtbar sind und folglich für einen einelementigen Quellen-Array taugen. Wir ersetzen im Beispiel aus Abschnitt 13.4.2.3.1 die Klasse **String** durch die Klasse **StringProperty**. Wegen des für jedes Element benötigten Quellen-Arrays muss zum Erstellen der beobachtbaren Liste eine Überladung der **FXCollections**-Methode **observableArrayList()** mit einem Parameter vom Typ **Callback<String, Observable[]>** verwenden werden, z.B.:

```
ObservableList<StringProperty> ols =
  FXCollections.observableArrayList(e -> new Observable[] {e});
```

Die zu implementierende **Callback<String, Observable[]>** - Methode **call()** erhält als Parameter ein Listenelement und hat als Rückgabe einen Array mit Objekten vom Typ **Observable** zu liefern. Im Beispiel erfolgt die Implementation per Lambda-Ausdruck, und die Rückgabe entsteht, indem das Listenelement in einen einelementigen Array verpackt wird.

Im folgenden Beispielprogramm erhält das **ObservableList<StringProperty>** - Objekt nach dem in Abschnitt 13.4.2.3.1 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 **Change**:

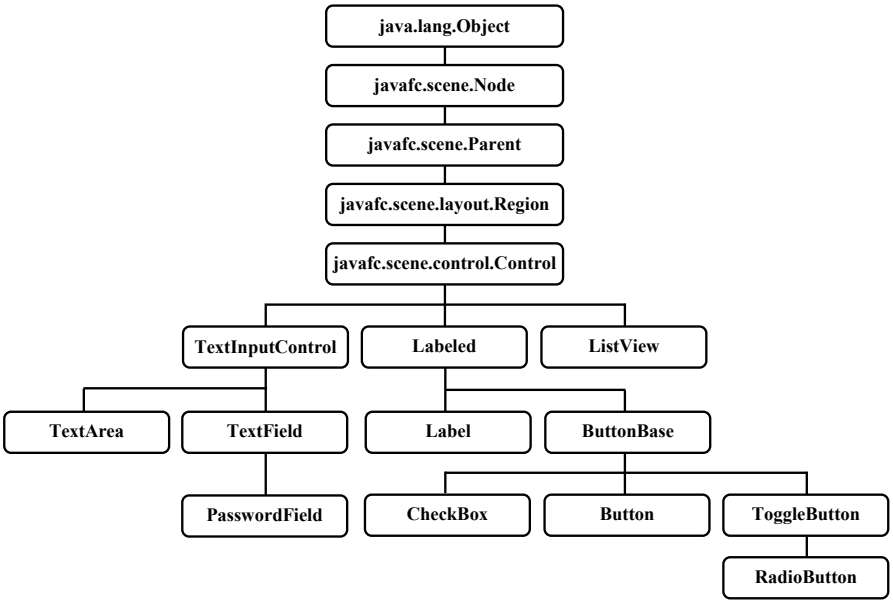
```
import javafx.beans.Observable;
import javafx.beans.property.*;
import javafx.collections.*;

public class ObservableListUpdates {
 public static void main(String[] args) {
  final 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());
     }
    }
   }
  });

  final StringProperty spa = new SimpleStringProperty("A");
  final 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
```

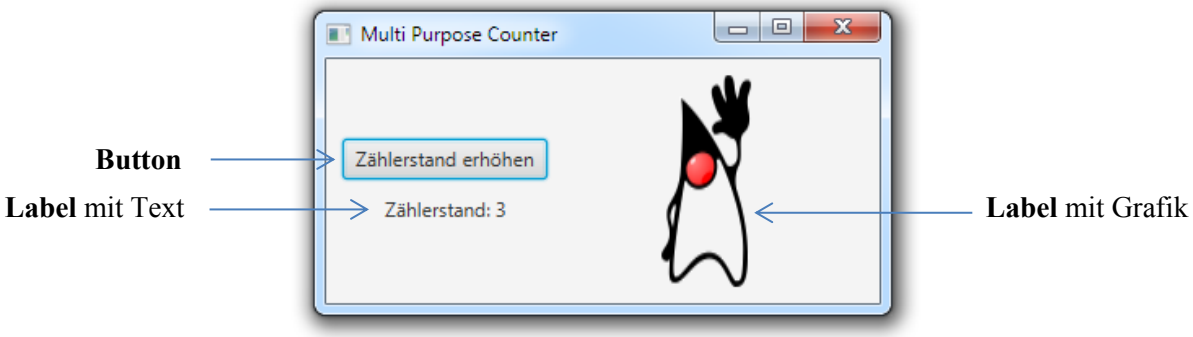
13.5 Einige Details zu elementaren Steuerelementen

Aus der stattlichen Sammlung von JavaFX-Steuerelementen werden im Manuskript einige besonders häufig benötigte Exemplare näher beschrieben. Deren Einordnung in der Klassenhierarchie zeigt die folgende Abbildung:



13.5.1 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:

```
final String lblPrefix = "Zählerstand: ";
final 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:

```
final 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"));
final ImageView imageView = new ImageView(icons[0]);
final 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]);
```

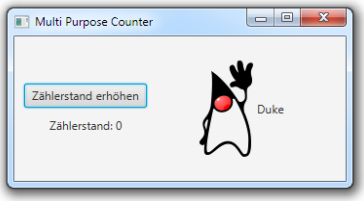
Neben **Label**-Objekten lassen sich auch diverse andere JavaFX-Komponenten mit Bildern verschönern (z.B. Befehlsschalter).

Text und Grafik können auch gemeinsam auftreten, wobei die Eigenschaft **ContentDisplay** über die relative Anordnung von Text und Grafik entscheidet. Im folgenden Codesegment

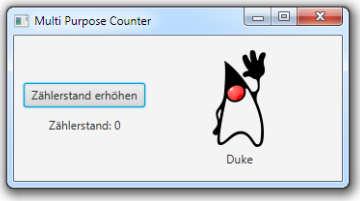
```
final Label lblIcon = new Label("Duke");
lblIcon.setGraphic(imageView);
lblIcon.setContentDisplay(ContentDisplay.TOP);
```

sorgt ein Aufruf der Setter-Methode **setContentDisplay()** mit dem Parameterwert **ContentDisplay.TOP** dafür, dass die Grafik *über* dem Text erscheint. Anschließend sind einige Anordnungen zu sehen:

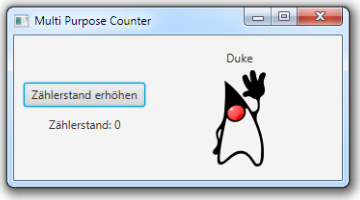
ContentDisplay.LEFT
(= Voreinstellung)



ContentDisplay.TOP



ContentDisplay.BOTTOM



Mit der Methode

public final void setGraphicTextGap(double *value***)**

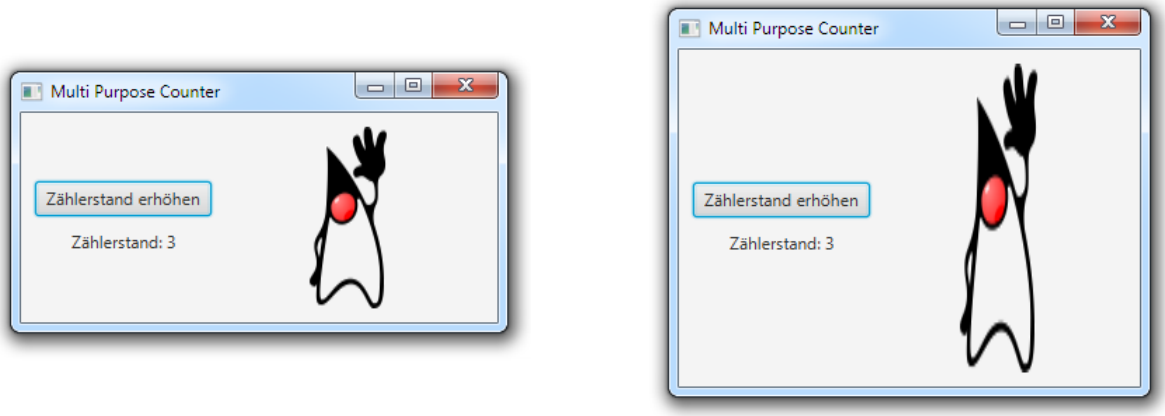
lässt sich der Abstand zwischen Text und Grafik verändern (Voreinstellung: 4), z.B.:

```
lblIcon.setGraphicTextGap(10);
```

Mit der in Abschnitt 13.4.2.1 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 13.4.2.2.1) dafür, dass der Innenrand des Containers berücksichtigt wird. Es resultiert eine Grafik mit dynamisch angepasster Höhe:



13.5.2 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:

```
final Button btnAdd = new Button("Zählerstand erhöhen");
```

Das im Abschnitt 13.5.1 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:

```
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:

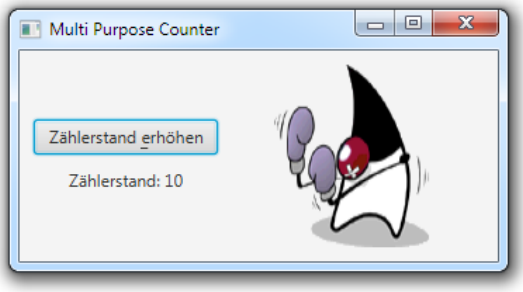
- Für das auf den ersten Unterstrich folgende Zeichen wird eine **Alt**-Tastenkombination festgelegt.
- Der erste Unterstrich wird im Programm nicht angezeigt.

Ist eine **Alt**-Tastenkombination vereinbart, wird im Programmablauf nach Betätigung der **Alt**-Taste das auslösende Zeichen unterstrichen. Solange dieser Zustand nicht eine weitere Betätigung der **Alt**-Taste aufgehoben wird, hat die Eingabe des auslösenden Zeichens denselben Effekt wie ein Mausklick auf das Steuerelement.

Im folgenden Code-Segment

```
final Button btnAdd = new Button("Zählerstand _erhöhen");
btnAdd.setMnemonicParsing(true);
```

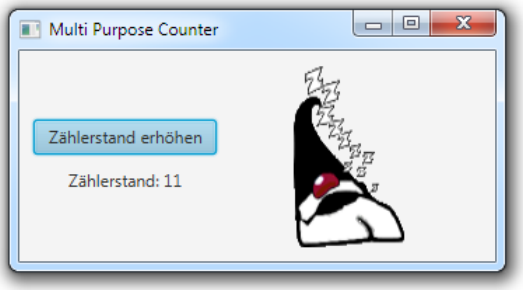
wird für einen Befehlsschalter mit Unterstrich im Beschriftungstext die (per Voreinstellung inaktive) Kurzwahldefinition aktiviert:



Für *einen* Schalter pro Fenster kann man zur weiteren Bedienungserleichterung das Auslösen per **Enter**-Taste ermöglichen, indem man ihn durch einen Aufruf der **setDefaultButton()** zum **Default Button** ernennt, z.B.:

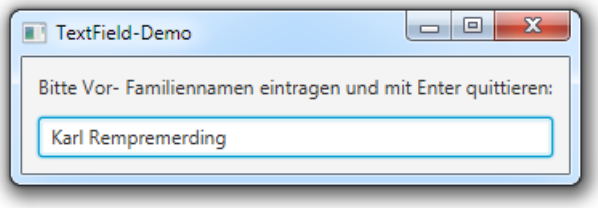
```
btnAdd.setDefaultButton(true);
```

Während (bei Verwendung des voreingestellten JavaFX-Designs) der aktiv gewählte Schalter an einem glimmenden Rand zu erkennen ist (siehe oben), strahlt dieses Glimmen beim Default Button in den Innenraum:

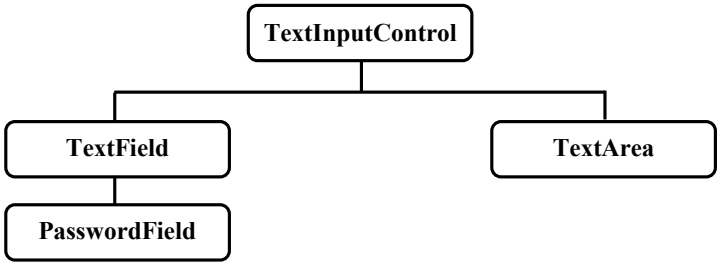


13.5.3 Einzeiliges Texteingabefeld

Um in einem JavaFX-Fenster eine einzelne Zeile mit purem Text zu erfassen, verwendet man die Klasse **TextField** , z.B.:

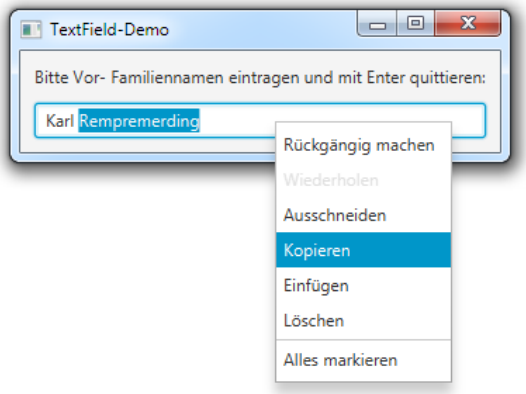


Sie stammt wie die (im Manuskript nicht behandelte) Klasse **TextArea** zur Erfassung von mehrzeiligem Text von der Klasse **TextInputControl** ab:



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.:

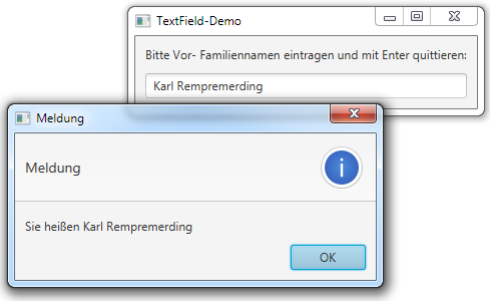


Der Quellcode des Beispielprogramms:

```
import javafx.application.Application;
 . . .
import javafx.stage.Stage;

public class TextFieldDemo extends Application {
 @Override
 public void start(Stage stage) {
  final int DIST = 10;
  final VBox root = new VBox(DIST);
  root.setPadding(new Insets(DIST, DIST, DIST, DIST));
  final Label label = new Label("Bitte Namen eintragen und mit Enter quittieren:");

  final 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();
       });

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("TextField-Demo");
    stage.show();
  }

  public static void main(String[] args) {
    launch(args);
  }
}
```

Das Tabulatorzeichen wird von einer **TextField**-Komponente *nicht* entgegengenommen. Drückt der Benutzer die **Enter**-Taste, während eine **TextField**-Komponente den Tastaturfokus besitzt, 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:



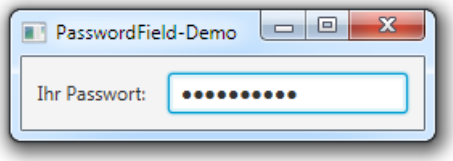
Statt der voreingestellten und im Beispiel angemessenen linksbündigen Ausrichtung des Textfeldinhalts kann mit der **TextField**-Methode **setAlignment()** unter Verwendung eines Parameterobjekts vom Aufzählungstyp **Pos** (aus dem Paket **javafx.geometry**) auch eine zentrierte oder rechtsbündige Ausrichtung gewählt werden, z.B.:

```
anzahl.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.:

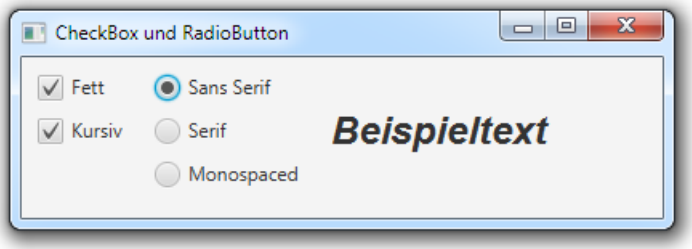


13.5.4 Umschalter

In diesem Abschnitt werden zwei Klassen für Umschalter vorgestellt, die beide von der Basisklasse **ToggleButton** abstammen:

- Für **Kontrollkästchen** steht die Klasse **CheckBox** zur Verfügung.
- Für ein **Optionsfeld** verwendet man Komponenten vom Typ **RadioButton**.

In folgendem Programm kann für den Text einer **Label**-Komponente über zwei Kontrollkästchen der Schriftschnitt und über ein Optionsfeld die Schriftart gewählt werden:



Den Quellcode des Programms finden Sie im Ordner

…\BspUeb\JavaFX\Steuerelemente\Umschalter

13.5.4.1 Kontrollkästchen

Im Beispielprogramm erhalten die beiden **CheckBox**-Komponenten per Konstruktor eine Beschriftung:

```
private final 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, so dass 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:

```
final GridPane root = new GridPane();
final VBox vboxCheck = new VBox(DIST);
root.add(vboxCheck, 0, 0);
vboxCheck.getChildren().add(cbBold);
vboxCheck.getChildren().add(cbItalic);
```

Bei den **Selected**-Eigenschaften der beiden **CheckBox**-Objekte registrieren wir einen per Methodenreferenz (siehe Abschnitt 12.1.2.1) 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, SIZE));
}
```

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** *posture*
  Die Werte im Enumerationstyp **FontWeight** stehen für aufsteigend geordnete Schwärzungsgrade: **THIN**, **EXTRA_LIGHT**, **LIGHT**, **NORMAL**, **MEDIUM**, **SEMI_BOLD**, **BOLD**, **EXTRA_BOLD**, **BLACK**.
- **javafx.scene.text.FontPosture** *weight*
  Die Werte im Enumerationstyp **FontWeight** sind **REGULAR** und **ITALIC**.
- **double** *size*
  Die Schriftgröße wird in der Einheit Punkt (= 1/72 Zoll) angegeben.

13.5.4.2 Optionsschalter

Die drei **RadioButton**-Objekte des Umschalter-Beispielprogramms erhalten per Konstruktor eine Beschriftung (siehe Einstieg von Abschnitt 13.5.4):

```
private final 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 einzeiligen **GridPane**- Root-Containers befindet:

```
final GridPane root = new GridPane();
final VBox vboxRadio = new VBox(DIST);
root.add(vboxRadio, 1, 0);
vboxRadio.getChildren().add(rbSans);
vboxRadio.getChildren().add(rbSerif);
vboxRadio.getChildren().add(rbMono);
```

Damit von den drei **RadioButton**-Objekten maximal eines ausgewählt sein kann, werden sie einem Objekt aus der Klasse **ToggleGroup** zugeordnet:

```
final ToggleGroup rbGroup = new ToggleGroup();
  .  .  .
rbGroup.getToggles().addAll(rbSans, rbSerif, rbMono);
rbSans.setSelected(true);
```

Über die Methode **getToggles()** erhält man eine beobachtbare Liste (vgl. Abschnitt 13.4.2.3), in die per **addAll()** die Optionsschalter aufgenommen werden. Mit der **RadioButton**-Methode **setSelected()** wird im Beispielprogramm dafür gesorgt, dass beim Programmstart der Optionsschalter zur schnörkellosen Schriftart ausgewählt ist.

Über das ausgewählte Element informiert ein **ToggleGroup**-Objekt über seine Eigenschaft **selectedToggle**, bei der wir einen per Methodenreferenz realisierten Change-Listener registrieren:

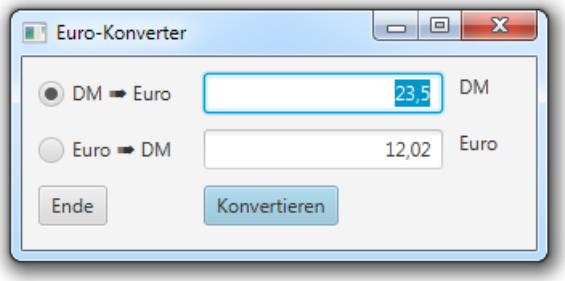
```
rbGroup.selectedToggleProperty().addListener(this::rbChanged);
```

In der realisierenden Methode `rbChanged()` wird passend zum ausgewählten Element die Schriftfamilie des **Label**-Objekts neu festgelegt:

```
private void rbChanged(ObservableValue<? extends Toggle> obs,Toggle old,Toggle nev) {
 final 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, fontPposture, SIZE));
}
```

13.5.5 Standardschaltfläche und Tastaturfokus

In diesem Abschnitt werden Techniken zur Verwaltung von Tastaturereignissen behandelt, die von Bedeutung für die Bedienbarkeit eines Programms sind. Als Beispiel dient ein Programm zur Währungsumrechnung, das Sie als Übungsaufgabe erstellen sollen (siehe Abschnitt 13.6):



Für den mit **Konvertieren** beschrifteten Befehlsschalter (mit dem Referenzvariablennamen `cbKonvertieren`) sollte mit der schon in Abschnitt 13.5.2 vorgestellten Methode **setDefaultButton()** festgelegt werden, dass sich die **Enter**-Taste bei aktivem Anwendungsfenster an ihn richten und (wie ein linker Mausklick auf die Schaltfläche) ein **ActionEvent** auslösen soll:

```
cbKonvertieren.setDefaultButton(true);
```

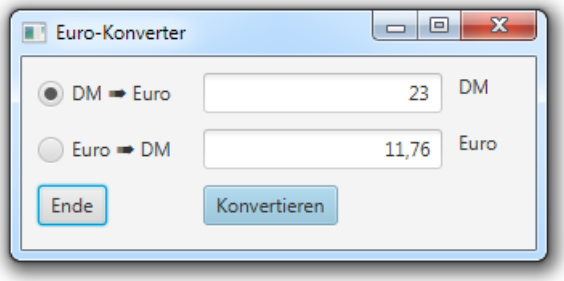
Der Benutzer kann die **Standardschaltfläche** (bei Verwendung des voreingestellten JavaFX-Designs) an der vollflächig aufgebrachten Hervorhebungsfarbe erkennen und manchen Griff zur Maus einsparen, was die Bedienung des Programms erleichtert.

Damit der Benutzer nach dem Programmstart sofort den ersten zu konvertierenden DM-Betrag eingeben kann, sollte die obere **TextField**-Komponente (mit dem Referenzvariablennamen `tfEingabe`) über die **Node**-Methode **requestFocus()** den **Tastaturfokus** anfordern. Diese Anforderung muss erfolgen, *nachdem* die Szene erstellt worden, z.B.:

```
final Scene scene = new Scene(root, 320, 120);
tfEingabe.requestFocus();
stage.setScene(scene);
```

Für eine Komponente den Tastaturfokus zu beanspruchen, kann auch im laufenden Programm sinnvoll sein. Ein Textfeld mit Tastaturfokus ist für den Benutzer an der dort vorhandenen Einfügemarke und an der hervorgehobenen Umrahmung zu erkennen.

Viele primär zur Mausbedienung konzipierte Steuerelemente (z.B. Befehlsschalter, Kontrollkästchen) können auch per Tastatur benutzt werden, wenn sie den Tastaturfokus besitzen. In der Regel ist dieser Zustand optisch gekennzeichnet, z.B. durch einen Zusatzrahmen. Im folgenden Zustand



des aktuellen Beispielprogramms ...

- hat der Schalter mit der Beschriftung **Ende** den Tastaturfokus und reagiert daher auf die **Leertaste**,
- während die mit **Konvertieren** beschriftete Standardschaltfläche auf die **Enter**-Taste reagiert.

Um den Tastaturfokus auf ein geeignetes Steuerelement zu übertragen, können die Benutzer einen Mausklick darauf setzen, oder den Fokus per Tastatur in der Fokussequenz wandern lassen:

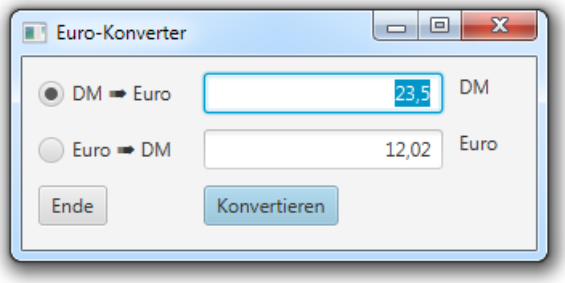
- Die Tabulatortaste ⇆ bewegt den Fokus zum nächsten Bedienelement in der Sequenz.
- Die Tastenkombination ⇧ + ⇆ bewegt den Fokus zum vorherigen Bedienelement in der Sequenz.

Ist für eine Komponente (aktuell) der Tastaturfokus nicht sinnvoll, kann sie mit der **Node**-Methode **setFocusable()** aus der Fokussequenz herausgenommen werden, z.B.:

```
cbBeenden.setFocusTraversable(false);
```

13.6 Übungsaufgaben zu Kapitel 13

1) Erstellen Sie den in Abschnitt 13.5.5 als Beispiel verwendeten Euro-DM-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 Escape-Sequenz lassen sich beliebige Unicode-Zeichen in einem Java-Programm verwenden, z.B.:

```
dm2euro = new RadioButton("DM "+'\u27a0'+" Euro");
```

14 GUI-Programmierung mit Swing

Zwar sollte für neue Projekte der aktuelle GUI-Standard JavaFX (siehe Kapitel 13) gegenüber der traditionellen Swing-Technik bevorzugt werden, doch existieren noch sehr viele Anwendungen mit Swing-GUI, die weiterhin gepflegt werden müssen, und Swing wird auf absehbare Zeit Bestandteil von Java SE bleiben. Daher kann es für angehende Java-Entwickler durchaus sinnvoll sein, sich mit Swing zu beschäftigen.

Die ursprüngliche, als ***A**bstract **W**indowing **T**oolkit* (AWT) bezeichnete GUI-Technologie wurde schon in Java 1.2 durch das *Swing Toolkit* erweitert und teilweise ersetzt. Es folgt eine kurze Gegenüberstellung der beiden Lösungen:

- **Abstract Windowing Toolkit** (AWT, enthalten im Paket **java.awt)**
  Das bereits in Java 1.0 vorhandene AWT ist zwar mittlerweile überholt, stellt aber immer noch einige Basisklassen für die Swing-Technologie zur Verfügung. Grundidee beim AWT-Entwurf war die möglichst weitgehende Verwendung von Steuerelementen des *Wirtsbetriebssystems*. Die an Komponenten des Wirtsbetriebssystem gekoppelten AWT-Bedienelemente werden als *schwergewichtig* bezeichnet. Aus der notwendigen Beschränkung auf den damals recht kleinen gemeinsamen Nenner der zu unterstützenden Plattformen resultierte ein beschränkter AWT-Funktionsumfang. Im aktuellen Kapitel werden aus dem AWT nur die nach wie vor relevanten Basisklassen berücksichtigt. Wer die AWT-Steuerelemente verwenden möchte, kann sich z.B. in Kröckertskothen (2001, Kap. 13) informieren.
- **Swing** (enthalten im Paket **javax.swing**)
  Mit Java 1.2 wurden die komplett in Java realisierten *leichtgewichtigen* Komponenten eingeführt. Während die *Top-Level-Fenster* nach wie vor schwergewichtig sind und die Verbindung zum Grafiksystem des Wirtsbetriebssystems herstellen, werden die Steuerelemente komplett von Java verwaltet und gezeichnet, was einige Vorteile bringt:
  - Weil die Beschränkung auf den kleinsten gemeinsamen Nenner verschiedener Betriebssysteme entfällt, stehen **mehr Komponenten** zur Verfügung.
  - Java-Anwendungen können auf allen Betriebssystemen ein **einheitliches Erscheinungsbild** bieten, müssen es aber nicht, denn:
  - Für die Swing-Komponenten kann (sogar vom Benutzer zur Laufzeit) ein **Look & Feel** gewählt werden (siehe Abschnitt 14.7.1 zu den verfügbaren Alternativen), während die AWT-Komponenten auf das GUI-Design des Betriebssystems festgelegt sind.

Swing hat viele weitere gute Eigenschaften, z.B.:

- Die Java-Entwicklungsumgebungen enthalten in der Regel einen grafischen GUI-Designer mit guter Swing-Unterstützung (z.B. Eclipse mit dem WindowBuilder).
- Durch Doppelpufferung wird ein Flackern beim Aufbau von komplexen Fensterinhalten verhindert. Swing-Fenster werden per Voreinstellung im Hintergrund aufgebaut und erst im fertigen Zustand angezeigt.
- QuickInfo-Fenster (Tool-Tipps)
- Gute Unterstützung für die Steuerung per Tastatur und für die Anpassung an verschiedene Sprachen und Konventionen (Lokalisierung)

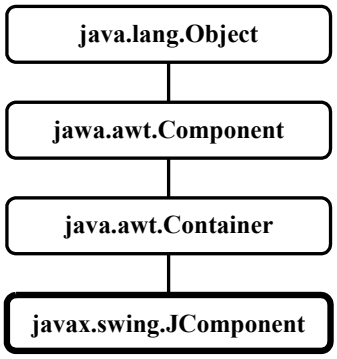
Den enorm wichtigen Multithreading-Bezügen von Swing gehen wir im aktuellen Kapitel aus dem Weg, holen deren Behandlung aber später nach (siehe Abschnitt 16.9).

14.1 Swing im Überblick

Wer nach der Lektüre dieses Kapitels weitere Informationen zur GUI-Entwicklung mit Swing benötigt, sollte neben der API-Dokumentation das Java-Tutorial (Oracle 2015) konsultieren.[1]

14.1.1 Komponenten

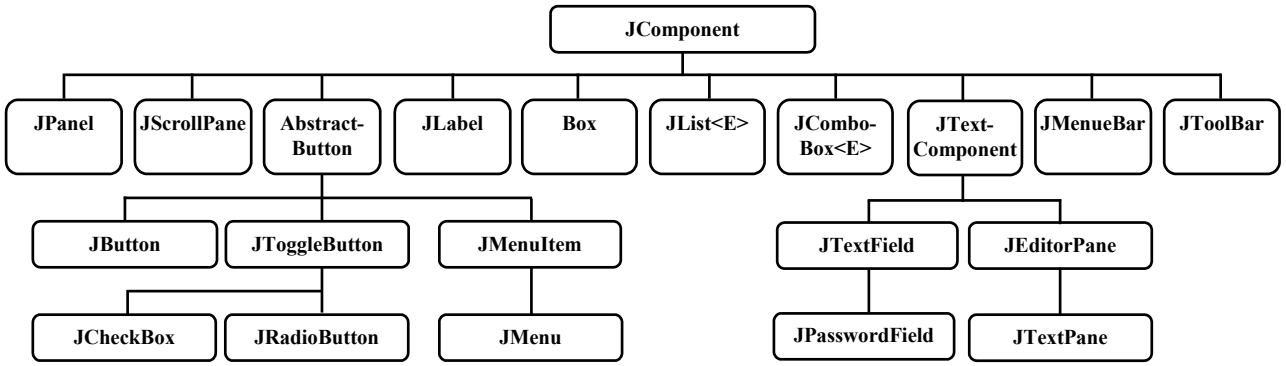
Die Swing-Komponenten stammen meist von der Klasse **javax.swing.JComponent** ab, die wiederum zahlreiche Handlungskompetenzen und Eigenschaften über folgende Ahnenreihe von AWT-Klassen erwirbt:



Komponenten sind natürlich auch Objekte, allerdings mit einigen zusätzlichen Kompetenzen:

- Visuelle Komponenten treten auf dem Bildschirm in Erscheinung (machen Grafikausgaben) und kommunizieren mit dem Benutzer.
- Sie empfangen elementare GUI-Ereignisse (z.B. Mausklicks, Tastendrücke) und bieten ihrerseits aufbereitete, abstraktere Ereignisse an (z.B. Änderung einer Listenzusammenstellung), über die sich andere Objekte informieren lassen können (siehe Abschnitt 14.5).
- Ihr Einsatz kann von Entwicklungsumgebungen durch GUI-Designer unterstützt werden. Durch die systematische Verfügbarkeit von Methoden zum Lesen und Setzen von Eigenschaften und die Einhaltung bestimmter Benennungsregeln kann ein GUI-Designer zu einer Komponente z.B. eine Tabelle mit Eigenschaftsausprägungen zur Entwurfszeit anbieten.

In der folgenden Abbildung sehen Sie die Abstammungsverhältnisse für etliche im Kapitel 14 behandelten **JComponent**-Abkömmlinge:



Die Klasse **JTextComponent** und ihre Erweiterungen befinden sich im Paket **javax.swing.text**, alle anderen Klassen befinden sich im Paket **javax.swing**. Wie an den Namen unschwer zu erkennen ist, stehen die meisten Klassen für Bedienelemente, die aus GUI-Systemen wohlbekannt sind (Befehlsschalter, Label etc.).

---

[1] Siehe: http://docs.oracle.com/javase/tutorial/uiswing/index.html

In der Sammlung befindet sich mit **JPanel** auch ein *Container*, der zur Aufnahme und damit zur Gruppierung von anderen Swing-Komponenten dient. Im Sinne einer flexiblen GUI-Gestaltung bietet Java die Möglichkeit, in einem Container neben „atomaren" Komponenten (z.B. **JButton**, **JLabel**) auch untergeordnete Container (in beliebiger Schachtelungstiefe) unterzubringen. Weil **JComponent** von **Container** abstammt, kann grundsätzlich jede **JComponent**-Spezialisierung andere Komponenten aufnehmen. Man sollte aber in Swing-Anwendungen neben den Hauptfenstern (Top-Level - Containern, siehe Abschnitt 14.1.2) nur **JPanel** (und daraus abgeleitete Klassen) als Container einsetzen (siehe Abschnitt 14.3.3).

Die Frage, welcher von den beiden Begriffen *Component* und *Container* in Javas GUI-Technologie dem anderen vorgeordnet ist, kann aufgrund des Klassenstammbaums nicht geklärt werden:

- **java.awt.Container** stammt von **java.awt.Component** ab.
- **javax.swing.JComponent** stammt von **java.awt.Container** ab.

Relevant sind aber letztlich nicht die (teilweise historisch bedingten) Namen, sondern die Methoden und Eigenschaften, die eine Klasse von ihren Vorfahren übernimmt.

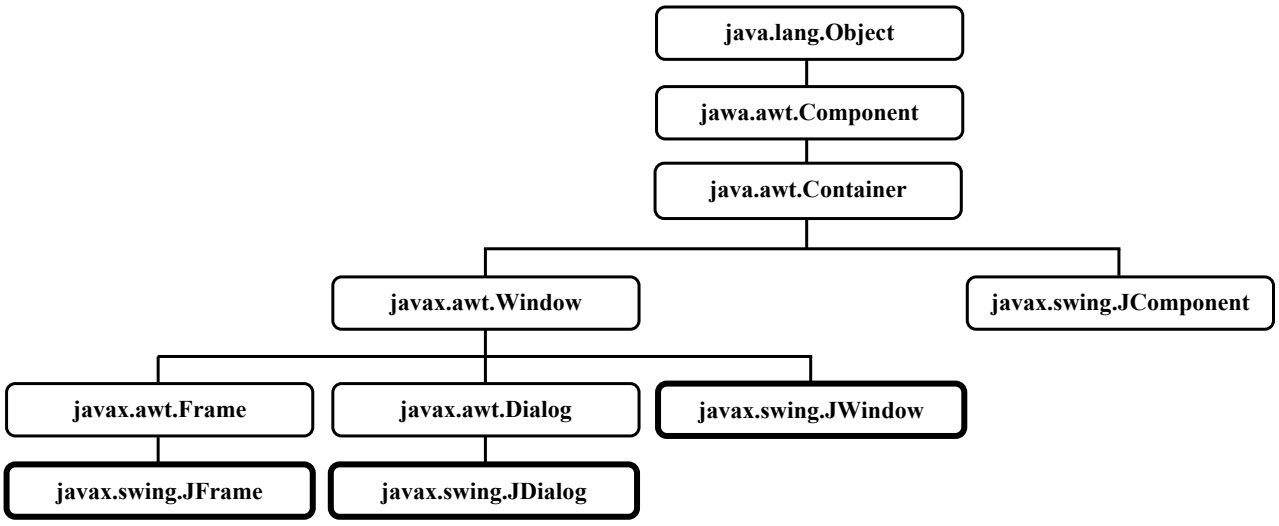
Aus Zeitgründen können leider viele attraktive Swing-Komponenten im Manuskript nicht behandelt werden, z.B.:

- **JTree**, **JTable**
- **JSplitPane**, **JTabbedPane**
- **JPopupMenu**

Gute Darstellungen findet man z.B. bei Krüger & Hansen (2011, 2014), Ullenboom (2012a) und im Java Tutorial (Oracle 2014).

14.1.2 Top-Level - Container

Jedes Programm mit Swing-GUI benötigt mindestens einen Top-Level - Container, der die Verbindung zu der vom Betriebssystem verwalteten Bedienoberfläche herstellt und die leichtgewichtigen Swing-Komponenten aufnimmt. Die Top-Level - Container im Paket **javax.swing** stammen nicht von **JComponent** ab, sondern haben einen etwas anderen Stammbaum:



14.1.2.1 Sorten

In Kapitel 14 werden alle Beispielprogramme als Top-Level - Container eine Komponente vom Typ **JFrame** benutzen, die ein **Rahmenfenster** mit folgender Ausstattung realisiert:

- **Rahmen**
  Wenn die Größe des Fensters nicht fixiert ist, kann sie über den Rahmen geändert werden.
- **Titelzeile**
  Sie enthält:
  - Fenstertitel
  - Bedienelemente zum Schließen, Minimieren und Maximieren des Fensters
  - Systemmenü (über die Java-Tasse am linken Rand der Titelzeile erreichbar)

Mit der Klasse **JDialog** werden *Dialogfenster* realisiert, denen im Vergleich zum Rahmenfenster die Titelzeilenbedienelemente zum Maximieren und zum Minimieren fehlen. Sie können zwar auch als selbständige Fenster einer Anwendung arbeiten, werden aber meist in Rahmenfenster-Anwendungen für bestimmte, meist temporäre Kommunikationsaufgaben benutzt.

Bei den mit der Klasse **JWindow** erzeugten Fenstern bestehen im Vergleich zu den Rahmenfenstern folgende Einschränkungen:

- kein Rahmen
- keine Titelzeile, also auch keine Bedienelemente zum Minimieren, Maximieren und Schließen sowie kein Systemmenü
- Benutzer können die Größe und die Position des Fensters nicht ändern.

Die Klasse **JWindow** ist also nur für spezielle Zwecke geeignet (z.B. Begrüßungsfenster, engl.: *splash screen*) und wird im Manuskript keine Rolle spielen.

14.1.2.2 Schichtaufbau

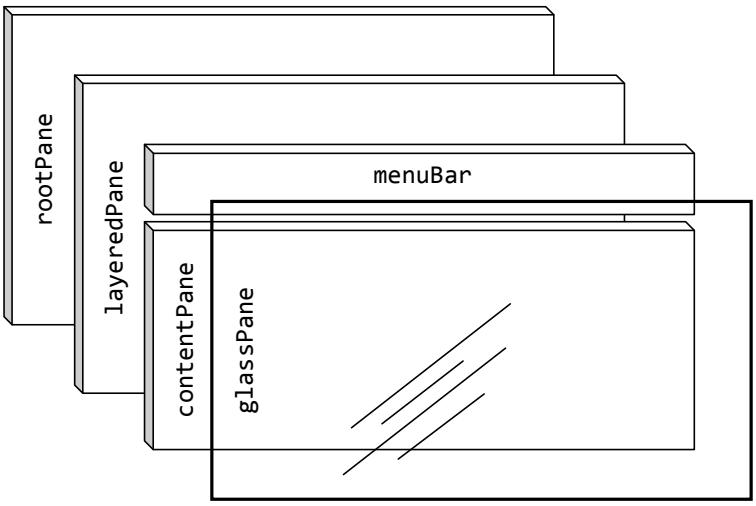
Swings Top-Level - Container enthalten ein Mitgliedsobjekt aus der Klasse **JRootPane**, das für die Verwaltung der Fensterbestandteile zuständig ist. Für einige Verabredungen (z.B. Auswahl einer voreingestellten Schaltfläche, welche auf die **Enter**-Taste wie auf einen Mausklick reagieren soll) werden wir dieses Objekt direkt ansprechen. Für die meisten Aufgaben bei der Fensterverwaltung sind Mitgliedsobjekte der Klasse **JRootPane** zuständig:

- Ein Objekt aus einer von **Container** abgeleiteten Klasse nimmt die Bedienelemente mit Ausnahme der Menüzeile auf.[1] Mit dieser so genannten **Inhaltsschicht** (engl.: *content pane*) werden wir häufig direkt kommunizieren.
- Für die optionale **Menüzeile** ist ein Objekt aus der Klasse **JMenuBar** zuständig. Es fehlt aus naheliegenden Gründen bei einem Top-Level - Container vom Typ **JWindow**.
- Für die Positionierung von Inhaltsschicht und Menüzeile sorgt ein Objekt aus der Klasse **JLayeredPane**. Durch seine Fähigkeit zur vertikalen Anordnung kann das Objekt das Menü vor anderen Bedienelementen erscheinen zu lassen.
- Ein Objekt aus einer von **Component** abgeleiteten Klasse schwebt als **Glasscheibe** (engl. *glass pane*) über der Inhaltsschicht und der optionalen Menüzeile.[1] Diese Glasscheibe er-

[1] Wie eine Überprüfung mit der **Object**-Methode **getClass()** zeigt, werden bei einem Top-Level - Container vom Typ **JFrame** sowohl die Inhaltsschicht als auch die Glasscheibe durch ein Objekt vom Typ **JPanel** realisiert.

möglicht das Übermalen von mehreren Komponenten sowie das Abfangen von GUI-Ereignissen. Hier landen z.B. Tool-Tip - Texte (vgl. Abschnitt 14.3.5.1).

Das folgende, aus dem Java-Tutorial (Oracle 2014) übernommene Schichtenmodell soll die Aufgaben der angesprochenen Komponenten eines Top-Level – Containers vom Typ **JFrame** oder **JDialog** illustrieren:



14.2 Beispiel für eine Swing-Anwendung

In folgendem Swing-Programm, das z.B. zur Verkehrszählung taugt, kommen zwei Label (mit einem Text bzw. einem Bild als Inhalt) sowie ein Befehlsschalter zum Einsatz:

14.2.1 Quellcode und erste Erläuterungen

Den folgenden Quellcode des Programms werden wir im weiteren Verlauf von Kapitel 14 vollständig besprechen:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class MPC {
  private final static String TITLE = "Multi Purpose Counter";
  private int numClicks = 0;
  private int iconInd = 0;
```

```
MPC() {
  // Über den JFrame-Konstruktor erhält das Rahmenfenster einen Titel.
  final JFrame frame = new JFrame("Multi Purpose Counter");

  // cont ist eine bequeme Referenz auf die Inhaltsschicht des JFrame-Fensters
  final Container cont = frame.getContentPane();

  // JLabel-Komponente mit Text initialisieren
  final String lblPrefix = "Zählerstand: ";
  final JLabel lblText = new JLabel(lblPrefix + "0");
  lblText.setToolTipText("Label mit Text");

  // Icons erzeugen
  final ImageIcon[] icons = new ImageIcon[3];
  icons[0] = new ImageIcon("duke.gif");
  icons[1] = new ImageIcon("fight.gif");
  icons[2] = new ImageIcon("snooze.gif");

  // JLabel mit Icon initialisieren
  final JLabel lblIcon = new JLabel(icons[0]);
  lblIcon.setToolTipText("Label mit Icon");

  // JButton-Komponente initialisieren
  final JButton cbCount = new JButton("Zählerstand erhöhen");
  cbCount.setMnemonic(KeyEvent.VK_A);

  // Der Schalter kann als Default Button auch mit Enter ausgelöst werden.
  frame.getRootPane().setDefaultButton(cbCount);

  // Quick-Info zum Befehlsschalter
  cbCount.setToolTipText("Befehlsschalter");

  // JPanel-Komponente erzeugen.
  // Dieser "immediate container" dient zur Positionierung von Komponenten.
  final JPanel panLeft = new JPanel();

  // Der Container wird "frei gestellt".
  panLeft.setBorder(BorderFactory.createEmptyBorder(30, 30, 30, 30));

  // Layout-Manager für die JPanel-Komponente erzeugen
  // Hier wird dafür gesorgt, dass alle Komponenten in einer Spalte stehen.
  panLeft.setLayout(new GridLayout(0, 1));

  // Der JButton und das Text-JLabel werden auf die JPanel-Komponente gesetzt.
  panLeft.add(cbCount);
  panLeft.add(lblText);

  // Der JPanel-Container wird auf den JFrame gesetzt (Position: Mitte).
  cont.add(panLeft, BorderLayout.CENTER);

  // Das Icon-Label wird auf den JFrame gesetzt (Position: Osten, also rechter Rand).
  cont.add(lblIcon, BorderLayout.EAST);
```

```
    // Für den JButton wird ein ActionListener registriert.
    // Dazu wird eine anonyme Klasse an Ort und Stelle definiert.
    cbCount.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
      numClicks++;
      lblText.setText(lblPrefix + numClicks);
      if (iconInd < icons.length-1)
       iconInd++;
      else
       iconInd = 0;
      lblIcon.setIcon(icons[iconInd]);
     }
    });

    // Der JFrame erhält einen Event Handler für Fenster-Ereignisse.
    frame.addWindowListener(new WiLi());

    // Diese Methode verhindert ein Schließen unter Missachtung des WindowListeners
    frame.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);

    // initiale JFrame-Größe festlegen
    frame.setSize(340, 150);

    // JFrame anzeigen
    frame.setVisible(true);
  }

  public static void main(String[] args) {
   new MPC();
  }

  // Der WindowListener für den JFrame (= Top Level Container) wird als innere Klasse def.
  private class WiLi extends WindowAdapter {
   @Override
   public void windowClosing(WindowEvent e) {
    if (JOptionPane.showConfirmDialog(e.getWindow(),
        "Wollen Sie nach "+ numClicks +
        " Klicks wirklich schon aufhören?",
        TITLE, JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION)
      System.exit(0);
   }
  }
}
```

Zu Beginn werden alle Klassen aus den Paketen **javax.swing**, **java.awt** und **java.awt.event** importiert, um sie bequem ansprechen zu können.

Das Programm besteht aus der startfähigen Klasse `MPC`. In ihrer **main()**-Methode wird ein Objekt dieser Klasse erzeugt:

```
public static void main(String[] args) {
  new MPC();
}
```

Die einzige und dazu noch sehr kurze Anweisung der **main**-Methode sollte ein doppeltes Erstaunen auslösen:

- Ein Ausdruck mit **new**-Operator und zugehörigem Konstruktoraufruf ist tatsächlich eine vollständige Anweisung, wenn ein Semikolon dahinter gesetzt wird. Sie gehört zu den Ausdrucksanweisungen (siehe Gosling et al. 2014, Abschnitt 148).
- Bisher wurde diese Anweisungsvariante nicht erwähnt, weil ihre Nützlichkeit kaum zu vermitteln gewesen wäre. Das im Konstruktor erzeugte `MPC`-Objekt bleibt nach Beendigung des Konstruktors aktiv und nützlich, weil bei seiner Kreation zusätzliche Threads zur Betreuung einer GUI-Anwendung entstanden sind, die eine Referenz auf das `MPC`-Objekt enthalten (siehe Abschnitt 14.5.5 über das Terminieren von GUI-Anwendungen).

Im Manuskript wurde mehrfach empfohlen, lokale Variablen und Instanzvariablen als **final** zu deklarieren, wenn laut Algorithmus keine Änderung vorgesehen ist (Abschnitte 3.3.10 und 4.2.5). Diese Empfehlung wurde aber in den Beispielprogrammen der Einfachheit halber selten umgesetzt. Im aktuellen Kapitel wird die Empfehlung beachtet in der Hoffnung, ...

- dass der syntaktische Zusatzaufwand nicht von den Lerninhalten ablenkt,
- und dass die Inkonsequenz beim Befolgen der **final**-Empfehlung nicht irritiert.

Das Hauptfenster der Klasse ist ein Objekt der Klasse **JFrame**. Weil dieses Rahmenfenster mehrere Komponenten enthält, hat der `MPC`-Konstruktor einige Arbeit:

- Das Fensterobjekt wird erzeugt und erhält seinen Titel per Konstruktorparameter:

  ```
  final JFrame frame = new JFrame(titel);
  ```

- Wie in Abschnitt 14.1.2.2 erläutert wurde, nimmt die Inhaltsschicht (engl.: *content pane*) des Rahmenfensters die Bedienelemente auf. Um bequem auf die Inhaltsschicht zugreifen zu können, wird mit der **JFrame**-Methode **getContentPane()** eine lokale Referenzvariable auf diesen Container angelegt:[1]

  ```
  final Container cont = frame.getContentPane();
  ```

  In der folgenden Anweisung wird die Inhaltsschicht per **add()** - Methodenaufruf gebeten, einen untergeordneten Container namens `panLeft` aus der Klasse **JPanel** (vgl. Abschnitt 14.3.3) aufzunehmen:

  ```
  cont.add(panLeft, BorderLayout.CENTER);
  ```

  Die Ortsangabe im zweiten **add()** - Parameter wird später im Zusammenhang mit dem **BorderLayout** erklärt. Seit Java 5 kann man den Aufnahmeantrag auch an das **JFrame**-Objekt richten

  ```
  frame.add(panLeft, BorderLayout.CENTER);
  ```

  und von der automatischen Weiterleitung an die Inhaltsschicht profitieren. Wir werden im Manuskript der Klarheit halber die letztlich zuständige Inhaltsschicht meist explizit ansprechen.

- Wie der Konstruktor die Bedienelemente des Rahmenfensters erzeugt, konfiguriert und positioniert sowie die Ereignisbehandlung vorbereitet, wird gleich im Detail erklärt.

---

[1] Die Methode **getContentPane()** hat den deklarierten Rückgabetyp **Container**, liefert aber de facto ein **JPanel**-Objekt. Solange die Kompetenzen der deklarierten Klasse **Container** genügen, sollte dieser generellere Typ verwendet und auf eine explizite Typumwandlung verzichtet werden.

- Am Ende seiner Tätigkeit legt der Konstruktor über die Methode **setSize()** der Klasse **java.awt.Window** noch eine initiale Größe für das Fenster fest und macht es dann mit der ebenfalls von **Window** geerbten Methode **setVisible()** sichtbar:

```
frame.setSize(340, 150);
frame.setVisible(true);
```

14.2.2 Thread-sichere GUI-Initialisierung

Wie in Kapitel 16 im Zusammenhang mit dem Thema *Multithreading* zu erfahren sein wird, ist der **setVisible(true)**-Aufruf am Ende des Konstruktors *nicht* Thread-sicher. Es könnte zu einer fehlerhaften Anzeige kommen, weil zwei Threads gleichzeitig auf das GUI zugreifen. Durch den **setVisible(true)** - Aufruf gelangen der Top-Level - Container und die enthaltenen Komponenten in den *realisierten* Zustand. Ab jetzt werden ihre Ereignisbehandlungs- oder Ausgabemethoden im **Ereignisverteilungs-Thread** aufgerufen, und eine Manipulation der Objekte aus einem anderen Thread muss unterbleiben. Weil das Risiko durch den **setVisible(true)**-Aufruf am Ende des Konstruktors gering ist, verzichten wir (in Übereinstimmung mit den meisten Lehrbüchern) vorläufig auf eine absolut wasserdichte, aber etwas aufwändigere Lösung. Nach dem **setVisible(true)**-Aufruf dürfen aber im Fensterkonstruktor, der im Haupt-Thread abläuft keine GUI-Manipulationen mehr vorgenommen werden.

So könnte eine risikofreie **main()**-Methode für unser Beispielprogramm aussehen:

```
public static void main(String[] args) {
 EventQueue.invokeLater(new Runnable() {
  public void run() {
   new MPC();
  }
 });
}
```

Bei der Kreation eines Bruchkürzungsprogramms mit Swing-GUI hat der WindowBuilder eine analoge **main()**-Methode für uns geschrieben (siehe Abschnitt 4.8).

Im weiteren Verlauf des Manuskripts wird Ihnen mindestens *ein* Programm begegnen, das tatsächlich mit einer fehlerhaften Anzeige reagiert, wenn der Fensterkonstruktor im Haupt-Thread ausgeführt wird (siehe z.B. Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).

14.2.3 Alternative Fensterkonstruktion

Auf vielen Webseiten, in vielen Lehrbüchern (z.B. Deitel & Deitel 2005; Krüger & Hansen 2014) und auch in der vorherigen Version dieses Manuskripts sind Swing-Beispielprogrammen mit einer alternativen Fensterkonstruktion zu finden. Dabei wird die Hauptklasse von **JFrame** abgeleitet, und das Anwendungsobjekt ist in Personalunion auch das Fensterobjekt des Programms, z.B.:

```
class MPC extends JFrame {
   . . .
}
```

Diese Vorgehensweise ist zwar zulässig, hat aber Nachteile;

- Man verschenkt die Möglichkeit, eine problemadäquate Basisklasse zu wählen.
- Die enorme Zahl von geerbten **JFrame**-Methoden kann zu Fehlern führen (z.B. versehentliches Überschreiben).
- Weil die Klasse **JFrame** das Interface **Serializable** implementiert, also das Serialisieren (siehe unten) ihrer Objekte erlaubt, sollte auch unserer **JFrame**-Ableitung eine statische Variable namens **serialVersionUID** mit einer Versionskennung definieren, damit es beim (De-)serialisieren nicht zu Fehlern aufgrund von Weiterentwicklungen der Klassendefinition kommen kann. Wenn eine solche Variable fehlt, warnt Eclipse:

  Wenn ein Serialisieren *nicht* in Frage kommt, kann die Warnung durch eine Marker-Annotation unterdrückt werden:

```
@SuppressWarnings("serial")
```

14.3 Bedienelemente (Teil 1)

Das Manuskript behandelt in zwei Abschnitten jeweils eine Zusammenstellung elementarer Bedienelemente.

14.3.1 Label

Mit Komponenten der Klasse **JLabel** realisiert man Bedienungshinweise in Schrift- und/oder Bildform. Das erste Label in unserem Beispielprogramm beschränkt sich auf eine Textanzeige:

```
final JLabel lblText = new JLabel(lblPrefix + "0");
```

Für Textänderungen im Programmablauf verwendet man die **JLabel**-Methode **setText()**, z.B.:

```
lblText.setText(lbPrefix + numClicks);
```

Zur Steuerung der horizontalen Textausrichtung innerhalb der vom **JLabel** belegten Rechteckfläche dient die Methode **setHorizontalAlignment()**, z.B.:[1]

```
status.setHorizontalAlignment(SwingConstants.CENTER);
```

Das zweite Label im Swing-Einstiegsbeispiel dient zur Anzeige von GIF-Dateien, die von **ImageIcon**-Objekten repräsentiert werden:

```
final ImageIcon[] icons = new ImageIcon[3];
icons[0] = new ImageIcon("duke.gif");
icons[1] = new ImageIcon("fight.gif");
icons[2] = new ImageIcon("snooze.gif");
```

Die **ImageIcon**-Objekte passen ganz gut in den Abschnitt über GUI-Design, doch soll der begrifflichen Klarheit halber darauf hingewiesen werden, dass es sich *nicht* um Komponenten handelt, weil sie z.B. keinerlei Ereignisse auslösen können. Neben dem GIF-Format (*Graphics Interchange Format*) werden auch die Formate JPEG (*Joint Photographic Experts Group*) und PNG (*Portable Network Graphics*) unterstützt.

Beim Erzeugen des zweiten Label-Objekts im Beispielprogramm wird ein **ImageIcon** als initiale Anzeige festgelegt:

```
final JLabel lblIcon = new JLabel(icons[0]);
```

---

[1] Die Anweisung stammt *nicht* aus dem Swing-Einstiegsbeispiel.

Um das Icon im Programmablauf auszutauschen, verwendet man die **JLabel**-Methode **setIcon()**, z.B.:

```
lblIcon.setIcon(icons[iconInd]);
```

Neben **JLabel**-Objekten lassen sich auch diverse andere Swing-Komponenten mit **ImageIcon**-Objekten verschönern (z.B. Befehlsschalter), wobei die Auswahl wiederum per Konstruktor oder per **setIcon()**-Methode erfolgt.

14.3.2 Befehlsschalter

Befehlsschalter werden in Swing durch die Klasse **JButton** realisiert. Die Syntax zum Deklarieren bzw. Erzeugen eines Schalters mit Beschriftung bietet keinerlei Überraschungen:

```
final JButton cbCount = new JButton("Zählerstand erhöhen");
```

Mit der **JButton**-Methode **setMnemonic()** kann eine **Alt**-Tastenkombination als Äquivalent zum Mausklick auf den Schalter festgelegt werden, z.B. **Alt+A**:

```
cbCount.setMnemonic(KeyEvent.VK_A);
```

Den **int**-wertigen Parameter der Methode **setMnemonic()** legt man am besten über die in der Klasse **KeyEvent** definierten VK-Konstanten (*Virtual Key*) fest.

Für *einen* Schalter pro Fenster kann man zur weiteren Bedienungserleichterung das Auslösen per **Enter**-Taste ermöglichen, indem man ihn zum **Default Button** ernennt. Dazu ist die Botschaft **setDefaultButton()** an das per **JFrame**-Methode **getRootPane()** zu ermittelnde **Root Pane** - Objekt des Fensters (vgl. Abschnitt 14.1.2.2) zu richten, z.B.:

```
frame.getRootPane().setDefaultButton(cbCount);
```

14.3.3 JPanel-Container

In einen Top-Level - Container (z.B. aus der Klasse **JFrame)** können neben normalen Steuerelementen auch untergeordnete Container aus der Klasse **JPanel**-Container eingefügt werden, die wiederum Komponenten und Container aufnehmen dürfen.[1] Für die Verteilung der in einem Container enthaltenen Komponenten auf die verfügbare Rechteckfläche ist der Layout-Manager des Containers verantwortlich (siehe Abschnitt 14.4). Mit Hilfe von verschachtelten **JPanel**-Containern mit passend konfigurierten Layout-Managern lässt sich eine flexible, problemadäquate Anordnung von Steuerelementen erreichen.

Im unserem Swing-Einstiegsbeispiel wird ein **JPanel**-Behälter namens `panLeft` verwendet, der für den linken Teil des Fensters zuständig sein soll:

```
final JPanel panLeft = new JPanel();
```

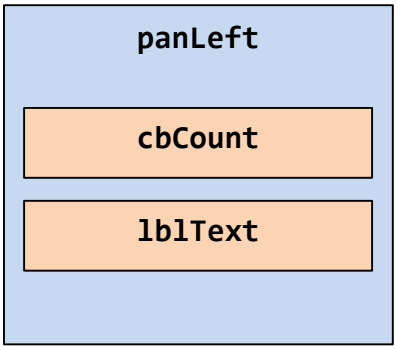
Um eine Komponente in einen Container aufzunehmen, benutzt man eine der zahlreichen **add()**-Überladungen. Durch die folgenden Anweisungen

```
panLeft.add(cbCount);
panLeft.add(lblText);
```

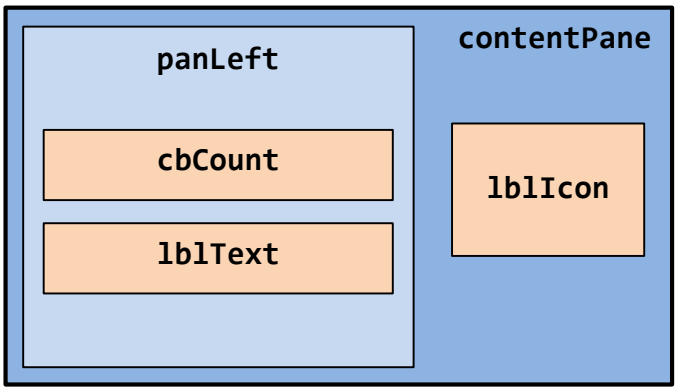
gelangen der Befehlsschalter und das Text-Label in den Container `panLeft`:

---

[1] Wie sich in Abschnitt 14.1.2 herausgestellt hat, ist auch die Inhaltsschicht eines **JFrame**-Fensters, die alle Swing-Komponenten mit Ausnahme der Menüzeile aufnimmt, Container aus der Klasse **JPanel**.



Wie dieser Container und das mit einem **ImageIcon** geschmückte **JLabel**-Objekt `lblIcon` auf die Inhaltsschicht des **JFrame**-Fensters gelangen, wird erst in Abschnitt 14.4 erklärt, weil dazu einiges Wissen um Layout-Manager erforderlich ist. Am Ende resultiert folgende Anordnung mit einem Top-Level - Container (**contentPane** des **JFrame**-Objekts), einem untergeordneten Container aus der Klasse **JPanel** und drei (atomaren) Komponenten:



Manche Layout-Manager ordnen die Komponenten in der Einfügereihenfolge horizontal oder vertikal an und beachten dabei (per Voreinstellung oder aufgrund einer speziellen Konfiguration) die **Komponenten-Orientierung** des Containers. Die Klasse **Container** und ihre Spezialisierungen besitzen ein Mitgliedsobjekt der Klasse **ComponentOrientation**, das per **set**-Methode festgelegt und per **get**-Methode ermittelt werden kann. Sollte es jemals erforderlich sein, die Sprachabhängige Voreinstellung zu ändern, kann dies also mit der **Component**-Methode **setComponentOrientation()** geschehen. Wenn nicht nur *ein* bestimmter Container betroffen sein soll, sondern die gesamte enthaltene Komponentenhierarchie, verwendet man die Methode **applyComponentOrientation()**.

14.3.4 Elementare Eigenschaften von Swing-Komponenten

Anschließend werden **JComponent**-Methoden beschrieben, die elementare Eigenschaften von Swing-Komponenten beeinflussen:

- **public void setAlignmentX(float** *alignmentX***)**
  public void setAlignmentY(float *alignmentY***)**
  Diese Methoden legt die von einer Komponente *gewünschte* horizontale bzw. vertikale Ausrichtung in der heimatlichen Container-Zelle fest. Mit dem Wert 0 des **float**-Parameters signalisiert eine Komponente ihre Präferenz für den linken bzw. oberen Rand. Dem maximalen Wert von 1 entspricht eine Vorliebe für den rechten bzw. unteren Rand, und der Wert 0,5 signalisiert eine Tendenz zur Mitte. Für einige spezielle Parameterwerte stehen Konstanten in der Klasse **Component** zur Verfügung, z.B.:
  ```
  lblText.setAlignmentX(Component.CENTER_ALIGNMENT);
  lblText.setAlignmentY(Component.TOP_ALIGNMENT);
  ```
  Mit den zugehörigen **get**-Methoden (z.B. **getAlignmentX()**) kann man die von einer Komponente gewünschten Orientierungen erfragen. Ob diese Wünsche auch Realität werden, hängt vom Layout-Manager des Containers ab, in dem sich eine Komponente befindet (vgl. Abschnitt 14.4). Während z.B. der **FlowLayout**-Manger die Ausrichtungswünsche der Komponenten ignoriert, werden sie vom **BoxLayout**-Manager nach Möglichkeit beachtet.
- **public void setMinimumSize(Dimension** *minimumSize***)**
  public void setMaxmumSize(Dimension *maximumSize***)**
  public void setPreferredSize(Dimension *preferredSize***)**
  Diese Methoden legen die von einer Komponente *gewünschte* minimale, maximale bzw. bevorzugte Größe über ein **Dimension**-Objekt fest, das öffentliche **int**-Felder für Breite (**width**) und Höhe (**height**) in der Maßeinheit Pixel besitzt, z.B.:[1]
  ```
  lblText.setMaximumSize(new Dimension(100, 50));
  ```
  Mit den zugehörigen **get**-Methoden (z.B. **getPreferredSize()**) kann man die von einer Komponente gewünschten Größen erfragen. Ob die Wünsche auch Realität werden, hängt von dem für eine Komponente zuständigen Layout-Manager ab (vgl. Abschnitt 14.4).
- **public void setBackground(Color** *color***)**
  public void setForeground(Color *color***)**
  Mit diesen Methoden lässt sich die Hinter- bzw. Vordergrundgrundfarbe über ein Objekt der Klasse **Color** setzen.
- **public void setFont(Font** *schriftart***)**
  Mit der Methode **setFont()** wird die Schriftart einer Komponente über ein Objekt der Klasse **Font** festgelegt, z.B.:
  ```
  lblText.setFont(new Font(Font.SANS_SERIF, Font.BOLD, 16));
  ```
  Durch Beschränkung auf die fünf *logischen* Schriftarten (**SERIF**, **SANS_SERIF**, **MONOSPACED**, **DIALOG**, **DIALOG_INPUT**), die generell in jeder JVM verfügbar sind und auf jeweils verfügbare *physikalische* Schriftarten abgebildet werden, vermeidet man eine Abhängigkeit von der lokalen Systemausstattung. Werden die Schriftarten über Konstanten der Klasse **Font** angesprochen (z.B. `Font.SANS_SERIF`) statt über Zeichenfolgen (z.B. `"Arial"`), sind Missverständnisse durch Tippfehler ausgeschlossen.

---

[1] Die bereits seit Java 1.0 vorhandene Klasse **Dimension** ist kein Musterbeispiel für die Datenkapselung der objektorientierten Programmierung, so dass in der API-Dokumentation gewarnt werden muss:

> Normally the values of `width` and `height` are non-negative integers. The constructors that allow you to create a dimension do not prevent you from setting a negative value for these properties. If the value of `width` or `height` is negative, the behavior of some methods defined by other objects is undefined.

- **public void setVisible(boolean** *visible***)**
  Mit dieser Methode lässt sich eine Komponente (un)sichtbar machen.

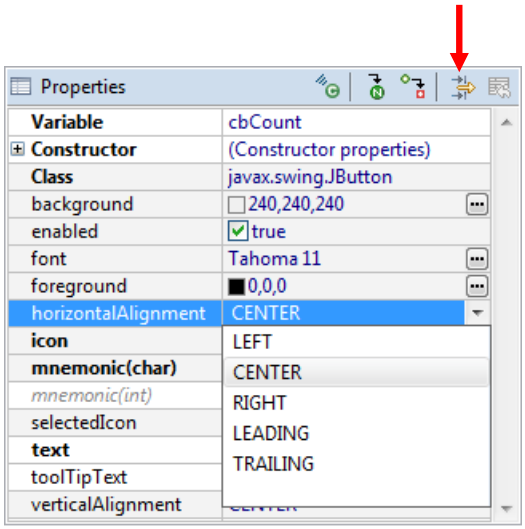
Mit den folgenden Methoden kann für den *Inhalt* einer Komponente (z.B. für den Text eines Labels) die horizontale bzw. vertikale Ausrichtung festgelegt werden. Sie sind zwar nicht in der Klasse **JComponent** definiert, aber in vielen Spezialisierungen (z.B. **JLabel**, **AbstractButton**, **JTextField**) vorhanden:

public void setHorizontalAlignment(int *alignment***)**
public void setVerticalAlignment(int *alignment***)**

Die erlaubten Werte sind über Konstanten im Interface **SwingConstants** ansprechbar:

LEFT, **CENTER**, **RIGHT**	Links, Mitte, Rechts
LEADING, **TRAILING**	Start- bzw. Endseite gemäß Schriftausrichtung

Um die Eigenschaften von Swing-Komponenten *zur Entwurfszeit* zu beeinflussen, müssen Java-Programmierer nicht unbedingt die **set**-Methoden direkt verwenden, weil visuelle Designwerkzeuge von Entwicklungsumgebungen diese Arbeit übernehmen. Der WindowBuilder in Eclipse präsentiert z.B. zu einer **JButton**-Komponente die folgende Tabelle mit den wichtigsten Eigenschaften:



Mit der Schaltfläche kann man zwischen der reduzierten und der vollständigen Eigenschaftsliste umschalten, wobei die seltener benötigten Eigenschaften an einer grauen und kursiven Beschriftung zu erkennen sind (im Beispiel: *mnemonic(int)*).

Wer die Eigenschaften von Komponenten *zur Laufzeit* beeinflussen möchte, kommt allerdings um den Aufruf der **set**-Methoden nicht herum.

14.3.5 Zubehör für Swing-Komponenten

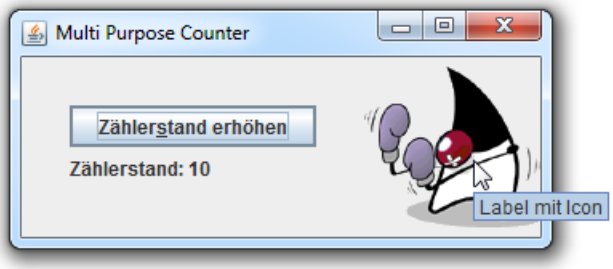
Es sind einige Möglichkeiten verfügbar, das optische Erscheinungsbild und die Bedienbarkeit von Swing-Komponenten zu verbessern.

14.3.5.1 Tool-Tip - Text

Mit der **JComponent**-Methode **setToolTipText()** kann man Tool-Tipps (QuickInfos) zu einzelnen Steuerelementen definieren, z.B.:

```
lblIcon.setToolTipText("Label mit Icon");
```

Diese erscheinen in einem PopUp-Fenster, wenn der Mauszeiger über einer betroffenen Komponente verharrt, z.B.:

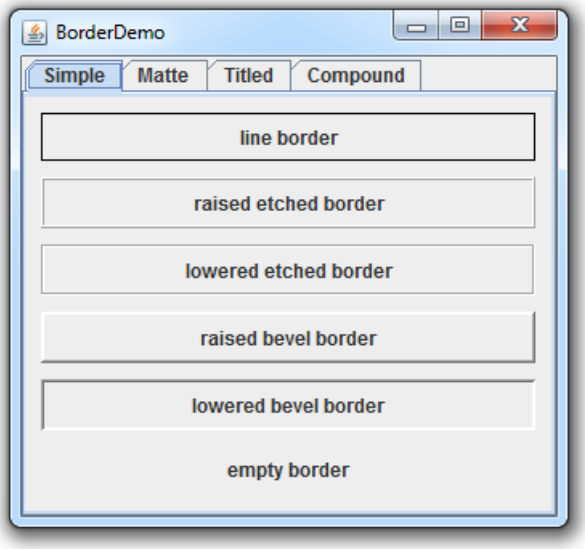


14.3.5.2 Rahmen

Mit der **JComponent**-Methode **setBorder()** lässt sich ein Rahmen festlegen, wobei die Klasse **BorderFactory** mit ihren statischen Methoden diverse Modelle herstellen kann. Im Einführungsbeispiel verschaffen wir dem **JPanel**-Container `panLeft`, der den Befehlsschalter und das Text-Label aufnimmt (siehe Abschnitt 14.3.3), etwas „Luft“:

```
panLeft.setBorder(BorderFactory.createEmptyBorder(30, 30, 30, 30));
```

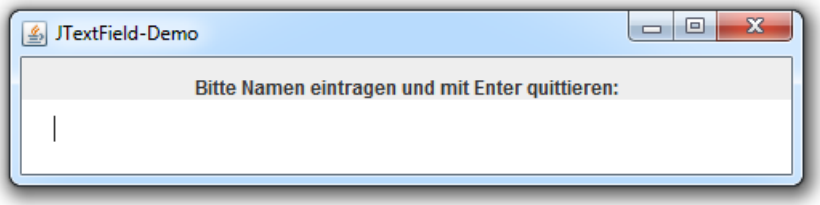
Den Quellcode zu der folgenden Rahmen-Musterschau:



finden Sie über diese Webseite:

http://docs.oracle.com/javase/tutorial/uiswing/components/border.html

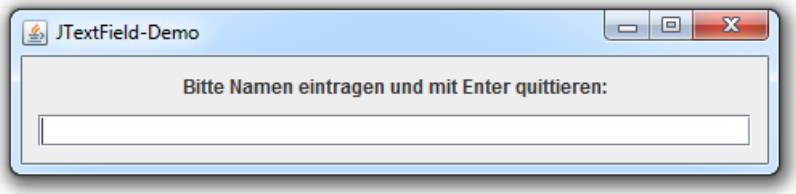
Obwohl die Methode **setBorder()** in der Klasse **JComponent** definiert ist, arbeitet sie nur bei **JPanel** und **JLabel** erwartungsgemäß (vgl. API-Dokumentation). So wird z.B. ein Rahmen um ein Texteingabefeld (**JTextField**, vgl. Abschnitt 14.6.1) von Swing wenig sinnvoll realisiert:



Um einer von **JLabel** und **JPanel** verschiedenen Komponente einen Rahmen zu verpassen, geht man am besten so vor:

- Komponente in einen **JPanel**-Container stecken.
- **setBorder()** auf den **JPanel**-Container anwenden.

Bei einem Texteingabefeld im eingerahmten **JPanel**-Container erhält man das gewünschte Ergebnis:

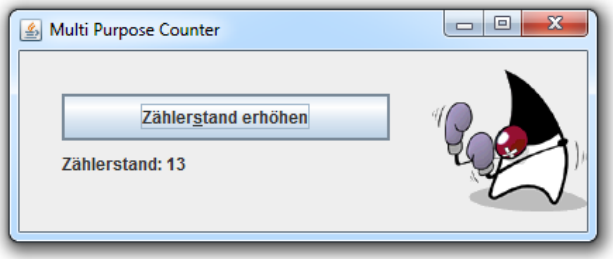


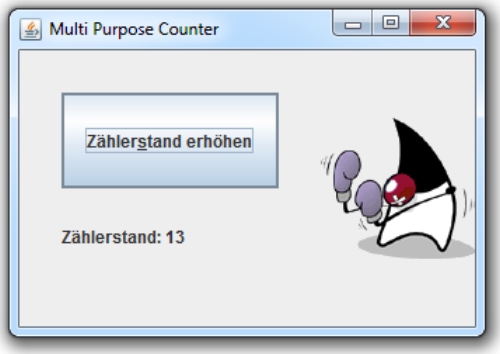
14.4 Layout-Manager

Der **Layout-Manager** eines Containers ist für die **Anordnung** und **Größe** der enthaltenen Komponenten zuständig und orientiert sich bei seiner Tätigkeit je nach Typ auch an den Wünschen der Komponenten zur Größe und Ausrichtung, welche über die **Component**-Methoden **getPreferredSize()**, **getMinimumSize()**, **getAlignmentX()** etc. in Erfahrung zu bringen sind. Für das Innenleben von enthaltenen (untergeordneten) Container-Komponenten sind deren Layout-Manager verantwortlich. Um einem übergeordneten Kollegen Auskunft geben zu können, muss ein Layout-Manager die minimale, maximale und bevorzugte Größe des eigenen Containers berechnen.

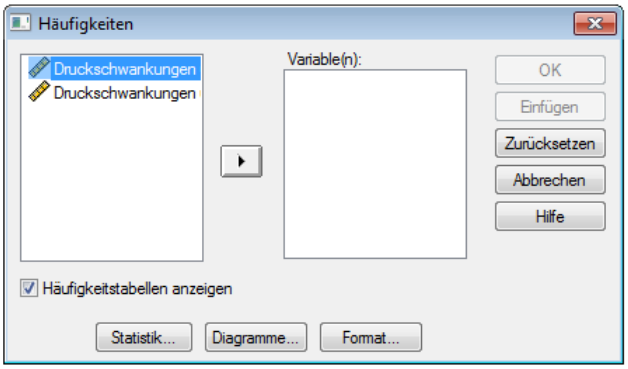
Durch Verschachteln von Containern, für die jeweils ein spezieller Layout-Manager zuständig ist, sollte sich fast jede Designidee verwirklichen lassen.

Bei einer Änderung der Fenstergröße sorgen die beteiligten Layout-Manager für die dynamische Anpassung der Platzaufteilung, z.B.:

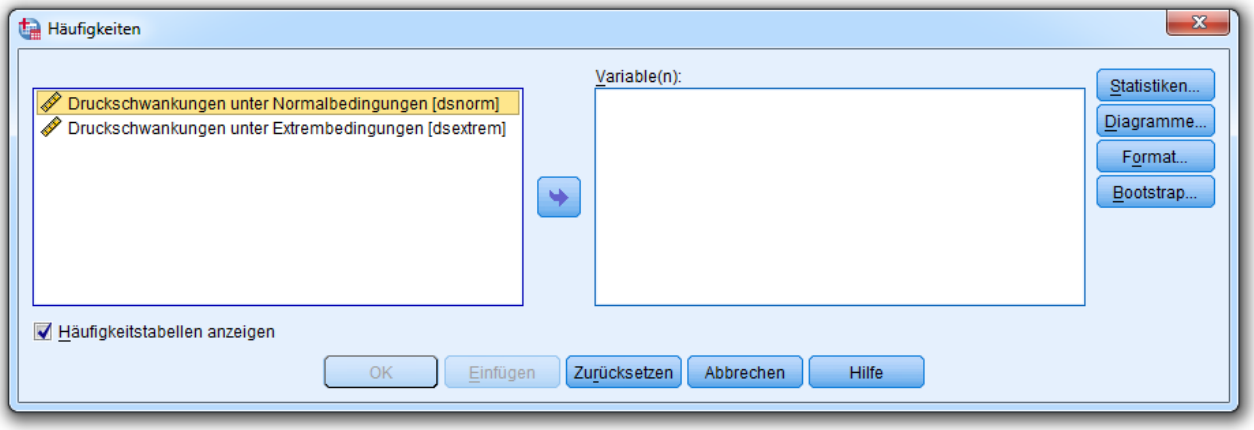




Wie nützlich größenvariable Fenster mit intelligenter Platzverwaltung für Benutzer sein können, zeigt der folgende Vergleich von zwei äquivalenten Dialogboxen aus verschiedenen Versionen des Statistikprogramms SPSS vor und nach der Neuentwicklung in Java. In SPSS 15 erschweren Dialogboxen mit fester Größe gelegentlich die Unterscheidung von Variablen mit identisch startender Bezeichnung:



Weil SPSS seit der Version 16 auf Java-Technik basiert, lässt sich die Breite der äquivalenten Dialogbox nun so wählen, dass die Bezeichnungen vollständig Platz finden:



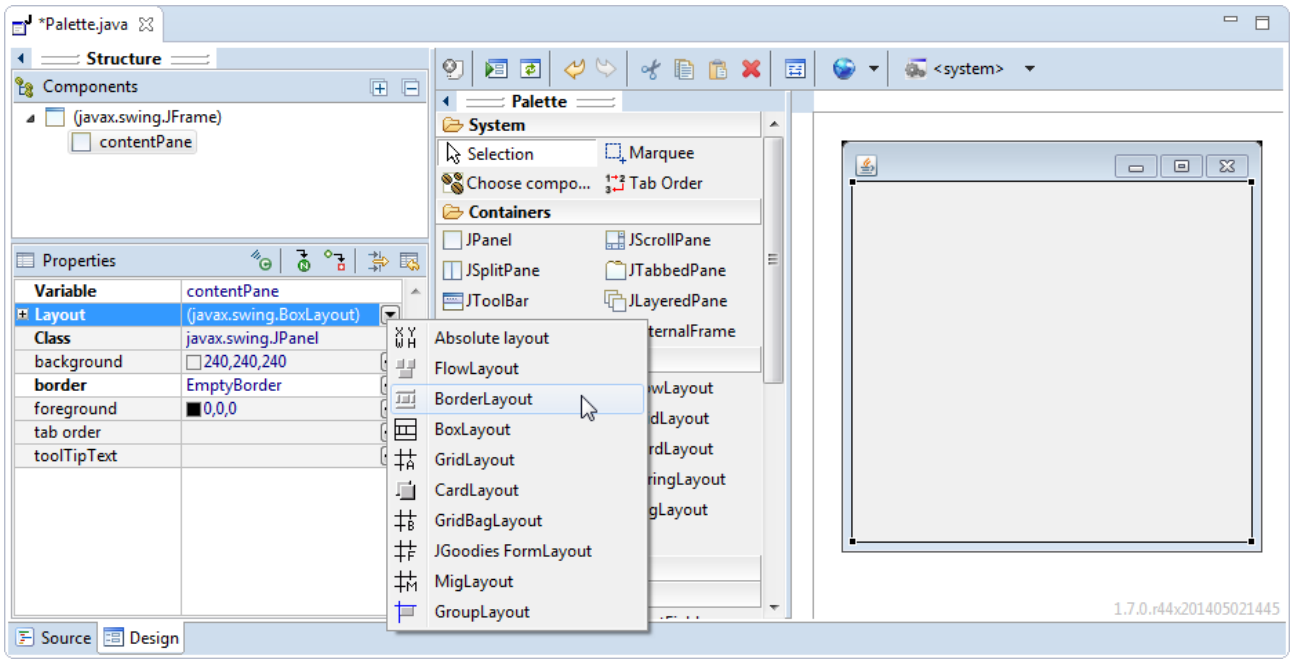
Man kann auf die Dienste eines Layout-Managers verzichten und alle Layout-Details (Positionen und Größen) manuell festzulegen (siehe Abschnitt 14.4.5).

```
cont.setLayout(null);
```

Dann sollte mit dem folgenden Aufruf der **JFrame**-Methode **setResizable()** eine Änderung der Fenstergröße durch den Benutzer verhindert werden:

```
frame.setResizable(false);
```

Im Alltag der Software-Entwicklung erleichtern Assistenten wie der WindowBuilder in Eclipse den Umgang mit den Layout-Managern, z.B. bei der Auswahl:



Um eine begründete Wahl zu treffen, muss man die Kandidaten natürlich kennen. Aus der WindowBuilder-Angebotsliste werden anschließend die ersten fünf Layout-Manager vorgestellt. Aus Zeitgründen können u.a. die die folgenden Layout-Manager *nicht* behandelt werden:

- **GridBagLayout**
  Dieser Layout-Manager ist ebenso flexibel wie kompliziert. Im Unterschied zum **GridLayout** (vgl. Abschnitt 14.4.2) lässt sich z.B. die Breite der Spalten getrennt festlegen.
- **CardLayout**
  Dieser Layout-Manager verwaltet die Komponenten im Container wie gestapelte Karten, von denen eine sichtbar ist.
- **GroupLayout**
  Dieser Layout-Manager wurde in Java 6 als Basis für den grafischen Fenster-Designer *Matisse* in der Entwicklungsumgebung *NetBeans* eingeführt, eignet sich aber auch für das direkte Kodieren.

Wer sie kennen lernen möchte, kann sich z.B. im Java-Tutorial informieren (Oracle 2014).[1]

Die anschließend beschriebenen Layout-Manager- Typen stammen alle direkt von der Klasse **Object** ab, so dass keine Spezialisierungsbeziehungen zu beachten sind.

14.4.1 BorderLayout

In unserem Swing-Einstiegsbeispiel bleiben bei nahezu beliebigen Veränderungen des Anwendungsfensters (siehe oben) die folgenden räumlichen Relationen erhalten:

- Das Text-Label befindet sich senkrecht unter dem Befehlsschalter.
- Das Icon-Label erscheint rechts neben den beiden anderen Komponenten und ist vertikal zentriert.

Eine wesentliche Voraussetzung für dieses Verhalten sind die beteiligten Container und ihre Layout-Manager. Das Anwendungsfenster (aus der Klasse **JFrame**) ist ein (schwergewichtiger) Top-

[1] Siehe: http://docs.oracle.com/javase/tutorial/uiswing/layout/index.html

Level - Container. Wie sich schon in Abschnitt 14.1.2 herausgestellt hat, besitzen **JFrame**-Fenster eine Scheibenstruktur, wobei die Inhaltsschicht (engl.: *content pane*) zur Verwaltung der im Rahmenfenster enthaltenen Steuerelemente dient (mit Ausnahme der Menüzeile).[1]

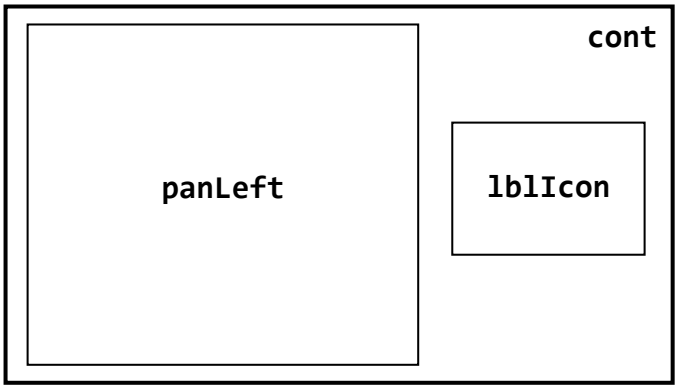
In dem zur **JFrame**-Inhaltsschicht gehörigen Container landen:

- ein untergeordneter Container aus der Klasse **JPanel** namens `panLeft`
- das **JLabel**-Objekt `lblIcon`

Diese beiden Komponenten gelangen über **add()**-Methodenaufrufe an ihrem Standort:

```
cont.add(panLeft, BorderLayout.CENTER);
cont.add(lblIcon, BorderLayout.EAST);
```

Es resultiert die folgende Anordnung:



Dafür sorgt der **Layout-Manager** der **JFrame**-Inhaltsschicht. Weil wir die Voreinstellung nicht geändert haben, handelt es sich um ein Objekt der Klasse **BorderLayout**. Dies wird deutlich in den Positionierungsparametern der obigen **add()**-Aufrufe:

- `panLeft` soll das Zentrum des `MPC`-Containers besetzen, das sich mangels West-Komponente am linken Rand befindet.
- `lblIcon` soll sich am östlichen `MPC`-Rand aufhalten.

Im Swing-Einstiegsbeispiel landen der Befehlsschalter und das Text-Label im **JPanel**-Container `panLeft`, dem ein **GridLayout**-Manager zugeteilt wird (siehe Abschnitt 14.4.2). Für ein gelungenes Layout ist oft das hierarchische Schachteln von Containern mit jeweils passend gewähltem Layout-Manager erforderlich.

Welche Plätze ein **BorderLayout**-Objekt insgesamt für einen Container verwalten kann, zeigt das folgende Beispielprogramm, das an jeder möglichen Position einen Befehlsschalter enthält:

---

[1] Zwar ist die **JFrame** - Inhaltsschicht über eine Referenzvariable vom Typ **Container** ansprechbar, geliefert von der **JFrame**-Methode **getContentPane()**, befragt man dieses Objekt jedoch über die Methode **getClass()** nach dem faktischen Typ, stellt sich die abgeleitete Klasse **JPanel** heraus. Wir haben es also (zumindest bei einem Rahmenfenster aus der Klasse **JFrame**) nur mit **JPanel**-Containern zu tun.

Für die Abstände zwischen den Komponenten (horizontal und vertikal jeweils 5 Pixel) wurde hier durch einen speziellen **BorderLayout**-Konstruktor gesorgt:

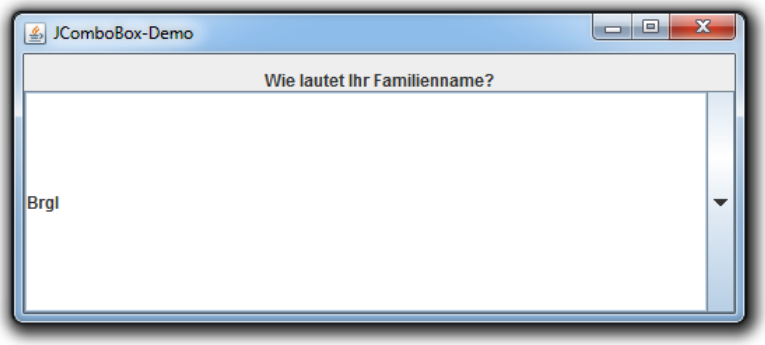
```
frame.getContentPane().setLayout(new BorderLayout(5,5));
```

Jede **BorderLayout** - Zelle kann nur *eine* Komponente aufnehmen. Setzt man eine Komponente in eine belegte Zelle, wird der Altinsasse verdrängt.

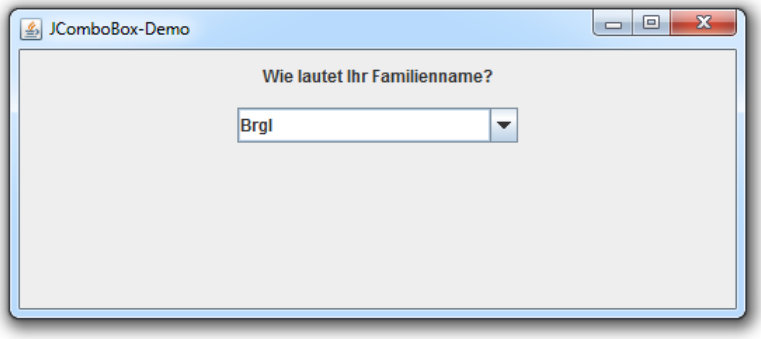
Welche Gestaltungsmöglichkeiten ein **BorderLayout** durch sein Verhalten bei teilweise unbesetzten Positionen bietet, sollten Sie durch Probieren herausfinden.

Wie das letzte Beispiel zeigt, belegt die in einem **BorderLayout**-Fach befindliche Komponente den gesamten dort verfügbaren Platz. Dies ist bei vielen Komponenten (z.B. **JPanel**, **JTextPane**) sehr sinnvoll, wobei man die „gierigste“ Komponente ins Zentrum setzen sollte, das bei Zunahme der Container-Fläche primär profitiert.

Bei manchen Bedienelementen ist eine monströse Größenzunahme aber unangemessen, z.B. bei einem Kombinationsfeld (**JComboBox**, siehe Abschnitt 14.6.4):



Der bei einem **JPanel**-Container voreingestellte **FowLayout** - Manager (siehe Abschnitt 14.4.3) respektiert die bevorzugte Größe der enthaltenen Bedienelemente. Daher kann es sinnvoll sein, ein Bedienelement mit **JPanel**-Hülle in ein **BorderLayout**-Fach zu stecken, z.B.:



14.4.2 GridLayout

Im Swing-Einstiegsbeispiel wird für das **JPanel**-Objekt `panLeft`, das den Befehlsschalter und das Text-Label enthält, mit der Methode **setLayout()** (geerbt von der Klasse **Container**) ein Layout-Manager aus der Klasse **GridLayout** engagiert, der im Allgemeinen eine ($z \times s$) - Komponentenmatrix verwaltet und im Beispiel dafür sorgt, dass alle `panLeft`-Komponenten bei linksbündiger Ausrichtung übereinander stehen. Dazu wird im **GridLayout**-Konstruktor *eine* Spalte und (über den Aktualparameterwert 0) eine unbestimmte Anzahl von Zeilen angekündigt:

```
panLeft.setLayout(new GridLayout(0, 1));
```

Ist ein solcher Layout-Manager einmal im Dienst, verteilt er die Komponenten automatisch auf die verfügbaren Zellen, so dass die **add()** - Methode ohne Platzanweisung auskommt:

```
panLeft.add(cbCount);
panLeft.add(lblText);
```

In der folgenden Anweisung wird dem Inhaltsbereich eines **JFrame**-Rahmenfensters ein **GridLayout** – Manager zugeordnet, der mit 2 Zeilen arbeitet (siehe ersten Konstruktorparameter) und die Anzahl der Spalten passend zur Anzahl der eingefügten Komponenten selbst ermittelt (Wert 0 im zweiten Konstruktorparameter):

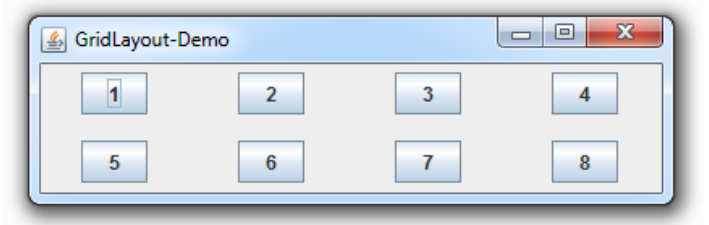
```
getContentPane().setLayout(new GridLayout(2,0,5,5));
```

Sobald der erste Parameter einen von 0 verschiedenen Wert besitzt, wird der 2. Parameter ignoriert. Der 2. Parameter beeinflusst das Layout also nur dann, wenn der 1. Parameter den Wert 0 besitzt. Durch die beiden letzten Konstruktorparameter wird in horizontaler und vertikaler Richtung ein Zwischenabstand von jeweils 5 Pixeln gewählt.

Ob die horizontale Verteilung der Komponenten von links nach rechts erfolgt oder umgekehrt, hängt von der Komponenten-Orientierung des Containers ab (siehe Abschnitt 14.3.3).

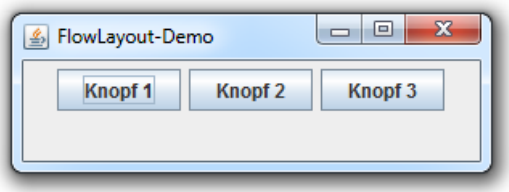
Von einem **GridLayout**-Manager wird der verfügbare Platz gleichmäßig auf alle Komponenten verteilt, wie das folgende Beispielprogramm mit acht phantasielos beschrifteten Schaltern zeigt:

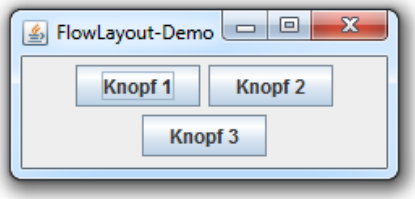
Wie beim **BorderLayout** belegt auch beim **GridLayout** jede Komponente den gesamten verfügbaren Platz in ihrer Zelle. Beim gleich zu behandelnden **FlowLayout**, das beim **JPanel**-Container voreingestellt ist, erhalten alle Komponenten hingegen ihre bevorzugte Größe. Daher kann es sinnvoll sein, Bedienelemente mit einer **JPanel**-Verpackung in die Zellen eines **GridLayouts** zu stecken, z.B.:



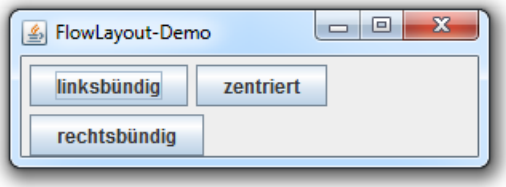
14.4.3 FlowLayout

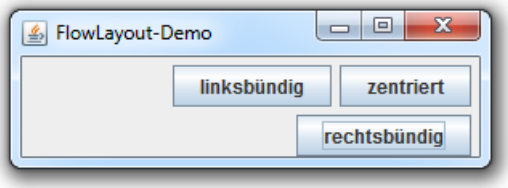
Das recht simple und betagte **FlowLayout**, dient beim **JPanel**-Container als Voreinstellung. Es ordnet die Komponenten nebeneinander an, bis ein „Zeilenumbruch“ erforderlich wird:[1]





An Stelle der voreingestellten horizontalen Zentrierung der enthaltenen Komponenten kann über die **FlowLayout**-Methode **setAlignment()** auch eine links- oder rechtsbündige Ausrichtung gewählt werden, z.B.:



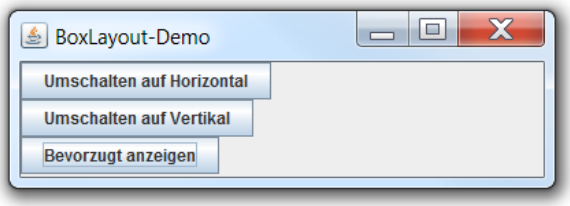


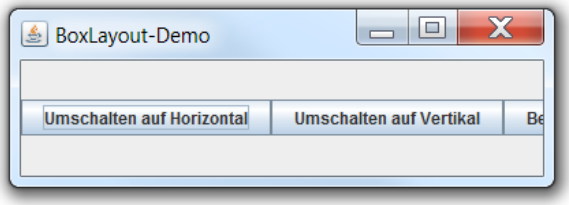
Im Unterschied zum **BorderLayout** und zum **GridLayout** erhalten beim **FlowLayout** die aufgenommenen Komponenten unabhängig von der verfügbaren Fläche nach Möglichkeit ihre bevorzugte Größe. Daher kann es sinnvoll sein, Bedienelemente mit einer **JPanel**-Verpackung in die Zellen eines **Border**- oder **GridLayouts** zu stecken.

Die von einer Komponente gewünschte Ausrichtung (eingestellt über die **JComponent**-Methoden **setAlignmentX()** bzw. **setAlignmentY()**, vgl. Abschnitt 14.3.4) wird vom **FlowLayout**-Manager *ignoriert*.

14.4.4 BoxLayout

Der **BoxLayout**-Manager kann als flexiblere Weiterentwicklung des betagten **FlowLayout**-Managers aufgefasst werden und dient dazu, die Komponenten im verwalteten Container vertikal (übereinander) oder horizontal (nebeneinander) anzuordnen, z.B.:





Er berücksichtigte dabei nach Möglichkeit die Wünsche der Komponenten bzgl. (minimaler, bevorzugter, maximaler) Größe und Ausrichtung. Bei horizontaler Anordnung findet im Unterschied zum **FlowLayout** *kein* „Zeilenumbruch“ statt.

---

[1] Ob die enthaltenen Komponenten von links nach rechts angeordnet werden oder umgekehrt, hängt von der Komponenten-Orientierung des Containers ab (siehe Abschnitt 14.3.3).

Im **BoxLayout**-Konstruktor wird zunächst der zu verwaltende Container und dann die gewünschte Orientierung der Komponenten (übereinander oder nebeneinander) gewählt, z.B.:

```
cont.setLayout(new BoxLayout(cont, BoxLayout.Y_AXIS));
```

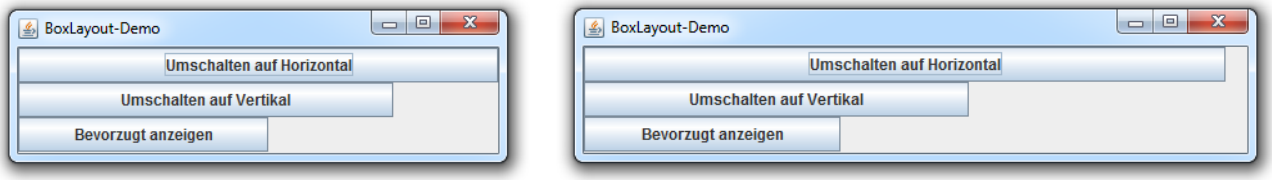
Dass man den Namen des Containers im **BoxLayout**-Konstruktor angeben muss, ist etwas merkwürdig.

Die Orientierung der Komponenten kann mit Hilfe von **BoxLayout**-Konstanten entweder absolut oder in Bezug auf die lokalspezifische Anordnung von Textelementen festgelegt werden:

- **X_AXIS**
  Horizontale Anordnung von links nach rechts
- **Y_AXIS**
  Vertikale Anordnung von oben nach unten
- **LINE_AXIS**, **PAGE_AXIS**
  Diese Optionen unterstützen die Internationalisierung von Software und beachten die Komponenten-Orientierung des betroffenen Containers (vgl. Abschnitt 14.3.3). Das **BoxLayout** ordnet die Komponenten analog zu den Wörtern in einer Zeile bzw. Spalte (**LINE_AXIS**) oder analog zu den Zeilen bzw. Spalten auf einer Seite an (**PAGE_AXIS**) und reagiert folgendermaßen auf die **ComponentOrientation** des Containers:

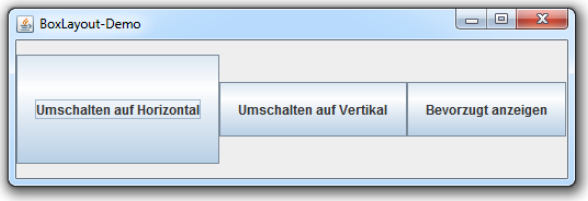
<table>
<tr><th rowspan="2">ComponentOrientation
(Textanordnung) des Containers</th><th colspan="2">Boxlayout-Komponentenanordnung bei</th></tr>
<tr><th>LINE_AXIS</th><th>PAGE_AXIS</th></tr>
<tr><td>horizontal (zeilenweise),
Wörter von links nach rechts
(Westeuropa)</td><td>von links nach rechts</td><td rowspan="2">von oben nach unten</td></tr>
<tr><td>horizontal (zeilenweise),
Wörter von rechts nach links
(Arabisch, Hebräisch)</td><td>von rechts nach links</td></tr>
<tr><td>vertikal (von oben nach unten)
Spalten von links nach rechts
(Mongolei)</td><td rowspan="2">von oben nach unten</td><td>von links nach rechts</td></tr>
<tr><td>vertikal (von oben nach unten)
Spalten von rechts nach links
(Japan, China, Korea)</td><td>von rechts nach links</td></tr>
</table>

Überschreitet bei einem vertikalen **BoxLayout** die Breite des Containers die bevorzugte Breite einer Komponente, dann wächst diese bis zu ihrer maximalen Breite an, z.B.:



In der Regel sind allerdings die bevorzugte und die maximale Breite einer Komponente identisch, so dass die Komponente trotz Zunahme der Container-Breite unverändert bleibt.

Überschreitet bei einem horizontalen **BoxLayout** die Höhe des Containers die bevorzugte Höhe einer Komponente, dann wächst diese bis zu ihrer maximalen Höhe, z.B.:

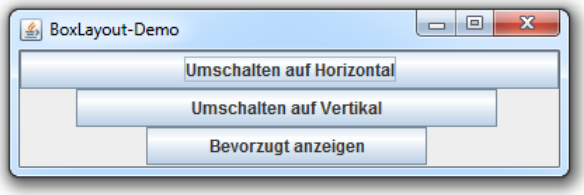


In der Regel sind allerdings die bevorzugte und die maximale Höhe einer Komponente identisch, so dass die Komponente trotz Zunahme der Container-Höhe unverändert bleibt.

Ein **BoxLayout**-Manager berücksichtigt auch die von den Komponenten gewünschte Ausrichtung in X- und Y-Richtung. Bisher waren alle Komponenten in X-Richtung linksbündig ausgerichtet und beim vertikalen **BoxLayout** dementsprechend am linken Container-Rand angeheftet (siehe oben). Mit der **JComponent**-Methode **setAlignmentX()** lässt sich eine Komponente z.B. in X-Richtung zentrieren:

```
horizontal.setAlignmentX(Component.CENTER_ALIGNMENT);
```

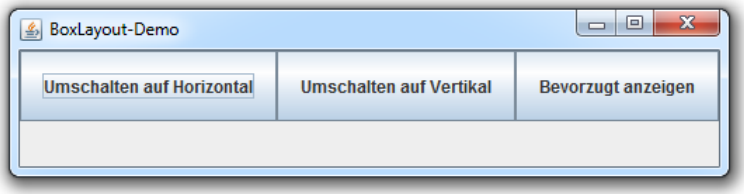
Sind *alle* Komponenten im Container so ausgerichtet, führt das vertikale **BoxLayout** zum folgenden Ergebnis:



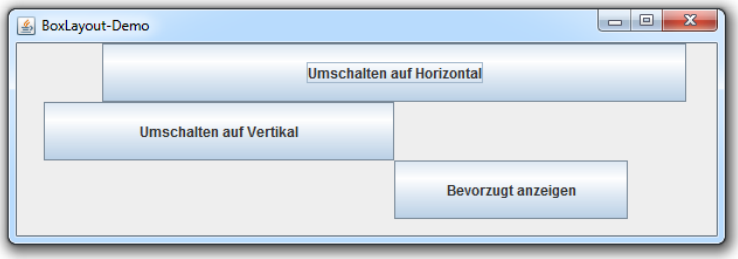
Analog hat bisher das horizontale **BoxLayout** die bei allen Komponenten voreingestellte Zentrierung in Y-Richtung respektiert (siehe oben). Mit der **JComponent**-Methode **setAlignmentY()** lässt sich eine Komponente z.B. an die Decke heften:

```
horizontal.setAlignmentY(Component.TOP_ALIGNMENT);
```

Sind *alle* Komponenten eines Containers in Y-Richtung TOP-orientiert, führt das horizontale **BoxLayout** zum folgenden Ergebnis:



Haben die Komponenten eines Containers *unterschiedliche* X- bzw. Y-Ausrichtungen, kommt es beim **BoxLayout** zu eventuell überraschenden Ergebnissen. Im folgenden Beispiel folgen vertikal eine zentrierte, eine rechts ausgerichtete und eine links ausgerichtete Komponente aufeinander:



Die beiden unteren Komponenten orientieren sich nicht am rechten bzw. linken Rand des Containers, sondern an der neutralen Position der ersten (zentrierten) Komponente.

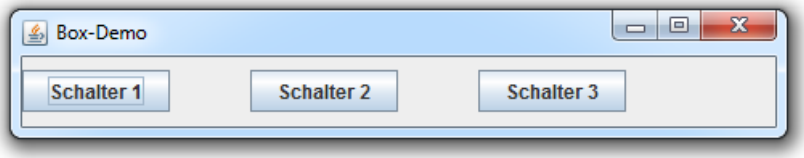
Die von **JComponent** abstammende Klasse **Box** realisiert einen Container mit voreingestelltem **BoxLayout** und bietet einige statische Methoden zur verbesserten Komponentenanordnung. Im **Box**-Konstruktor ist die gewünschte Komponentenausrichtung anzugeben, z.B.:

```
final Box boxContainer = new Box(BoxLayout.X_AXIS);
```

Mit der statischen **Box**-Methode **createHorizontalStrut()** erhält man eine unsichtbare Komponente mit fester Breite. Durch Einfügen solcher Komponenten, z.B.:

```
final JButton cb1 = new JButton("Schalter 1");
boxContainer.add(cb1);
boxContainer.add(Box.createHorizontalStrut(50));
final JButton cb2 = new JButton("Schalter 2");
boxContainer.add(cb2);
boxContainer.add(Box.createHorizontalStrut(50));
final JButton cb3 = new JButton("Schalter 3");
boxContainer.add(cb3);
```

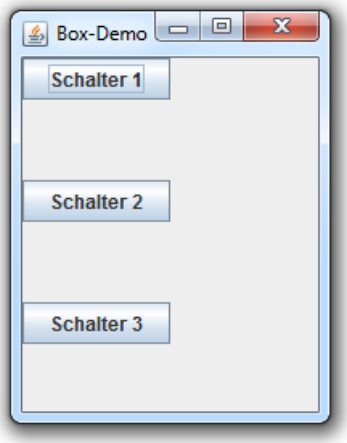
schafft man feste Abstände zwischen horizontal angeordneten Komponenten:



Mit der statischen **Box**-Methode **createVerticalStrut()** erhält man eine unsichtbare Komponente mit fester Höhe. Durch Einfügen solcher Komponenten, z.B.:

```
final Box boxContainer = new Box(BoxLayout.Y_AXIS);
final JButton cb1 = new JButton("Schalter 1");
boxContainer.add(cb1);
boxContainer.add(Box.createVerticalStrut(50));
final JButton cb2 = new JButton("Schalter 2");
boxContainer.add(cb2);
boxContainer.add(Box.createVerticalStrut(50));
final JButton cb3 = new JButton("Schalter 3");
boxContainer.add(cb3);
```

schafft man feste Abstände zwischen vertikal angeordneten Komponenten:



Mit der statischen **Box**-Methode **createHorizontalGlue()** erhält man eine unsichtbare Komponente mit variabler Breite, die bei einer Verbreiterung des Containers Platz absorbiert und bei einer Verkleinerung wieder abgibt. Durch Einfügen solcher Komponenten, z.B.

```
boxContainer.add(Box.createHorizontalGlue());
```

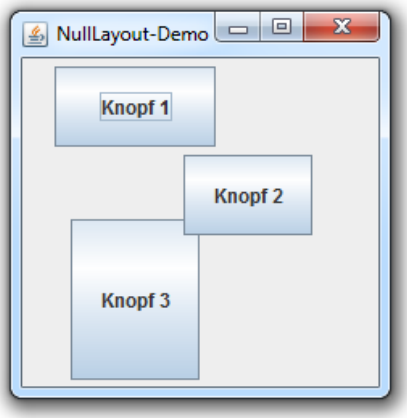
schafft man variable (elastische) Abstände zwischen horizontal angeordneten Komponenten:

Mit der **Box**-Methode **createVerticalGlue()** erreicht man dieselbe Elastizität in vertikaler Richtung.

Die von statischen **create**-Methoden der Klasse **Box** erzeugten unsichtbaren Hilfskomponenten taugen nicht nur in **Box**-Objekten, sondern lassen sich in beliebige Container einfügen, die von einem **BoxLayout**-Manager betreut werden.

14.4.5 Freies Layout

Man kann auf die Dienste eines Layout-Managers verzichten und die Positionen sowie Größen der in einem Container enthaltenen Komponenten individuell festlegen, so dass z.B. auch Merkwürdigkeiten wie überlappende Befehlsschalter möglich werden:



Ein freies Layout lässt sich folgendermaßen realisieren:

- Den voreingestellten Layout-Manager abschalten mit **setLayout(null)**
- Positionen und Größen der Komponenten z.B. mit der **Component**-Methode **setBounds()** festlegen:

  public void setBounds(int *x*, **int** *y*, **int** *width*, **int** *height***)**

  Mit den vier **int**-Parametern wählt man die Position und die Größe:

  x neue X-Koordinate der Komponente
  y neue Y-Koordinate der Komponente

width neue Breite der Komponente
height neue Höhe der Komponente

Dies wird in folgendem Programm demonstriert:

```
import java.awt.*;
import javax.swing.*;

class NullLayoutDemo {
 NullLayoutDemo() {
  final JFrame frame = new JFrame("NullLayout-Demo");

  final Container cont = frame.getContentPane();
  cont.setLayout(null);

  final JButton k1 = new JButton("Knopf 1");
  k1.setBounds(20,5,100,50);
  cont.add(k1);
  final JButton k2 = new JButton("Knopf 2");
  k2.setBounds(100,60,80,50);
  cont.add(k2);
  final JButton k3 = new JButton("Knopf 3");
  k3.setBounds(30,100,80,100);
  cont.add(k3);

  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(210, 240);
  frame.setVisible(true);
 }

 public static void main(String[] arg) {
  new NullLayoutDemo();
 }
}
```

14.5 Ereignisbehandlung

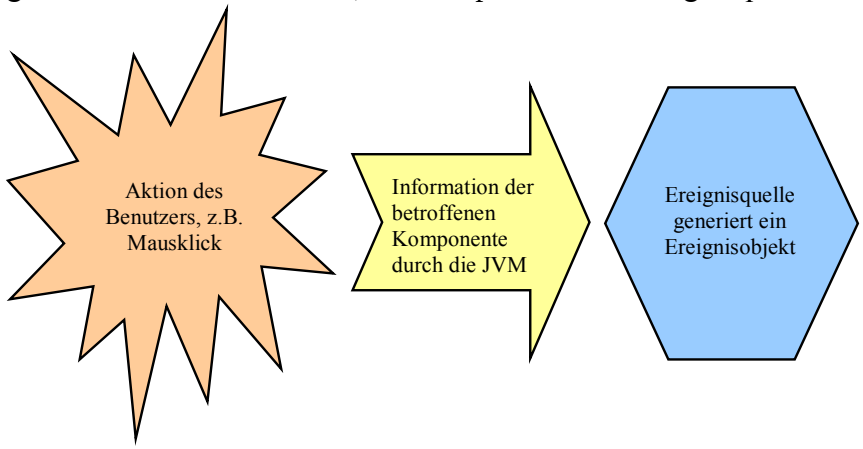
Die Besonderheit einer Komponente im Unterschied zu einem gewöhnlichen Objekt besteht neben ihrem optischen Auftritt in der Fähigkeit, über **Ereignisse** zu kommunizieren. Sie kann elementare GUI-Ereignisse (z.B. Mausklicks, Tastendrücke) empfangen und aufbereitete, abstraktere Ereignisse (z.B. Änderung einer Listenzusammenstellung) anbieten. Andere Objekte können sich bei einer Komponente für eine Ereigniskategorie registrieren lassen, um ggf. benachrichtigt zu werden. Wie das im Detail geschieht, wird nun beschrieben.

14.5.1 Das Delegationsmodell

Bei der in Java benutzten Ereignisbehandlung nach dem Delegationsmodell sind folgende Objekte beteiligt:

- **Ereignisquelle**
  Dies ist eine Komponente, die Ereignisse feststellen und weiterleiten kann. Im **Swing**-Demoprogramm, das wir seit Abschnitt 14.2 besprechen, kann z.B. der Befehlsschalter `cbCount` diese Rolle spielen. Auslöser ist letztlich der Benutzer, von dessen Tätigkeit (z.B. Mausklick) die Komponente durch Vermittlung des Betriebssystems und des Java-

Laufzeitsystems erfährt. Daraufhin generiert die Komponente ein **Ereignisobjekt**. Aus der Sicht des Programms ist es also sinnvoll, die *Komponente* als Ereignisquelle aufzufassen.



Neben dem Befehlsschalter enthält das Beispielprogramm noch eine zweite potentielle Ereignisquelle: das Rahmenfenster. Hier führt z.B. ein Mausklick des Benutzers auf das Schließkreuz in der Titelzeile zu einem Window-Closing - Ereignis.

- **Ereignisobjekt**
  Zu jeder Ereignisart gehört in Java eine Ereignisklasse. Stellt die Quellkomponente ein Ereignis von bestimmter Art fest, generiert sie ein Objekt der zugehörigen Ereignisklasse und stellt es dem anschließend vorzustellenden Ereignisempfänger zur Verfügung. Dieser kann über Methoden der Ereignisklasse nähere Informationen über das Ereignis ermitteln. Bei einem Mausereignis (siehe unten) lässt sich z.B. über **getX()** und **getY()** der Ort des Geschehens feststellen. Eine Komponente kann in der Regel Ereignisobjekte aus verschiedenen Klassen generieren (z.B. **MouseEvent**, **ActionEvent**, **KeyEvent**). Über die Abstammungsverhältnisse der Ereignisklassen informiert Abschnitt 14.5.2.

  Zu jeder Ereignisklasse gehört ein Interface, das die Existenz von Methoden mit bestimmten Definitionsköpfen vorschreibt. Diese Methoden werden auch als *event handler* bezeichnet.

  Eine Ereignisklasse (z.B. **WindowEvent**) ist oft für *mehrere* Ereignisarten zuständig (z.B. Fenster wird aktiviert, Fenster wird geschlossen), und bei jeder Ereignisart wird eine bestimmte Methode aus dem Interface zur Ereignisklasse aufgerufen (siehe Abschnitt 14.5.2). Jede Ereignisart besitzt eine eindeutige Kennung (Event ID, siehe unten).

- **Ereignisempfänger**
  Ein Ereignis wird in der Regel *nicht* „direkt an der Quelle“ behandelt. Stattdessen wird es an Objekte gemeldet, die sich zuvor bei der Quelle als Ereignisempfänger (*event listener*) für die betroffene Ereignisklasse haben registrieren lassen.

  Nur Objekte einer entsprechend gerüsteten (das zur Ereignisklasse gehörige Interface implementierenden) Klasse können bei einer Ereignisquelle als Empfänger für die Ereignisklasse registriert werden. Bei der Ereignismeldung wird die zur speziellen Ereignisart gehörige Methode aufgerufen.

  Meist genügt es, bei einer Quelle für eine Ereignisklasse *einen* Empfänger zu registrieren. Es sind aber auch Mehrfachregistrierungen erlaubt.

Tritt bei der Quelle ein Ereignis auf, wird die zuständige Behandlungsmethode der registrierten Empfänger aufgerufen und erhält dabei als Aktualparameter eine Referenz zum Ereignisobjekt.

Diese Architektur mag auf den ersten Blick unnötig komplex erscheinen, hat aber z.B. dann Vorteile, wenn in einem Programm mehrere Komponenten als Quelle für dieselbe Ereignisklasse in Frage

kommen und sich *ein* Empfänger um die Ereignisbehandlung kümmern soll. Dieser kann beim Ereignisobjekt mit einer **getSource()** - Anfrage die Ereignisquelle erfragen, um differenziert reagieren zu können. In diesem Fall würden separate Ereignisbehandlungsmethoden eventuell zu einer unerwünschten Code-Wiederholung führen.

14.5.2 Ereignisarten und Ereignisklassen

Potentielle Empfänger für ein Objekt der Ereignisklasse **WindowEvent** müssen das zugehörige Interface **WindowListener** implementieren, zu dem etliche Methoden gehören, wie die API-Dokumentation zeigt:

`void`	**`windowActivated`**`(WindowEvent e)` Invoked when the Window is set to be the active Window.
`void`	**`windowClosed`**`(WindowEvent e)` Invoked when a window has been closed as the result of calling dispose on the window.
`void`	**`windowClosing`**`(WindowEvent e)` Invoked when the user attempts to close the window from the window's system menu.
`void`	**`windowDeactivated`**`(WindowEvent e)` Invoked when a Window is no longer the active Window.
`void`	**`windowDeiconified`**`(WindowEvent e)` Invoked when a window is changed from a minimized to a normal state.
`void`	**`windowIconified`**`(WindowEvent e)` Invoked when a window is changed from a normal to a minimized state.
`void`	**`windowOpened`**`(WindowEvent e)` Invoked the first time a window is made visible.

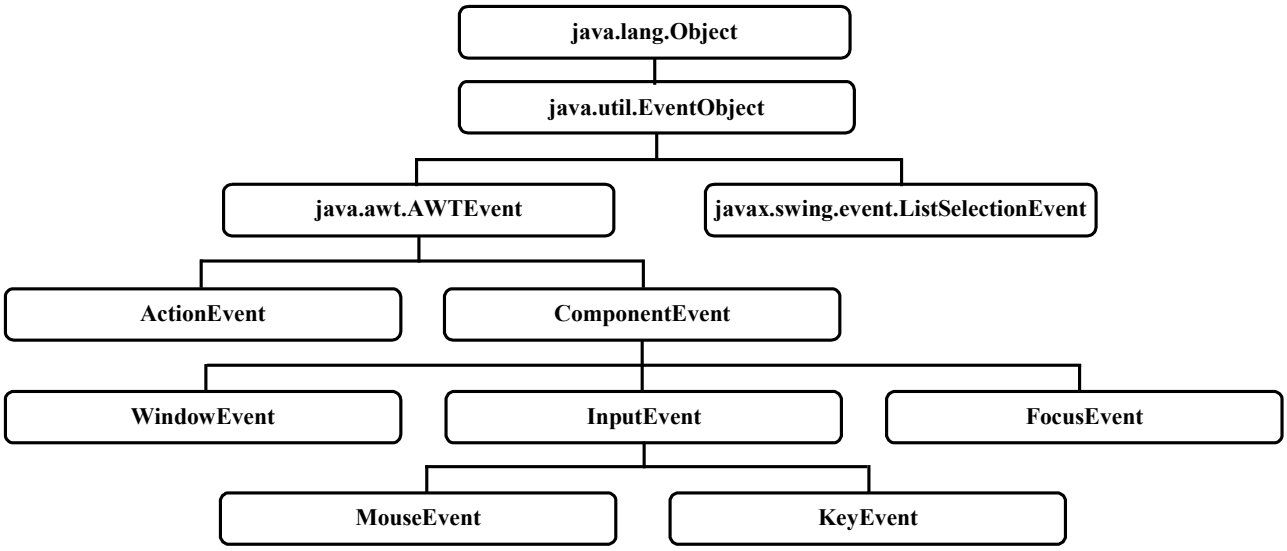
Aus den Namen der Methoden wird klar, dass die Ereignisklasse **WindowEvent** bei *mehreren* Ereignisarten zum Einsatz kommt, z.B. beim Schließen und Verkleinern eines Fensters. Die Java-Designer haben sich für eine überschaubare Anzahl von Ereignisklassen entschieden, die jeweils für mehrere, zusammengehörige Ereignisarten zuständig sind.

Die von einem Ereignis betroffene Komponente (also die Ereignisquelle, vgl. Abschnitt 14.5.1) erfährt von der JVM die exakte Ereignisart (**Event ID**) und kennt nun ...

- die Ereignisklasse
  Damit ist klar, an welche Empfänger das Ereignis weitergeleitet werden muss.
- die bei den Empfängern aufzurufende Ereignisbehandlungsmethode (den Event Handler)

Das an die Event Handler übergebene Ereignisobjekt lässt sich mit der Methode **getID()** zur Ereignisart befragen.

In der folgenden Abbildung sehen Sie die Abstammungsverhältnisse der im Manuskript behandelten Ereignisklassen:



Klassen ohne Paketangabe gehören zum Paket **java.awt.event**.

14.5.3 Ereignisempfänger registrieren

Besitzt ein Objekt die nötigen Voraussetzungen, kann es bei einer Ereignisquelle durch einen ereignisklassenspezifischen Methodenaufruf als Empfänger registriert werden. In unserem Swing - Demoprogramm wird beim Fensterobjekt (aus der Klasse **JFrame**) für die Ereignisklasse **WindowEvent** mit der Methode **addWindowListener()** ein neu erzeugtes Objekt aus der Klasse `WC` als Empfänger eingetragen. Die folgende Anweisung findet sich im `MPC`-Konstruktor:

```
addWindowListener(new WiLi());
```

Damit die Klasse `WiLi` den Anforderungen an einen **WindowEvent**-Empfänger genügt, hat sie das Interface **WindowListener** (siehe Abschnitt 14.5.2) zu implementieren (direkt oder indirekt). In Abschnitt 14.5.4 wird die Definition der Klasse `WiLi` beschrieben.

Zu jeder Ereignisklasse gehört eine spezielle Registrierungsmethode. Für welche Ereignisklassen eine Komponente als Quelle in Frage kommt, ist also an der Verfügbarkeit von Registrierungsmethoden zu erkennen. So ist z.B. in der Klasse **javax.swing.JList<E>** die Methode **addListSelectionListener()** vorhanden, um Empfänger für ein **ListSelectionEvent** (Benutzer wechselt in einer Liste den gewählten Eintrag) zu registrieren. In der Klasse **javax.swing.JButton** fehlt eine solche Methode erwartungsgemäß.

In der folgenden Tabelle ist für einige Ereignisklassen festgehalten:

- Das zugrunde liegende Benutzerverhalten
- Das von einem Ereignisempfänger zu implementierende Interface
- Die zuständige Empfänger-Registrierungsmethode

Außerdem sind die im nächsten Abschnitt zu beschreibenden *Adapterklassen* angegeben, die für viele Ereignisklassen zur Vereinfachung der Empfängerklassen - Implementation verfügbar sind:

Ereignisklasse	**Mögliche Auslöser**	**Empfänger-Interface, zugeh. Adapterklasse, Registrierungsmethode**
ActionEvent	Der Benutzer klickt auf einen Befehlsschalter, drückt die **Enter**-Taste in einem Textfeld oder wählt einen Menüeintrag.	ActionListener addActionListener()
ComponentEvent	Position, Größe oder Sichtbarkeit einer Komponente ändern sich.	ComponentListener ComponentAdapter addComponentListener()
WindowEvent	Der Benutzer (de)aktiviert, öffnet, schließt oder (de)ikonisiert ein Fenster.	WindowListener WindowAdapter addWindowListener()
MouseEvent[1]	Benutzer drückt eine Maustaste, während sich die Maus über einer Komponente befindet.	MouseListener MouseAdapter addMouseListener()
MouseEvent[1]	Der Benutzer bewegt die Maus über einer Komponente.	MouseMotionListener MouseMotionAdapter addMouseMotionListener()
KeyEvent	Der Benutzer drückt eine Taste, während eine Komponente den Tastaturfokus besitzt.	KeyListener KeyAdapter addKeyListener()
ItemEvent	Das gewählte Item in einem Kombinationsfeld oder der Zustand eines Umschalters (Kontrollkästchen oder Optionsschalter) wechseln.	ItemListener addItemListener()
FocusEvent	Eine Komponente erhält den Tastaturfokus.	FocusListener FocusAdapter addFocusListener()
ListSelectionEvent	Der Benutzer wechselt in einer Liste den gewählten Eintrag.	ListSelectionListener addListSelectionListener()

Bei einer Ereignisquelle können zu einer Ereignisklasse auch *mehrere* Ereignisempfänger registriert werden, so dass Ereignisobjekte ggf. an mehrere Empfänger gesendet werden.

Außerdem ist es möglich, eine Empfängerregistrierung aufzuheben, z.B. mit der Methode **removeActionListener()**.

14.5.4 Adapterklassen

In unserem Swing-Einstiegsbeispiel soll der **WindowEvent**-Empfänger zum Fensterobjekt nur dann aktiv werden, wenn ein Benutzer das Fenster *schließen* und damit die Anwendung beenden möchte. In diesem Fall wird ein **WindowEvent** mit bestimmter **Event ID** erzeugt und beim Empfänger nur die **WindowListener**-Methode **windowClosing()** benötigt. Um in solchen Fällen überflüssigen Programmieraufwand zu vermeiden, besitzt das Java-API zu vielen Ereignis-Schnittstellen eine so genannte *Adapterklasse*. Diese implementiert das zugehörige Interface mit *leeren* Metho-

[1] Ein **MouseEvent** wird an registrierte **MouseListener** *und* an registrierte **MouseMotionListener** weitergeleitet.

den. Leitet man eine eigene Klasse aus einer Adapterklasse ab, ist also das fragliche Interface erfüllt, und man muss nur die wirklich benötigten Methoden durch Überschreiben funktionstüchtig machen.

In unserem Swing-Einstiegsbeispiel wird die Ereignisempfängerklasse `WC` aus der zum Interface **WindowListener** gehörigen Adapterklasse **WindowAdapter** abgeleitet:

```
private class WiLi extends WindowAdapter {
  @Override
  public void windowClosing(WindowEvent e) {
    if (JOptionPane.showConfirmDialog(e.getWindow(),
         "Wollen Sie nach "+ numClicks +
         " Klicks wirklich schon aufhören?",
         titel, JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION)
      System.exit(0);
    }
}
```

Es wird lediglich die tatsächlich benötigte Methode **windowClosing()** implementiert. Diese wird aufgerufen, wenn der Benutzer per Titelzeilen-Schließkreuz oder Systemmenü seine Absicht bekundet, das Fenster zu schließen. In diesem Fall fragt die `WiLi`-Implementation beim Benutzer nach, ob er allen Ernstes aufhören möchte (siehe Abschnitt 3.8 zur Klasse **JOptionPane**). Nach einer Bestätigung dieser Absicht, wird das Programm über die statische **System**-Methode **exit()** beendet.

Die `WiLi`-Definition befindet sich komplett innerhalb der `MPC`-Definition, ohne zu einer `MPC`-Methode zu gehören. Damit ist `WiLi` eine *innere Klasse* (siehe Abschnitt 4.9.1.1), und wie bei anderen Klassenmitgliedern (z.B. Feldern und Methoden) ist ein Zugriffsmodifikator erlaubt und sinnvoll. Im Beispiel wird die Mitgliedsklasse `WiLi` durch den Zugriffsmodifikator **private** für den internen Gebrauch reserviert.

14.5.5 Schließen von Fenstern und Beenden von GUI-Programmen

Man muss nicht unbedingt einen eigenen Ereignisempfänger definieren, wenn beim Schließen eines **JFrame**-Fensters lediglich die Anwendung beendet werden soll. Seit Java 1.3 kann mit folgendem Aufruf der der **JFrame**-Methode **setDefaultCloseOperation()** für diese **WindowEvent**-Behandlung gesorgt werden:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Allerdings ist es nicht immer sinnvoll, ein Programm ohne weitere Maßnahmen zu beenden (z.B. ohne Prüfung auf ungesicherte Dokumente).

Ein **JFrame**-Fenster lässt sich auch dann vom Benutzer per Fenstertitelzeilensymbol oder Systemmenü schließen, wenn im Programm weder ein **WindowListener** registriert, noch die **setDefaultCloseOperation()** - Methode aufgerufen wird. Allerdings verschwindet dabei nur das *Fenster* vom Bildschirm, während die Anwendung weiterläuft. War die Anwendung z.B. aus einem Konsolenfenster über das Werkzeug **java.exe** gestartet worden, erhält der Benutzer *keine* neue Eingabeaufforderung. Dazu muss er die immer noch aktive Anwendung erst beenden, z.B. mit der Tastenkombination **Strg+C**.

Mit Hilfe eines **WindowEvent**-Handlers kann das Schließen eines **JFrame**-Fensters *nicht* verhindert werden, wir gewinnen vielmehr die Möglichkeit, auf dieses Ereignis zu reagieren. Um einer

JFrame-Komponente zu verbieten, auf Benutzerwunsch hin von der Bildfläche zu verschwinden, verwendet man folgenden Aufruf der **JFrame**-Methode **setDefaultCloseOperation()**:

```
frame.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
```

Der gerade begonnene Exkurs zur **Terminierung von GUI-Anwendungen** soll noch etwas fortgeführt werden. Im Beispielprogramm beschränkt sich die **main()**-Methode darauf, ein Objekt aus der Klasse `MPC` zu erzeugen, und endet nach sehr kurzer Zeit mit der Rückkehr des Konstruktoraufrufs:

```
public static void main(String[] args) {
 new MPC();
}
```

Während unsere *Konsolen*programme nach dem Verlassen der **main()**-Methode beendet waren, geht es beim Swing-Demoprogramm zu diesem Zeitpunkt erst richtig los. Wie lange der Spaß andauert, hängt vom Verhalten des Benutzers und von den Ereignisbehandlungsmethoden ab.

Um dieses, mit unseren bisherigen Vorstellungen unvereinbare Verhalten erklären zu können, müssen wir das Konzept der **Threads** (Ausführungsfäden) einbeziehen, das bei der Programmierung eine wichtige Rolle spielt und im weiteren Kursverlauf noch ausführlich zur Sprache kommen wird. Ein Programm (Prozess) kann in mehrere *nebenläufige Ausführungsfäden* zerlegt werden, was bei Java-Programmen regelmäßig geschieht. Nachdem Sie ein Java-Programm aus einer Konsole gestartet haben, können Sie unter Windows mit der Tastenkombination **Strg+Pause** eine Liste seiner aktiven Threads anfordern(, ohne das Programm dabei abzubrechen).

Ein Java-Programm (ob mit oder ohne GUI) endet genau dann, wenn eine der folgenden Bedingungen eintritt:

- Alle Benutzer-Threads sind abgeschlossen.
  Neben den Benutzer-Threads kennt Java noch so genannte *Daemon*-Threads, die ein Programm *nicht* am Leben erhalten können.
- Ein Thread ruft die Methode **System.exit()** oder die Methode **Runtime.exit()** auf, und der **Security Manager**[1] hat nicht gegen eine Beendigung des Programms einzuwenden. Nach der Instruktion

  ```
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  ```

  an eine **JFrame**-Komponente, ruft diese automatisch die Methode **System.exit()** auf, wenn das Fenster geschlossen wird.

Während ein Konsolenprogramm nur *einen* Benutzer-Thread besitzt (namens **main**), tauchen bei GUI-Programmen zusätzliche Benutzer-Threads auf, z.B. **AWT-EventQueue-0**. Sobald in der **main()**-Methode unseres Beispielprogramms das Anwendungsfenster (ein **JFrame**-Abkömmling) erzeugt wird, starten die GUI-Threads. Während anschließend mit der Methode **main()** auch der Thread **main** endet, leben die GUI-Threads weiter.

Dort befindet sich auch Referenzen auf das `MPC`-Objekt und seine Member-Objekte, so dass diese vom Garbage Collector unbehelligt bleiben.

[1] Diesen zentralen Bestandteil der Java-Sicherheitsarchitektur können wir aus Zeitgründen nicht behandeln.

14.5.6 Optionen zur Definition von Ereignisempfängern

In der aktuellen Java-Praxis werden meist anonyme Klassen als Ereignisempfänger verwendet. Seit Java 8 kann diese Praxis durch Lambda-Ausdrücke vereinfacht werden. Anschließend werden diese und weitere Optionen beschrieben.

14.5.6.1 Innere Klasse als Ereignisempfänger

Wer den vollständigen Quellcode des Swing-Einstiegsbeispiels aufmerksam liest, wird feststellen, dass die `WiLi`–Klassendefinition im Rumpf der `MPC`-Klassendefinition steht, was `WiLi` zur **eingeschachtelten Klasse** (engl. *nested class*) bzw. zur **Mitgliedklasse** macht. Weil die Klasse `WiLi` *keinen* **static**-Modifikator erhalten hat, ist es eine **innere Klasse** (vgl. Abschnitt 4.9.1.1):

```
class MPC {
  .  .  .
 private class WiLi extends WindowAdapter {
   @Override
   public void windowClosing(WindowEvent e) {
     if (JOptionPane.showConfirmDialog(e.getWindow(),
         "Wollen Sie nach "+ numClicks +
         " Klicks wirklich schon aufhören?",
         titel, JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION)
       System.exit(0);
     }
 }
}
```

Dank dieser Konstruktion können die privaten `MPC`-Instanzvariablen in den `WiLi`-Methoden angesprochen werden, was im Beispiel durch den Zugriff auf `numClicks` demonstriert wird. Trotz des nicht ganz überzeugenden Beispiels können Sie sich bestimmt vorstellen, welchen Nutzen innere Klassen gerade bei der Definition von Ereignisempfängern haben.

Im Vergleich zur anschließend vorgestellten Ereignisbehandlung durch eine anonyme Klasse, hat die Verwendung einer *inneren* Klasse den (selten relevanten) Vorteil, dass *mehrere* Objekte erzeugt werden können.

14.5.6.2 Anonyme Klasse als Ereignisempfänger

Nun haben wir unser Swing-Einstiegsbeispiel fast vollständig durchleuchtet mit Ausnahme der Ereignisbehandlung zur Schaltfläche:

```
cbCount.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
       numClicks++;
       lblText.setText(lblPrefix + numClicks);
       if (iconInd < icons.length-1)
       iconInd++;
       else
       iconInd = 0;
       lblIcon.setIcon(icons[iconInd]);
    }
  });
```

Hier wird bei der Ereignisquelle `cbCount` für die Ereignisklasse **ActionEvent** ein Empfänger registriert, wobei nicht nur das Objekt dynamisch erzeugt, sondern die gesamte Klasse an Ort und Stelle definiert wird. Es entsteht eine so genannte *anonyme Klasse* (siehe Abschnitt 12.1.1.2).

Wie bei einer inneren Klasse haben wir auch bei einer anonymen Klasse die Möglichkeit, in ihren Methoden auf Felder und Methoden der umgebenden Klasse zuzugreifen, was im Beispiel außerordentlich hilfreich ist.

Weil eine anonyme Klasse innerhalb einer Methode definiert wird, also eine *lokale Klasse* ist (vgl. Abschnitt 4.9.2), kann sie auch auf finale lokale Variablen der umgebenden Methode zugreifen. Java 8 hat in diesem Zusammenhang zwei Verbesserungen gebracht (vgl. Abschnitt 4.9.2):

- Es genügt die *effektive* Finalität.
- Der Zugriff gelingt auch bei *Parametern* der umgebenden Methode.

Von einer anonymen Klasse lässt sich nur *ein* Objekt erstellen, das aber sehr wohl als Ereignisempfänger bei mehreren Komponenten registriert werden kann.

Übrigens verwendet auch der im WindowBuilder von Eclipse enthaltene Quellcode-Generator anonyme Klassen zur Ereignisbehandlung, was schon in Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.** zu beobachten war.

14.5.6.3 Ereignisempfänger per Lambda-Ausdruck definieren

Wie man die traditionelle Ereignisbehandlung durch eine anonyme Klasse seit Java 8 mit Hilfe eines Lambda-Ausdrucks vereinfachen kann, wurde schon in Abschnitt 12.1.1.3 demonstriert. Hier ist die Lambda-Variante der Ereignisbehandlung für das Swing-Einstiegsbeispiel zu sehen:

```
cbCount.addActionListener(e ->  {
      numClicks++;
      lblText.setText(lblPrefix + numClicks);
      if (iconInd < icons.length-1)
      iconInd++;
      else
      iconInd = 0;
      lblIcon.setIcon(icons[iconInd]);
  });
```

Im Unterschied zu einer inneren oder anonymen Klasse kann man beim Lambda-Ausdruck keine Felder oder zusätzliche Methoden definieren, was nur selten ein Nachteil ist.

14.5.6.4 Do-It-Yourself – Ereignisbehandlung

Zur Behandlung der von Instanz-Komponenten oder vom Rahmenfenster generierten Ereignisse muss eine Anwendungsklasse nicht unbedingt *Fremd*klassen beauftragen, sondern kann den Job auch selbst übernehmen, sofern sie die erforderlichen Ereignis-Interfaces erfüllt. Dies wird in Abschnitt 14.5.7.2 an einem Beispielprogramm zum Umgang mit Mausereignissen demonstriert.

Ein Vorteil dieser Vorgehensweise besteht darin, dass eine Klasse eingespart wird, was wiederum das Laden des Programms prinzipiell beschleunigen sollte. Allerdings widerspricht die Definition von funktionsüberladenen Klassen dem ***S**ingle **R**esponsibility **P**rinciple* (SRP, siehe Abschnitt 4.1.1.1). Es drohen schlecht zu wartende Programme und Kopfschmerzen.

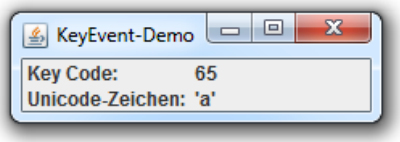
14.5.7 Tastatur- und Mausereignisse

14.5.7.1 Die Klasse KeyEvent für Tastaturereignisse

Das folgende Beispiel demonstriert den Umgang mit der Ereignisklasse **KeyEvent** und der Klasse **KeyAdapter**, die das Interface **KeyListener** implementiert. Es kommt eine anonyme Klasse zum Einsatz, die unter Angabe der Basisklasse definiert wird:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class KeyEventDemo {
 KeyEventDemo() {
  final JFrame frame = new JFrame("KeyEvent-Demo");

  final JLabel keyCode = new JLabel(" Key Code:");
  final JLabel uChar = new JLabel(" Unicode-Zeichen:");
  frame.add(keyCode, BorderLayout.NORTH);
  frame.add(uChar, BorderLayout.SOUTH);

  frame.addKeyListener(new KeyAdapter() {
   @Override
   public void keyPressed(KeyEvent e) {
    keyCode.setText(" Key Code:                "+e.getKeyCode());
   }
   @Override
   public void keyTyped(KeyEvent e) {
    uChar.setText(" Unicode-Zeichen:  '"+e.getKeyChar()+"'");
   }
   @Override
   public void keyReleased(KeyEvent e) {
    keyCode.setText(" Key Code:");
    uChar.setText(" Unicode-Zeichen:");
   }
  });

  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(240, 70);
  frame.setVisible(true);
 }

 public static void main(String[] args) {
  new KeyEventDemo();
 }
}
```

Das Programm verarbeitet als **KeyEvent**-Empfänger die bei aktivem **JFrame**-Rahmenfenster auftretenden Tastaturereignisse zu Protokollausgaben, z.B.:



Ein Tastendruck löst das Ereignis mit der EventID **KeyEvent.KEY_PRESSED** aus und bewirkt einen Aufruf der **KeyListener**-Methode **keyPressed()**, falls ein Empfänger registriert ist. Wird eine Taste losgelassen, kommt das Ereignis mit der EventID **KeyEvent.KEY_RELEASED** und ein Aufruf der **KeyListener**-Methode **keyReleased()** an registrierte Empfänger zustande. Beiden **KeyListener**-Methoden wird per Aktualparameter ein **KeyEvent**-Objekt übergeben. Mit der **KeyEvent**-Methode **getKeyCode()** bringt man den virtuellen Key Code der betroffenen Taste in Erfahrung (siehe API-Dokumentation zur Klasse **KeyEvent**).

Entspricht einer Taste(nkombination) ein Unicode-Zeichen, kommt es zum Ereignis mit der EventID **KeyEvent.KEY_TYPED** und damit zu einem Aufruf der **KeyListener**-Methode **keyTyped()** an registrierte Empfänger, der wiederum ein **KeyEvent**-Ereignisobjekt übergeben wird. Das Unicode-Zeichen lässt sich mit der **KeyEvent**–Methode **getKeyChar()** ermitteln.

14.5.7.2 Die Klasse MouseEvent für Mausereignisse

Im folgenden Beispiel wird der Umgang mit der Ereignisklasse **MouseEvent** sowie mit den (beiden!) zugehörigen Schnittstellen **MouseListener** und **MouseMotionListener** demonstriert. Außerdem wird gezeigt (wie in Abschnitt 14.5.6.4 angekündigt), dass eine Anwendungsklasse nicht unbedingt *Fremd*klassen mit der Ereignisbehandlung beauftragen muss, sondern den Job auch selbst übernehmen kann, sofern sie die erforderlichen Ereignis-Schnittstellen erfüllt:

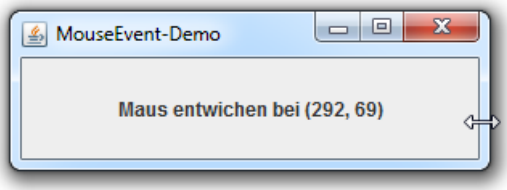
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MouseEventDemo implements MouseListener, MouseMotionListener {
   private final JFrame frame;
   private final JLabel status = new JLabel();
   private final Color defColor;

   MouseEventDemo() {
    frame = new JFrame("MouseEvent-Demo");

    status.setHorizontalAlignment(SwingConstants.CENTER);
    frame.getContentPane().add(status, BorderLayout.CENTER);

    frame.addMouseListener(this);
    frame.addMouseMotionListener(this);

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    defColor = frame.getContentPane().getBackground();

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 100);
    frame.setVisible(true);
   }

   public void mouseClicked(MouseEvent e) {
    status.setText("Mausklick bei ("+e.getX()+", "+e.getY()+")");
   }
```

```
    public void mousePressed(MouseEvent e) {
      frame.getContentPane().setBackground(Color.RED);
      status.setText("Maustaste gedrückt bei ("+e.getX()+", "+e.getY()+")");
    }
    public void mouseReleased(MouseEvent e) {
      frame.getContentPane().setBackground(defColor);
    }
    public void mouseEntered(MouseEvent e) {
      status.setText("Maus eingedrungen bei ("+e.getX()+", "+e.getY()+")");
    }
    public void mouseExited(MouseEvent e) {
      status.setText("Maus entwichen bei ("+e.getX()+", "+e.getY()+")");
    }
    public void mouseDragged(MouseEvent e) {
      status.setText("Maus gezogen bei ("+e.getX()+", "+e.getY()+")");
    }
    public void mouseMoved(MouseEvent e) {
      status.setText("Maus bewegt bei ("+e.getX()+", "+e.getY()+")");
    }

    public static void main(String[] arg) {
      new MouseEventDemo();
    }
}
```

Beim Registrieren der Ereignisempfänger gibt ein Objekt der Klasse `MouseEventDemo` mit dem Schlüsselwort **this** sich selbst an:

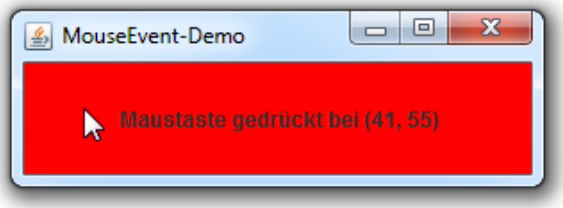
```
frame.addMouseListener(this);
frame.addMouseMotionListener(this);
```

Mit dem Programm lassen sich diverse Mausereignisse beobachten, wobei die Ereignisbehandlungsmethoden das übergebene **MouseEvent**-Objekt mit den Methoden **getX()** und **getY()** nach dem genauen Tatort befragen, z.B.:

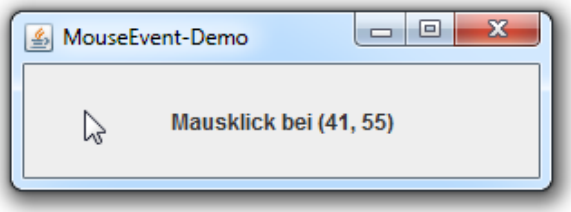


Beim Loslassen der Maustaste werden die Methoden **mouseReleased()** und **mouseClicked()** unmittelbar nacheinander aufgerufen. Um beide Methoden im Beispielprogramm sichtbar zu machen, wird die Hintergrundfarbe der Inhaltsschicht von der Methode **mousePressed()** auf Rot und von der Methode **mouseReleased()** wieder auf den Voreinstellungswert gesetzt. Wird die Maustaste gedrückt und wieder losgelassen, ergeben sich folgende Momentaufnahmen des Programms:

Nach dem Drücken der Maustaste



Nach dem Loslassen der Maustaste

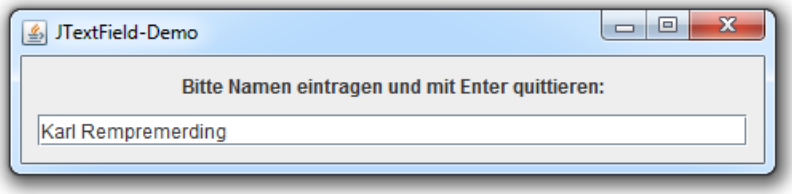


14.6 Bedienelemente (Teil 2)

In diesem Abschnitt werden einige weitere Swing-Komponenten vorgestellt. Eine Darstellung *aller* Komponenten verbietet sich aus Platzgründen und ist auch nicht erforderlich, weil Sie bei Bedarf in der API-Dokumentation und im Java-Tutorial (Oracle 2014) die benötigten Informationen finden.

14.6.1 Einzeiliges Texteingabefeld

Im Rahmen der **JOptionPane**-Klassenmethode **showInputDialog()** (vgl. Abschnitt 3.8) konnten Sie schon eine einzeilige Textkomponente besichtigen. In einem **JFrame**-Fenster kann dieses elementare Bedienelement mit Hilfe einer **JTextField**-Komponente realisiert werden, z.B.:



Der Quellcode des Beispielprogramms:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JTextFieldDemo {
 JTextFieldDemo() {
  final String titel = "JTextField-Demo";
  final JFrame frame = new JFrame(titel);
  final Container cont = frame.getContentPane();

  final JLabel label =
   new JLabel("Bitte Namen eintragen und mit Enter quittieren:");
  label.setHorizontalAlignment(JTextField.CENTER);
  label.setBorder(BorderFactory.createEmptyBorder(10, 0, 0, 0));
  cont.add(label, BorderLayout.NORTH);

  final JTextField name = new JTextField(40);
  name.addActionListener(
   new ActionListener() {
     public void actionPerformed(ActionEvent e) {
      JOptionPane.showMessageDialog(((JComponent)e.getSource()).getParent(),
       "Sie heißen "+name.getText(), titel, JOptionPane.INFORMATION_MESSAGE);
     }
    });
```

```
    final JPanel pan = new JPanel();
    pan.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
    pan.add(name);
    cont.add(pan, BorderLayout.CENTER);


    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
  }

  public static void main(String[] args) {
    new JTextFieldDemo();
  }
}
```

Im Beispielprogramm erfährt der **JTextField**-Konstruktor die gewünschte Spaltenzahl und berechnet daraus die bevorzugte Breite der Komponente, wobei für eine Spalte die Breite des Buchstabens *m* in der eingestellten Schriftart angenommen wird:

```
private final JTextField name = new JTextField(40);
```

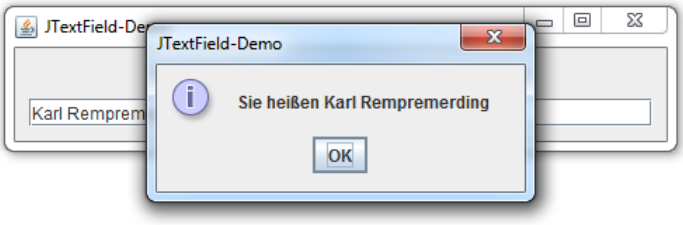
Um das Texteingabefeld zu beranden, kommt das in Abschnitt 14.3.5 vorgeschlagene Verfahren zum Einsatz:

- Es wird ein **JPanel**-Objekt erzeugt und mit dem gewünschten Rand versehen.
- Das Texteingabefeld wird in den **JPanel**-Container eingefügt.

Die gesamte Konstruktion landet schließlich in der Zentralregion des Inhaltsschicht-Containers im **JFrame**-Rahmenfenster, wo per Voreinstellung ein **BorderLayout** - Manager tätig ist:

```
final JPanel pan = new JPanel();
pan.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
pan.add(name);
cont.add(pan, BorderLayout.CENTER);
```

Sobald der Benutzer die **Enter**-Taste drückt, während eine **JTextField**-Komponente den Tastaturfokus besitzt, wird ein **ActionEvent** ausgelöst. Im Beispiel präsentiert der im Rahmen einer anonymen Klasse (vgl. Abschnitt 14.5.6.2) realisierte Event Handler daraufhin einen Benachrichtigungs-Standarddialog (vgl. Abschnitt 3.8) mit dem erfassten Text, den er über die **JTextField**-Methode **getText()** ermittelt:



Im ersten Aktualparameter des Methodenaufrufs **JOptionPane.showMessageDialog()** wird eine Referenz auf das elterliche Fenster des Standarddialogs übergeben:

```
((JComponent)e.getSource()).getParent()
```

Daran orientiert das Laufzeitsystem den Erscheinungsort des Dialogfensters. In früheren Beispielen haben wir mit dem Parameterwert **null** auf einen Ortswunsch verzichtet.

Statt der voreingestellten und im Beispiel angemessenen linksbündigen Ausrichtung des Textfeldinhalts kann mit der **JTextField**-Methode **setHorizontalAlignment()** auch eine zentrierte oder rechtsbündige Ausrichtung gewählt werden, z.B.:

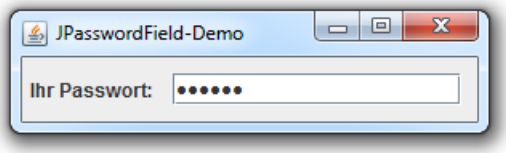
```
anzahl.setHorizontalAlignment(JTextField.RIGHT);
```

Rechtsbündige Textfelder sind z.B. bei der Erfassung von Zahlen zu bevorzugen.

Mit **setEditable(false)** wird für eine **JTextField**-Komponente festgelegt, dass sie vom Benutzer *nicht* geändert werden darf, was in bestimmten Situationen sinnvoll sein kann.

Ansonsten bringt das Beispielprogramm noch einen Nachtrag zur **JFrame**-Rahmenfensterkomponente: Statt wie in den bisherigen Beispielen die initiale Fenstergröße mit **setSize()** festzulegen, wird das Fenster über die (von **java.awt.Window** geerbte) Methode **pack()** aufgefordert, seine Größe passend zu den enthaltenen Komponenten einzurichten. Man erspart sich das lästige Schätzen der benötigten Fenstergröße und erhält auch nach späteren Änderungen bei der Komponentenausstattung ein gutes Erscheinungsbild.

Zum Erfassen von **Passwörtern** steht die von **JTextField** abstammende Komponente **JPasswordField** bereit, die im Unterschied zu ihrer Basisklasse für jedes eingegebene Zeichen einen Punkt anzeigt, z.B.:



Auch die Klasse **JPasswordField** besitzt einen Konstruktor, der die gewünschte Spaltenzahl entgegen nimmt und mit der Breite des Buchstabens *m* multipliziert, um so die bevorzugte Komponentenbreite zu ermitteln:

```
final JPasswordField pw = new JPasswordField(16);
```

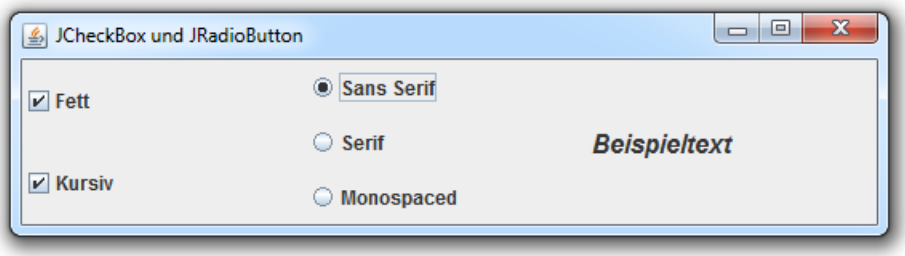
Das erfasste Passwort kann mit der **JPasswordField**-Methode **getPassword()** als **char**-Array extrahiert werden.

14.6.2 Umschalter

In diesem Abschnitt werden zwei Klassen für Umschalter vorgestellt, die beide von der Basisklasse **JToggleButton** abstammen:

- Für **Kontrollkästchen** steht die Swing-Komponente **JCheckBox** zur Verfügung.
- Für ein **Optionsfeld** verwendet man Komponenten vom Typ **JRadioButton**.

In folgendem Programm kann für den Text einer **JLabel**-Komponente über zwei Kontrollkästchen der Schriftschnitt und über ein Optionsfeld die Schriftart gewählt werden:



Den Quellcode des Programms finden Sie im Ordner

…\BspUeb\Swing\Umschalter

14.6.2.1 Kontrollkästchen

Im Beispielprogramm erhalten die beiden **JCheckBox**-Komponenten per Konstruktor eine Beschriftung:

```
private final JCheckBox chkBold, chkItalic;
private final JLabel lblBeispiel;
  .  .  .
final JFrame frame = new JFrame("JCheckBox und JRadioButton");
final Container cont = frame.getContentPane();
cont.setLayout(new GridLayout(1, 0));

final JPanel panCheck = new JPanel();
panCheck.setLayout(new GridLayout(0, 1));
cont.add(panCheck);

lblBeispiel = new JLabel("Beispieltext");
cont.add(lblBeispiel);

chkBold = new JCheckBox("Fett");
chkItalic = new JCheckBox("Kursiv");
panCheck.add(chkBold);
panCheck.add(chkItalic);

final CheckHandler cbHandler = new CheckHandler();
chkBold.addItemListener(cbHandler);
chkItalic.addItemListener(cbHandler);
```

Mit anderen Konstruktor-Überladungen lässt sich …

- alternativ oder ergänzend ein Bild vereinbaren
- ein initial markiertes Kontrollkästchen erzeugen

Aus Layout-Gründen werden die beiden Kontrollkästchen in einem eigenen **JPanel**-Container untergebracht, der am linken Rand des **JFrame**-Fensters Platz nehmen soll. Der bei einem **JFrame**-Fenster (genauer: bei seiner Inhaltsschicht) voreingestellte **BorderLayout** - Manager wird im Beispiel ersetzt durch einen **GridLayout** - Manager, der mit einer Zeile arbeitet (vgl. Abschnitt 14.4.2). In seine erste (linke) Zelle wird der **JPanel**-Container mit den Kontrollkästchen eingefügt.

Ändert sich der Zustand eines Umschalters, wird ein **ItemEvent** generiert, und die registrierten Ereignisempfänger erhalten die Botschaft **itemStateChanged()**. Dies ist die einzige Methode im Interface **ItemListener**, welches **ItemEvent**-Empfänger implementieren müssen.

Im Beispiel agiert für beide Kontrollkästchen dasselbe Objekt aus der intern definierten Klasse `CheckHandler` als **ItemEvent**-Empfänger:

```
private class CheckHandler implements ItemListener {
 public void itemStateChanged(ItemEvent e) {
  final int oldStyle = lblBeispiel.getFont().getStyle();
  int newStyle = 0;

  if (e.getSource() == chkBold) {
   if (e.getStateChange() == ItemEvent.SELECTED) {
```

```
        newStyle = oldStyle + Font.BOLD;
       } else {
        newStyle = oldStyle - Font.BOLD;
       }
      } else if (e.getSource() == chkItalic) {
       if (e.getStateChange() == ItemEvent.SELECTED) {
        newStyle = oldStyle + Font.ITALIC;
       } else {
        newStyle = oldStyle - Font.ITALIC;
       }
      }
      lblBeispiel.setFont(lblBeispiel.getFont().deriveFont(newStyle));
     }
    }
```

Der **ItemEvent**-Handler stellt mit **getSource()** die Ereignisquelle fest und passt sein Verhalten an. Er verwendet dabei einige Methoden zum Umgang mit Schriftarten, die in einem Java-Programm als Objekte der Klasse **Font** aus dem Paket **java.awt** vertreten sind:

- Die **Component**-Methode **getFont()** stellt die von einer Komponente verwendete Schriftart fest.
- Die **Font**-Methode **getStyle()** ermittelt den Stil (Schnitt) einer Schriftart, wobei die **int**-wertige Rückgabe folgendermaßen zu interpretieren ist:

int-Wert	Stil	**Font**-Konstante
0	Standard	PLAIN
1	Fett	BOLD
2	Kursiv	ITALIC
3	Fett + Kursiv	

- Mit den diversen Überladungen der **Font**-Methode **deriveFont()** gewinnt man eine neue Schriftart als Variante des angesprochenen **Font**-Objekts. Alle nicht per **deriveFont()** - Aktualparameter modifizierten Eigenschaften des angesprochenen Objekts werden übernommen. Man kann z.B. bequem eine neue Schrift erzeugen, die sich von einer bestimmten vorhandenen Schrift nur durch die Größe unterscheidet.
- Mit der **Component**-Methode **setFont()** wird die Schriftart einer Komponente festgelegt.

Ob das Quell-Kontrollkästchen ein- oder ausgeschaltet wurde, ermittelt der Event Handler mit der **ItemEvent**-Methode **getStateChange()**.

14.6.2.2 Optionsschalter

Auch die drei Optionsschalter des Umschalter-Beispielprogramms haben einen gemeinsamen **ItemListener**. Zudem sorgt ein Objekt aus der Klasse **ButtonGroup** dafür, dass stets nur *ein* Gruppenmitglied aktiviert ist:

```
private final JRadioButton rbSans, rbSerif, rbMono;
private final JLabel lblBeispiel;
private final Font fontSans = new Font(Font.SANS_SERIF, Font.PLAIN, 16),
                   fontSerif = new Font(Font.SERIF, Font.PLAIN, 16),
                   fontMono = new Font(Font.MONOSPACED, Font.PLAIN, 16);
  . . .
```

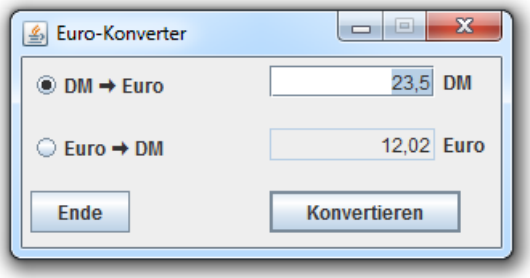
```
final JPanel panRadio = new JPanel();
panRadio.setLayout(new GridLayout(0, 1));
cont.add(panRadio);

lblBeispiel = new JLabel("Beispieltext");
cont.add(lblBeispiel);
lblBeispiel.setFont(fontSerif);

rbSans = new JRadioButton("Sans Serif", false);
rbSerif = new JRadioButton("Serif", true);
rbMono = new JRadioButton("Monospaced", false);

panRadio.add(rbSans); panRadio.add(rbSerif); panRadio.add(rbMono);

final ButtonGroup rbGroup = new ButtonGroup();
rbGroup.add(rbSans); rbGroup.add(rbSerif); rbGroup.add(rbMono);

final RadioHandler rbHandler = new RadioHandler();
rbSans.addItemListener(rbHandler);
rbSerif.addItemListener(rbHandler);
rbMono.addItemListener(rbHandler);
```

Im **ItemEvent**-Handler der Optionsschalter kommen die in Abschnitt 14.6.2.1 vorgestellten Schriftverwaltungsmethoden zum Einsatz:

```
private class RadioHandler implements ItemListener {
 public void itemStateChanged(ItemEvent e) {
  final int style = lblBeispiel.getFont().getStyle();
  Font font = null;
  if (e.getSource() == rbMono) {
   font = fontMono;
  } else if (e.getSource() == rbSerif) {
   font = fontSerif;
  } else if (e.getSource() == rbSans) {
   font = fontSans;
  }
  lblBeispiel.setFont(font.deriveFont(style));
 }
}
```

14.6.3 Standardschaltfläche und Tastaturfokus

In diesem Abschnitt werden Techniken zur Verwaltung von Tastaturereignissen behandelt, die von Bedeutung für die Bedienbarkeit eines Programms sind. Als Beispiel dient ein Programm zur Währungsumrechnung, das Sie als Übungsaufgabe erstellen sollen (siehe Abschnitt 14.8):



Für den mit **Konvertieren** beschrifteten Befehlsschalter (mit dem Referenzvariablennamen `cbKonvertieren`) sollte mit der schon in Abschnitt 13.5.2 vorgestellten Methode **setDefaultButton()** festgelegt werden, dass sich die **Enter**-Taste bei aktivem Anwendungsfenster an ihn richten und (wie ein linker Mausklick auf die Schaltfläche) ein **ActionEvent** auslösen soll:

```
frame.getRootPane().setDefaultButton(cbKonvertieren);
```

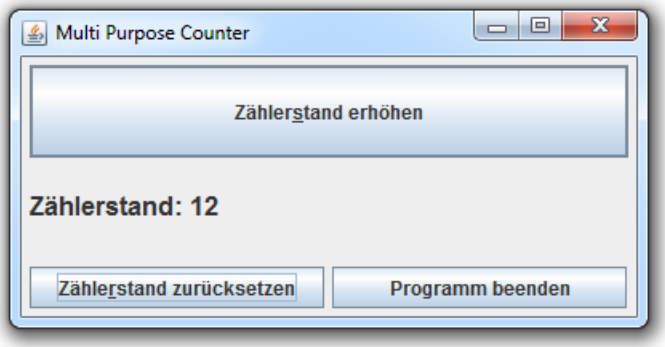
Ein **setDefaultButton()** - Aufruf ist an die **RootPane**-Schicht des **JFrame**-Fensters zu richten. Der Benutzer kann die **Standardschaltfläche** an einem betonten (breiten und dunklen) Rahmen erkennen (siehe oben) und manchen Griff zur Maus einsparen, was die Bedienung des Programms erleichtert.

Damit der Benutzer nach dem Programmstart sofort den ersten zu konvertierenden DM-Betrag eingeben kann, sollte die obere **JTextField**-Komponente (mit dem Referenzvariablennamen `tfEingabe`) über die **Component**-Methode **requestFocusInWindow()** den **Tastaturfokus** anfordern. Diese Anforderung muss erfolgen, *nachdem* die Komponente per **pack()** realisiert worden ist, und *bevor* das Fenster per **setVisible()** sichtbar gemacht worden ist, z.B.:

```
frame.pack();
tfEingabe.requestFocusInWindow();
frame.setVisible(true);
```

Ein Textfeld mit Tastaturfokus ist für die Benutzer an der dort vorhandenen Einfügemarke zu erkennen.

Viele primär zur Mausbedienung konzipierte Steuerelemente (z.B. Befehlsschalter, Kontrollkästchen) können auch per Tastatur benutzt werden, wenn sie den Tastaturfokus besitzen. In der Regel ist dieser Zustand optisch gekennzeichnet, z.B. durch einen Zusatzrahmen um die Beschriftung. Im folgenden Beispiel hat der Schalter mit der Beschriftung **Zählerstand zurücksetzen** den Tastaturfokus und reagiert daher auf die **Leertaste**, während die an einer fetten Umrahmung zu erkennende Standardschaltfläche mit **Zählerstand erhöhen** beschriftet ist und auf die **Enter**-Taste reagiert:

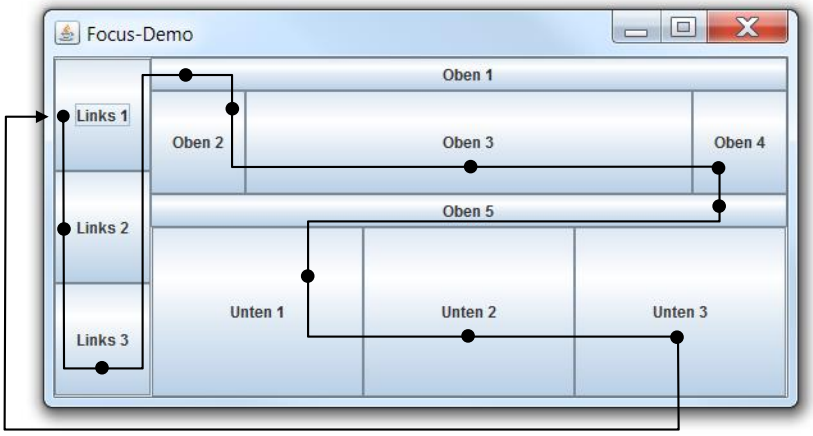


Um den Tastaturfokus auf ein geeignetes Steuerelement zu übertragen, können die Benutzer einen Mausklick darauf setzen, oder den Fokus per Tastatur in der Fokussequenz wandern lassen:

- Die Tabulatortaste ⇆ bewegt den Fokus zum nächsten Bedienelement in der Sequenz.
- Die Tastenkombination ⇧ + ⇆ bewegt den Fokus zum vorherigen Bedienelement in der Sequenz.

Die Fokussequenz für die in einem Container enthaltenen Steuerelemente wird durch ein Objekt aus der **FocusTraversalPolicy** - Klassenhierarchie festgelegt. Für die Top-Level-Container in Swing kommt per Voreinstellung die Klasse **LayoutFocusTraversalPolicy** zum Einsatz, und untergeord-

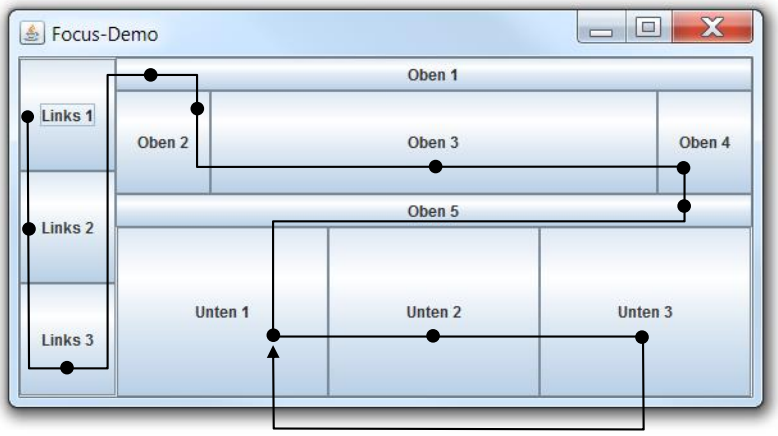
nete Container übernehmen diese Einstellung. Dabei resultiert eine sinnvolle Fokussequenz unter Berücksichtigung der Layout-Manager. Im folgenden Beispielprogramm arbeitet im Rahmenfenster das voreingestellte **BorderLayout**, wobei sich in seiner WEST- und seiner CENTER-Zelle **JPanel**-Container mit einem einspaltigen **GridLayout** - Manager befinden. Während der linke **JPanel**-Container drei Befehlsschalter enthält, befinden sich im rechten Container zwei untergeordnete **JPanel**-Container. Der obere Container verwendet ein mit fünf Befehlsschaltern befülltes **BorderLayout** und der untere ein mit drei Befehlsschaltern befülltes einzeiliges **GridLayout**. In der folgenden Abbildung ist die resultierende Fokussequenz zu sehen:



Gelegentlich ist es sinnvoll, für die in einem Container enthaltenen Komponenten eine *lokale* Fokusschleife zu bilden, die per ⇆ nicht mehr verlassen werden kann. Dazu muss der Container mit der Methode **setFocusCycleRoot()** und dem Aktualparameter **true** zur Wurzel einer Fokusschleife ernannt werden. Im Beispielprogramm wird dafür gesorgt, dass die drei Schalter im einzeiligen **GridLayout** (mit den Beschriftungen `Unten 1`, `Unten 2` und `Unten 3`) einen lokalen Zirkel bilden:

```
panUnten.setFocusCycleRoot(true);
```

Es resultiert die folgende Fokussequenz:



Statt eine **LayoutFocusTraversalPolicy**-Klasse aus der Standardbibliothek zu verwenden, kann man auch ein eigenes Regelwerk definieren (siehe Krüger & Hansen 2014, S. 237ff).

Ist für eine Komponente (aktuell) der Tastaturfokus nicht sinnvoll, kann sie mit der **Component**-Methode **setFocusable()** aus der Fokussequenz herausgenommen werden, z.B.:

```
tfAusgabe.setFocusable(false);
```

Mit der **Component**-Methode **transferFocus()** kann eine Komponente den Fokus an die nächste Station in der Fokussequenz weitergeben. Im Währungskonverterbeispiel (siehe oben) könnte ein Aufruf dieser Methode dazu genutzt werden, nach einer Konvertierung (**ActionEvent** beim Schalter `cbKonvertieren`) sofort die Eingabe des nächsten Betrags zu ermöglichen:

```
cbKonvertieren.transferFocus();
```

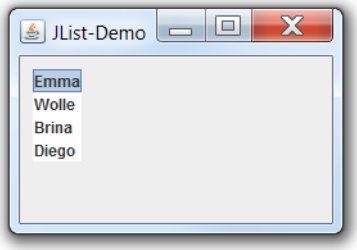
Eine direkte Fokus-Anforderung durch die Zielkomponente ist allerdings oft einfacher, weil die aktuelle Fokussequenz dabei keine Rolle spielt:

```
tfEingabe.requestFocus();
```

14.6.4 Listen

14.6.4.1 Einfach

Mit einer Komponente aus einer Konkretisierung der generischen Klasse **JList<E>** kann man dem Benutzer eine Liste mit markierbaren Einträgen präsentieren, z.B.:



Sind keine Listenänderungen im Programmablauf geplant, eignet sich z.B. beim Erstellen eines **JList<String>** - Objekts die Konstruktorüberladung mit einem Parameter vom Typ **String[]**, z.B.

```
JList<String> listFest =
 new JList<>(new String[] {"Eimer", "Wand", "Brille"});
```

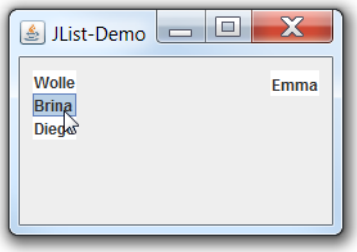
Ist Variabilität in der Listenzusammenstellung gefragt, stellt man dem **JList<E>** - Objekt ein Objekt zu Seite, dessen Klasse das **ListModel<E>** - Interface implementiert. Die beiden Objekte arbeiten als **Model-View** - Team zusammen, wobei …

- sich das **JList<E>** - Objekt um die Anzeige und die Benutzerinteraktion kümmert,
- das **ListModel<E>** - Objekt die Listeneinträge verwaltet (z.B. Elemente ergänzt oder löscht),
- beide (hinter den Kulissen) über **ListDataEvent**-Ereignisse kommunizieren

Statt eine eigene Klasse zu definieren und dabei das **ListModel<E>** - Interface zu implementieren, verwenden wir die Klasse **DefaultListModel<E>**, die alle benötigten Kompetenzen besitzt. Als Konkretisierung für den Typformalparameter verwenden wir die Klasse **String**:

```
private final DefaultListModel<String> modelLinks, modelRechts;
private final JList<String> listLinks, listRechts;
  .  .  .
modelLinks = new DefaultListModel<String>();
modelRechts = new DefaultListModel<String>();
modelLinks.addElement("Emma"); modelLinks.addElement("Wolle");
modelLinks.addElement("Brina"); modelLinks.addElement("Diego");
listLinks = new JList<String>(modelLinks);
listRechts = new JList<String>(modelRechts);
```

Zur Demonstration der möglichen Dynamik verwenden wir ein Beispielprogramm mit *zwei* Listen, wobei die linke Liste mögliche Teilnehmer eines Hunderennens enthält, und die rechte Liste initial leer ist. Durch Markieren (Anklicken) eines Listeneintrags soll der Benutzer dessen Wechsel in die jeweils andere Liste veranlassen können, z.B.:



Mit dem folgenden Methodenaufruf

```
listLinks.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

wird im Beispielprogramm veranlasst, dass nur *ein* Listenelement ausgewählt sein kann (analog zu einer Serie von Optionsschaltern). Daneben beherrscht die Klasse **JList<E>** auch:

- **ListSelectionModel.SINGLE_INTERVAL_SELECTION**
  Es kann eine Teilmenge hintereinander liegende Listeneinträge gewählt werden.
- **ListSelectionModel.MULTIPLE_INTERVAL_SELECTION**
  Es kann eine beliebige Teilmenge der Listeneinträge gewählt werden. Dies ist die Voreinstellung.

Wie die beiden Listen auf die von einem **BorderLayout**-Manager verwaltete Inhaltsschicht des **JFrame**-Fensters gelangen, muss nach etlichen Layout-Beispielen nicht mehr erläutert werden.

Durch die Wahl eines Eintrags initiiert der Benutzer beim betroffenen **JList<String>** - Objekt ein **ListSelectionEvent**, über das ein registrierter **ListEventListener** durch Aufruf seiner Methode **valueChanged()** informiert wird. Im Beispielprogramm wird bei beiden **JList<String>** - Komponenten ein Objekt der inneren Klasse `ListSelectionHandler` registriert. Seine **valueChanged()**-Methode entfernt aus der Sender-Liste das aktuell markierte Element und fügt es in die alternative Liste ein:

```
final ListSelectionHandler lsh = new ListSelectionHandler();
listLinks.addListSelectionListener(lsh);
listRechts.addListSelectionListener(lsh);
  .  .  .
```

```
private class ListSelectionHandler implements ListSelectionListener {
 public void valueChanged(ListSelectionEvent e) {
  if (e.getValueIsAdjusting() == true)
   return;
  if (((JList<String>)e.getSource()).getSelectedIndex() < 0)
   return;
  String wahl = ((JList<String>)e.getSource()).getSelectedValue();
  if (e.getSource() == listLinks) {
   modelLinks.removeElement(wahl);
   modelRechts.addElement(wahl);
  } else {
   modelRechts.removeElement(wahl);
   modelLinks.addElement(wahl);
  }
 }
}
```

Die `ListSelectionEvent`-Behandlung wird gleich abgebrochen, wenn die **getValueIs-Adjusting()** - Methode ein noch andauerndes Wählverhalten des Benutzers meldet:

```
if (e.getValueIsAdjusting() == true)
 return;
```

Im Beispielprogramm wird bei der Ereignisbehandlung, die durch das Markieren eines Elements angestoßen wird, genau dieses Element aus der Liste entfernt. Dabei ändert sich die Auswahl, so dass ein weiteres **ListSelectionEvent** stattfindet. Auf das Sekundärereignis soll nicht reagiert werden. Es ist daran zu erkennen, dass in der Senderliste kein Element markiert ist, so dass **get-SelectedIndex()** den Wert -1 liefert:

```
if (((JList<String>)e.getSource()).getSelectedIndex() < 0)
 return;
```

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

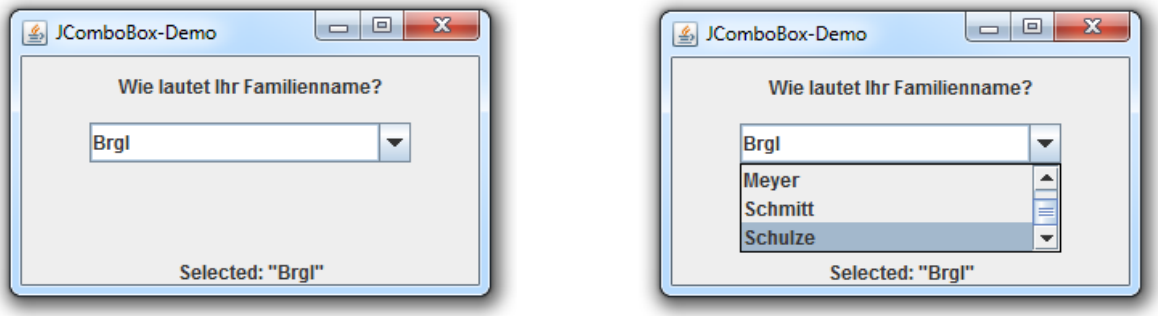
…\BspUeb\Swing\JList

14.6.4.2 Kombiniert

Die **JComboBox<E>** - Komponente bietet eine Kombination aus einem einzeiligen Textfeld und einer Dropdown-Liste, wobei normalerweise nur *ein* Element sichtbar ist. Um seine Wahl zu treffen, hat der Benutzer zwei Möglichkeiten:

- die Bezeichnung der gewünschten Option eintragen und mit **Enter** quittieren
- die Dropdown-Liste aufklappen und die gewünschte Option markieren

Im folgenden Programm wird die Angabe des Familiennamens durch eine Liste mit den häufigsten Namen erleichtert:



Die Kontrollausgabe am unteren Fensterrand ist allerdings nicht für Benutzer gedacht, sondern soll die Tätigkeit des **ItemEvent**-Handlers (siehe unten) sichtbar machen.

Zur Kreation eines Objekts der generischen Klasse **JComboBox<E>** stehen (wie bei der Klasse **JList<E>**, siehe Abschnitt 14.6.4.1) Konstruktoren zur Verfügung, die als Parameter einen Array mit den Listenelementen oder ein Listenverwaltungsobjekt im Sinne der Model-View - Architektur entgegen nehmen. Im Beispielprogramm wird die erste Alternative genutzt:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JComboBoxDemo {
 JComboBoxDemo() {
  final String titel = "JComboBox-Demo";
  final JFrame frame = new JFrame(titel);
  final Container cont = frame.getContentPane();

  final JLabel label = new JLabel("Wie lautet Ihr Familienname?");
  label.setHorizontalAlignment(JTextField.CENTER);
  label.setBorder(BorderFactory.createEmptyBorder(10, 0, 0, 0));
  cont.add(label, BorderLayout.NORTH);

  final String[] auswahl = {"Müller", "Meyer", "Schmitt", "Schulze"};
  final JComboBox<String> name = new JComboBox<String>(auswahl);
  name.setMaximumRowCount(3);
  name.setEditable(true);
  name.setSelectedItem("");

  final JPanel pan = new JPanel();
  name.setPreferredSize(new Dimension(200, 25));
  pan.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
  pan.add(name);
  cont.add(pan, BorderLayout.CENTER);

  final String lblSelectedPrefix = "Selected: ";
  final JLabel lblSelected = new JLabel(lblSelectedPrefix+"\"\"");
  lblSelected.setHorizontalAlignment(JTextField.CENTER);
  cont.add(lblSelected, BorderLayout.SOUTH);
```

```
    name.addItemListener(new ItemListener() {
        public void itemStateChanged(ItemEvent e) {
         if(e.getStateChange() == ItemEvent.SELECTED)
           lblSelected.setText(lblSelectedPrefix + "\"" + e.getItem() + "\"");
        }
       });

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 180);
    frame.setVisible(true);
  }

  public static void main(String[] args) {
    new JComboBoxDemo();
  }
}
```

Per Voreinstellung funktioniert ein **JComboBox<E>** - Bedienelement als reine *Dropdown*-Liste, bietet also *kein* Texteingabefeld. Dies wird im Beispiel mit dem Methodenaufruf **setEditable(true)** geändert. Außerdem wird mit dem Methodenaufruf **setSelectedItem("")** verhindert, dass in der Eingabezeile das erste Element der versteckten Liste als Vorgabe erscheint.

Im Beispiel wird das **JComboBox<String>** - Bedienelement aus folgenden Gründen in einen **JPanel**-Container gesteckt, der seinerseits im Zentrum des **BorderLayouts** zur Inhaltsschicht des **JFrame** - Top-Level-Container landet:

- Der voreingestellte **FlowLayout** - Manager des **JPanel**-Containers respektiert die bevorzugte Größe des **JComboBox<E>** - Objekts, die im Beispiel mit folgendem Aufruf der **JComponent**-Methode **setPreferredSize()** unter Verwendung eines **Dimension**-Objekts eingestellt wird:

  ```
  name.setPreferredSize(new Dimension(200, 25));
  ```

  Demgegenüber erzeugt der **BorderLayout**-Manager durch Verwendung der gesamten verfügbaren Fläche ein monströses Bedienelement, z.B.:

- Man kann per **setBorder()** einen Rand für den **JPanel**-Container und damit letztlich für das **JComboBox<E>** - Bedienelement festgelegen (siehe Abschnitt 14.6.1 zum Rand bei Bedienelementen).

Ein **JComboBox<E>** - Bedienelement erlaubt (wie die Komponenten aus den Klassen **JCheckBox** und **JRadioButton**) das Registrieren von **ItemEvent**-Empfängern, wobei die zugehörige Methode **itemStateChanged()** bei jeder Selektion *und* bei jeder Deselektion aufgerufen wird. Einen eingetippten Text wählt der Benutzer durch Quittieren mit der **Enter**-Taste.

Im Beispielprogramm wird der Empfänger nur bei einem positiven Auswahlereignis aktiv, wobei zur Differentialdiagnose die **ItemEvent**-Methode **getStateChange()** dient:

```
if(e.getStateChange() == ItemEvent.SELECTED) {
  .  .  .
}
```

Alternativ zum **ItemEvent**-Empfänger kann ein **ActionListener** registriert werden, der ein **ActionEvent** erhält, wenn der Benutzer ein Listenelement gewählt oder das Editieren der Textzeile abgeschlossen hat.

Über die Methode **setMaximumRowCount()** wird einem **JComboBox<E>** - Objekt mitgeteilt, wie viele Einträge es maximal anzeigen soll, z.B.:

```
name.setMaximumRowCount(3);
```

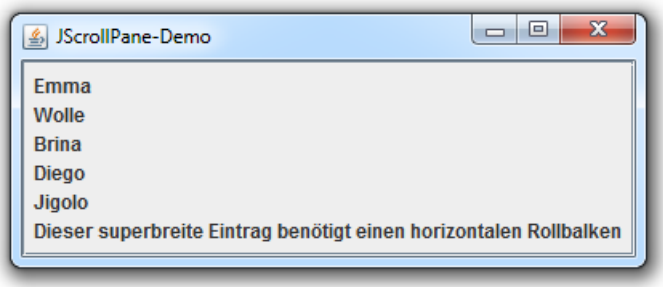
Sind mehr Einträge vorhanden, erscheint automatisch ein vertikaler Rollbalken (siehe Beispiel).

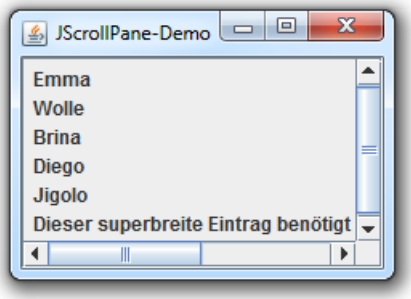
14.6.5 Rollbalken

Damit eine Komponente bei Platznot automatisch einen vertikalen und/oder horizontalen Rollbalken erhält, erstellt man ein **JScrollPane**-Objekt und beauftragt es mit der Anzeige der potentiell voluminösen Komponente. Im folgenden Beispielprogramm

```
import javax.swing.*;

class JScrollPaneDemo {
 JScrollPaneDemo() {
  final JFrame frame = new JFrame("JScrollPane-Demo");
  final JList<String> liste = new JList<String>(
   new String[] {"Emma","Wolle","Brina","Diego","Jigolo",
    "Dieser superbreite Eintrag benötigt einen horizontalen Rollbalken"});
  final JPanel panel = new JPanel();
  panel.add(liste);
  frame.getContentPane().add(new JScrollPane(panel), java.awt.BorderLayout.CENTER);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(225, 180);
  frame.setVisible(true);
 }

 public static void main(String[] args) {
  new JScrollPaneDemo();
 }
}
```

wird eine **JList<String>** - Komponente bei Bedarf durch eine **JScrollPane**-Komponente mit Rahmen versorgt:

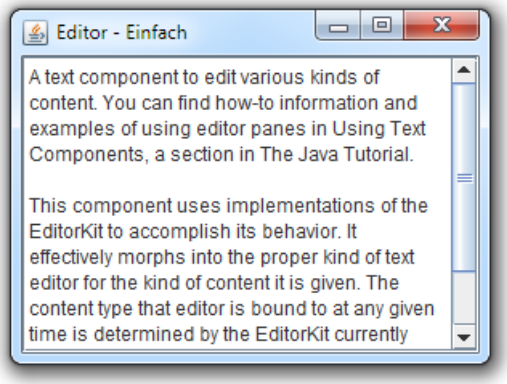




Ein mit Rollbalken versorgter **JPanel**-Container kann natürlich eine reichhaltige, verschachtelte Innenausstattung mit diversen Komponenten besitzen.

14.6.6 Ein (fast) kompletter Editor als Swing-Komponente

Ein Objekt aus der Klasse **JTextPane** als *Bedienelement* zu bezeichnen, ist ein wenig untertrieben, weil es sich hier um einen recht brauchbaren Texteditor handelt:

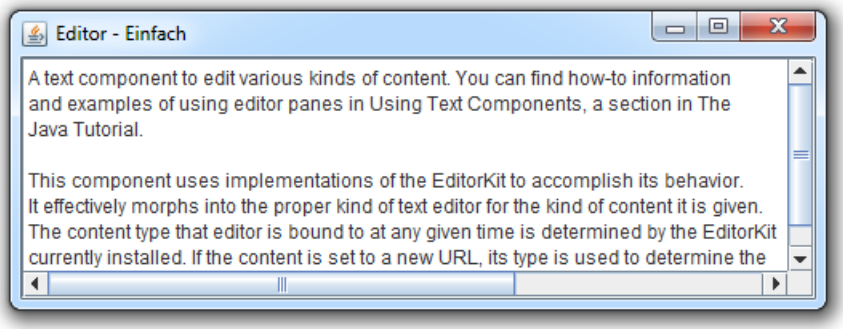


Für das abgebildete Programm, das immerhin schon den Datenaustausch via Zwischenablage beherrscht, muss man erstaunlich wenig Aufwand betreiben:

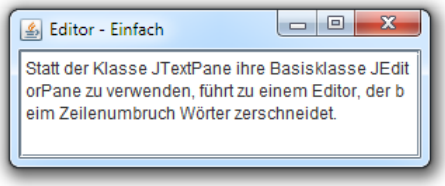
```
import javax.swing.*;

class Editor {
 public Editor() {
  final JFrame frame = new JFrame("Editor - Einfach");
  JTextPane text = new JTextPane();
  frame.getContentPane().add(new JScrollPane(text), java.awt.BorderLayout.CENTER);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(300, 200);
  frame.setVisible(true);
 }
 public static void main(String[] arg) {
  new Editor();
 }
}
```

Um die **JTextPane**-Komponente mit **Rollbalken** zu versorgen, wird sie in eine **JScrollPane**-Komponente verpackt. Eine **JTextPane**-Komponente verwendet (wie z.B. auch der Notepad-Editor in Windows) ausschließlich einen *vertikalen* Rollbalken.

Mit der **JTextPane**-Basisklasse **JEditorPane** erhält man auch einen *horizontalen* Rollbalken, z.B.:



Es kommt allerdings trotz Rollbalken manchmal zum Zeilenumbruch, wobei Wörter zerschnitten werden, z.B.:

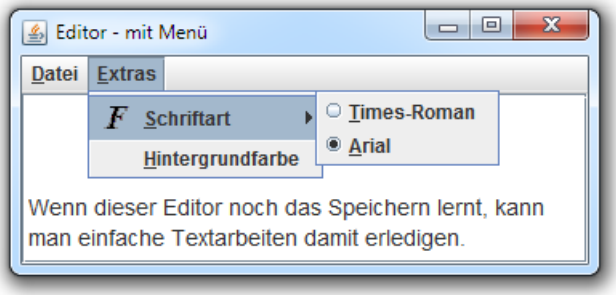


Wir verwenden daher die Klasse **JTextPane** und verzichten auf den horizontalen Rollbalken.

In den nächsten Abschnitten werden wir unserem Editor in Richtung Praxistauglichkeit weiterentwickeln.

14.6.7 Menüzeile, Menü und Menü-Item

Um das Potential der **JTextPane**-Komponente besser auszuschöpfen, erweitern wir das in Abschnitt 14.6.6 begonnene Programm um ein Datei- und ein Extras-Menü:



14.6.7.1 Menüzeile

Als Menüzeile verwenden wir einen Spezial-Container der Klasse **JMenuBar**, den ein Aufruf der **JFrame**-Methode **setJMenuBar()** auf die Layered Pane des **JFrame**-Fensters befördert (siehe Abschnitt 14.1.2 zum Aufbau eines Top-Level - Containers):

```
final JMenuBar menuBar = new JMenuBar();
frame.setJMenuBar(menuBar);
```

14.6.7.2 Menü

Die Menüs werden als Komponenten der Klasse **JMenu** erstellt, bei Bedarf mit einer **Alt**-Tastenkombination zum schnellen Öffnen versehen und dann auf die Menüzeile gesetzt oder in ein Untermenü aufgenommen (siehe unten):

```
final JMenu fileMenu = new JMenu("Datei");
fileMenu.setMnemonic(KeyEvent.VK_D);
menuBar.add(fileMenu);
```

Eine **JMenu**-Komponente kann über ihre **add()** - Methode andere **JMenu**-Komponenten als Untermenüs aufnehmen, z.B.:

```
final JMenu toolsMenu = new JMenu("Extras");
toolsMenu.setMnemonic(KeyEvent.VK_E);
final JMenu fontMenu = new JMenu("Schriftart");
fontMenu.setMnemonic(KeyEvent.VK_S);
toolsMenu.add(fontMenu);
```

Um ein Menü mit einem Icon zu verzieren, kann man so vorgehen:

- Man erstellt eine Grafikdatei passender Größe in einem unterstützten Format (GIF, JPEG, PNG) oder findet ein legal verwendbares Exemplar. Die Firma Oracle stellt zahlreiche Icons (meist in einer Größe von 16 × 16 bzw. 24 × 24 Pixeln) auf der folgenden Webseite zur Verfügung:

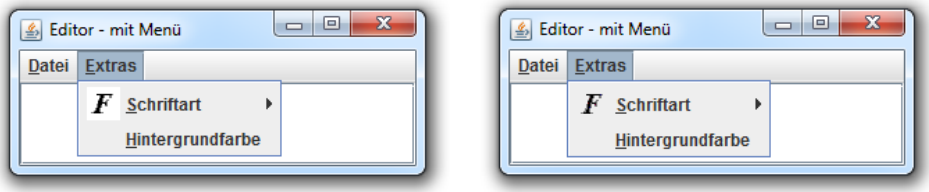
  http://www.oracle.com/technetwork/java/index-138612.html

- Für das `fontMenu` sollte ein bescheidenes, selbst erstelltes Icon (mit 24 × 24 Pixeln) genügen:

  In der Regel ist es sinnvoll, die Hintergrundfarbe eines Icons transparent zu setzen, z.B.:

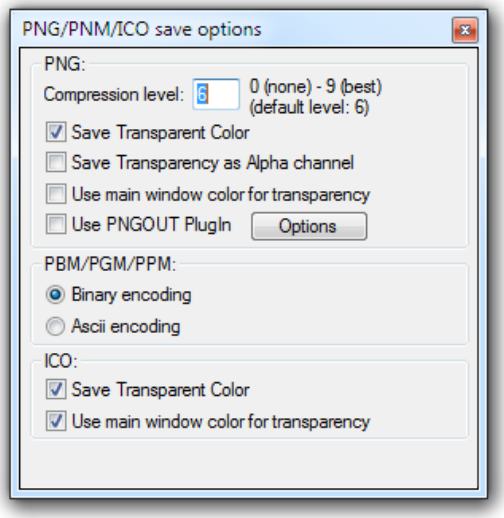
  ohne Transparenz mit Transparenz

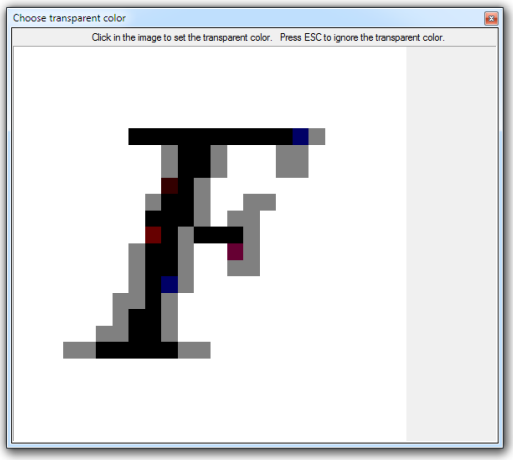
  Dies kann z.B. mit diversen kostenlosen Programmen geschehen, unter Windows z.B. mit **IrfanView**. Dort erscheint nach Wahl des Menübefehls

  File > Save as

  neben dem Dateiauswahldialog ein Zusatzfenster, wo das Kontrollkästchen **Save Transparent Color** markiert werden muss,

  damit nach dem Quittieren des Dateiauswahldialogs im folgenden Fenster die Transparenzfarbe per Mausklick festgelegt werden kann:



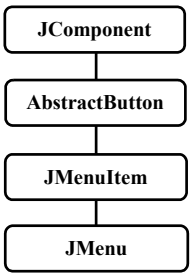
- Schließlich wird aus dem Icon ein **ImageIcon**-Objekt erzeugt und dem Menü mit der **AbstractButton**-Methode **setIcon()** zugeordnet, z.B.:

```
fontMenu.setIcon(new ImageIcon("font.gif"));
```

Analog werden wir gleich das Menü-Item **Datei > Öffnen** verzieren.

14.6.7.3 Menü-Item

Menüeinträge, die nicht für Untermenüs sondern für wählbare Aktionen stehen sollen, werden über Komponenten der Klasse **JMenuItem** realisiert. Diese Klasse stammt von **AbstractButton** ab und ist die Basisklasse von **JMenu**:



Ihre Objekte reagieren mit einem Ereignis der Klasse **ActionEvent**, wenn sie vom Benutzer gewählt werden.

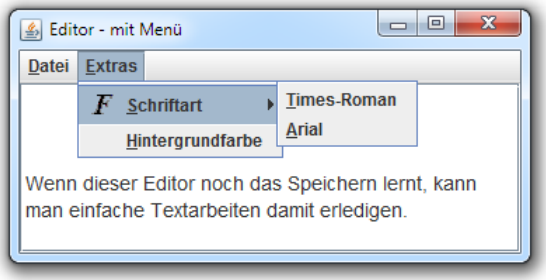
Jede **JMenu**-Komponente kann mit ihrer **add()**-Methode neben Untermenüs auch Menü-Items aufnehmen, z.B.:

```
final JMenuItem timesItem = new JMenuItem("Times-Roman");
timesItem.setMnemonic(KeyEvent.VK_T);
fontMenu.add(timesItem);
```

Für jedes Menü-Item wird ein **ActionListener** registriert, z.B. unter Verwendung einer anonymen Klasse:

```
timesItem.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
   text.setFont(new Font(Font.SERIF, Font.PLAIN, 14));
  }
 });
```

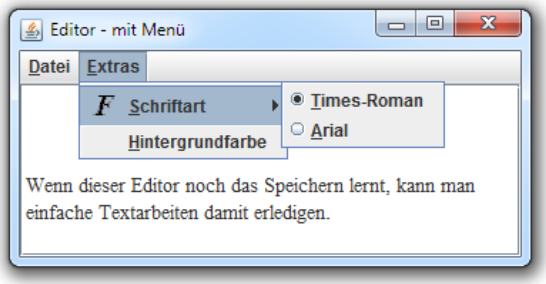
Das Schriftartenmenü des Beispielprogramms enthält Menü-Items mit Optionsschalterlogik, was aber bisher optisch nicht zum Ausdruck kommt:



Um den Mangel zu beheben, ersetzen wir die Klasse **JMenuItem** durch ihre Ableitung **JRadioButtonMenuItem**

```
final JMenuItem timesItem = new JRadioButtonMenuItem("Times-Roman");
final JMenuItem arialItem = new JRadioButtonMenuItem("Arial");
```

und sorgen dafür, dass im aufgeklappten Menü die aktuell gewählte Schriftart markiert ist, z.B.:



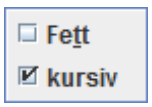
Mit einem **ButtonGroup**-Objekt (vgl. Abschnitt 14.6.2.2) wird sichergestellt, dass stets nur *ein* Schriftart-Menü-Item markiert ist:

```
final ButtonGroup bgFonts = new ButtonGroup();
bgFonts.add(timesItem);
bgFonts.add(arialItem);
```

Bei der Wahl eines Schriftart-Menü-Items sorgt der Ereignisempfänger in seiner Methode **actionPerformed()** dafür, dass der **JTextPane**-Komponente die gewünschte Schriftart und dem gewählten Menü-Item die Markierung zugewiesen wird, z.B.:

```
timesItem.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
   text.setFont(new Font(Font.SERIF, Font.PLAIN, 14));
   timesItem.setSelected(true);
  }
 });
```

Soll bei einem Menü-Item mit Umschaltlogik ein Kontrollkästchen den aktuellen Schaltzustand anzeigen, verwendet man die Klasse **JCheckBoxMenuItem**, z.B. mit dem folgenden Ergebnis:



Um ein Menü-Item mit einem Icon zu verzieren, geht man genauso vor wie bei einem Menü (siehe Abschnitt 14.6.7.2). In der folgenden Anweisung erhält das Editor - Menü-Item **Datei > Öffnen** ein Icon unter Verwendung der Datei **open.gif**:

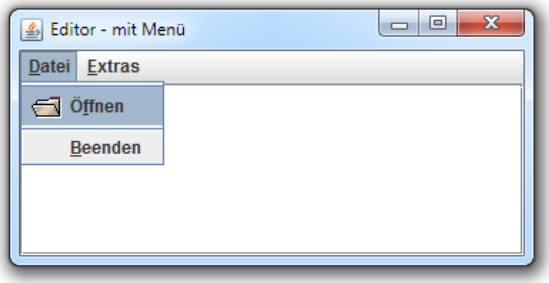
```
openItem.setIcon(new ImageIcon("open.gif"));
```

14.6.7.4 Separatoren

Um eine Trennlinie zwischen zwei
Menüeinträgen zu erzeugen, ruft man an passender Stelle die **JMenu**-Methode **addSeparator()** auf, z.B.:

```
fileMenu.addSeparator();
```

Bis zur Praxistauglichkeit unseres Editors ist noch einiges zu tun, doch wir kommen voran:



14.6.8 Standarddialog zur Dateiauswahl

Im **ActionEvent**-Handler zum Menü-Item

Datei > Öffnen

unseres Editors

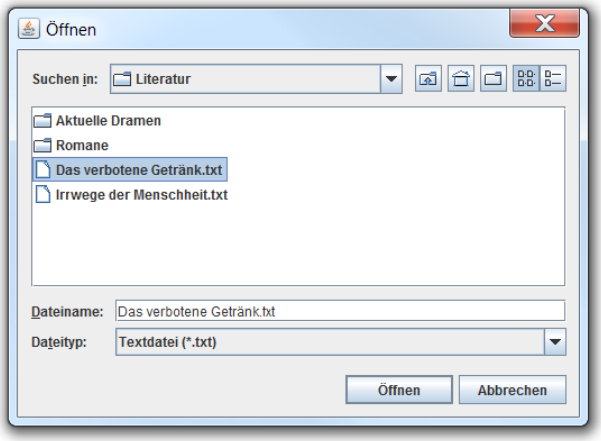
```
openItem.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
       getFileName();
       if (file != null)
         readText();
     }
   });
```

wird die private Methode `getFileName()` aufgerufen. Diese erkundigt sich beim Benutzer nach der zu öffnenden Datei und verwendet ein Objekt der Klasse **JFileChooser**, um einen Standarddialog zur Dateiauswahl anzubieten:

```
private void getFileName() {
 final FileNameExtensionFilter filter =
  new FileNameExtensionFilter("Textdatei (*.txt)", "txt");
 final JFileChooser fc = new JFileChooser();
 fc.setFileFilter(filter);
 if (fc.showOpenDialog(frame) == JFileChooser.APPROVE_OPTION)
  file = fc.getSelectedFile();
}
```

Mit einem Objekt der Klasse **FileNameExtensionFilter** und der **JFileChooser**-Methode **setFileFilter()** wird dafür gesorgt, dass per Voreinstellung (neben Ordnern) nur Dateien mit der Namenserweiterung **.txt** angezeigt werden.

Wir erhalten ohne großen Aufwand einen passablen Dialog



mit wichtigen Bedienungsoptionen für die Benutzer, z.B.

- Unterordner per Doppelklick öffnen
- zur nächst höheren Ebene wechseln
- **Dateityp** ändern
- neuen Ordner erstellen
- Ansicht zwischen Liste und Details umschalten
- gewählte Datei **öffnen**.

Im **ActionListener** zum `openItem` wird nach dem erfolgreichen Ermitteln einer zu öffnenden Datei mit der Methode `readText()` deren Inhalt eingelesen, wobei die wesentliche Arbeit von der **JTextPane**-Methode **read()** übernommen wird:[1]

```
private void readText() {
 try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(file)) ) {
  text.read(bis, null);
 } catch (final Exception e) {
  JOptionPane.showMessageDialog(frame,
   "Fehler beim Lesen:\n" + e, "Fehler",
    JOptionPane.ERROR_MESSAGE);
 }
}
```

Mit den Klassen **BufferedInputStream** und **FileInputStream** werden wir uns in Kapitel 15 ausführlich beschäftigen.

Den vollständigen Quellcode zur aktuellen Ausbaustufe des Beispielprogramms finden Sie im Ordner

…\BspUeb\Swing\Editor\E1 (mit Menü)

Neben dem Dateiauswahldialog sowie den Nachrichten- bzw. Bestätigungsdialogen der Klasse **JOptionPane** (vgl. Abschnitt 3.8) kennt die Swing-Bibliothek als weiteren Standarddialog noch die Farbauswahl über die Klasse **JColorChooser**. Im Rahmen einer Übungsaufgabe (siehe Abschnitt 14.8) sollen Sie mit Hilfe dieser Klasse einen **ActionListener** zum Menüitem **Extras > Hintergrundfarbe** realisieren.

---

[1] Diese Methode verwendet die ANSI-Kodierung (IANA-Bezeichnung: **Cp1252**, vgl. Abschnitt 15.4.1.2), was zu Problemen beim Lesen von Dateien mit UTF-8 - Kodierung führt.

14.6.9 Symbolleisten

Besonders häufig benötigte Funktionen bieten GUI-Programme meist (auch) über Symbolleisten an. In Swing werden Symbolleisten über Container der Klasse **JToolbar** realisiert. Hier bringt man in der Regel Symbolschalter aus der altbekannten Klasse **JButton** unter (siehe Abschnitt 13.5.2), doch sind auch andere Komponenten erlaubt (z.B. **JTextField**).

Damit eine Symbolleiste vom Benutzer per Maus abgerissen und neu positioniert werden kann, sollte sie zu einem Container mit **BorderLayout** gehören (vgl. Abschnitt 14.4.1):

- Die über Symbolleistenschalter veranlassten Aktionen sollten die Komponente im **BorderLayout**-Zentrum betreffen.
- Die Symbolleiste sollte sich initial an einer peripheren Position des **BorderLayout**s befinden (Norden, Osten, Süden oder Westen).

In unserem Editor, den wir seit Abschnitt 14.6.6 als Demonstrationsbeispiel benutzen, benutzt das Rahmenfenster ein **BorderLayout** mit der **JTextPane**-Komponente im Zentrum. Wir ergänzen nun eine Symbolleiste, die das Öffnen einer Textdatei, das Beenden des Programms und die Wahl einer Hintergrundfarbe erlaubt, wobei es sich um eine willkürliche Zusammenstellung ohne Anspruch auf vorbildliche Bedienungslogik handelt.

Die Symbolleisten-Komponente erhält per Konstruktor eine Titelzeilenbeschriftung für den selbständigen Zustand und startet im Norden des **BorderLayout**s:

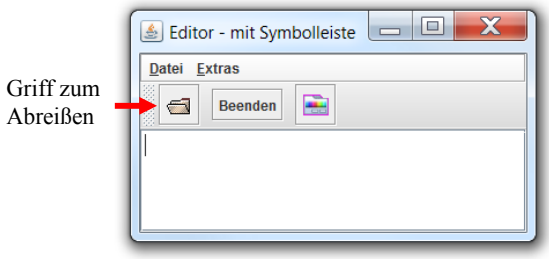
```
final JToolBar toolBar = new JToolBar("Editor");
frame.getContentPane().add(toolBar, BorderLayout.NORTH);
```

Um Menüzeile und Symbolleiste optisch zu trennen, erhält die Menüzeile einen Rahmen:

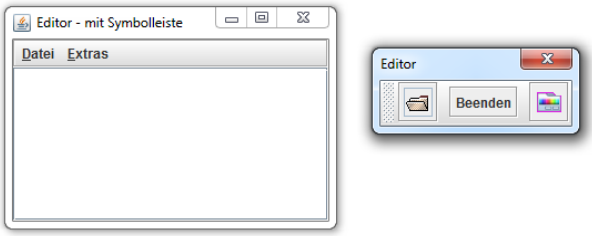
```
menuBar.setBorder(BorderFactory.createEtchedBorder());
```

Unter diesen Voraussetzungen lässt sich die Symbolleiste aus der initialen Position lösen, im frei schwebenden Zustand halten oder an einer (anderen) peripheren Position anheften:

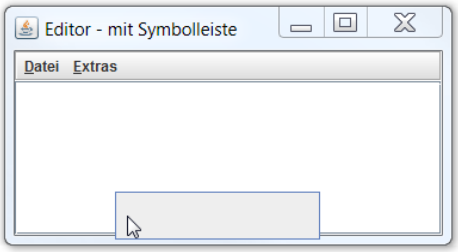
Symbolleiste im Norden angeheftet



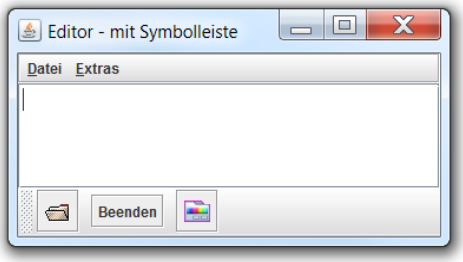
Frei schwebende Symbolleiste



Benutzer wünscht die südliche Position



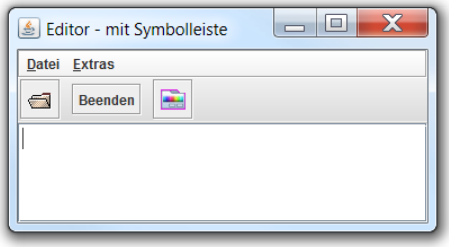
Symbolleiste im Süden angeheftet



Ist die voreingestellte Beweglichkeit unerwünscht, kann sie mit der **JToolBar**-Methode **setFloatable()** für eine bestimmte Symbolleiste verhindert werden, z.B.:

```
toolBar.setFloatable(false);
```

Benutzer erkennen eine *fixierte* Symbolleiste am Fehlen einer Griffzone:



Es ist nachzutragen, wie die eben schon zu bewundernde Ausstattung der Symbolleiste vorgenommen wurde. In der Regel erhalten Symbolleistenschalter ein Icon (z.B. in der Größe 24 × 24), doch sind auch Beschriftungen möglich:

```
final JButton openButton = new JButton(new ImageIcon("open.gif"));
toolBar.add(openButton);
final JButton exitButton = new JButton("Beenden");
toolBar.add(exitButton);
final JButton colorButton = new JButton(new ImageIcon("color.gif"));
toolBar.add(colorButton);
```

Die Symbolleistenschalter können den **ActionEvent**-Empfänger jeweils vom funktionsgleichen Menü-Item übernehmen:

```
openButton.addActionListener(openItem.getActionListeners()[0]);
colorButton.addActionListener(colorItem.getActionListeners()[0]);
colorButton.addActionListener(colorItem.getActionListeners()[0]);
```

Wie bei Menüeinträgen kann auch bei Symbolleistenelementen ein Separator zur Gruppierung verwendet werden, z.B.:

```
toolBar.addSeparator();
```

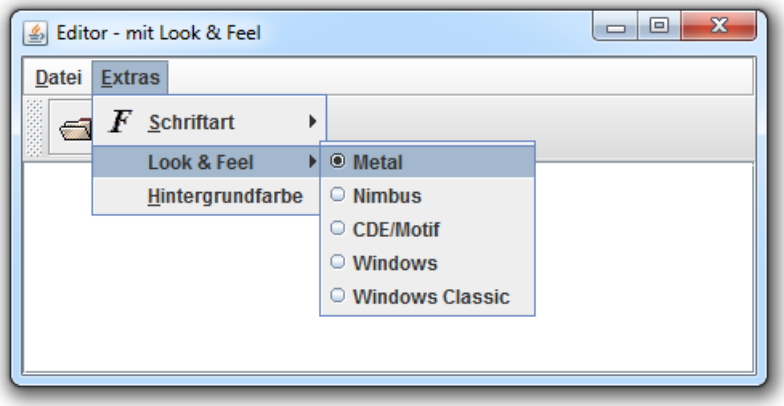
14.7 Weitere Swing-Techniken

14.7.1 Look & Feel umschalten

Wer den in Abschnitt 14.6.8 gezeigten Dateiauswahldialog lieber im Windows-Design erleben möchte, kann die Swing-Option nutzen, das Look & Feel einer Anwendung zwischen folgenden Alternativen umzuschalten (Stand: Java 8, Update 73, unter Windows 7):

- **Metal**
  Traditioneller Swing-Standard
- **Nimbus** (aufpoliertes Swing-Design)
  Mit Java 6 Update 10 wurde Nimbus als aufpolierte Alternative zu Metal eingeführt.
- **CDE/Motif**
  Ein traditioneller X-11 – Window-Manager unter UNIX
- **Windows**
- **Windows Classic**

Bevor man ein solches Look & Feel - Menü präsentieren kann,



ist etwas Arbeit angesagt.

Neben einem **JMenu**-Objekt und einem **JRadioButtonMenuItem**-Array zur Aufnahme der aktuell verfügbaren Look & Feel - Optionen deklarieren wir einen Array der Klasse **UIManager.LookAndFeelInfo**, die als statische Mitgliedklasse (vgl. Abschnitt 4.9.1.2) im Rumpf der Klasse **UIManager** definiert ist und daher einen Doppelnamen trägt:

```
private final JRadioButtonMenuItem[] lafItems;
private final UIManager.LookAndFeelInfo[] lafs;
```

Im Konstruktor der Anwendung werden die Objekte generiert und initialisiert:

```
final JMenu lafMenu = new JMenu("Look & Feel");
lafs = UIManager.getInstalledLookAndFeels();
lafItems = new JRadioButtonMenuItem[lafs.length];
final ButtonGroup bgLaf = new ButtonGroup();
final UIActionHandler uah = new UIActionHandler();
for (int i = 0; i < lafs.length; i++) {
 lafItems[i] = new JRadioButtonMenuItem(lafs[i].getName());
 bgLaf.add(lafItems[i]);
 lafItems[i].addActionListener(uah);
 if (UIManager.getLookAndFeel().getName() == lafs[i].getName())
  lafItems[i].setSelected(true);
 lafMenu.add(lafItems[i]);
}
```

Durch einen Aufruf der statischen **UIManager**-Methode **getInstalledLookAndFeels()** füllen wir den **LookAndFeelInfo**-Array und verwenden die Informationen anschließend zum dynamischen Menüaufbau. Anzahl und Inhalt der Items im `lafMenu` hängen also von der Ausstattung der aktuellen JVM ab, so dass unser Editor von zukünftigen Neuerungen profitieren wird. Mit der **UIManager**-Methode **getLookAndFeel()** wird aktuelle Look & Feel ermittelt, um das zugehörige **JRadioButtonMenuItem** als gewählt markieren zu können.

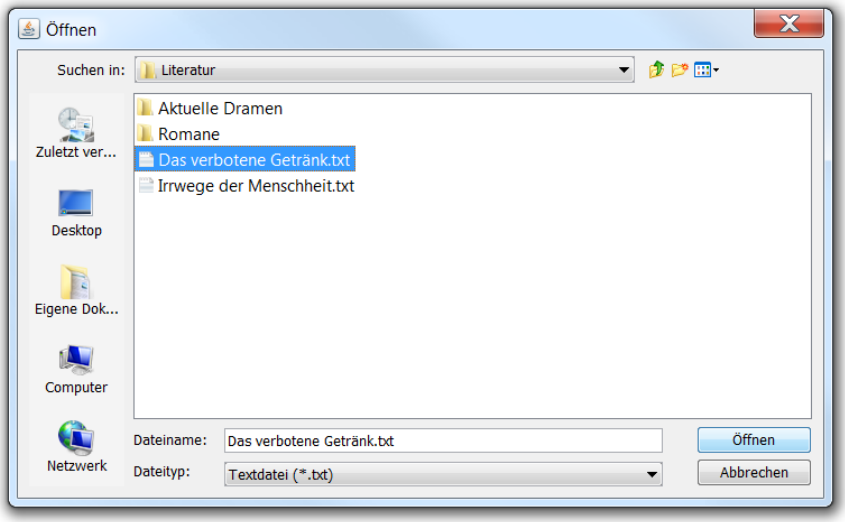
Für den **ActionListener** zu den Look & Feel - Menüitems definieren wir eine interne Klasse namens `UIActionHandler` (siehe Abschnitt 4.9.1.1 zu inneren Klassen):

```
private class UIActionHandler implements ActionListener {
 public void actionPerformed(ActionEvent e) {
  for (int i = 0; i < lafs.length; i++) {
   if (e.getSource() == lafItems[i])
    try {
     UIManager.setLookAndFeel(lafs[i].getClassName());
     SwingUtilities.updateComponentTreeUI(Editor.this.frame);
     lafItems[i].setSelected(true);
     break;
    } catch (Exception ex) {
     JOptionPane.showMessageDialog(text, "Fehler beim UI-Management", titel,
       JOptionPane.ERROR_MESSAGE);
    }
  }
 }
}
```

Dem Aufruf der statischen **UIManager**-Methode **setLookAndFeel()** wird als Aktualparameter passend zum gewählten Menüitem der Name der Klasse übergeben, die das gewünschte Look & Feel realisiert. Um das neue Look & Feel wirksam werden zu lassen, ist noch ein Aufruf der statischen **SwingUtilities**-Methode **updateComponentTreeUI()** erforderlich, die als Parameter eine Referenz auf das Rahmenfenster benötigt. Die innere Klasse `UIActionHandler` besitzt über das Schlüsselwort **this** mit vorangestelltem Klassennamen eine Referenz auf das Anwendungsobjekt und kann auf dessen Instanzvariable `frame`, die das Anwendungsfenster repräsentiert, zugreifen:

```
SwingUtilities.updateComponentTreeUI(Editor.this.frame);
```

Bei aktivem **Windows** - Look & Feel verwendet der Editor den folgenden Dateiauswahldialog:



Den vollständigen Quellcode zur aktuellen Ausbaustufe des Editors finden Sie im Ordner

…\BspUeb\Swing\Editor\E4 (mit Look & Feel)

Ist Nimbus als Look & Feel eingestellt, misslingt die Wahl der Hintergrundfarbe für die **JTextPane**-Komponente über die Methode **setBackground()**. Eine zuvor unter einer anderen Look & Feel - Einstellung gewählte Hintergrundfarbe verschwindet beim Wechsel zu Nimbus. In der API-Dokumentation wird auf solche Einschränkungen hingewiesen:[1]

[1] Siehe: http://docs.oracle.com/javase/8/docs/api/javax/swing/JComponent.html

It is up to the look and feel to honor this property, some may choose to ignore it.

14.7.2 Zwischenablage

Mit Hilfe der System-Zwischenablage (vom Betriebssystem verwaltet) können Anwender auf bequeme Weise Daten zwischen Anwendungen oder auch innerhalb einer Anwendung übertragen. In Abhängigkeit von den beteiligten Programmen werden dabei unterschiedliche Datenformate unterstützt (z.B. Zeichenfolgen, Bitmaps, Dateilisten). Markierte Daten werden per (Kontext)menü oder Tastaturbefehl (unter Windows: **Strg+C** bzw. **Strg+X**) in die Zwischenablage kopiert und beim Ausschneiden anschließend am alten Ort gelöscht. Viele Quellanwendungen legen die Daten sogar in mehreren Formaten in der Zwischenablage ab (z.B. einfachen und formatierten Text). Beim ebenfalls per (Kontext)menü oder Tastaturbefehl (unter Windows: **Strg+V**) anzufordernden Einfügen prüft die Zielanwendung, ob sie eines der verfügbaren Formate verarbeiten kann.

Die von **JTextComponent** abstammenden Komponenten (z.B. **JTextField** und **JTextPane**) erlauben dem Benutzer den Zwischenablagentransfer von Zeichenfolgen über die systemüblichen Tastenkombinationen und bieten dem Programmierer über die Methoden **cut()**, **copy()** und **paste()** einen bequeme Möglichkeit zur Nutzung der Zwischenablage.

Sollen aber *beliebige* Daten transportiert werden (z.B. Dateilisten, Bilder), dann ist eine etwas nähere Beschäftigung mit der Zwischenablagen-Unterstützung im Swing-Framework erforderlich. Von den beteiligten Typen (alle aus dem Paket im Paket **java.awt.datatransfer**) kommen einige auch beim Ziehen und Ablegen von Daten per Maus (engl.: *Drag & Drop*) zum Einsatz (siehe Abschnitt 14.7.3).

Die Daten in der Zwischenablage werden durch Objekte der Klasse **DataFlavor** beschrieben, die im Wesentlichen jeweils einen so genannten **MIME-Type** kapseln. Ursprünglich zur Beschreibung von Mail-Erweiterungen gedacht (*Multipurpose Internet Mail Extensions*), wird das MIME-Schema mittlerweile recht universell zur Deklaration von digitalen Inhalten verwendet, z.B. klassifiziert als **text/plain** oder **image/jpeg**).

Damit ein Objekt den gesamten Zwischenablageninhalt repräsentieren kann, muss seine Klasse die Methoden der Schnittstelle **Transferable** implementieren:

- **public DataFlavor[] getTransferDataFlavors()**
  Man erhält einen Array der Klasse **DataFlavor** und kennt damit alle in der Zwischenablage vorhandenen Formate.
- **public boolean isDataFlavorSupported(DataFlavor** *flavor***)**
  Man erfährt, ob in der Zwischenablage Daten eines bestimmten Typs vorhanden sind.
- **public Object getTransferData(DataFlavor** *flavor***)**
  Enthält die Zwischenablage Daten eines bestimmten Typs, werden diese geliefert.

Für den Zugriff auf die Zwischenablage verwendet man ein Objekt der Klasse **Clipboard** aus dem Paket **java.awt.datatransfer**, dessen Adresse mit Hilfe der Klasse **Toolkit** aus dem Paket **java.awt.Toolkit** zu ermitteln ist:

```
private final Clipboard clip;
private Transferable clipContent;
  .  .  .
clip = Toolkit.getDefaultToolkit().getSystemClipboard();
```

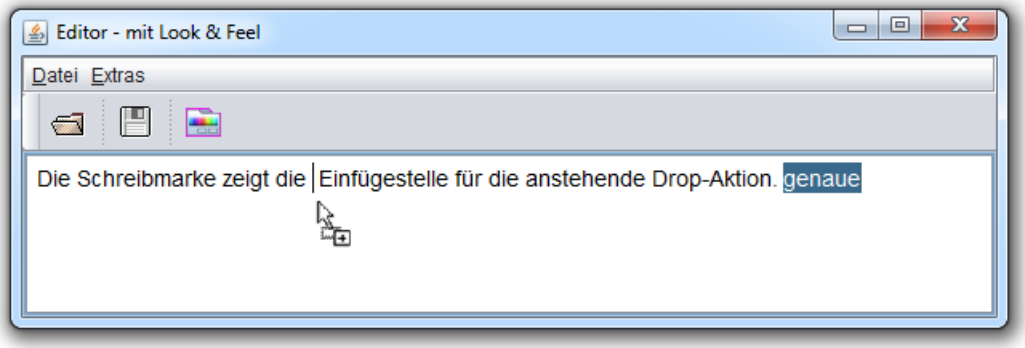
Das **Clipboard**-Objekt beherrscht u.a. die folgenden Methoden:

- **public Transferable getContents(Object** *requestor***)**
  Über diese Methode verschafft man sich ein Objekt, das die gesamte Zwischenablage (mit allen vorhandenen Formaten) repräsentiert und die Methoden der Schnittstelle **Transferable** beherrscht. Weil der **getContents()**-Parameter derzeit keine Rolle spielt, verwendet man stets den Wert **null**, z.B.:
  ```
  clipContent = clip.getContents(null);
  ```
- **public void setContents(Transferable** *contents*, **ClipboardOwner** *owner***)**
  Diese Methode setzt den Zwischenablageinhalt auf ein **Transferable**-Objekt und registriert einen Besitzer der Zwischenablage, wobei ggf. ein Altinhaber über den Eigentumsverlust informiert wird. Spielen die Besitzverhältnisse keine Rolle, gibt man im zweiten Parameter den Wert **null** an.
- **public void addFlavorListener(FlavorListener** *listener***)**
  Möchte man informiert werden, wenn sich die in der Zwischenablage verfügbaren Datenformate ändern, registriert man beim **Clipboard**-Objekt einen Ereignisempfänger, der das Interface **FlavorListener** erfüllt, also die Methode **flavorsChanged()** implementiert.

14.7.3 Ziehen & Ablegen (Drag & Drop)

Benutzer von gut entworfenen GUI-Programmen können in vielen Situationen ihre Absichten auf intuitive Weise (ohne Handbuchstudium) artikulieren, indem sie Daten (z.B. Texte, Listeneinträge, Farben) per Maus von einer Quelle zu einem Ziel bewegen und per **Strg**-Taste signalisieren, ob kopiert oder verschoben werden soll. Dabei kommen sowohl programminterne als auch programmübergreifende Transporte in Frage. Den beteiligten GUI-Komponenten wird einiges an kommunikativen Kompetenzen abverlangt. Weil alle von **JComponent** abstammenden Komponenten bestens vorbereitet sind, lässt sich in Java - Programmen das Ziehen & Ablegen ohne Strapazen für den Programmierer realisieren.

Etliche Klassen (u.a. **JTextField** und **JTextPane**) beherrschen die Empfängerrolle bereits ab Werk, z.B.:

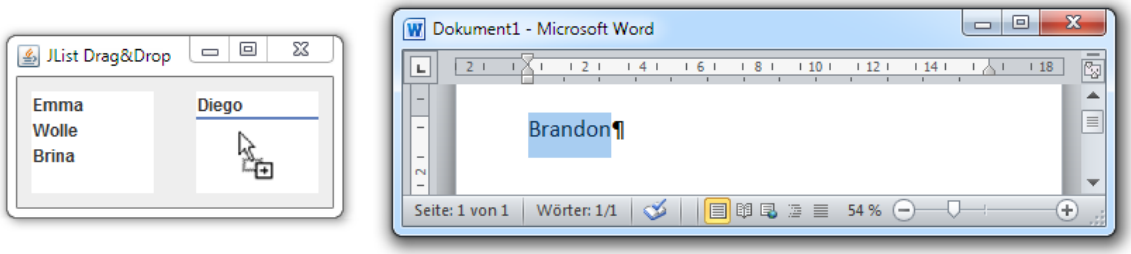


Es genügt ein einfacher Aufruf der Methode **setDragEnabled()**, um auch die Quellfunktionalität zu aktivieren, z.B. beim Editor, den wir seit Abschnitt 14.6.6 sukzessive ausbauen:

```
text.setDragEnabled(true);
```

Auch bei der Klasse **JList<E>** lässt sich auf diese Weise die Drag-Funktionalität einschalten. Das Implementieren macht hier etwas Arbeit und vermittelt dabei nützliche Einblicke in die (Drag & Drop) - Unterstützung des Swing-Frameworks. Im folgenden Beispielprogramm mit zwei Komponenten vom Typ **JList<String>** können Einträge *zwischen* den Listen oder *innerhalb* einer Liste verschoben oder kopiert werden:

Außerdem ist sogar der **String**-Datenaustausch mit anderen Java- und Windows-Programmen möglich, z.B.:



Ein **JList<String>** - Objekt benötigt beim Drag & Drop einen Transporthelfer aus einer geeigneten Spezialisierung der Klasse **javax.swing.TransferHandler**. Im Beispielprogramm wird für den Transportjob die innere Klasse `ListTransferHandler` definiert, deren Methoden anschließend erläutert werden:

```
private class ListTransferHandler extends TransferHandler {
 private JList<String> quellListe, zielListe;
 private int quellIndex, zielIndex;
   .  .  .
}
```

Im Beispielprogramm akzeptieren beide **JList<String>** - Objekte dasselbe `ListTransferHandler`-Objekt als Transporthelfer:

```
final ListTransferHandler lth = new ListTransferHandler();
listeLinks.setTransferHandler(lth);
listeRechts.setTransferHandler(lth);
```

Dieses Objekt kennt Quelle und Ziel eines Transports, so dass zwischen listeninternen und listenübergreifenden Bewegungen unterschieden werden kann.

14.7.3.1 TransferHandler-Methoden für die Drag-Rolle

Über eine **JList<String>** - Komponente in der Drag-Rolle teilt ein `ListTransferHandler` auf Befragen per **getSourceActions()** mit, dass Kopieren und Verschieben möglich seien:

```
public int getSourceActions(JComponent c) {
 return TransferHandler.COPY_OR_MOVE;
}
```

Wird die Drag-Rolle real, holt der Transporthelfer bei der Quelle die Daten ab (hier: eine Zeichenfolge) und verpackt sie in ein Objekt der Klasse **StringSelection**, welche die Schnittstelle **Transferable** (siehe Abschnitt 14.7.2) implementiert:

```
protected Transferable createTransferable(JComponent quelle) {
 quellListe = (JList<String>) quelle;
 quellIndex = quellListe.getSelectedIndex();
 return new StringSelection((String) quellListe.getSelectedValue());
}
```

Zum Abschluss einer *Verschiebung* sorgt der Transporthelfer dafür, dass die Quelle (genauer: deren Model-Objekt) das betroffene Listenelement löscht:

```
protected void exportDone(JComponent quelle, Transferable data, int action) {
 if (action == TransferHandler.MOVE) {
  if (zielListe == quellListe  &&  zielIndex < quellIndex)
   quellIndex++;
  ((DefaultListModel<String>) quellListe.getModel()).remove(quellIndex);
 }
}
```

14.7.3.2 TransferHandler-Methoden für die Drop-Rolle

Über ein **JList<String>** - Objekt in der Drop-Rolle berichtet ein `ListTransferHandler` auf Befragen mit **canImport()**, dass ein Import nur bei Textdaten möglich sei:

```
public boolean canImport(TransferHandler.TransferSupport info) {
 if (info.isDataFlavorSupported(DataFlavor.stringFlavor))
  return true;
 else
  return false;
}
```

Bei der Spezifikation von unterstützten Datentypen kommen finalisierte statische Felder der in Abschnitt 14.7.2 vorgestellten Klasse **DataFlavor** zum Einsatz.

In der Methode **importData()** ermittelt der Transporthelfer den vom Benutzer via Mausposition angegebenen Einfügeindex für die Empfangsliste, extrahiert die Daten und übergibt sie an das Model-Objekt des Empfängers:

```
public boolean importData(TransferHandler.TransferSupport info) {
   JList.DropLocation dl = (JList.DropLocation) info.getDropLocation();
   zielListe = (JList<String>) info.getComponent();

   DefaultListModel<String> listModel =
      (DefaultListModel<String>) zielListe.getModel();
   zielIndex = dl.getIndex();

   Transferable t = info.getTransferable();
   String element;
   try {
      element = (String) t.getTransferData(DataFlavor.stringFlavor);
   } catch (Exception e) { return false; }
   listModel.add(zielIndex, element);
      return true;
}
```

Um von einem **JList<E>** - Objekt die bestmögliche Drop-Rückmeldung für den Benutzer zu erhalten (z.B. Anzeige der Einfügestelle), setzt man den **DropMode** wie im folgenden Beispiel:

```
listeLinks.setDropMode(DropMode.ON_OR_INSERT);
listeRechts.setDropMode(DropMode.ON_OR_INSERT);
```

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

…\BspUeb\Swing\Drag & Drop

14.8 Übungsaufgaben zu Kapitel 14

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Bei der AWT/Swing - Ereignisbehandlung gehört zu jeder Event-ID eine eigene Ereignisklasse.
2. Methoden einer anonymen oder inneren Klasse dürfen auf die Mitglieder der umgebenden Klasse zugreifen (auch auf die privaten).
3. Auch zu anonymen Klassen und Mitgliedsklassen erstellt der Compiler jeweils eine eigene Bytecode-Datei.
4. Ein Swing-Programm endet mit dem Schließen seines Hauptfensters.

2) Ermitteln Sie mit Hilfe eines Programms die Event-IDs zu folgenden Ereignissen:

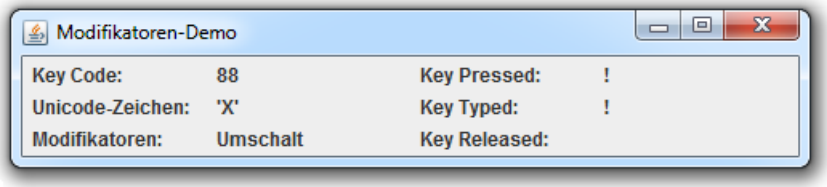
- Eine Schaltfläche wurde betätigt.
- Über einer Komponente wurde eine Maustaste gedrückt bzw. losgelassen.
- Eine Taste wurde gedrückt bzw. losgelassen.
- Das Anwendungsfenster wurde aktiviert.

Sie werden feststellen, dass alle Maustasten dieselben Ereigniskennungen liefern. Mit Hilfe der **MouseEvent**-Methode **getButton()** kann man die Tasten aber doch unterscheiden.

3) Erstellen Sie ein Tastatur-Demonstrationsprogramm, das für jede gedrückte Taste(nkombination) ausgibt:

- den virtuellen Key Code der zuletzt gedrückten Taste
- Unicode-Zeichen (falls definiert)
- gedrückte Modifikator-Tasten (Umschalt, Steuerung, Alt)

So ähnlich sollte Ihr Programm z.B. auf die Tastenkombination **Umschalt+x** reagieren:



Verwenden Sie zur Anordnung der Komponenten ein **GridLayout**.

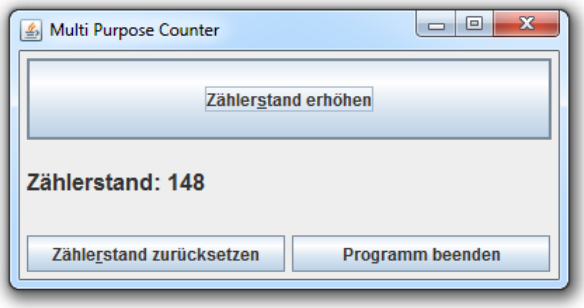
4) Zu einer anonymen Klasse lässt sich nur *eine* Instanz erzeugen. Es ist aber durchaus möglich, ein solches Objekt z.B. als **ActionListener** für mehrere Befehlsschalter zu verwenden, indem der zuerst versorgte Schalter mit **getActionListeners()** nach dem zuständigen Ereignisempfänger befragt und die erhaltene Referenz anschließend wieder verwendet wird. Fertigen Sie ein entsprechendes Beispielprogramms mit zwei Befehlsschaltern an.

5) Vergleichen Sie die Ereignisbehandlung bei GUI-Anwendungen mit der in Kapitel 11 vorgestellten Ausnahmebehandlung. Welche Unterschiede sind vorhanden?

6) Welche Layout-Manager sind bei den Swing-Container-Klassen **JFrame**, **JPanel** und **Box** jeweils voreingestellt?

7) Warum existiert zur Ereignisklasse **ActionEvent** bzw. zum Interface **ActionListener** keine Adapterklasse (analog zur Klasse **WindowAdapter** beim Interface **WindowListener**)?

8) Erstellen Sie eine seriösere Variante des Zählprogramms aus Abschnitt 14.2, z.B. mit folgender Bedienoberfläche:



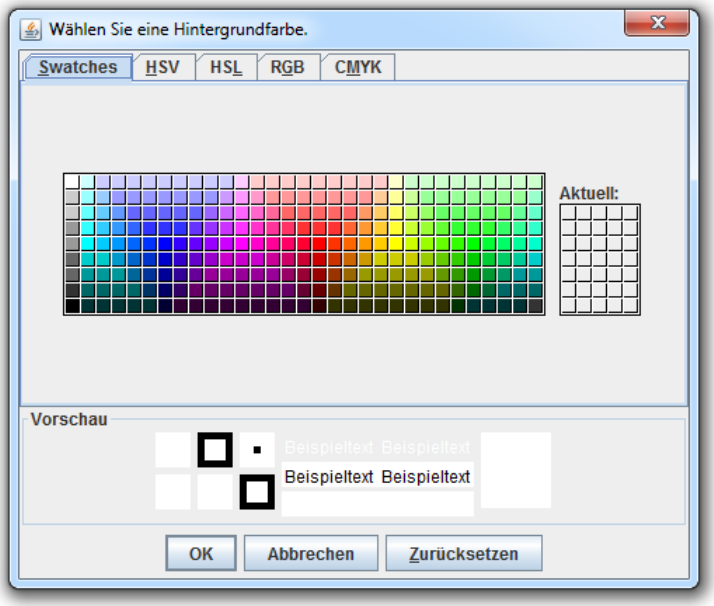
Bieten Sie statt der verspielten Icon-Anzeige Schalter an, um den Zählerstand zurück zu setzen und das Programm zu beenden. Verhindern Sie mit der Methode **setMinimumSize()** der **JFrame**-Basisklasse **java.awt.Window**, das beim Verkleinern des Rahmenfensters Bedienelemente verschwinden. Über die **JComponent**-Methode **setFont()** können Sie die Schriftart der Zählerstandanzeige beeinflussen, z.B.:

```
lblCounter.setFont(new Font(Font.SANS_SERIF, Font.BOLD, 16));
```

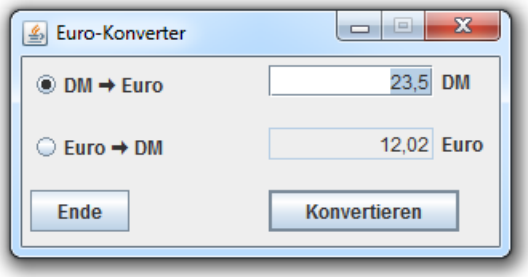
9) Realisieren Sie für das Editor-Beispielprogramm ausgehend vom Entwicklungsstand in

…\BspUeb\Swing\Editor\E1 (mit Menü)

einen **ActionListener** zum Menü-Item **Extras > Hintergrundfarbe** unter Verwendung der Klasse **JColorChooser**, die einen attraktiven Standarddialog zur Farbauswahl bietet:



10) Erstellen Sie den in Abschnitt 14.6.3 als Beispiel verwendeten Euro-DM-Konverter:

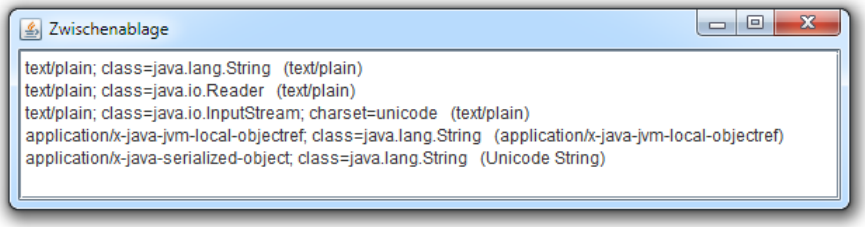


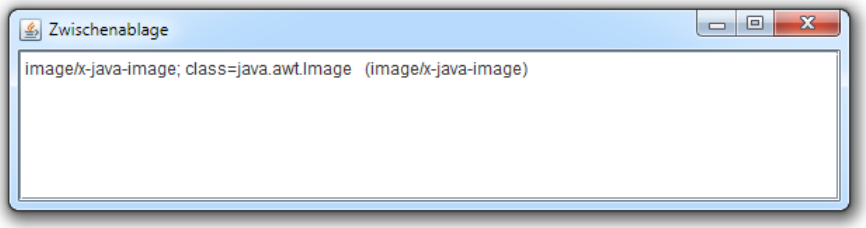
Hinweise:

- Die untere **JTextField**-Komponente soll nur zur Ausgabe dienen. Daher wurde via **setEditable(false)** festgelegt, 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 Escape-Sequenz lassen sich beliebige Unicode-Zeichen in einem Java-Programm verwenden, z.B.:

```
dm2euro = new JRadioButton("DM "+'\u27a0'+" Euro", true);
```

11) Erstellen Sie eine Swing-Anwendung, welche die Liste der aktuell verfügbaren Zwischenablageformate ausgibt, sobald sich diese Liste ändert, z.B.




Zwischenablage
image/x-java-image; class=java.awt.Image (image/x-java-image)

15 Ein-/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 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. Im Wesentlichen geht es darum, Zeichenfolgen, primitive Werte (Typ **byte**, **int**, **double** etc.) oder ganze Objekte in eine Datei auf der Festplatte schreiben bzw. von dort lesen. Außerdem werden wir uns mit der Verwaltung von Dateien und Verzeichnissen beschäftigen.

Vorausblick auf zwei verwandte Themen:[1]

- In einem späteren Kapitel werden Sie mit den **Netzwerkverbindungen** weitere, außerordentlich wichtige Datenquellen bzw. -senken kennen lernen und dabei von Ihren Kenntnissen über die sequentielle Datenstromtechnik profitieren.
- Im Kapitel über **Datenbankprogrammierung** werden anspruchsvolle Datenverwaltungstechniken vorgestellt, die sich auch im Netzwerk- und Mehrbenutzerkontext bewähren. Dabei überlassen wir 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 15.1.1) orientierte Ein-/Ausgabetechnik. Die als Ergänzung zur Datenstromtechnik konzipierte **Channel-Technik** kommt nicht explizit zum Einsatz, wird allerdings von einigen API-Klassen im Hintergrund verwendet.[2]

Unsere Beispielprogramme verwenden regelmäßig Klassen aus den Paketen **java.io** und **java.nio.file**, so dass sich der Import dieser Pakete meistens lohnt:

```
import java.io.*;
import java.nio.file.*;
```

---

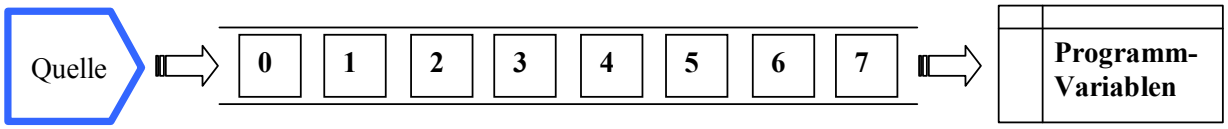
[1] Leider sind die Kapitel über Netzwerk- und Datenbankprogrammierung mangels Zeit für die erforderliche Aktualisierung derzeit nicht im Manuskript enthalten.

[2] Channel-Technik kommt z.B. dann ins Spiel, wenn über die statischen Methoden **newOutputStream()** oder **newInputStream()** der Klasse **Files** ein Datenstromobjekt angefordert wird (siehe Abschnitte 15.3.1.3 und 15.3.2.3).

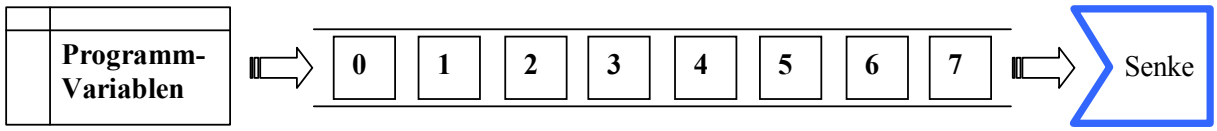
15.1 Grundlagen

15.1.1 Datenströme

In Java wird die sequentielle Datenein- und -ausgabe über sogenannte *Ströme* (engl. *streams*) abgewickelt. Ein Programm liest Bytes[1] aus einem **Eingabestrom**, der aus einer Datenquelle (z.B. Datei, Array, Eingabegerät, anderes Programm, Netzwerkverbindung) gespeist wird:



Ein Programm **schreibt** Bytes in einen **Ausgabestrom**, der die Werte von Programmvariablen zu einer Datensenke befördert (z.B. Datei, Array, Ausgabegerät, anderes Programm, 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 haben zwar eine konzeptionelle Verwandtschaft mit den sequentiellen Strömen basierend (nicht nur) auf den Elementen von Kollektionen, 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.

15.1.2 Beispiel

Das folgende Programm schreibt einen **byte**-Array in eine Datei und liest die Daten anschließend wieder zurück:

---

[1] Wenn in Kapitel 15 der Namensteil *Byte* auftaucht, ist keine Java-Wrapper-Klasse gemeint, sondern eine 8 Bit umfassende Informationseinheit der Datenverarbeitung.

Quellcode	Ausgabe
`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.txt");` `   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);` `  }` `}`	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` vom Typ **OutputStream** erzeugt und mit der vom **Path**-Objekt `file` bezeichneten 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` vom Typ **InputStream** 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).

In einem Datenstromklassen-Konstruktor sowie beim Lesen und Schreiben von Daten kann es zu einer von **IOException** abstammenden Ausnahme kommen, die entweder in einer **try-catch** - Anweisung abgefangen oder im Definitionskopf der nutzenden Methode deklariert werden muss (vgl. Abschnitt 11.5). 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.

15.1.3 Klassifikation der Stromverarbeitungsklassen

Das Paket **java.io** enthält vier abstrakte Basisklassen, von denen die für uns relevanten Stromverarbeitungsklassen abstammen:

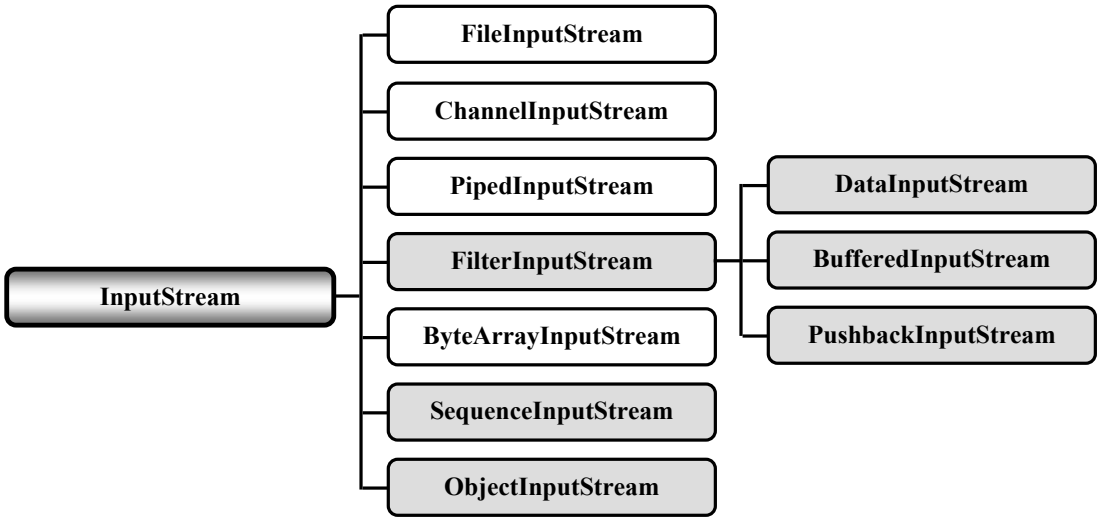
- **InputStream** und **OutputStream**
  Die Klassen aus den zugehörigen Hierarchien verarbeiten **Ströme mit Bytes** als Elementen. Byte-Ströme werden für das Schreiben und Lesen von **binären Daten** verwendet, die keine Sequenz von Zeichen darstellen, sondern z.B. die Werte primitiver Datentypen oder komplette Objekte.
- **Reader** und **Writer**
  Die Klassen aus den zugehörigen Hierarchien verarbeiten **Zeichenströme**, die aus einer Folge von Zeichen in einer bestimmten Kodierung (z.B. UTF-8 oder ANSI) bestehen. Landet ein Zeichenstrom in einer Datei, kann diese anschließend mit jedem Texteditor bearbeitet werden, der die verwendete Kodierung versteht.

Während die Byte-orientierten Klassen schon zur ersten Java-Generation gehörten, wurden die zeichenorientierten Klassen erst mit der Version 1.1 eingeführt, um Probleme mit der Internationalisierung von Java-Programmen zu lösen. Wo alte und neue Lösungen zur Verarbeitung von Textdaten konkurrieren, sollten die zeichenorientierten Klassen den Vorzug erhalten.

Bei den Abkömmlingen der vier abstrakten Basisklassen sind nach der *Funktion* zu unterscheiden:

- **Ein- bzw. Ausgabeklassen**
  Sie haben **direkten Kontakt zu Datenquellen bzw. -senken**. Sollen z.B. Bytes aus einer Datei gelesen bzw. in eine Datei geschrieben werden, kommen die Klassen **FileInputStream** bzw. **FileOutputStream** zum Einsatz.
- **Transformationsklassen**
  Sie dienen zum **Transformieren** von Datenströmen 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:



15.1.4 Aufbau und Verwendung der Transformationsklassen

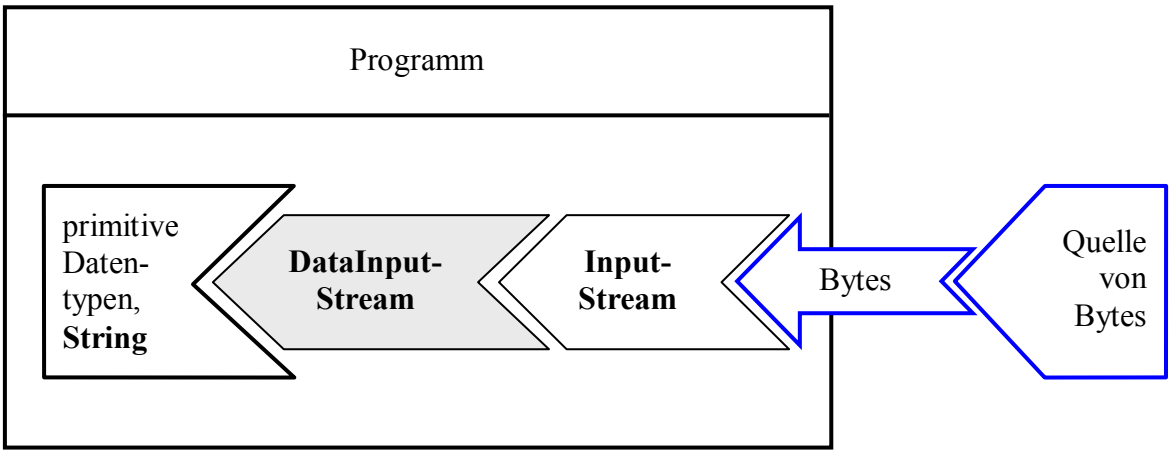
Eine Transformations- bzw. Filterklasse baut auf einer anderen Datenstromklasse auf und stellt Methoden für eine erweiterte Funktionalität zur Verfügung. Wie diese Zusammenarbeit organisiert wird, betrachten wir am Beispiel der Eingabetransformationsklasse **DataInputStream** aus der **InputStream**-Hierarchie. 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 beliebigen 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 bezeichnet (zu den Klassen **Files** und **Path** siehe Abschnitt 15.2.1):

```
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 15.1.5), 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[] egal) {
      Path file = Paths.get("demo.dat");
      int[] arr = {1024,2048,4096,8192};

      try (DataOutputStream dos =
           new DataOutputStream(Files.newOutputStream(file))) {
         for(int ele : arr)
            dos.writeInt(ele);
      } catch (IOException ioe) {
         System.err.println(ioe);
         System.exit(1);
      }

      try (DataInputStream dis =
           new DataInputStream(Files.newInputStream(file))) {
         for(int ele : arr)
            System.out.println(ele = dis.readInt());
      } catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Ausgabe

```
1024
2048
4096
8192
```

Am Beispiel **DataInputStream** sollen noch einmal wichtige Merkmale einer Transformations- bzw. Filterklasse zusammengefasst werden:

- Die Klasse **DataInputStream** besitzt eine im Konstruktor initialisierte Instanzvariable vom Typ **InputStream**, über die der Kontakt zu einem angeschlossenen Datenstromobjekt hergestellt wird.
- Die **DataInputStream**-Eingabemethoden beauftragen den eingebundenen **InputStream**, Bytes in hinreichender Menge zu beschaffen. Diese werden dann zu Werten eines primitiven Datentyps zusammengesetzt.
- **DataInputStream**-Objekte können mit jedem **InputStream**-Objekt kooperieren. Bei 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.

Die Exception Handler des obigen Beispielprogramms schreiben diagnostische Ausgaben 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 15.3.1.6). Beide Standardströme werden automatisch zur Verfügung gestellt und müssen weder geöffnet noch geschlossen werden.

15.1.5 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. Geöffnete Dateien können 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 Java-Datenstromobjekte beherrschen die Methode **close()**, die ggf. Zwischenspeicher entleert, den Strom schließt und die assoziierten Ressourcen frei gibt. Danach ist das Stromobjekt zum Lesen bzw. Schreiben von Daten nicht mehr zu gebrauchen.

Einen expliziten **close()** - Aufruf zu unterlassen, hat *manchmal keine* negativen Konsequenzen, weil die vom Garbage Collector ausgeführte Methode **finalize()** einen **close()** - Aufruf enthält (z.B. bei den Klassen **FileInputStream** und **FileOutputStream**). Es gibt jedoch gute 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 2015) heißt es dazu unmissverständlich:

  > 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 15.3.1.5).

Um allen Problemen aus dem Weg zu gehen, sorgt man dafür, dass jeder Strom so früh wie möglich durch einen **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.

Ein Transformationsobjekt gibt einen **close()** - Aufruf an den zugrunde liegenden Datenstrom weiter, so dass 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).
- Seit Java 7 lässt sich das Schließen der in einem **try**-Block benötigten Ressourcen automatisieren (*try with resources*). Beteiligte Klassen müssen das Interface **AutoCloseable** implementieren, was bei den Datenstromklassen (den Ableitungen der Klassen **InputStream**, **OutputStream**, **Reader** und **Writer**) der Fall ist. Bei dieser Technik erfolgt der **close()** - Aufruf automatisch hinter den Kulissen (siehe Beispiel in Abschnitt 15.1.2, vgl. Abschnitt 11.8.2).

15.2 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 15.2.1). 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.

Eine Verwendung der mit Java 6 kompatiblen Technik kommt bei älteren, noch weiter zu pflegenden Projekten in Frage. Daher wird im Manuskript auch die ältere Technik beschrieben (siehe Abschnitt 15.2.2).

---

[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.

15.2.1 Dateisystemzugriffe über das NIO.2 - API

Wir beschränken uns auf die Typen **Path**, **Paths** und **Files** aus dem Paket **java.nio.file**.

15.2.1.1 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),[1]
- 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* definiert, so dass es keinen Konstruktor zu diesem Typ gibt. In diese Presche springt die Klasse **Paths** mit der statischen Methode **get()**:

public static Path get(String *first***, String...** *more***)**

In der Aktualparameterliste darf dem obligatorischen *ersten* Knotennamen eine beliebig lange Liste weiterer Knotennamen folgen (zum Serienparameter siehe Abschnitt 4.3.1.3.3). 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 Pfadangabe startet mit dem Heimatverzeichnis des Benutzers `otto`. Wenn generell das Heimatverzeichnis des angemeldeten Benutzers gemeint ist, kann der Pfad mit Hilfe der statischen **System**-Methode **getProperty()** allgemeingültig notiert werden:

```
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.:

```
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 abgeleitetes **Path**-Objekt anfordern:

---

[1] Weil der Rückwärtsschrägstrich in Java-Syntax die Escape-Sequenzen einleitet, muss der Wurzelknoten eines Windows-Laufwerks mit dem doppelten Rückwärtsschrägstrich (C:\\) oder mit dem Vorwärtsschrägstrich (C:/) notiert werden.

- **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.

- **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
  - das Parameterobjekt einen Dateinamen enthält.

  Dann resultiert ein absoluter Pfad, der auf die Datei zeigt, z.B.:

Quellcodesegment	**toString()** - Ergebnis
`Path ordner = Paths.get("U:/Eigene Dateien");` `Path datei = ordner.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
  - 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`

- **public File toFile()**
  Liefert das dem **Path**-Objekt entsprechende **File**-Objekt (vgl. Abschnitt 15.2.2)

Von der Methode **toUri()** erhält man ein **URI**-Objekt (***U**niform **R**esource **I**dentifier*), das z.B. zum Öffnen einer Datei durch einen WWW-Browser verwendet werden kann, 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 lexikographischen Priorität 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 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.

15.2.1.2 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 Pfadnamen gibt, z.B.:

```
if (Files.exists(ordner))
 System.out.println(ordner + " existiert bereits.");
else
 if (Files.notExists(ordner))
  System.out.println(ordner + " existiert noch nicht");
 else
  System.out.println(ordner + " 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 Pfadnamen vorhanden war, beweist die Rückgabe **true** der statischen **Files**-Methode **notExists()**.

Wie im Java-Tutorial (Oracle 2015) 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*).

15.2.1.3 Verzeichnis anlegen

Um das Verzeichnis

U:\Eigene Dateien\Java\io

anzulegen, erzeugen wir zunächst ein passendes **Path**-Objekt (vgl. Abschnitt 15.2.1.1):

```
Path ordner = 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(ordner);
} catch (FileAlreadyExistsException ae) {
   System.err.println(ordner + " existiert, ist aber kein Verzeichnis.");
   System.exit(1);
} catch (IOException ioe) {
   System.err.println(ioe);
   System.exit(1);
}
```

Von beiden Methoden sind die folgenden 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.

15.2.1.4 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 explizit anzulegen, wozu die statische **Files**-Methode **createFile()** bereitsteht. Das als Parameter benötigte **Path**-Objekt zur Datei

kann man aus einem vorhanden **Path**-Objekt zum Verzeichnis und einem Dateinamen über die **Path**-Methode **resolve()** gewinnen:

```
Path file = ordner.resolve("Ausgabe.txt");
try {
 Files.createFile(file);
} catch (FileAlreadyExistsException ae) {
 System.err.println(file + " existiert bereits.");
} catch (IOException ioe) {
 System.err.println(ioe);
}
```

15.2.1.5 Attribute von Dateisystemobjekten ermitteln

Mit diversen statischen Methoden der Service-Klasse **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***)**
  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.:
  ```
  2016-02-11T13:19:56.112304Z
  ```
  Zum Parameter **LinkOption** folgt eine Erläuterung hinter der Auflistung.
- **public static long size(Path** *path***)**
  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 4.3.1.3.3) 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, so dass die ermittelten Attributausprägungen vom Verweisziel stammen. Mit dem Parameterwert **LinkOption.NOFOLLOW_LINKS** unterbleibt die Auflösung, so dass die Attributausprägungen vom Link stammen. 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.:

In den folgenden Anweisungen werden Informationen über eine Datei gesammelt:

```
Path ordner = Paths.get("U:", "Eigene Dateien", "Java", "io");
Path datei = ordner.resolve("Ausgabe.txt");
  .  .  .
System.out.println("Eigenschaften von    " + datei);
System.out.println(" Größe in Bytes:     " + Files.size(datei));
System.out.println(" Letzte Änderung:    " + Files.getLastModifiedTime(datei));
System.out.println(" Datei:              " + Files.isRegularFile(datei));
System.out.println(" Verzeichnis:        " + Files.isDirectory(datei));
```

Ausgabe:

```
Eigenschaften von    U:\Eigene Dateien\Java\io\Ausgabe.txt
 Größe in Bytes:     3
 Letzte Änderung:    2016-02-11T13:19:56.112304Z
 Datei:              true
 Verzeichnis:        false
```

Statt für mehrere Attribute eines Dateisystemobjekts jeweils eine zeitaufwändige Anfrage an das Dateisystem zu richten, sollte man über die **Files**-Methode **readAttributes()** ein Paket mit den Basis-, DOS- oder POSIX-Attributen eines Dateisystemobjekts ermitteln, z.B.:

```
BasicFileAttributes attr = Files.readAttributes(file, BasicFileAttributes.class);
```

Mit dem zweiten Parameter wird der gewünschte Rückgabetyp ermittelt, wobei sich oft die Klasse **BasicFileAttributes** anbietet, die 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());
```

15.2.1.6 Zugriffsrechte für Dateien ermitteln

Mit den statischen Methoden **isReadable()**, **isWritable()** und **isExecutable()** der Serviceklasse **Files** kann man feststellen, ob das aktive Programm 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 auf den ermittelten Status verlassen.

Beispiel:

Quellcodesegment	Ausgabe
`System.out.println(Files.isWritable(datei));`	`true`

15.2.1.7 Attribute ändern

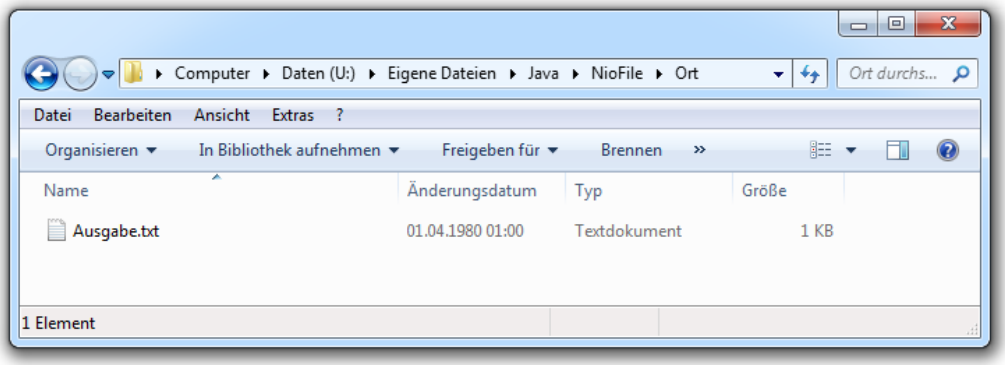
Über die Service-Klasse **Files** lassen sich einige Attribute von Dateisystemobjekten ändern. Wir beschränken uns auf das Datum der letzten Modifikation:

public static Path setLastModifiedTime(Path *path***, FileTime...** *time***)**
throws IOException

Zum Erstellen des benötigten **FileTime**-Objekts ist die Methode **fromMillis()** 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. Neben dem Kreationsdatum können auch diverse DOS- bzw. POSIX-Dateiattribute (z.B. Hidden, Owner) gesetzt werden (siehe Java-Tutorial, Oracle 2015).[1]

[1] Siehe Webseite (besucht am 14.02.2016): http://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html

15.2.1.8 Verzeichniseinträge auflisten

Zu einem Ordner liefert die statische **Files**-Methode **newDirectoryStream()** ein Objekt aus einer Klasse, welche u.a. die Interfaces **AutoClosable** und **Iterable<Path>** beherrscht.[1] Für das geöffnete Verzeichnis werden Ressourcen belegt, so dass ein möglichst frühes Schließen erforderlich ist, was am besten in einer **try**-Anweisung mit automatischer Ressourcenfreigabe geschieht. Im folgenden Beispiel wird das Stromobjekt in einer **for**-Schleife für Kollektionen dazu verwendet, um über die Einträge im Verzeichnis zu iterieren:

```
try (DirectoryStream<Path> stream = Files.newDirectoryStream(ordner)) {
   for (Path path: stream)
      System.out.println(path.getFileName());
} catch (Exception e) {
   System.err.println(e);
}
```

15.2.1.9 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 in Abschnitt 15.3.1.2). 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);
 }
}
```

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:[2]

- **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 von Betriebs- bzw. Dateisystem unterstützt wird.

---

[1] 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.

[2] Wie bei einem Serienparameter üblich, darf man die Aktualparameterliste auch komplett weglassen (vgl. Abschnitt 4.3.1.3.3)

- **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.

15.2.1.10 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 2015), beschränken wir uns auf zwei Beispiele.[1]

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(datei, datei.resolveSibling("Umbenannt.txt"));
```

Auf die Angabe von Optionen wird hier verzichtet, was bei einem Serienparameter erlaubt ist (vgl. Abschnitt 4.3.1.3.3).

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:

```
Files.move(ordner.resolve("Umbenannt.txt"),
                          ordner.getParent().resolve("Verschoben.txt"));
```

Das **Path**-Objekt zu einem (vom Wurzelknoten verschiedenen) Verzeichnis liefert über die Methode **getParent()** das übergeordnete Verzeichnis (siehe Abschnitt 15.2.1.1).

Im **CopyOption**-Serienparameter sind 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, so dass 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 der Option **REPLACE_EXISTING** passieren kann. Wenn keine atomare Ausführung möglich ist, wird eine Ausnahme vom Typ **AtomicMoveNotSupportedException** geworfen.

---

[1] Webseite (besucht am 02.02.2015): http://docs.oracle.com/javase/tutorial/essential/io/move.html

15.2.1.11 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 15.2.1.8) aufgesucht und gelöscht:

```
try (DirectoryStream<Path> stream = Files.newDirectoryStream(ordner)) {
   for (Path path: stream)
      Files.delete(path);
} catch (IOException ioe) {
   System.err.println(ioe);
}
```

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()**.

15.2.1.12 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();
```

Genau genommen 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())
    zeigeKap(store);
} catch (IOException e) {
  e.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 abstract long getTotalSpace()**
  throws IOException
  Man erhält die Gesamtkapazität in Bytes.
- **public abstract 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 zeigeKap(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.format("%-20s %12d %12d %12d\n", store, gesamt, belegt, frei);
}
```

Eine typische Ausgabe (Windows-Rechner mit zwei Festplattenpartitionen, einer eingelegten CD und einer verbundenen Netzfreigabe):

```
Laufwerk              Gesamt (MB)  Belegt (MB)    Frei (MB)
System (C:)                228833       161746        67087
Daten (E:)                 753865       447881       305984
ElsterFormular (M:)           177          177            0
(Z:)                        51200        26259        24940
```

15.2.1.13 Weitere Optionen

Aus Zeitgründen können einige attraktive Neuerungen im NIO.2 - API nur erwähnt werden (siehe Kapitel *Basic I/O* im Java-Tutorial, Oracle 2015):[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)

15.2.2 Dateisystemzugriffe über die Klasse File aus dem Paket java.io

Der Umgang mit Dateien und Verzeichnissen (z.B. Erstellen, auf Existenz prüfen, Löschen, Attribute lesen und setzen) wird in Java 6 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};
   . . .
 }
}
```

[1] Webseite (besucht am 01.02.2015): http://docs.oracle.com/javase/tutorial/essential/io/

15.2.2.1 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 gewöhnliche Schrägstrich 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).

15.2.2.2 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** bereit steht:

```
String name = ordner + "/Ausgabe.txt";
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);
  System.exit(1);
 }
}
```

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.

15.2.2.3 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 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("\nEigenschaften der Datei " + name);
System.out.println("  Vollst. Pfad:       " + f.getAbsolutePath());
DateFormat df = DateFormat.getInstance();
Date d = new Date(f.lastModified());
String time = df.format(d);
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 U:/Eigene Dateien/Java/FileDemo/AusDir/Ausgabe.txt
  Vollst. Pfad:       U:\Eigene Dateien\Java\FileDemo\AusDir\Ausgabe.txt
  Letzte Aenderung:   18.02.16 04:16
  Groesse in Bytes:   3
  Schreiben moeglich: true
```

Für die formatierte Ausgabe der **lastModified()** - Rückgabe unter Berücksichtigung der lokalen Zeitzone sorgen ein **Date**- und ein **DateFormat**-Objekt.

15.2.2.4 Attribute ändern

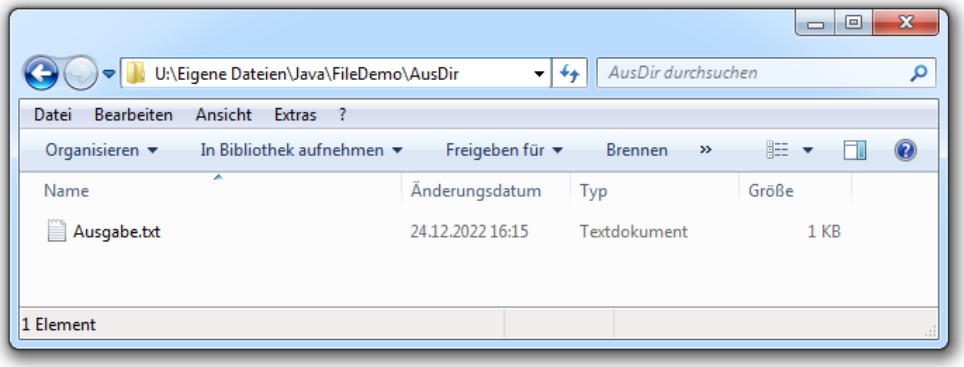
Man kann etliche Attribute von Dateien oder Ordnern ändern, z.B.:

- **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

Hier werden die Anforderungen an ein **File**-Objekt gerichtet, das eine Datei repräsentiert:

```
Date d = null;
DateFormat df = DateFormat.getInstance();
try {
 d = df.parse("24.12.22 16:15");
} catch (Exception e) {}
f.setLastModified(d.getTime());
f.setWritable(false);
```

Windows hat nichts dagegen, das Datum der letzten Änderung in die Zukunft zu verlegen:



15.2.2.5 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("Dateien im akt. Verzeichnis:");
for (File fi : files)
 System.out.println(" " + fi.getName());
```

Anschließend werden die Datei- oder Verzeichnisnamen mit Hilfe der **File**-Methode **getName()** ausgegeben:

```
Dateien im akt. Verzeichnis:
.classpath
 .project
 .settings
 FileDemo.class
 FileDemo.java
 FileFilter.class
 FileFilter.java
```

Eine alternative **listFiles()** - Überladung liefert eine *gefilterte* Liste mit **File**-Verzeichniseinträgen, z.B.:

```
files = curDir.listFiles(new FileFilter("java"));
System.out.println("Dateien im akt. Verzeichnis mit Extension .java:");
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.*;

public class FileFilter implements FilenameFilter {
  private String ext;

  public FileFilter(String ext_) {ext = ext_;}

  public boolean accept(File dir, String name) {
    return name.toLowerCase().endsWith("."+ext);
  }
}
```

Um den **FilenameFilter** - Interface-Vertrag zu erfüllen, muss `FileFilter` die Methode **accept()** implementieren. Im Beispiel resultiert folgende Ausgabe:

```
Dateien im akt. Verzeichnis mit Extension .java:
 FileDemo.java
 FileFilter.java
```

15.2.2.6 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.

15.2.2.7 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() + " geloescht");
else {
 System.out.println("Fehler beim Löschen des Ordners " + dir.getName());
}
```

Damit ein Verzeichnis gelöscht werden kann, muss es leer sein.

15.3 Klassen zur Verarbeitung von Byte-Strömen

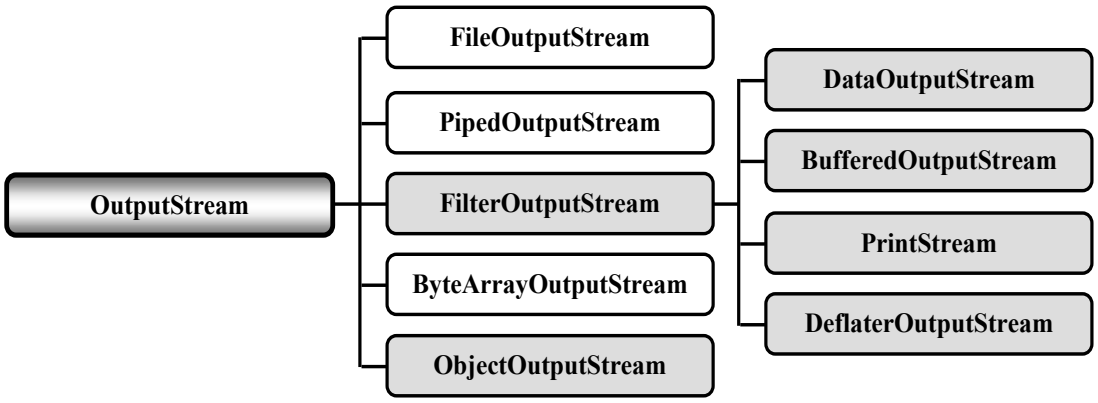
In Java 1.0 stammten *alle* Ein-/Ausgabeklassen von **InputStream** oder **OutputStream** ab. Diese Klassen haben sich zur Ein- bzw. Ausgabe von Bytes, primitiven Datenwerten und Objekten bewährt, aber bei der Behandlung von Unicode-Zeichen und vor allem beim Internationalisieren von Java-Software Probleme 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 15.3.1.6).

15.3.1 Die OutputStream-Hierarchie

15.3.1.1 Überblick

In der folgenden Abbildung sehen Sie den für uns relevanten Teil der Klassenhierarchie zur Basisklasse **OutputStream**, wobei die Ausgabeklassen (in direktem Kontakt mit Datensenken) mit einem weißen Hintergrund und die Ausgabe*transformations*klassen mit einem grauen Hintergrund dargestellt sind:



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**. In Abschnitt 16.2.7.2 wird die Realisation einer Produzenten-Konsumenten - Kooperation mit der Pipe-Technik beschrieben.

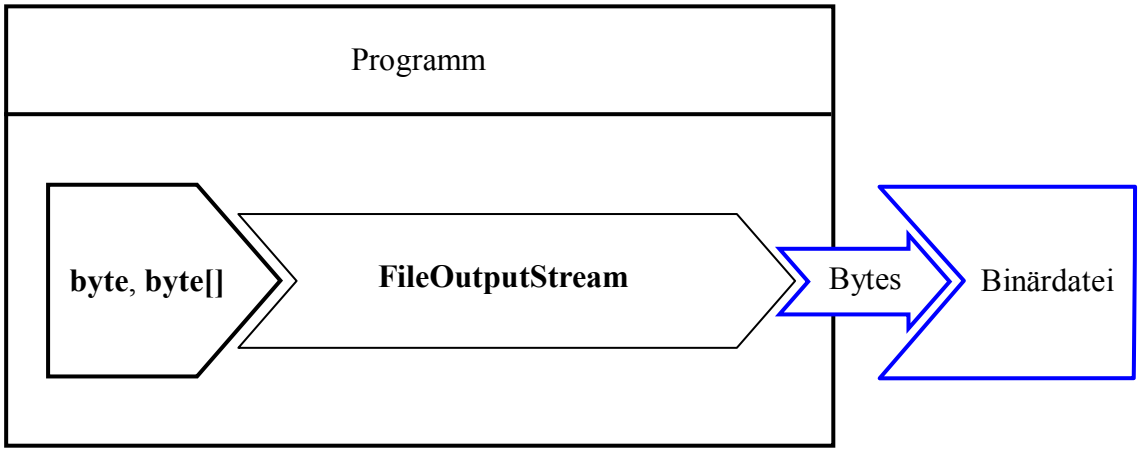
Mit der Transformationsklasse **ObjectOutputStream** können komplette Objekte in einen Byte-orientierten Ausgabestrom geschrieben werden. Sie wird zusammen mit ihrem Gegenstück **ObjectInputStream** in Abschnitt 15.6 behandelt.

Mit den folgenden Klassen werden wir uns im Manuskript *nicht* beschäftigen:

- Objekte der Ausgabeklasse **ByteArrayOutputStream** schreiben Bytes in einen **byte**-Array, also in eine programminterne Datensenke.
- 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).

15.3.1.2 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 15.2.2) 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 (obligatorisch abzufangende) 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 15.2.1.1).

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.

Weil **FileOutputStream**-Objekte nur **byte**-Variablen oder -Arrays befördern können, werden sie oft mit Filterobjekten (z.B. aus der Klasse **DataOutputStream**) kombiniert, die flexiblere Ausgabemethoden bieten (siehe Abschnitt 15.3.1.4). 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 15.2.1.9). 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 15.3.2.2):

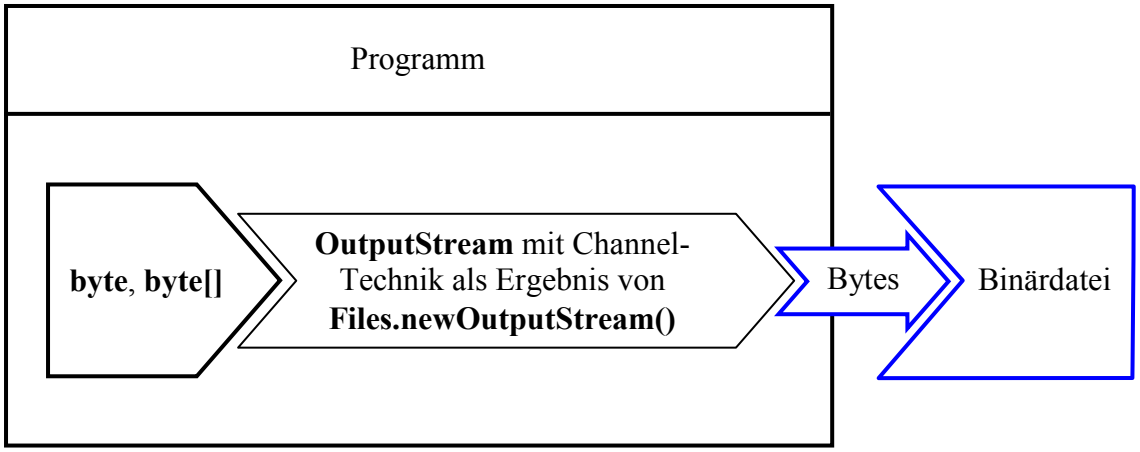
```
import java.io.*;

class FileCopy {
 public static void main(String[] args) {
  String quelle = "quelle.dat", ziel = "ziel.dat";
  final int BUFLEN = 1048576; // Ein Megabyte (1024*1024 Bytes) als Puffergröße
  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:");
   while (true) {
    nread = fis.read(buffer, 0, Math.min(BUFLEN, fis.available()));
    if (nread == 0)
     break;
    else {
     fos.write(buffer, 0, nread);
     total += nread;
     System.out.print("*");
    }
   }
   zeit = System.currentTimeMillis() - zeit;
   System.out.println("\nEs wurden "+total+" Bytes kopiert. "+
     "(Benötigte Zeit: "+zeit+" Millisekunden.)");
  } catch (IOException e) {
   System.err.println(e.getMessage());
  }
 }
}
```

In der **while**-Schleife wird mit der **FileInputStream**-Methode **read()** aus der Quelldatei jeweils ein Megabyte oder aber die per **available()** - Aufruf ermittelte Restmenge (vgl. Abschnitt 15.3.2.2) 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. Nach einem erfolgreichen Programmlauf wird die Transportleistung und die benötigte Zeit protokolliert, z.B.:

```
Kopieren von quelle.dat in ziel.dat gestartet:
**********************************************************************************
**********************************
Es wurden 126619130 Bytes kopiert. (Benoetigte Zeit: 2812 Millisekunden.)
```

15.3.1.3 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 15.7.1):

public static OutputStream newOutputStream(Path *path***, OpenOption...** *options***)**
throws IOException

Eine Inspektion des API-Quellcodes (in den Klassen **Files**, **FileSystemProvider** und **Channels**) zeigt, dass im Hintergrund bei den Dateizugriffen die moderne Channel-Technik zum Einsatz kommt, wobei ein Geschwindigkeitsvorteil möglich, aber nicht garantiert ist.

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.

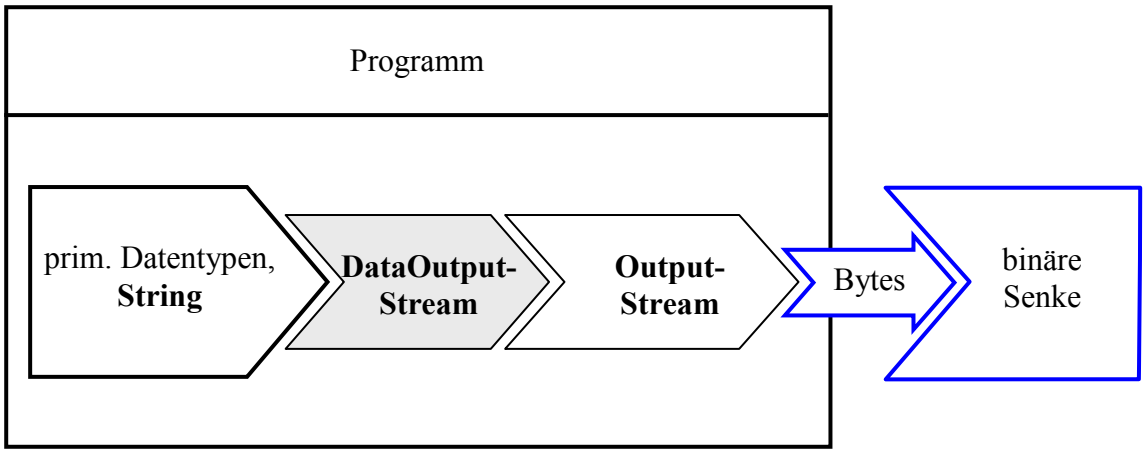
Im Vergleich zu einem traditionell per Konstruktor erzeugten **FileOutputStream** bestehen folgende Vorteile:

- Es lassen sich Öffnungsoptionen für die Datei angeben (siehe Abschnitt 15.7.1).
- Es kommt die moderne Channel-Technik zum Einsatz.
- Im Unterschied zu einem **FileOutputStream** ist die Nutzung durch mehrere Threads (Ausführungsfäden, siehe Kapitel 16) erlaubt.

Ein Einsatzbeispiel für die Methode **newOutputStream()** war schon in Abschnitt 15.1.2 zu sehen.

15.3.1.4 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 15.4.1.2) zum Einsatz kommt. Diese Methode ist angemessen, sofern die resultierenden Zeichen später mit der **DataInputStream**-Methode **readUTF()** wieder eingelesen werden sollen (vgl. Abschnitt 15.3.2.4). Für universell verwendbare Textdateien ist die Klasse **OutputStreamWriter** mit einstellbarer und normkonformer Kodierung weit besser geeignet.

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 bezeichnet:

```
import java.io.*;
import java.nio.file.*;

class DataOutputStreamDemo {
 public static void main(String[] args) {
  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");
  } catch (IOException ioe) {
   System.err.println(ioe);
   System.exit(1);
  }
  try (DataInputStream dis = new DataInputStream(Files.newInputStream(file))) {
   System.out.println("readInt() - Ergebnis:\t\t"+dis.readInt()+
                      "\nreadDouble() - Ergebnis:\t"+dis.readDouble()+
                      "\nreadUTF() - Ergebnis:\t\t"+dis.readUTF());
  } catch (IOException ioe) {
   System.err.println(ioe);
  }
 }
}
```

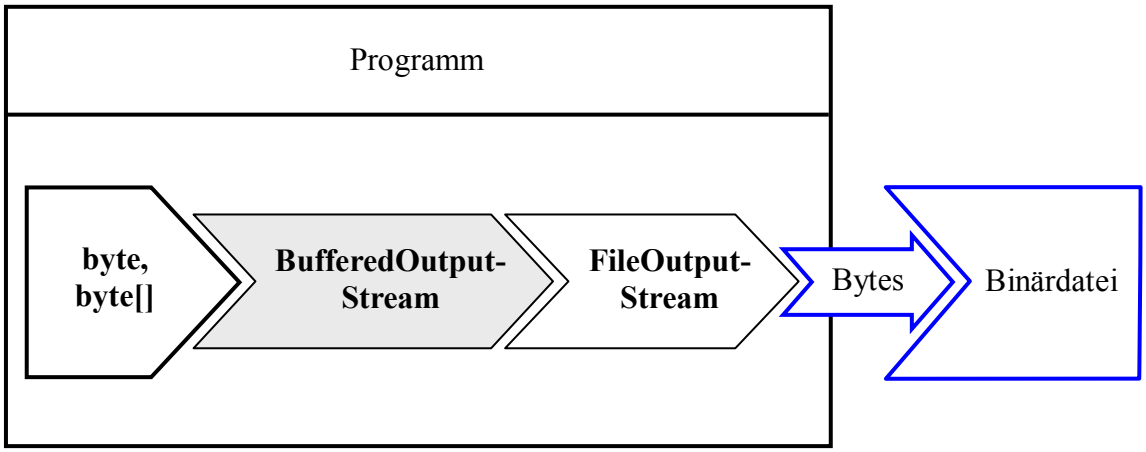
Ein **DataInputStream** holt in Kooperation mit einem **InputStream** die Werte zurück (vgl. Abschnitt 15.3.2):

```
readInt() - Ergebnis:          4711
readDouble() - Ergebnis:       3.141592653589793
readUTF() - Ergebnis:          DataOutputStream-Demo
```

15.3.1.5 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 15.1.4). 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 500.000 **write()** - Aufrufen jeweils ein einzelnes Byte in eine Datei, zunächst ungepuffert, dann unter Verwendung eines **BufferedOutputStream**-Objekts:

```
import java.io.*;

class BufferedOutputStreamFile {
 final static long ANZAHL = 500_000;
 public static void main(String[] args) {
  long time;

  try (FileOutputStream fos = new FileOutputStream("Buffer.dat")) {
   time = System.currentTimeMillis();
   for (int i = 1; i <= ANZAHL; i++)
    fos.write(i);
   System.out.println("Zeit für die ungepufferte Ausgabe: " +
     (System.currentTimeMillis() - time));
  } catch (IOException ioe) {
   System.err.println(ioe);
   System.exit(1);
  }

  try (BufferedOutputStream bos = new BufferedOutputStream(
                                  new FileOutputStream("Buffer.dat"))) {
   time = System.currentTimeMillis();
   for (int i = 1; i <= ANZAHL; i++)
    bos.write(i);
   System.out.println("Zeit für die gepufferte Ausgabe:   " +
     (System.currentTimeMillis() - time));
  } catch (IOException ioe) {
   System.err.println(ioe);
  }
 }
}
```

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: 842
Zeit für die gepufferte Ausgabe:   16
```

Noch drastischer ist der Zeitgewinn durch die gepufferten Ausgabe, wenn sich die Ausgabedatei auf einer Freigabe im lokalen Netzwerk befindet:[2]

```
Zeit für die ungepufferte Ausgabe: 254976
Zeit für die gepufferte Ausgabe:   47
```

Wegen des erheblichen Performanzvorteils sollte also ein Ausgabepuffer eingesetzt werden, wenn zahlreiche Schreibvorgänge mit jeweils kleinem Volumen stattfinden. Das `FileCopy`-Beispielprogramm in Abschnitt 15.3.1.2 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()** geschehen, die **flush()** aufruft und anschließend den zugrunde lie-

[1] Rechner mit Intel Core i7 860 (2,8 GHz)
[2] Netzwerkanbindung mit 1,0 GBit/s

genden **OutputStream** schließt, wie ein Blick in den Quellcode der **BufferedOutputStream**-Basisklasse **FilterOutputStream** zeigt:[1]

```
public void flush() throws IOException {
    out.flush();
}

public void close() throws IOException {
    try (OutputStream ostream = out) {
        flush();
    }
}
```

Im Unterschied zu **FileOutputStream** überschreiben weder **FilterOutputStream** noch **BufferedOutputStream** die von **Object** geerbte **finalize()** - Methode, so dass beim Terminieren eines **BufferedOutputStream**-Objekts per Garbage Collector *kein* **close()** - und insbesondere *kein* **flush()** - Aufruf erfolgt. Auf die **finalize()** - Methode sollte man sich allerdings generell *nicht* verlassen, weil ihr Aufruf nicht garantiert ist (vgl. Abschnitt 15.1.5).

15.3.1.6 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 15.1.3 sind bei der Zeichenstromverarbeitung die Klassen aus der später ins Java-API aufgenommenen **Writer**-Hierarchie zu bevorzugen. Diese haben bei der Textausgabe (Datentypen **String**, **char**) den Vorteil, dass für die Umsetzung des Java-internen Unicodes in die bevorzugte Textkodierung der Ausgabe ein Kodierungsschema gewählt werden kann (siehe Abschnitt 15.4.1.2). Allerdings ist die Klasse **PrintStream** trotzdem *nicht* überflüssig, weil z.B. der per **System.out** ansprechbare Standardausgabestrom ein **PrintStream**-Objekt ist. Dies gilt auch für den Standard*fehler*ausgabestrom, der über die Klassenvariable **System.err** ansprechbar ist.[2]

Ein **PrintStream**-Objekt kann mit Hilfe seiner vielfach überladenen Methoden **print()** und **println()** Werte mit beliebigem Datentyp ausgeben, z.B.:

Quellcode	Ausgabe
`class PrintStreamConsole {` `  public static void main(String[] args) {` `  PrintStreamConsole wob = new PrintStreamConsole();` `  System.out.println("Ein PrintStream kann Variablen"` `  +"\nbel. Typs verarbeiten,"` `  +"\nz.B. die double-Zahl\n" + "  " + Math.PI` `  +"\noder auch das Objekt\n" + "  " + wob);` `  }` `}`	`Ein PrintStream kann Variablen` `bel. Typs verarbeiten,` `z.B. die double-Zahl` `  3.141592653589793` `oder auch das Objekt` `  PrintStreamDemo@16f0472`

Seit Java 5.0 (alias 1.5) ist auch die **PrintStream**-Methode **printf()** (alias **format()**) zur formatierten Ausgabe verfügbar, die schon in Abschnitt 3.2.2 vorgestellt wurde.

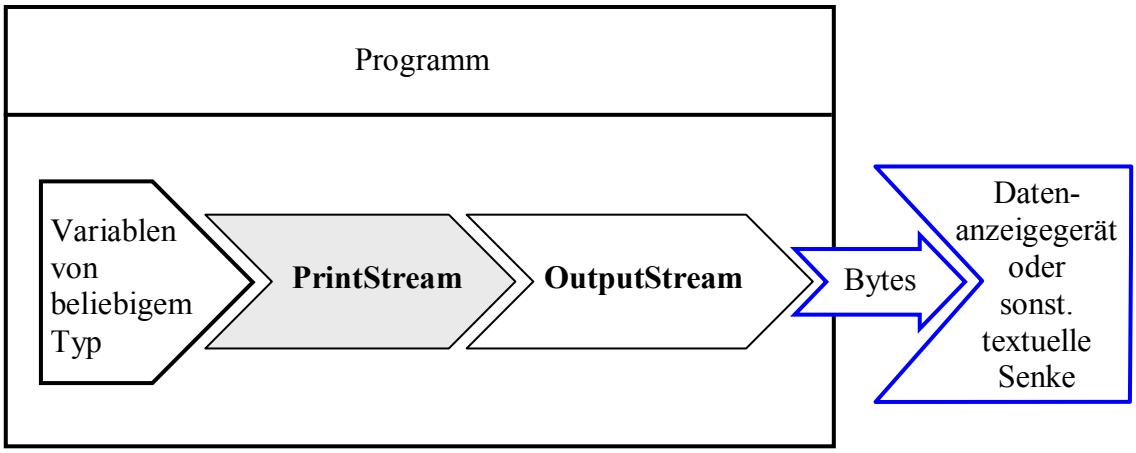
---

[1] Sie finden diese Methodendefinition in der Datei **FilterOutputStream.java**, die wiederum im Archiv **src.zip** mit den API-Quelltexten steckt. Das Quelltextarchiv kann bei der JDK-Installation auf die Festplatte befördert werden.

[2] Wird z.B. ein Ausnahmeobjekt über die Methode **printStackTrace()** beauftragt, die Aufrufsequenz auszugeben, dann landet diese im Fehlerausgabestrom.

Im Unterschied zu anderen **OutputStream**-Unterklassen werfen die **PrintStream**-Methoden *keine* **IOException**. Stattdessen setzen sie ein Fehlersignal, das mit **checkError**() abgefragt werden kann. Es wäre in der Tat recht umständlich, jeden Aufruf der Methode **System.out.println()** in einen überwachten **try**-Block zu setzen.

Generell kann man die **PrintStream**-Arbeitsweise folgendermaßen darstellen:



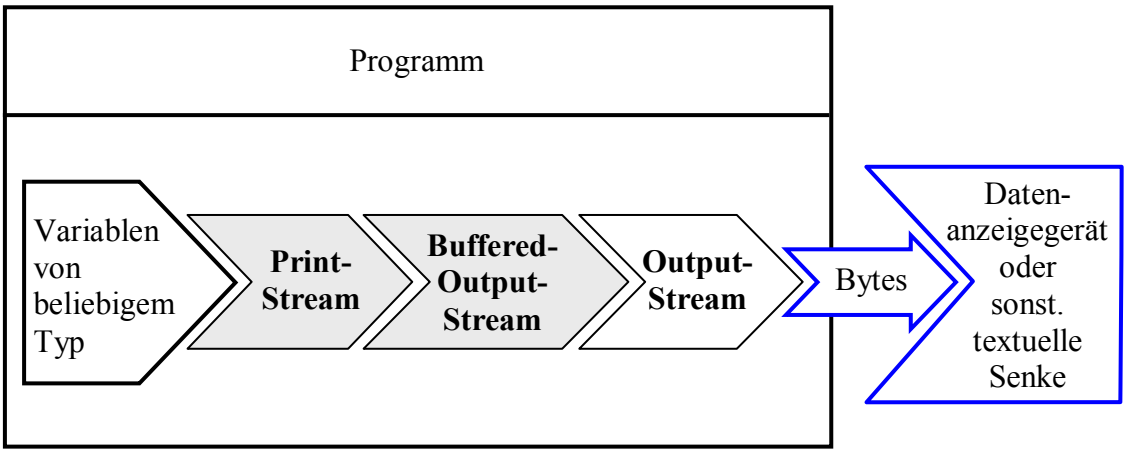
Im nächsten Beispiel ist zu sehen, wie mit Hilfe der Transformationsklasse **PrintStream** Werte primitiver Datentypen in eine *Textdatei* geschrieben werden (über einen **FileOutputStream**):

```
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:



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 15.3.1.5).
- Impliziter Aufruf der **PrintStream**-Methode **close()** durch Verwendung einer **try**-Anwendung 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 Ausgabe eines Newline-Zeichens (**\n**)
  - nach Ausführen 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 gutes Beispiel für die Kombination aus einem **PrintStream** und einem **BufferedOutputStream** ist der per **System.out** ansprechbare Standardausgabestrom, der analog zu folgendem Codefragment initialisiert wird:

```
FileOutputStream fdout =
    new FileOutputStream(FileDescriptor.out);
BufferedOutputStream bos =
    new BufferedOutputStream(fdout, 128);
PrintStream ps =
    new PrintStream(bos, true);
System.setOut(ps);
```

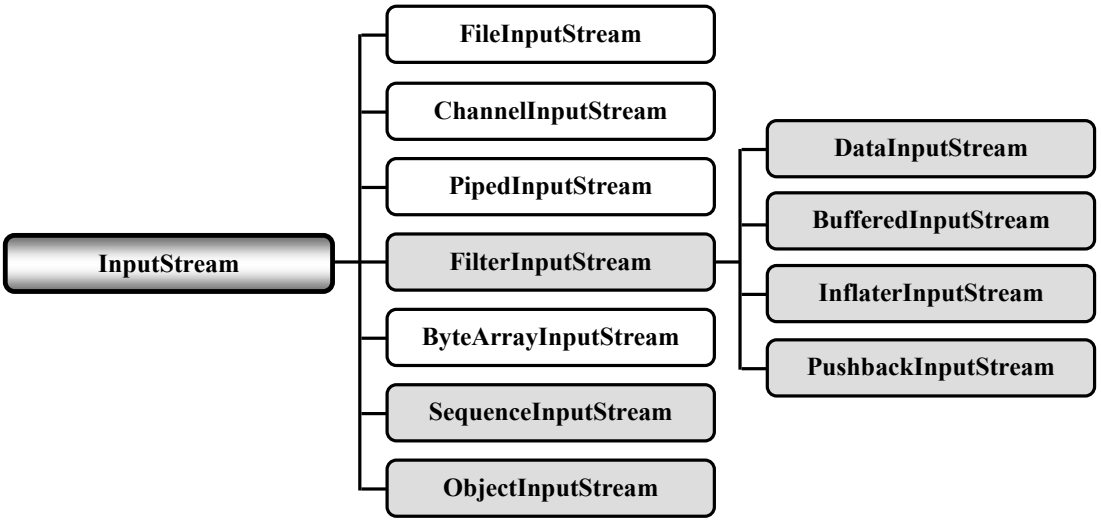
Mit der statischen Variablen **out** der Klasse **FileDescriptor** wird der Bezug zur Konsole hergestellt. Im **PrintStream**-Konstruktor wird für den **autoFlush**-Parameter der Wert **true** verwendet. Über die **System**-Methode **setOut()** kann ein selbst entworfener Strom als Standardausgabe in Betrieb genommen werden.

Bei der Textausgabe ist in der Regel die modernere und flexiblere Klasse **PrintWriter** zu bevorzugen (siehe Abschnitt 15.4.1.5). 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.

15.3.2 Die InputStream-Hierarchie

15.3.2.1 Überblick

Um Ihnen das Blättern zu ersparen, wird die schon in Abschnitt 15.1.3 gezeigte Abbildung zur **InputStream**-Hierarchie wiederholt (Eingabeklassen mit weißem Hintergrund und Eingabe*transformations*klassen mit grauem Hintergrund):



Mit der Transformationsklasse **ObjectInputStream** werden wir uns in Abschnitt 15.6 näher beschäftigen. Sie ermöglicht das Einlesen kompletter Objekte aus einem Byte-Strom.

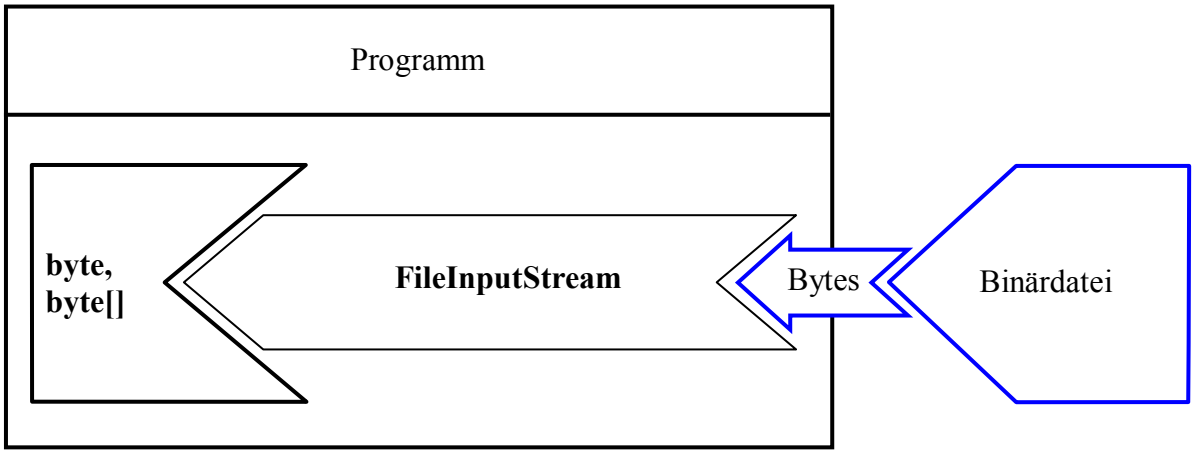
Durch ein Tandem aus einem **PipedOutputStream** und einem verbundenen **PipedInputStream** lässt sich ein unidirektionaler Datentransfer zwischen zwei Threads (Ausführungsfäden) einrichten. Der erste Thread schreibt Bytes in den **PipedOutputStream** und der zweite Thread liest aus dem verbundenen **PipedInputStream**. In Abschnitt 16.2.7.2 wird die Realisation einer Produzenten-Konsumenten - Kooperation mit der Pipe-Technik beschrieben.

Die folgenden Klassen können nur kurz vorgestellt werden:

- **ByteArrayInputStream**
  Objekte dieser Eingabeklasse lesen Bytes aus einem **byte**-Array.
- **SequenceInputStream**
  Mit Hilfe dieser Transformationsklasse kann man mehrere Objekte aus **InputStream**-Unterklassen hintereinander koppeln und als einen einzigen Strom behandeln. Ist das Ende eines Eingabestroms erreicht, wird er geschlossen, und der nächste Strom in der Sequenz wird geöffnet.
- **BufferedInputStream**
  Analog zum **BufferedOutputStream** (siehe Abschnitt 15.3.1) 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 zurück zu stellen, was z.B. dann sinnvoll ist, wenn nach einer vorausschauenden Prüfung die eigentliche Verarbeitung durch ein anderes Stromobjekt erfolgen soll.

15.3.2.2 FileInputStream

Mit einem **FileInputStream** kann man Bytes aus einer Datei lesen:



Ein Beispielprogramm mit **FileInputStream**-Beteiligung war schon im Abschnitt 15.3.1.2 (über den **FileOutputStream)** zu sehen.

Im **FileInputStream**-Konstruktor kann die anzusprechende Datei über ein **File**-Objekt (siehe Abschnitt 15.2.2) oder über einen **String** spezifiziert werden:

- **public FileInputStream**(**File** *file*)
- **public FileInputStream(String** *name***)**

Scheitert das Öffnen der Datei, werfen die Konstruktoren eine Ausnahme vom Typ **FileNotFoundException**, auf die ein Aufrufer vorbereitet sein muss.

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 15.2.1.1).

Eine Auswahl der **FileInputStream**-Methoden:

- **int read()**
  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.
- **int read(byte[]** *b***)**
  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.
- **int available()**
  Laut API-Dokumentation schätzt **available()**, wie viele Bytes eine Methode aus dem Strom lesen kann, ohne auf Daten warten zu müssen.

Im folgenden Codefragment kommen die drei beschriebenen Methoden zum Einsatz:

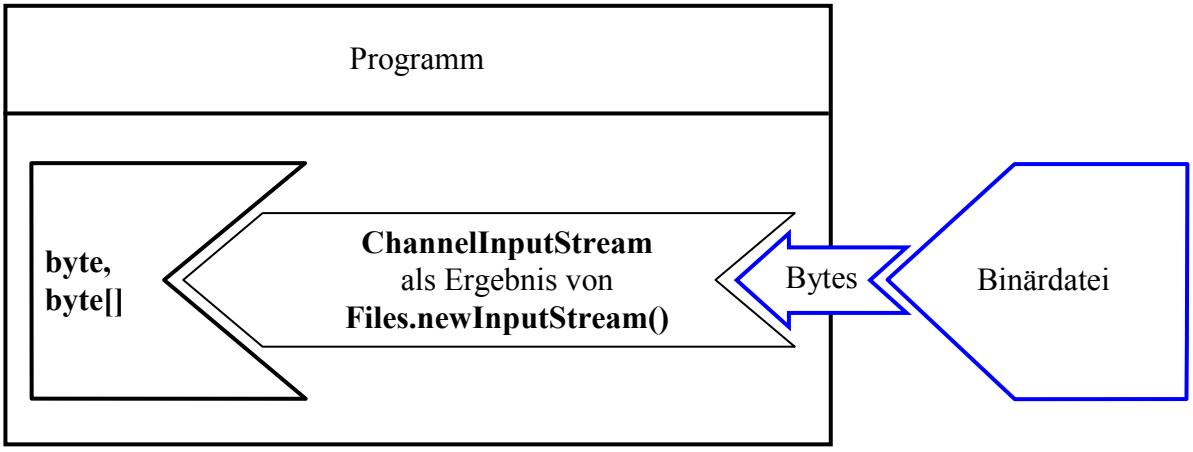
```
int anfang, gelesen;
byte[] rest = new byte[10];
try (FileInputStream fis = new FileInputStream(name)) {
   System.out.println("Verfügbar in Datei:    " + fis.available());
   anfang = fis.read();
   System.out.println("Verfügbar nach read(): " + fis.available());
   gelesen = fis.read(rest);
   System.out.print("Eingelesen:            " + anfang);
   for (int i = 0; i < gelesen; i++)
      System.out.print(rest[i]);
} catch (IOException ioe) {
      System.err.println(ioe);
      System.exit(1);
}
```

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
```

15.3.2.3 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 15.7.1):

public static InputStream newInputStream(Path *path*, **OpenOption...** *options*)
throws IOException

Über die Methode **getClass()** befragt, nennt das Rückgabeobjekt als seinen Typ die von **InputStream** abstammende Klasse **ChannelInputStream** im Paket **sun.nio.ch**. Offenbar kommt im Hintergrund bei den Dateizugriffen die moderne Channel-Technik zum Einsatz, wobei ein Geschwindigkeitsvorteil möglich, aber nicht garantiert ist.

Wird kein **OpenOption** - Parameter angegeben, ist der Enumerationswert **StandardOpenOption.READ** in Kraft

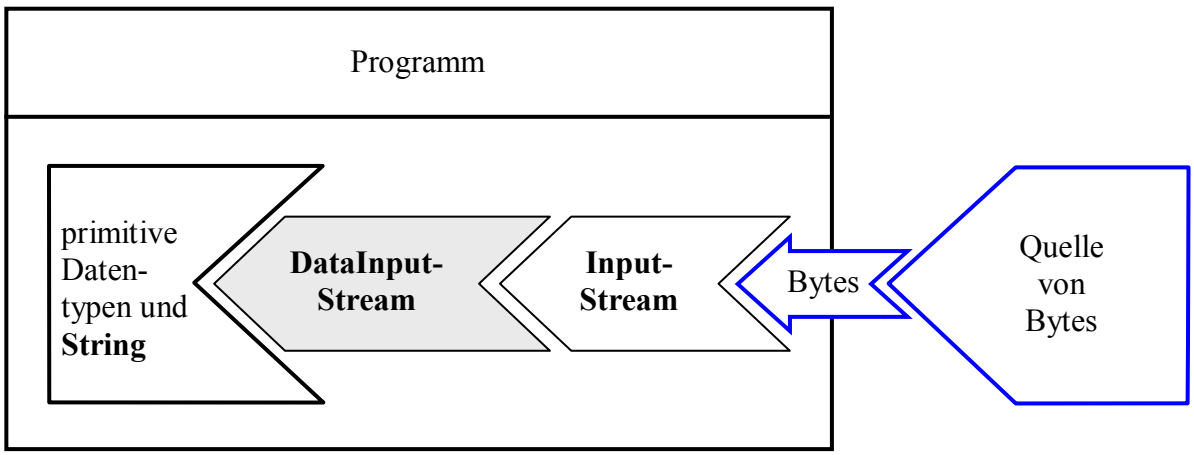
Im Vergleich zu einem traditionell per Konstruktor erzeugten **FileInputStream** bestehen folgende Vorteile:

- Es lassen sich Öffnungsoptionen für die Datei angeben (siehe Abschnitt 15.7.1).
- Es kommt die moderne Channel-Technik zum Einsatz.
- Im Unterschied zu einem **FileInputStream** ist die Nutzung durch mehrere Threads (Ausführungsfäden, siehe Kapitel 16) erlaubt.

Ein Einsatzbeispiel für die Methode **newInputStream()** war schon in Abschnitt 15.1.2 zu sehen.

15.3.2.4 DataInputStream

Die Transformationsklasse **DataInputStream** liest primitive Datentypen sowie **String**-Objekte aus einem Bytestrom und ist uns zusammen mit ihrem Gegenstück **DataOutputStream** schon in Abschnitt 15.3.1.3 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 15.4.1.2) vorausgesetzt wird. Diese Methode ist nur dann angemessen, wenn die Zeichen mit der **DataOutputStream**-Methode **writeUTF()** geschrieben worden sind (vgl. Abschnitt 15.3.1.4).

15.4 Klassen zur Verarbeitung von Zeichenströmen

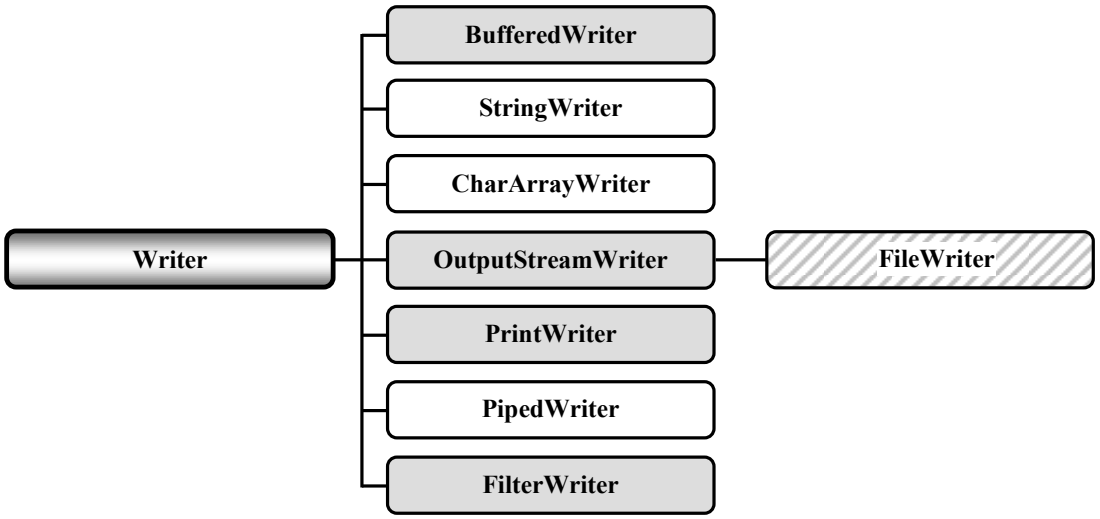
Java verwendet intern zur Repräsentation von Zeichen den Unicode, und die Klassen zur Verarbeitung von Zeichenströmen müssen beim Lesen und Schreiben zwischen der internen Repräsentation und der extern benötigten (z.B. der lokalüblichen) Zeichenkodierung vermitteln.

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 in der Regel gegenüber den älteren Bytestromklassen (vor allem wegen der unproblematischen Internationalisierung) zu bevorzugen.

15.4.1 Die Writer-Hierarchie

15.4.1.1 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 Klasse **FileWriter** mit einer Datei verbunden ist *und* eine Transformationsfunktion besitzt, ist sie mit schraffiertem Hintergrund dargestellt.

Bei den folgenden **Writer**-Subklassen beschränken wir uns auf kurze Hinweise:

- **StringWriter** und **CharArrayWriter**
  StringWriter schreiben in einen dynamisch wachsenden **StringBuffer** (siehe Abschnitt 5.2.2). Im folgenden Beispiel werden die auszugebenden Zeichen von einem **PrintWriter** (siehe Abschnitt 15.4.1.5) geliefert:

Quellcode	Ausgabe
`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());` `  }` `}`	`Zeile 1` `Zeile 2` `Zeile 3` `Zeile 4` `Zeile 5`

  CharArrayWriter-Objekte verhalten sich im Wesentlichen genauso.

- **PipedWriter**
  Diese Klasse ist das zeichenorientierte Analogon zu Klasse **PipedOutputStream**.

- **FilterWriter**
  Diese abstrakte Basisklasse bietet sich dazu an, eigene Transformationsklassen für zeichenorientierte Ausgabeströme abzuleiten.

15.4.1.2 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 Auswahl der ab Java SE 7 unterstützten Kodierungen (*Basic Encoding Set*, enthalten in **.../lib/rt.jar**):[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>7-bit-ASCII-Code
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-8850-1 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 generell voreingestellt (auch bei anderen Datentypen).
Beim großen griechischen Delta (\u0394) wird ausgegeben: 03 94</td></tr>
<tr><td>UTF-16LE</td><td>UnicodeLittleUnmarked</td><td>Für alle Unicode-Zeichen werden 16 Bit in Little-Endian - Reihenfolge ausgegeben: Das niederwertige Byte zuerst.
Bei großen griechischen Delta (\u0394) wird ausgegeben: 94 03</td></tr>
<tr><td>Windows-1252</td><td>Cp1252</td><td>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.</td></tr>
<tr><td>IBM850</td><td>Cp850</td><td>MS-DOS Latin-1
MS-DOS-Codepage zur Verwendung in Westeuropa</td></tr>
</table>

Dass Klassen aus dem Paket **java.io** (z.B. **OutputStreamWriter**) für die Kodierungen andere Namen benutzen als Klassen aus den Paketen **java.nio.*** (z.B. **Files**), ist sehr bedauerlich. Bisher sind mir allerdings bei der versehentlichen Benutzung einer Bezeichnung aus der falschen Serie noch keine Probleme begegnet.

---

[1] Die Angaben stammen von der Webseite
http://docs.oracle.com/javase/8/docs/technotes/guides/intl/encoding.doc.html
Dort werden noch weitere unterstützte Kodierungsschemata aufgelistet.

Bei den folgenden Überladungen des **OutputStreamWriter**-Konstruktors kann das gewünschte Kodierungsschema über seinen IANA-Namen oder über ein **Charset**-Objekt angegeben werden:

- **public OutputStreamWriter(OutputStream** *out***, String** *charsetName*)
  throws UnsupportedEncodingException
- **public OutputStreamWriter(OutputStream** *out***, Charset** ***cs***)

Wird im **OutputStreamWriter**-Konstruktor kein Kodierungsschema angegeben,

public OutputStreamWriter(OutputStream *out*)

dann entscheidet die Systemeigenschaft **file.encoding**. Ihr Wert kann mit der **System**-Methode **getProperty()** ermittelt werden, z.B.:

Quellcodesegment	Ausgabe
`String denc = System.getProperty("file.encoding");` `System.out.println(denc);`	`UTF-8`

Weil die voreingestellte Kodierung vom Wirtbetriebssystem abhängt, sollte ein Java-Programm in der Regel die Kodierung festlegen.[1]

Zur Ausgabe stehen die folgenden, teilweise von **Writer** geerbten **write()** - Überladungen zur Verfügung:

- **public void write(int** *c***)**
  throws IOException
  Die beiden niederwertigen Bytes des Aktualparameters 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.

Im folgenden Programm werden die oben beschriebenen Kodierungsschemas 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:

---

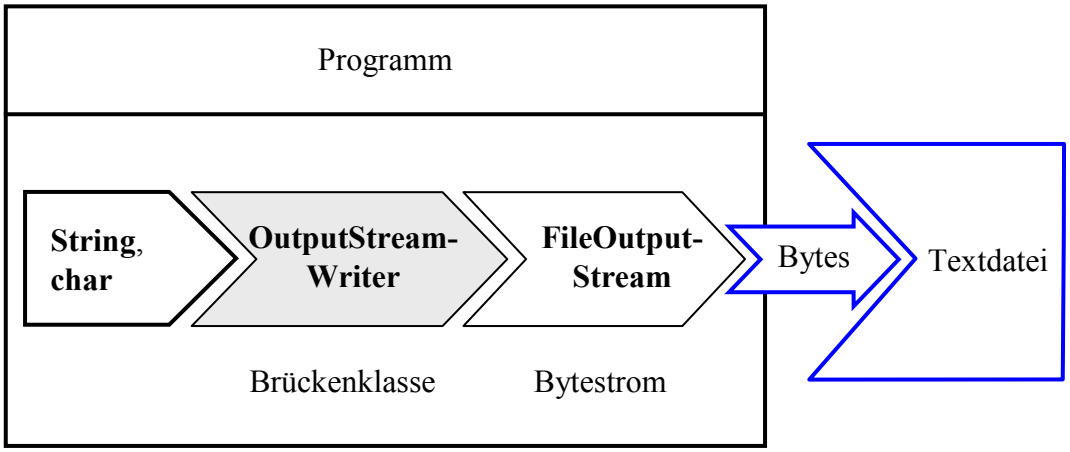
[1] Üblicherweise wird für Mac OS X und Linux als voreingestellte Kodierung UTF-8 erwartet, für Windows hingegen Cp1252. Auf meinem Rechner mit Windows 7 und Java 8 (genauer: 1.8.0_66) hat die Systemeigenschaft **file.encoding** allerdings den Wert UTF-8.

```
import java.io.*;
import java.nio.file.*;

class OutputStreamWriterDemo {
 public static void main(String[] args) throws IOException {
  String[] encodings = {"ASCII", "ISO8859_1", "UTF8",
                        "UnicodeBigUnmarked", "UnicodeLittleUnmarked",
                        "Cp1252", "Cp850"};
  Files.deleteIfExists(Paths.get("test.txt"));
  for (int i = 0; i < encodings.length; i++) {
   try (OutputStreamWriter osw = new OutputStreamWriter(
                           new FileOutputStream("test.txt", true), encodings[i])) {
    osw.write(encodings[i] + " ae = ä" + System.lineSeparator());
   }
  }
 }
}
```

Für das Unicode-Zeichen \u00E4 wird jeweils ausgegeben:

Kodierungsschema	Byte(s) in der Ausgabe (Hexadezimal)
ASCII	3F
ISO8859_1	E4
UTF8	C3 A4
UnicodeBigUnmarked	00 E4
UnicodeLittleUnmarked	E4 00
Cp1252	E4
Cp850	84

Das Beispielprogramm arbeitet mit folgender Datenstromarchitektur:



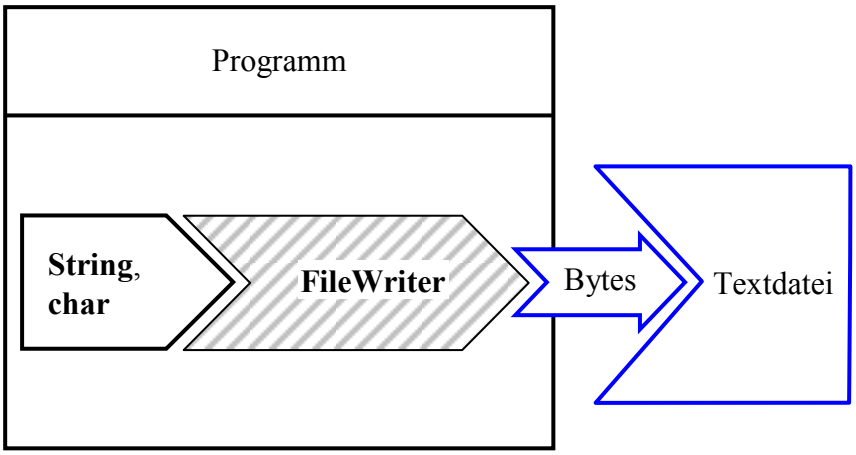
OutputStreamWriter (und auch die Ableitung **FileWriter**, siehe Abschnitt 15.4.1.3) sammeln die per Unicode-Wandlung entstandenen Bytes zunächst in einem internen Puffer (Größe: 8192 Bytes), den ein Objekt der Klasse **StreamEncoder** aus dem Paket **sun.nio.cs** verwaltet. Daher muss auf jeden Fall vor dem Ableben eines **OutputStreamWriter**-Objekts (z.B. beim Programmende) der Puffer geleert werden. Dazu stehen mehrere Möglichkeiten bereit:

- 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**-Anwendung 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 keine Pufferentleerung statt.

15.4.1.3 FileWriter

Die von **OutputStreamWriter** abgeleitete Klasse **FileWriter** im Paket **java.io** eignet sich zur Ausgabe von **String**- und **char**-Variablen in eine Textdatei, wenn das voreingestellte Kodierungsschema (siehe Abschnitt 15.4.1.2) akzeptabel ist:



In obiger Darstellung der **Writer**-Hierarchie (siehe Abschnitt 15.4.1.1) erhielt die Klasse **FileWriter** einen schraffierten Hintergrund, denn ihre Objekte …

- stehen (über ein Instanzobjekt aus der Klasse **FileOutputStream**) mit einer Datei in Kontakt, so dass sie als Ausgabestrom arbeiten können,
- transformieren den eingehenden Zeichenstrom in einen Bytestrom, so dass sie als Filterobjekte bezeichnet werden können.

Wichtiger als die akademische Bemerkung zur korrekten Klassifikation der Klasse **FileWriter** sind ihre Konstruktoren. Die per **String**- oder **File**-Objekt bestimmte Datei 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 15.2.1.1).

Zur Ausgabe von Textdaten stehen die von **Writer** bzw. **OutputStreamWriter** geerbten **write()** -Überladungen zur Verfügung. Das folgende Programm schreibt ein einzelnes Zeichen und eine Zeichenfolge jeweils in eine eigene Zeile, und trennt die beiden Zeilen durch die 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");
  }
 }
}
```

Ist das voreingestellte Kodierungsschema ungeeignet, muss man auf die **FileWriter**-Bequemlichkeit verzichten, einen **OutputStreamWriter** mit passendem Kodierungsschema erstellen und einen **FileOutputStream** dahinter schalten (siehe Abschnitt 15.4.1.2).

15.4.1.4 BufferedWriter

Am Ende von Abschnitt 15.4.1.2 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.

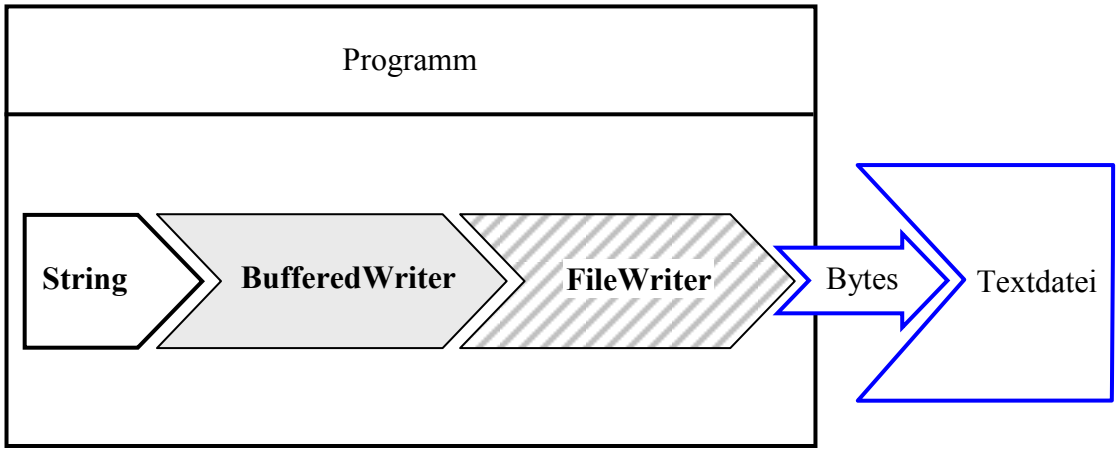
Vom folgenden Programm werden mit Hilfe eines **FileWriter**s (siehe Abschnitt 15.4.1.3) zweimal jeweils 5.000.000 Zeilen in eine Textdatei geschrieben, zunächst ohne und dann mit zwischengeschaltetem **BufferedWriter**. Weil die erste Ausgabe unabhängig von der verwendeten Technik stets länger dauert, findet zunächst ein „Warmlaufen“ statt:

```
import java.io.*;

class BufferedWriterDemo {
 public static void main(String[] args) throws IOException {
  final int ANZAHL = 5_000_000;

  try (FileWriter fw = new FileWriter("test1.txt")) {
   long time = System.currentTimeMillis();
   for (int i = 1; i <= ANZAHL; i++)
    fw.write("Zeile " + i + System.lineSeparator());
   System.out.println("Benötigte Zeit beim Warmlaufen:            "
     +(System.currentTimeMillis()-time));
  }

  try (FileWriter fw = new FileWriter("test2.txt")) {
   long time = System.currentTimeMillis();
   for (int i = 1; i <= ANZAHL; i++)
    fw.write("Zeile " + i + System.lineSeparator());
   System.out.println("Benötigte Zeit ohne BufferedWriter:        "
     +(System.currentTimeMillis()-time));
  }

  try (BufferedWriter bw =  new BufferedWriter(new FileWriter("test3.txt"))) {
   long time = System.currentTimeMillis();
   for (int i = 1; i <= ANZAHL; i++)
    bw.write("Zeile " + i + System.lineSeparator());
   System.out.println("Benötigte Zeit mit BufferedWriter:         "
     +(System.currentTimeMillis()-time));
  }
 }
}
```

Dass der Gewinn durch den **BufferedWriter** recht bescheiden ausfällt, liegt vermutlich an der eingangs erwähnten Byte-Pufferung durch den **FileWriter**:[1]

```
Benötigte Zeit beim Warmlaufen:           831
Benötigte Zeit ohne BufferedWriter:       767
Benötigte Zeit mit BufferedWriter:        589
```

Die performantere Datenstrompipeline hat folgende Architektur:



[1] Die Beobachtungen wurden auf einem Rechner mit der CPU Intel Core i7 860 (2,8 GHz) durchgeführt.

15.4.1.5 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** weiter lebt (vgl. Abschnitt 15.3.1.6). 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 in 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.

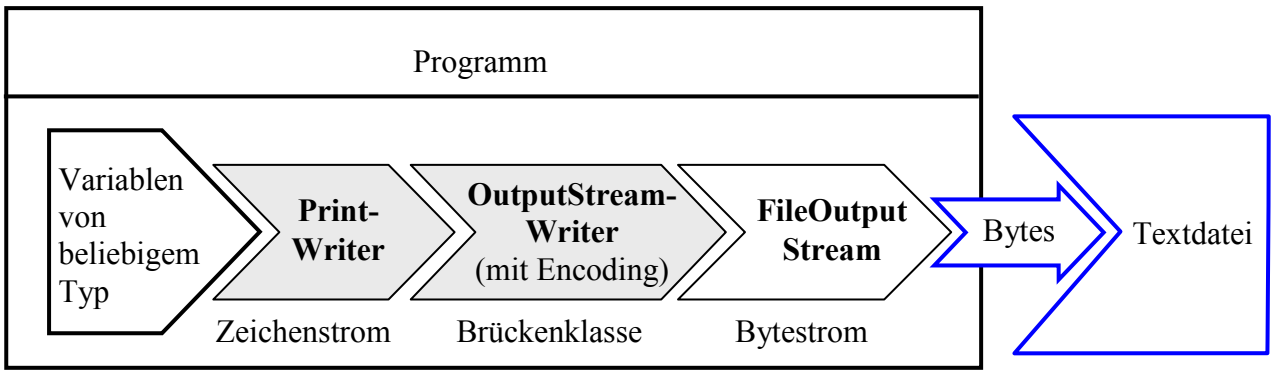
Wie die folgende Auswahl der **PrintWriter**-Konstruktoren zeigt, dürfen die angekoppelten Datenstromobjekte von den Basisklassen **OutputStream** oder **Writer** abstammen:

- **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 einem Kodierungsschema in Byte-Sequenzen übersetzt. Diese Bytes werden auf dem Weg zur Datensenke zwischengespeichert (vgl. Abschnitt 15.4.1.2). 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 *interaktiven* Anwendung unbedingt erforderlich, sollte aber bei einer Dateiausgabe aus Performanzgründen vermieden werden.

Gibt man im **PrintWriter**-Konstruktor *explizit* einen **OutputStreamWriter** an, kann man die Kontrolle über das Kodierungsschema übernehmen. Diese Möglichkeit stellt den entscheidenden Vorteil der Klasse **PrintWriter** gegenüber dem Vorgänger **PrintStream** dar (vgl. Abschnitt 15.4.1.2). Bei der Ausgabe von numerischen Daten in eine Textdatei spielt die Wahlfreiheit beim Kodierungsschema natürlich keine Rolle. Insgesamt ist die Klasse **PrintWriter** bei der Ausgabe in Textdateien zu bevorzugen, weil sich damit alle Aufgaben bewältigen lassen.

Mit der folgenden Ausgabestromkonstruktion wird die Flexibilität der **Writer**-Subklassen bei der Ausgabe in eine Textdatei ausgeschöpft:

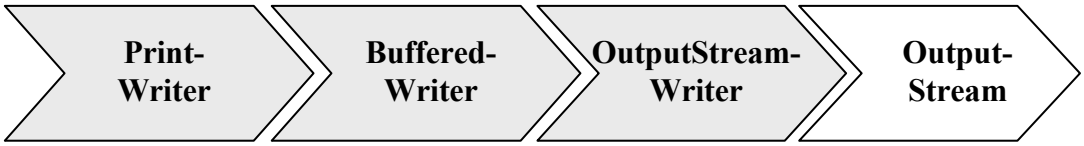


Wird ein **PrintWriter** auf einen Ausgabestrom aus der **OutputStream**-Hierarchie gesetzt, dann nimmt der Konstruktor insgeheim einen **BufferedWriter**, der Zeichen zwischenspeichert (siehe

Abschnitt 15.4.1.4), und einen **OutputStreamWriter** mit dem voreingestellten Kodierungsschema, der Bytes zwischenspeichert (siehe Abschnitt 15.4.1.2), in Betrieb:[1]

```
public PrintWriter(OutputStream out, boolean autoFlush) {
 this(new BufferedWriter(new OutputStreamWriter(out)), autoFlush);
  . . .
}
```

Damit arbeitet insgesamt folgendes Gespann:



Vor dem Terminieren eines **PrintWriter**-Objekts müssen die Zwischenspeicher unbedingt entleert werden, z.B. per **close()** - Aufruf in einer **finally**-Klausel:

```
import java.io.*;
class PrintWriterFile {
 public static void main(String[] egal) throws IOException {
  PrintWriter pw = null;
  try {
   FileOutputStream fos = new FileOutputStream("pw.txt");
   pw = new PrintWriter(fos);
   for (int i = 1; i <= 3000; i++) {
    pw.println(i);
   }
  } finally {
   if (pw != null)
    pw.close();
  }
 }
}
```

Ohne **close()** - Aufruf fehlen in der Ausgabe des Beispielprogramms ca. 1500 Zeilen. Eben war noch einmal die traditionelle Ressourcenfreigabe aus Java 6 zu sehen, weil sich dabei der Effekt einer vergessenen **close()** - Anweisung leicht demonstrieren lässt. Über die seit Java 7 verfügbare **try-**Anweisung mit automatischer Ressourcenfreigabe kann man dem Compiler den **close()** - Aufruf an den **PrintWriter** überlassen und Aufwand sparen:

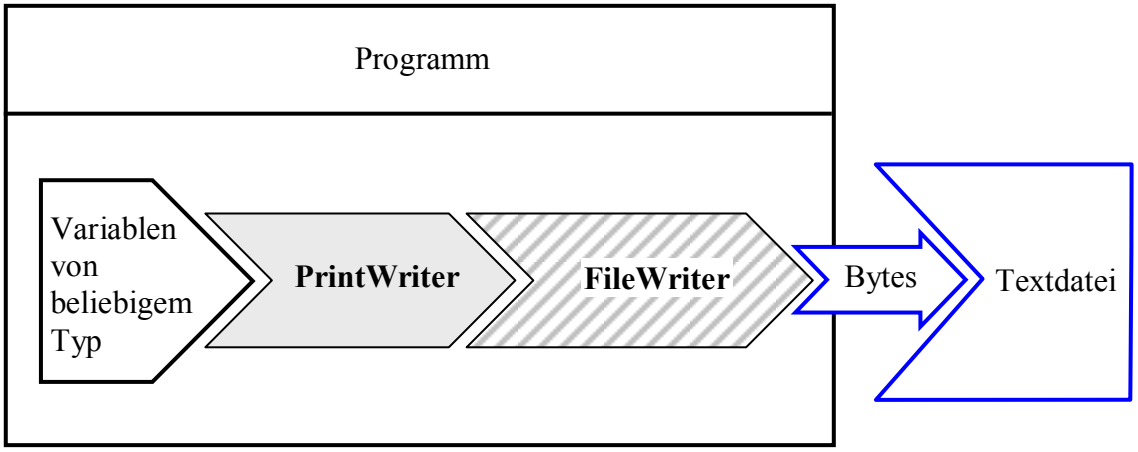
```
import java.io.*;
class PrintWriterFile {
 public static void main(String[] args) throws IOException {
  try (PrintWriter pw = new PrintWriter(
                  new FileOutputStream("pw.txt"))) {
   for (int i = 1; i <= 3000; i++) {
    pw.println(i);
   }
  }
 }
}
```

Der **PrintWriter** reicht den **close()** - Aufruf an die zugrunde liegenden Datenströme weiter, so dass im Beispiel auch die Dateiverbindung geschlossen wird.

---

[1] Sie finden diese Definition in der Datei **PrintWriter.java**, die wiederum im Archiv **src.zip** mit den API-Quelltexten steckt. Das Quelltextarchiv kann bei der JDK-Installation auf die Festplatte befördert werden.

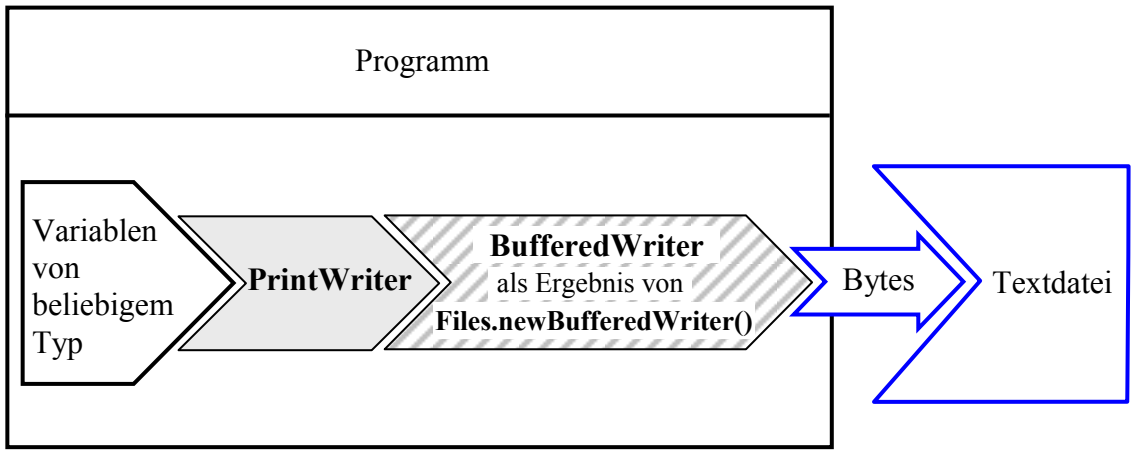
Wenn das voreingestellte Kodierungsschema (siehe Abschnitt 15.4.1.2) akzeptabel ist, taugt auch ein **FileWriter** (siehe Abschnitt 15.4.1.3) als Verbindungsstück zwischen einem **PrintWriter** und einer Textdatei:



Im Wesentlichen sollte sich dieses Gespann genauso verhalten wie die Kombination aus einem **PrintStream** und einem **FileOutputStream**, die ebenfalls mit dem voreingestellten Kodierungsschema arbeitet.

15.4.1.6 BufferedWriter mit Dateianschluss per NIO.2 - API

Wer beim gepufferten Schreiben von Zeichen die Verbindung zur Textdatei über das NIO.2 - API (vgl. Abschnitt 15.2.1) 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 15.7.1):

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 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 dem gewünschten Kodierungsschema 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 Öffnungsoptionen für die Datei angeben (siehe Abschnitt 15.7.1).
- Über das im Hintergrund per **newOutputStream()** erzeugte Ausgabeobjekt kommt die moderne Channel-Technik zum Einsatz.

Zur Ausgabe in eine Textdatei mit UTF-8 - Kodierung eignet sich die folgende Überladung ohne **Charset**-Parameter:

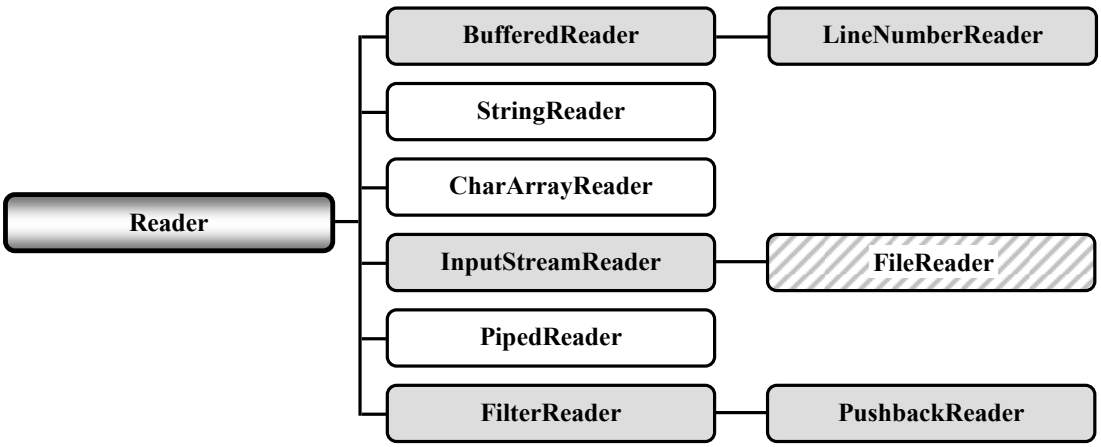
public static BufferedWriter newBufferedWriter(Path *path*, **OpenOption...** *options*)
throws IOException

15.4.2 Die Reader-Hierarchie

In der **Reader**-Hierarchie finden sich diverse Klassen zur Verarbeitung von zeichenorientierten *Eingabe*strömen.

15.4.2.1 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.

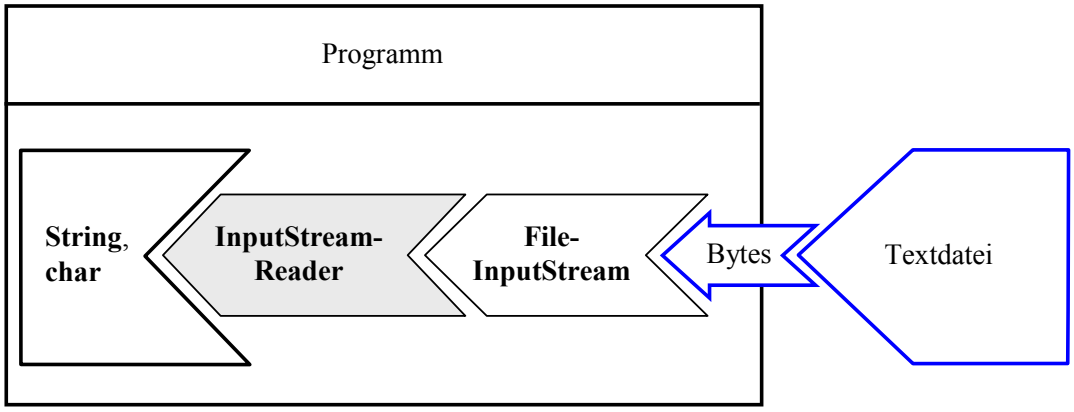
Bei den meisten **Reader**-Unterklassen beschränken wir uns auf kurze Hinweise:

- **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. **char**-Array.
- **PipedReader**
  Objekte dieser Klasse lesen Zeichen aus einer Pipe, 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 zurück zu stellen, 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 in Abschnitt 15.5 die (unmittelbar aus **java.lang.Object** abgeleitete) Klasse **Scanner** ansehen.

15.4.2.2 Brückenklasse InputStreamReader

Die Brückenklasse **InputStreamReader** ist das Gegenstück zur Klasse **OutputStreamReader**, wandelt also Bytes unter Verwendung eines einstellbaren Kodierungsschemas (vgl. Abschnitt 15.4.1.2) in Unicode-Zeichen:



Wie beim **OutputStreamReader** findet zur Beschleunigung der Konvertierung automatisch eine Pufferung des Byte-Stroms statt.

15.4.2.3 FileReader und BufferedReader

Die Klasse **FileReader** ist das Gegenstück zur Klasse **FileWriter**. Sie erhielt in obiger Darstellung der **Reader**-Hierarchie einen schraffierten Hintergrund, denn ihre Objekte …

- stehen (über ein Instanzobjekt aus der Klasse **FileInputStream**) mit einer Datei in Kontakt, so dass sie als Eingabestrom arbeiten können,

- transformieren (als **InputStreamReader**-Abkömmlinge) den eingehenden Byte-Strom in einen Zeichenstrom, so dass sie als Filterobjekte bezeichnet werden können.

Bei der Wandlung von Bytes in Unicode-Zeichen kommt das voreingestellte Kodierungsschema zum Einsatz (siehe Abschnitt 15.4.1.2). Wer ein alternatives Kodierungsschema benötigt, muss auf die **FileReader**-Bequemlichkeit verzichten, einen **InputStreamReader** mit passendem Kodierungsschema erstellen und mit einem **InputStream** kombinieren.

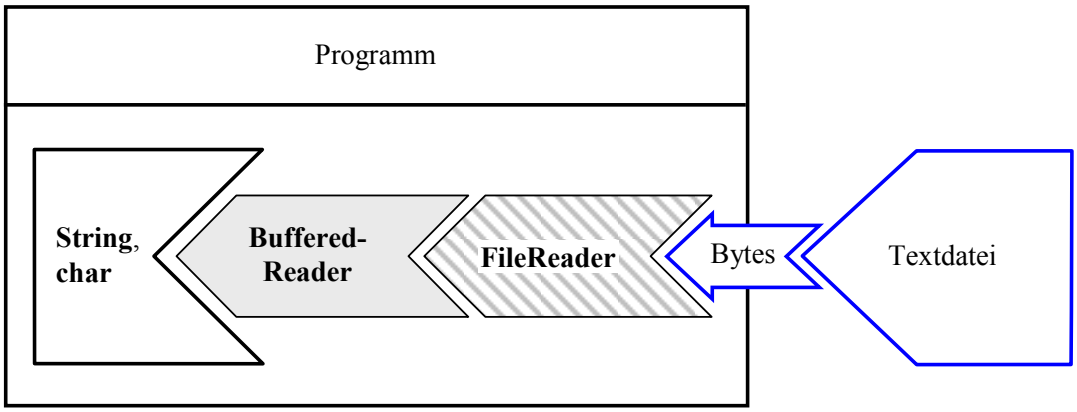
In der Regel setzt man vor den **FileReader** noch einen **BufferedReader**, der für eine Beschleunigung sorgt und außerdem die bei Dateien mit Zeilenstruktur sehr nützliche Methode **readLine()** zum Einlesen einer kompletten Zeile bietet, z.B.:

Quellcode	Ausgabe

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

class FileReaderDemo {
 public static void main(String[] args) throws IOException {
  ArrayList<String> als = new ArrayList<String>();
  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));
  }
 }
}
```

```
Zeile 100
```

Das Beispielprogramm arbeitet mit folgender Datenstrom-Architektur:

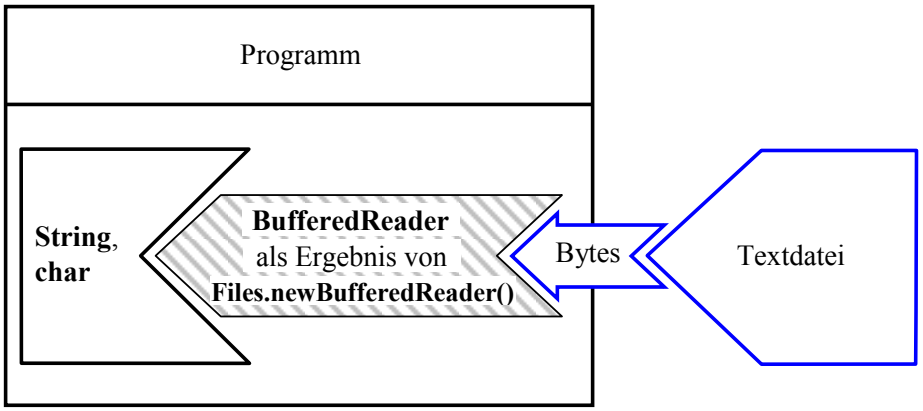


Die bei einem **BufferedReader** voreingestellte Puffergröße von 8192 Zeichen lässt sich per Konstruktor ändern.

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 15.2.1.1).

15.4.2.4 BufferedReader mit Dateianschluss per NIO.2 - API

Wer beim gepufferten Lesen von Zeichen die Verbindung zur Textdatei über das NIO.2 - API (vgl. Abschnitt 15.2.1) 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 API-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 dem gewünschten Kodierungsschema 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 15.4.2.3) 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 moderne Channel-Technik zum Einsatz.

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 15.4.2.3 lässt sich mit Hilfe der **Files**-Methode **newBufferedReader()** vereinfachen und sollte anschließend dank Channel-Technik effektiver arbeiten:

Quellcode	Ausgabe

```
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.*;

class BufferedReaderNio2 {
 public static void main(String[] args) throws IOException {
  ArrayList<String> als = new ArrayList<String>();
  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));
  }
 }
}
```

```
Zeile 100
```

15.5 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 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 Objekt der Klasse **BigDecimal** zu interpretieren, und wirft bei Misserfolg eine **InputMismatchException**.
- **public String next()**
  Die Methode holt das nächste Token.

Ob noch ein Token vorhanden und vom gewünschten Typ ist, kann mit einer entsprechenden Methode festgestellt werden, z.B.:

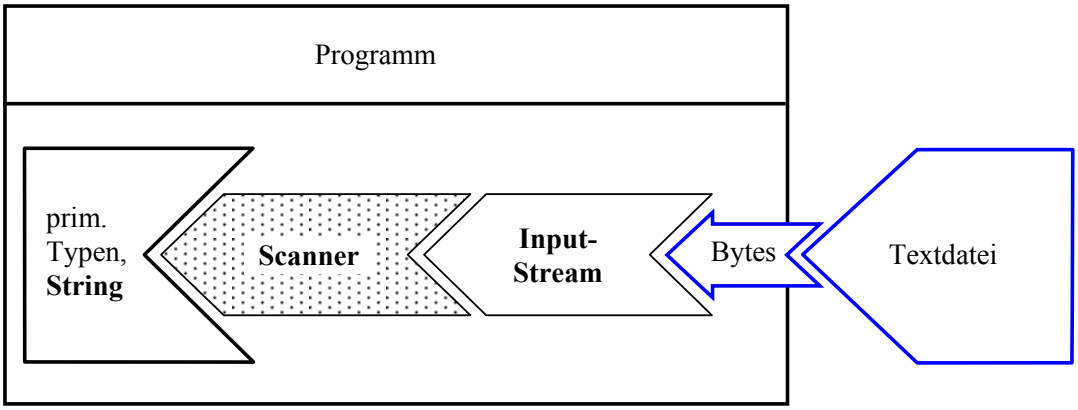
- **public boolean hasNext()**
  Prüft, ob noch ein Token vorhanden ist
- **public boolean hasNextInt()**
  Prüft, ob das nächste Token als **int**-Wert interpretierbar ist
- **public boolean hasNextDouble()**
  Prüft, ob das nächste Token als **double**-Wert interpretierbar ist

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 ein Kodierungsschema (vgl. Abschnitt 15.4.1.2) angegeben werden kann:

- **public Scanner(File** *source***)**
- **public Scanner(File** *source***, String** *charsetName***)**
- **public Scanner(Path** *source***)**
- **public Scanner(Path** *source***, String** *charsetName***)**
- **public Scanner(InputStream** *source***)**
- **public Scanner(InputStream** *source***, String** *charsetName***)**

Obwohl die Klasse **Scanner** weder von **InputStream** noch von **Reader** abstammt, benutzen wir doch die gewohnte Darstellung zur Veranschaulichung der Funktionsweise:



Das folgende Programm liest Zahlen und **String**-Objekte aus einer Textdatei:

```
import java.util.*;

class ScannerFile {
 public static void main(String[] args) {
  try (Scanner input = new Scanner(
                       new java.io.FileInputStream("daten.txt"), "Cp1252")) {
   while (input.hasNext())
    if (input.hasNextInt())
     System.out.println("int-Wert:\t" + input.nextInt());
    else
     if (input.hasNextDouble())
      System.out.println("double-Wert:\t" + input.nextDouble());
     else
      System.out.println("Text:\t\t" + input.next());
  } catch(java.io.IOException ioe) {
   System.err.println(ioe);
  }
 }
}
```

Mit den Eingabedaten

```
4711    3,1415926
Nicht übel!
13      9,99
```

in einer ANSI-kodierten Textdatei (vgl. Abschnitt 15.4.1.2 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("datenUS.txt"), "Cp1252")) {
 input.useLocale(Locale.US);
  . . .
}
```

Wir haben den Einsatz der Klasse **Scanner** für die Datenerfassung via Konsole bereits in Abschnitt 3.4.1 erwogen. Das folgende Programm nimmt zwei reelle Zahlen *a* und *b* von der Standardeingabe (ein **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("Argumente (durch Leerzeichen getrennt):  ");
  try {
   basis = input.nextDouble();
   exponent = input.nextDouble();
   System.out.println(basis+" hoch "+exponent +
                      " = " + Math.pow(basis, exponent));
  } catch(Exception 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:

```
import util.io.*;

public class Simput {
 static boolean error;
 static 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 = "Eingabe konnte nicht konvertiert werden.";
   System.out.println("Falsche Eingabe!\n");
  }
  return eingabe;
 }
 .  .  .
}
```

15.6 Objektserialisierung

Nach vielen Mühen und lästigen Details im aktuellen Kapitel über Datenströme kommt nun als Belohnung für die Ausdauer eine angenehm einfache Lösung für eine anspruchsvolle und wichtige Aufgabe. Wer objektorientiert programmiert, möchte natürlich auch objektorientiert speichern und laden. Erfreulicherweise können Objekte tatsächlich meist genauso 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 *Objektbaum* (alias: *Objektgraph*) beteiligt. Ein mehrfach referenziertes Objekt wird dabei Ressourcen schonend nur *einmal* einbezogen.

Die zuständigen Klassen gehören zur **OutputStream**- bzw. **InputStream**-Hierarchie und hätten schon früher behandelt werden können. Für ein attraktives und wichtiges Spezialthema ist aber auch die Platzierung am Ende des Datenstromkapitels (sozusagen als Krönung) nicht unangemessen.

Voraussetzungen für die Serialisierbarkeit einer Klasse:

- 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.

Für das Schreiben von Objekten ist die Byte-orientierte Ausgabetransformationsklasse **ObjectOutputStream** zuständig, für das Lesen die Byte-orientierte Eingabetransformationsklasse **ObjectIn-**

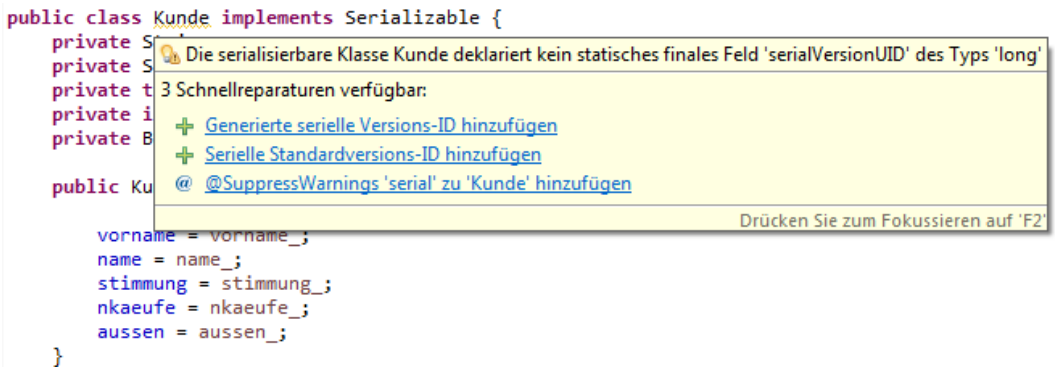
putStream. Ein komplexes Objektensemble in einen Byte-Strom zu verwandeln bzw. von dort zu rekonstruieren, ist eine komplexe Aufgabe, die aber automatisiert abläuft.

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 = -798747263086382088L;
 private String vorname;
 private String name;
 private transient 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 Aufgaben anzuraten (vgl. Abschnitt 3.3.7.2).

Unsere Entwicklungsumgebung Eclipse reklamiert die fehlende **serialVersionUID** zur serialisierbaren Klasse `Kunde`:



Ist eine solche Versionsangabe vorhanden, wird sie beim Serialisieren eines Objekts automatisch mit gespeichert und beim Deserialisieren überprüft, um Fehler durch veraltete, inkompatible Objektversionen zu verhindern. Akzeptiert man die angebotene Schnellreparatur **Generierte serielle Versions-ID hinzufügen**, dann wird der Mangel bequem und professionell behoben:

```
public class Kunde implements Serializable {
 private static final long serialVersionUID = -798747263086382088L;
  . . .
}
```

Die Eclipse-Warnung ist Ihnen vielleicht schon einmal begegnet. Sie erscheint z.B. dann, wenn eine selbst definierte Klasse die Schnittstelle **Serializable** von ihrer Basisklasse übernimmt.

In folgendem Beispielprogramm wird ein Objekt der serialisierbaren Klasse `Kunde` mit der **ObjectOutputStream**-Methode **writeObject()** in eine Datei befördert und anschließend mit der **ObjectInputStream**-Methode **readObject()** von dort zurück geholt. Außerdem wird demonstriert, dass die beiden Klassen 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.bin"))) {
   oos.writeInt(1);
   oos.writeObject(kunde);
  }

  try (ObjectInputStream ois =  new ObjectInputStream(
                                new FileInputStream("test.bin"))) {
   int anzahl = ois.readInt();
   Kunde unbekannt = (Kunde) ois.readObject();
   System.out.printf("Zahl der eingelesenen Fälle: %d\n\n", anzahl);
   unbekannt.prot();
  }
 }
}
```

Beim Schreiben eines Objekts wird auch seine Klasse samt **serialVersionUID** festgehalten. Beim Lesen eines Objekts wird zunächst die zugehörige Klasse festgestellt und nötigenfalls in die virtuelle Maschine geladen. Ist die **serialVersionUID** kompatibel, wird das Objekt auf dem Heap angelegt, und die Instanzvariablen erhalten ihre rekonstruierten Werte.

Als **transient** deklarierte Instanzvariablen werden von **writeObject()** und von **readObject()** übergangen. Damit eignet sich dieser Modifikator z.B. für ...

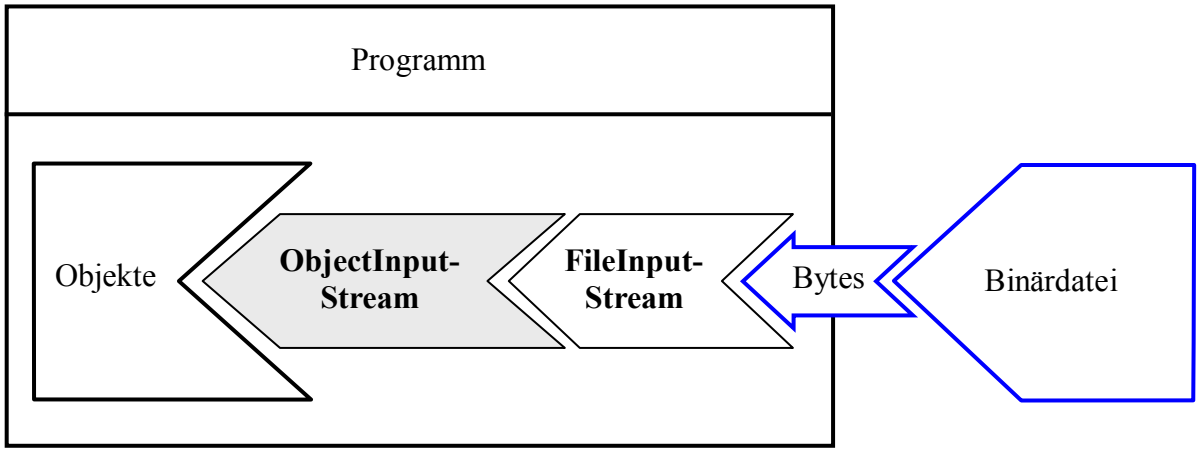
- Variablen, die aus Datenschutzgründen nicht in einer Datei gespeichert werden sollen,
- Variablen, deren temporärer Charakter ein Speichern sinnlos macht.

In der Klasse `Kunde` ist die Instanzvariable `stimmung` als **transient** deklariert, was zu folgender Ausgabe des Beispielprogramms führt:

```
Zu sichern:

Name: Fritz Orth
Stimmung: 1
Anz.Einkaeufe: 13, Aussenstaende: 426.89

Zahl der eingelesenen Fälle: 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.

In der folgenden Abbildung wird die Rekonstruktion eines Objekts skizziert:



15.7 Daten lesen und schreiben über die NIO.2 - Klasse Files

Die Klasse **Files** im Paket **java.nio** aus dem NIO.2 -API beherrscht nicht nur Dateisystemzugriffe (siehe Abschnitt 15.2.1), sondern auch das Lesen und Schreiben von Daten. Außerdem kann sie zu einer per **Path**-Objekt identifizierten Datei ein Datenstromobjekt liefern und noch einige sonstige Aufgaben erledigen. Wird beim Schreiben und Lesen von Zeichen über eine **Files**-Methode oder ein per **Files** erstelltes Datenstromobjekt keine Kodierung angegeben, dann kommt die UTF-8 - Kodierung zum Einsatz.

15.7.1 Öffnungsoptionen

Bei etlichen **Files**-Methoden zum Schreiben in eine Datei sowie zum Erstellen eines Datenstromobjekts zu einer Datei akzeptiert ein Parameter vom Interface-Typ **OpenOption** eine Serie von Werten, um Optionen für das Öffnen der Datei festzulegen. 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:

- **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.
- **WRITE**
  Mit dieser Option wird eine Datei zum Schreiben geöffnet.

- **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.
- **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.
- **READ**
  Mit dieser Option wird eine Datei zum Lesen geöffnet.
- **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);
```

15.7.2 Lesen und Schreiben von kleinen Dateien

In diesem Abschnitt werden statische Ein-/Ausgabemethoden der Klasse **Files** vorgestellt, die sich laut Java-Tutorial (Oracle 2015) im Unterschied zur Verwendung eines **Stream**-Objekts (siehe Abschnitte 15.3 und 15.4) 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:

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:

```
Path imFile = Paths.get("Emma.jpg");
byte[] imBytes = Files.readAllBytes(Paths.get("Emma.jpg"));
```

Mit der Methode **readAllLines()** befördert man *alle* Zeilen einer Textdatei in eine Kollektion vom Typ **List<String>**:

---

[1] Webseite (besucht am 01.02.2015): http://docs.oracle.com/javase/tutorial/essential/io/file.html#common

public static List<String> readAllLines(Path *path***, Charset** *cs***)**
throws IOException

Die gewünschte Zeichenkodierung wird über ein **Charset**-Objekt gewählt (siehe z.B. 15.4.1.2).

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

15.7.3 Datenstrom zu einem Path-Objekt erstellen

Zu einem **Path**-Objekt, das eine Datei bezeichnet, kann man über statische Fabrikmethoden der Klasse **Files** Datenstromobjekte für Byte- bzw. Zeichenströme erstellen (siehe Abschnitt 15.3 bzw. 15.4):

- **public static OutputStream newOutputStream(Path** *path***, OpenOption...** *options***)**
throws IOException
Man erhält einen Byte-orientieren Ausgabestrom, der mit einer Datei verbunden ist (siehe Abschnitt 15.3.1.3).
- **public static InputStream newInputStream(Path** *path***, OpenOption...** *options***)**
throws IOException
Man erhält einen Byte-orientieren Eingabestrom, der mit einer Datei verbunden ist (siehe Abschnitt 15.3.2.3).
- **public static BufferedWriter newBufferedWriter(Path** *path***, Charset** *cs***,**
OpenOption... *options***)**
throws IOException
Man erhält einen gepufferten, zeichenorientieren Ausgabestrom (siehe Abschnitt 15.4.1.6), der einen durch die eben vorgestellte **Files**-Methode per **newOutputStream()** erstellten **OutputStream** für die Verbindung zur Datensenke verwendet.

- **public static BufferedReader newBufferedReader(Path** *path***, Charset** *cs***)**
  throws IOException
  Man erhält einen gepufferten zeichenorientieren Eingabestrom (siehe Abschnitt 15.4.2.4) der einen durch die eben vorgestellte **Files**-Methode per **newInputStream()** erstellten **InputStream** für die Verbindung zur Datenquelle verwendet.

Zu **newBufferedWriter()** und **newBufferedReader()** existieren Überladungen ohne **Charset**-Parameter, wobei die UTF-8 - Kodierung zum Einsatz kommt.

Im Vergleich zu den traditionellen, per Konstruktor erstellten Datenstromklassen (z.B. **FileOutputStream**, **FileInputStream**, **BufferedWriter**, **BufferedReader**) haben die Produkte der **Files**-Fabrikmethoden folgende Vorteile:

- Bei den Dateizugriffen kommt die moderne Channel-Technik zum Einsatz, wobei ein Geschwindigkeitsvorteil möglich, aber nicht garantiert ist.
- Für die Ausgabedateien können Öffnungsoptionen gesetzt werden (vgl. Abschnitt 15.7.1).

15.7.4 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.nio.file.*;

class ProbeContentType {
 public static void main(String[] args) {
  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("%-20s %s\n", path.getFileName(),
                            Files.probeContentType(path));
  } catch (Exception e) {
   System.err.println(e);
  }
 }
}
```

wird der MIME-Type für alle Dateien in einem Verzeichnis aufgelistet:

```
Inhaltstyp der Dateien im Verzeichnis U:\Eigene Dateien\Java\Test:

ausgabe.text        null
Begriffe.pdf        application/pdf
Druck.xlsx          application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
Konstruktoren.doc   application/msword
logo.gif            image/gif
```

Wie man durch Umbenennen einer Datei verifizieren kann, orientiert sich **probeContentType()** unter Windows nicht am Dateiinhalt, sondern an der Namenserweiterung, so dass für die Textdatei **ausgabe.text** *kein* MIME-Typ ermittelt werden kann.

15.7.5 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 der Datei gezählt (vgl. Abschnitt 12.2.5.4.3):

```
Stream<String> sol = Files.lines(Paths.get("U:/Eigene Dateien/Java/test.txt"));
System.out.println("Anzahl der Zeilen: " + sol.count());
```

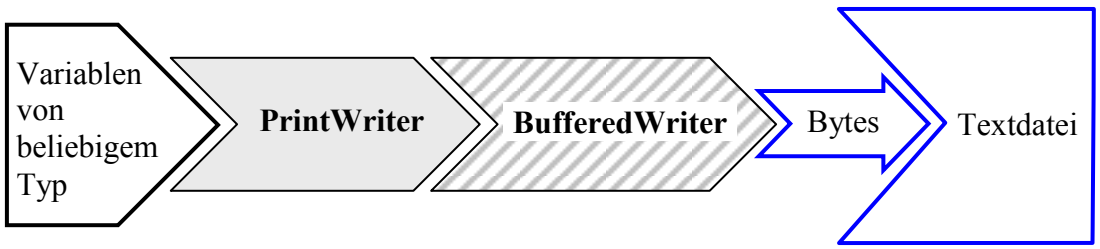
Es ist zu beachten, dass die Klasse **Stream<String>** zu keiner Datenstrom-Hierarchie im Sinne des aktuellen Kapitels 15 gehört, sondern einen Strom im Sinne der mit Java 8 eingeführten Erweiterung um Techniken der funktionalen Programmierung repräsentiert (siehe Kapitel 12).

15.8 Empfehlungen zur Ein- und Ausgabe

Weil die Ein-/Ausgabe - Behandlung in Java durch die Vielzahl beteiligter Klassen etwas unübersichtlich ist, folgt in diesem Abschnitt eine rezeptartige Beschreibung wichtiger Spezialfälle beim Schreiben in Dateien bzw. Lesen aus Dateien.

15.8.1 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 15.4.1.6) in Kombination mit einem **PrintWriter** (siehe Abschnitt 15.4.1.5), 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("U:/Eigene Dateien/Java/io/Ausgabe.txt")))) {
   pw.println(4711);
   pw.printf("%4.2f", Math.PI);
   String ls = System.getProperty("line.separator");
   pw.println(ls + "Nichts ist unmöglich!");
  }
 }
}
```

Über Parameter der Methode **newBufferedWriter()** wählt man:

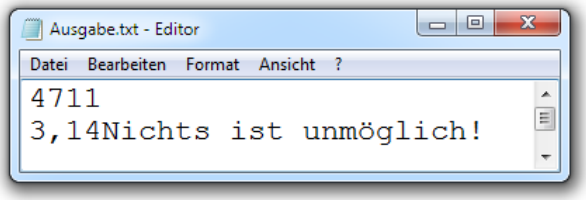
- die Ausgabedatei per **Path**-Objekt (NIO.2 - API)
- das Kodierungsschema über den Parameter **Charset**
  Verzichtet man auf diesen Parameter, wird die Kodierung UTF-8 verwendet.
- Öffnungsoptionen über den Parameter **OpenOption** (siehe Abschnitt 15.7.1)

Die **PrintWriter**-Methode **printf()** (alias **format()**) ermöglicht eine flexible Formatierung der Ausgabe.

Um in eine Textausgabedatei einen Zeilenwechsel einzufügen, sollten Sie *nicht* (wie bei der Konsolenausgabe) die Escape-Sequenz **\n** schreiben, sondern die Plattform-spezifische Zeilenschaltung, die mit dem folgenden Aufruf der statischen **System**-Methode **getProperty()** zu ermitteln ist:

```
System.getProperty("line.separator")
```

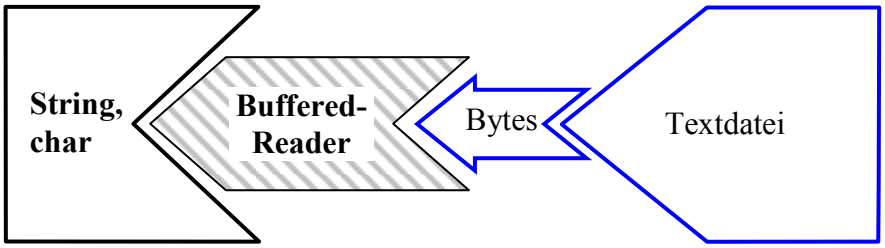
Anderenfalls erkennt z.B. unter Windows der Standardeditor **Notepad** die Zeilenwechsel nicht:



Die **PrintWriter**-Methode **println()** schließt ihre Ausgabe korrekt mit der Plattform-spezifischen Zeilenschaltung ab.

15.8.2 Textzeilen einlesen

Um Texte aus einer Datei zu lesen, eignet sich ein durch die statische Methode **newBufferedReader()** der Klasse **Files** erstellter **BufferedReader** (siehe Abschnitt 15.4.2.4), 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"))) {
   String s;
   while ((s=br.readLine()) != null)
    ls.add(s);
  }
  for(String s : ls)
   System.out.println(s);
 }
}
```

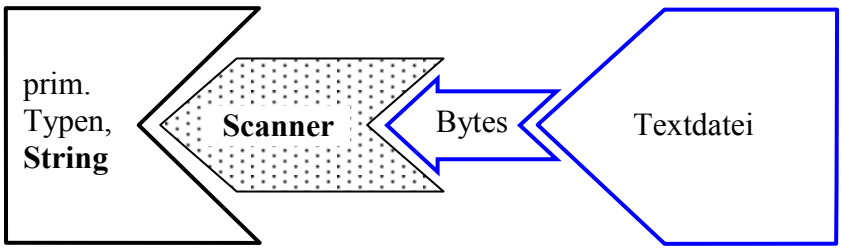
Über Parameter der Methode **newBufferedReader()** wählt man:

- die Eingabedatei per **Path**-Objekt (NIO.2 - API)
- das Kodierungsschema (Voreinstellung: UTF-8)

Zum Lesen von Zeichenfolgen kommt auch die Klasse **Scanner** in Frage (siehe Abschnitte 15.5 und 15.8.3), die den Eingabestrom aufgrund wählbarer Trenneichen in Bestandteile (Tokens) zerlegen kann und ebenfalls die Wahl eines Kodierungsschemas erlaubt.

15.8.3 Zahlen und Zeichenfolgen aus einer Textdatei lesen

Um primitive Datentypen und separierte Zeichenfolgen aus einer Textdatei zu lesen, kann man ein Objekt aus der Klasse **Scanner** im Paket **java.util** verwenden (siehe Abschnitt 15.5):



Beispiel:

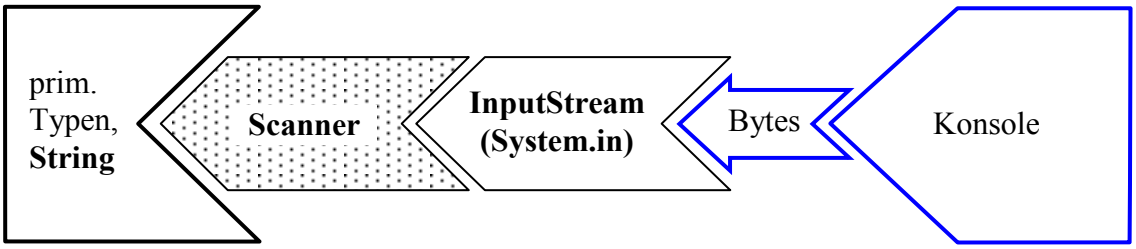
```
import java.nio.file.Paths;
import java.util.*;

class ZahlenScannen {
 public static void main(String[] args) {
  try (Scanner input = new Scanner(Paths.get("Eingabe.txt"))) {
   while (input.hasNext())
    if (input.hasNextInt())
     System.out.println("int-Wert:\t" + input.nextInt());
    else
     if (input.hasNextDouble())
      System.out.println("double-Wert:\t" + input.nextDouble());
     else
      System.out.println("Text:\t\t" + input.next());
  } catch(java.io.IOException ioe) {
   System.err.println(ioe);
  }
 }
}
```

Über Parameter des **Scanner**-Konstruktors wählt man:

- die Eingabedatei per **Path**-Objekt (NIO.2 - API)
- das Kodierungsschema (Voreinstellung: UTF-8)

15.8.4 Eingabe von der Konsole

In Abschnitt 15.5 wird beschrieben, wie man Tastatureingaben mit Hilfe der Klasse **Scanner** entgegen nimmt:

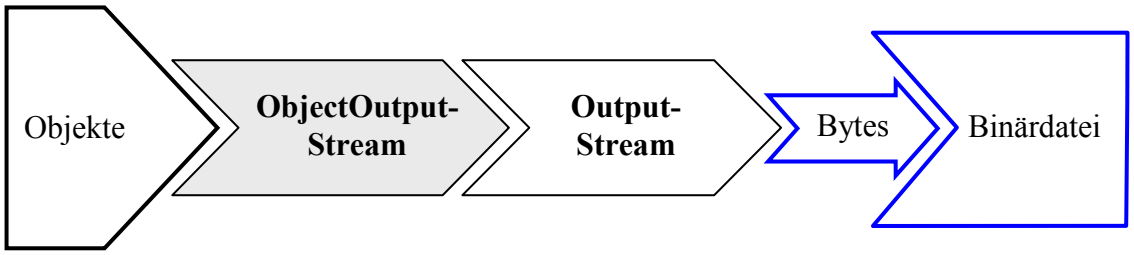


Beispiel:

```
Scanner input = new Scanner(System.in);
System.out.print("Ihr Alter: ");
int alter = input.nextInt();
```

15.8.5 Objekte (de)serialisieren

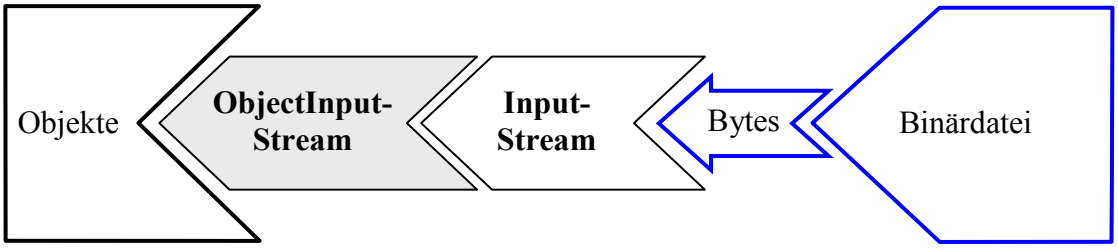
Um Objekte in eine Datei zu schreiben oder aus einer zu Datei lesen, verwendet man die in Abschnitt 15.6 vorgestellten Klassen **ObjectOutputStream** und **ObjectInputStream**. Hier ist die Ausgabe zu sehen:



Beispiel:

```
try (ObjectOutputStream oos = new ObjectOutputStream(
                              new FileOutputStream("test.bin"))) {
   oos.writeObject(kunde);
}
```

Mit der folgenden Datenstromkonstruktion holt man Objekte zurück:



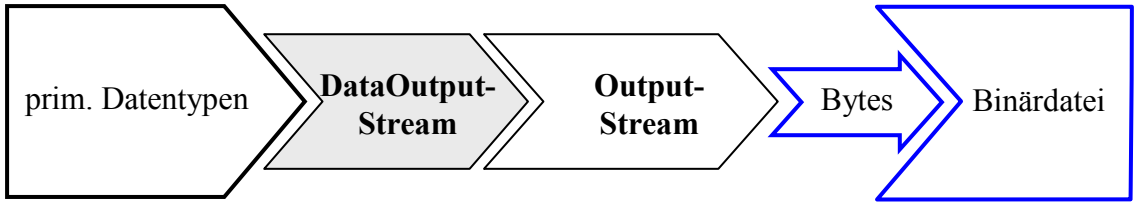
Beispiel:

```
try (ObjectInputStream ois=new ObjectInputStream(new FileInputStream("test.bin"))) {
   Kunde unbekannt = (Kunde) ois.readObject();
}
```

Wer die Verbindung zur Aus- bzw. Eingabedatei über das NIO.2 - API (vgl. Abschnitt 15.2.1) herstellen möchte, lässt sich von der statischen **Files**-Methode **newOutputStream()** einen **OutputStream** bzw. von der Methode **newInputStream()** einen **InputStream** liefern.

15.8.6 Primitive 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 Klasse **FileOutputStream** (siehe Abschnitt 15.3.1.4):

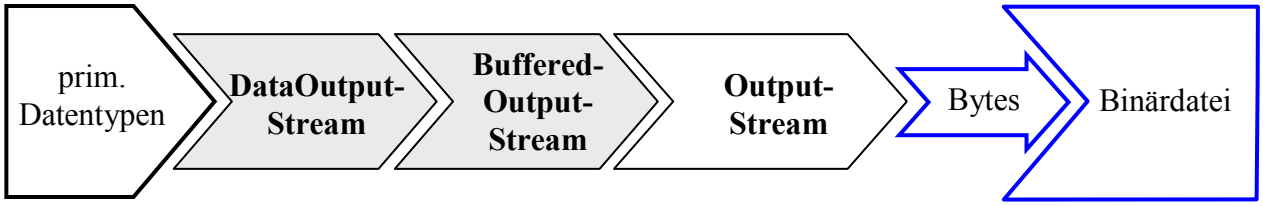


Um die Verbindung zur Ausgabedatei über das NIO.2 - API (vgl. Abschnitt 15.2.1) herzustellen, lässt man sich von der statischen **Files**-Methode **newOutputStream()** einen **OutputStream** liefern.

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:



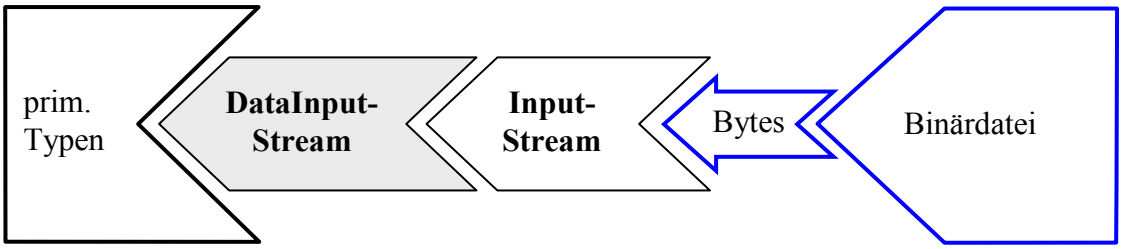
Im folgenden Beispiel wird ein Puffer mit 16384 Bytes Kapazität verwendet:

```
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 (siehe Beispiel in Abschnitt 15.1.2).

15.8.7 Primitive 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 Klasse **FileInputStream**:

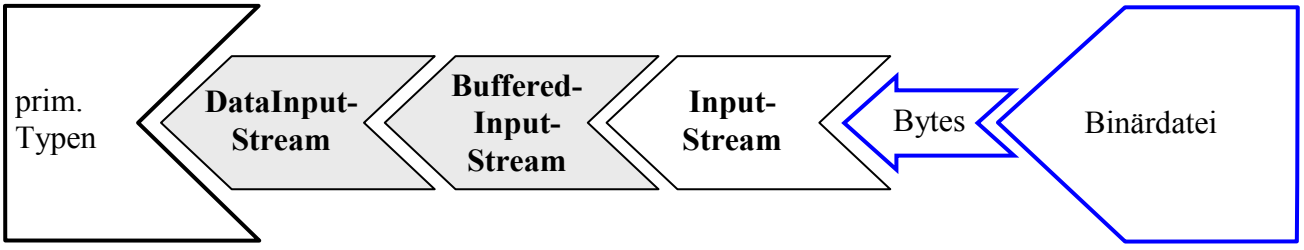


Um die Verbindung zur Eingabedatei über das NIO.2 - API (vgl. Abschnitt 15.2.1) herzustellen, lässt man sich von der statischen **Files**-Methode **newOnputStream()** einen **InputStream** liefern.

Beispiel:

```
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:

```
DataInputStream dis = new DataInputStream(
                        new BufferedInputStream(
                          new FileInputStream("demo.dat"), 16384));
```

15.9 Übungsaufgaben zu Kapitel 15

1) Welche von den 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 verwenden die von **Writer** abstammenden Klassen per Voreinstellung die UTF-8 - Kodierung.
5. Bei der Textausgabe verwendet ein durch die **Files**-Fabrikmethode **newBufferedWriter()** erstelltes Datenstromobjekt per Voreinstellung die UTF-8 - Kodierung.

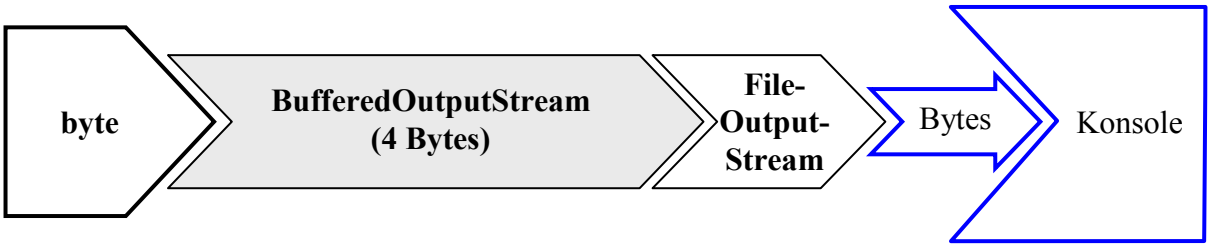
2) Die **FileInputStream**-Methode **read()** versucht, ein Byte aus der angeschlossenen Datei zu lesen. Warum verwendet sie den Rückgabetyp **int**?

3) Erstellen Sie ein Programm zur Demonstration der Ausgabepufferung. Um mitverfolgen zu können, wie bei erschöpfter Pufferkapazität Daten weitergeleitet werden, sollte Sie als Senke die Konsole verwenden.

Wie Sie aus dem Abschnitt 15.3.1.6 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, so dass 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.:

```
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 zusätzlich per Ton die Ankunft eines Bytes im Puffer gemeldet. Wird ein Konsolenprogramm in Eclipse 3.x 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));
  }
 }
}
```

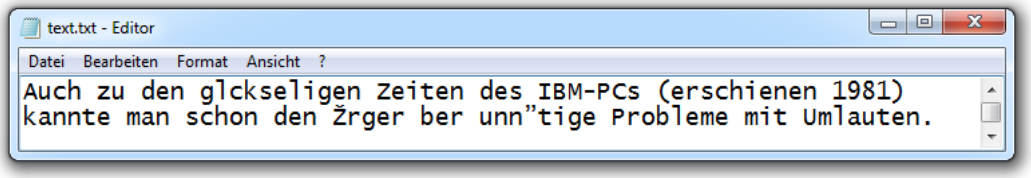
5) Als Byte-orientierte Eingabetransformationsklasse kann **DataInputStream** in Kooperation mit beliebigen **InputStream**-Abkömmlingen die Werte primitiver Datentypen lesen. In Java zeigt die Referenzvariable **System.in** auf ein Objekt aus der Klasse **InputStream** (genauer: auf ein Objekt aus einer Klasse, die von der abstrakten Klasse **InputStream** abstammt). Nun kann man hoffen, mit einem **DataInputStream**-Objekt, das auf **System.in** aufsetzt, Werte primitiver Datentypen von der Tastatur entgegen nehmen zu können. In folgendem Programm wird versucht, beim Benutzer einen **short**-Wert (Ganzzahl mit 2 Bytes) zu erfragen:

Quellcode	Ausgabe
`import java.io.*;` `class DisSysIn {` `  public static void main(String[] args) throws IOException {` `   short zahl;` `   DataInputStream dis = new DataInputStream(System.in);` `   System.out.print("Bitte eine Zahl eingeben: ");` `   zahl = dis.readShort();` `   System.out.println("Gelesen: " + zahl);` `  }` `}`	`Zahl eingeben: 0` `Gelesen: 12301`

Wie ein Blick auf den Beispieldialog zeigt, ist das Verfahren nicht zu gebrauchen.

- Wie ist das Ergebnis zu erklären?
- Welcher **short**-Wert resultiert, wenn der Benutzer eine leere Eingabe mit Enter abschickt?
- Wie sollte man numerische Daten von der Konsole lesen?
- Wozu sollte man die Klasse **DataInputStream** verwenden?

6) Schreiben Sie ein Programm, das den Text



in der Datei

...\BspUeb\IO\MS-DOS\text.txt

einlesen und korrekt in einem **JOptionPane**-Meldungsfenster darstellen kann:

7) 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:

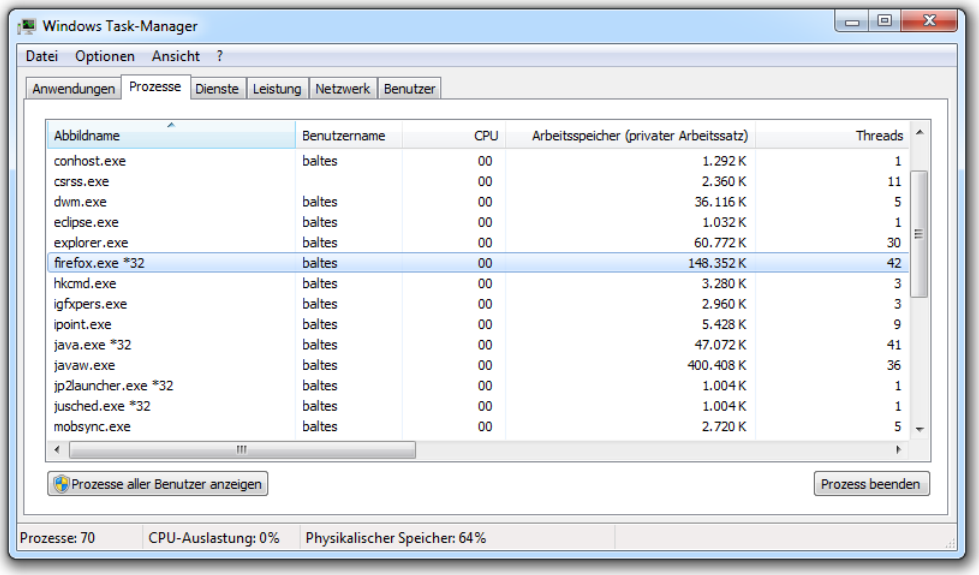
```
nr temp alter gewicht
1 12,3 74,5 123,9
2 11,2 34,4 156,7
3  7,2 83,5 142,1
4 45,2 17,2 129,8
5  1,2 44,4 216,7
6 17,2 23,5 132,1
7 12,2 42,1 182,2
```

In der ersten Zeile stehen die Namen der Merkmale.

16 Multithreading

Wir sind längst daran gewöhnt, dass moderne Betriebssysteme mehrere Programme (Prozesse) parallel betreiben 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. Sofern nur *ein* Prozessor vorhanden ist, der den einzelnen Programmen bzw. Prozessen reihum vom Betriebssystem zur Verfügung gestellt wird, reduziert sich zwar die Ausführungsgeschwindigkeit jedes Programms im Vergleich zum Solobetrieb, doch ist in den meisten Anwendungen 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 in einem anderen Fenster arbeiten. Wie unter Windows ein Blick in die Prozessliste mit dem Task-Manager zeigt, sind z.B. bei einer typischen Verwendung des Internet-Browsers Firefox ca. 40 Threads aktiv, wobei die Anzahl ständig schwankt:



Die Multithreading-Technik kommt aber nicht nur dann in Frage, wenn eine Anwendung mehrere Aufgaben gleichzeitig erledigen soll. Sind auf einem Rechner *mehrere* Prozessoren oder Prozessorkerne verfügbar, dann sollten aufwändige Einzelaufgaben (z.B. das Rendern einer 3D-Ansicht, Virenanalyse einer kompletten Festplatte) in Teilaufgaben zerlegt werden, um *alle* verfügbaren CPU-Kerne einzuspannen und Zeit zu sparen. Mittlerweile (2016) sind 4 Kerne guter Standard und dank Intels Hyper-Threading - Technologie sieht das Betriebssystem bei einer Quad-Core - CPU in der Regel sogar 8 logische Kerne.

Multi-Core - CPUs erhöhen den Druck auf die Software-Entwickler, per Multithreading für gut skalierende Anwendungen zu sorgen, die auf einem Quad-Core - Rechner deutlich schneller laufen als auf einem Single-Core - Rechner. Die Möglichkeit zum (Quasi-)parallelbetrieb mehrerer Programmfunktionen ist aber unabhängig von der Zahl verfügbarer CPU-Kerne in vielen Situationen für die Anwender sehr nützlich.

Beim Multithreading ist allerdings eine sorgfältige Einsatzplanung erforderlich, denn:

- Thread-Wechsel sind mit einem gewissen Zeitaufwand verbunden und sollten daher nicht zu häufig stattfinden.
- In der Regel erfordert das Synchronisieren von Threads einige Aufmerksamkeit beim Programmierer (siehe Abschnitt 16.2). Hier kommt es oft zu Fehlern, die zudem aufgrund variabler Folgen schwer zu analysieren sind.

Während jeder *Prozess* einen eigenen Adressraum besitzt, laufen die *Threads* eines Programms im selben Adressraum ab, so dass sie gelegentlich auch als *leichtgewichtige Prozesse* bezeichnet werden. Sie haben einen gemeinsamen Heap-Speicher, wobei aber jeder Thread als selbständiger Kontrollfluss bzw. Ausführungsfaden einen eigenen Stack-Speicher benötigt.

Vor der Einführung von Java gehörte die Unterstützung von Threads zur HighTech-Programmierung, weil man die Single-Thread-Architektur der damals üblichen Programmiersprachen durch direkte Betriebssystemaufrufe erweitern musste. In Java ist die Multithreading-Unterstützung in die Sprache bzw. das API eingebaut und von jedem Programmierer ohne großen Aufwand zu nutzen. Um diese Technik möglichst gut zu beherrschen, 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.

16.1 Start und Ende eines Threads

Das erste Beispiel soll im klassischen Produzenten-Konsumenten - Paradigma den Start und das Ende eines Threads sowie die Koordination von zwei Threads veranschaulichen, wobei von den potentiellen Vorteilen einer Multi-Thread - 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.

16.1.1 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`:

```
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 offen() {
   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 vom Konsumenten verwendet und im entsprechenden Kontext näher erläutert:

- `offen()`
  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) {
  final Lager lager = new Lager(100);
  final ProThread pt = new ProThread(lager);
  final 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 pl;

 ProThread(Lager pl) {
  super("Produzent");
  this.pl = pl;
 }

 public void run() {
  while (pl.offen()) {
   pl.ergaenze((int) (5 + Math.random()*100));
   try {
    Thread.sleep((int) (1000 + Math.random()*3000));
   } catch(InterruptedException ie) {interrupt();}
  }
 }
}
```

In der Klasse `ProThread` enthält die **run()**-Methode eine **while**-Schleife, die bis zum Eintreten einer Terminierungsbedingung läuft.

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 enger Zusammenarbeit mit dem Wirtsbetriebssystem, wobei ein Thread zwischen den Zuständen **ready** und **running** wechselt (siehe Abschnitt 16.4.1).

Wenn seine **run()** - Methode beendet ist, befindet sich ein Thread im Zustand **terminated** und kann *nicht* erneut gestartet werden. Mögliche Gründe für das Ende der **run()** - Methode:

- Alle Anweisungen wurden ausgeführt.
- Die Methode wurde mit **return** vorzeitig verlassen.
- Es ist eine **RuntimeException** aufgetreten und unbehandelt geblieben.
  Man kann einen Thread mit einem **UncaughtExceptionHandler** ausstatten, damit im Falle einer unbehandelten Ausnahme noch Abschlussarbeiten erledigt werden (siehe Ullenboom 2012a, Abschnitt 14.3.9).

Es ist möglich, aber nicht empfehlenswert, einen Thread von außen mit der (mittlerweile abgewerteten) Methode **stop()** abzuwürgen (siehe Abschnitt 16.3.2).

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:

```
pl.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 nicht für eine präzise Programmablaufsteuerung.

Weil von der Methode **sleep()** potentiell eine **InterruptedException** zu erwarten ist, muss sie in einem **try**-Block ausgeführt werden. Die in Abschnitt 16.3 näher zu beschreibende **Thread**-Methode **interrupt()** wird oft dazu eingesetzt, einen per **sleep()** in den Schlaf oder per **wait()** (siehe Abschnitt 16.2.2) in den Wartezustand geschickten Thread wieder zu aktivieren.

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:

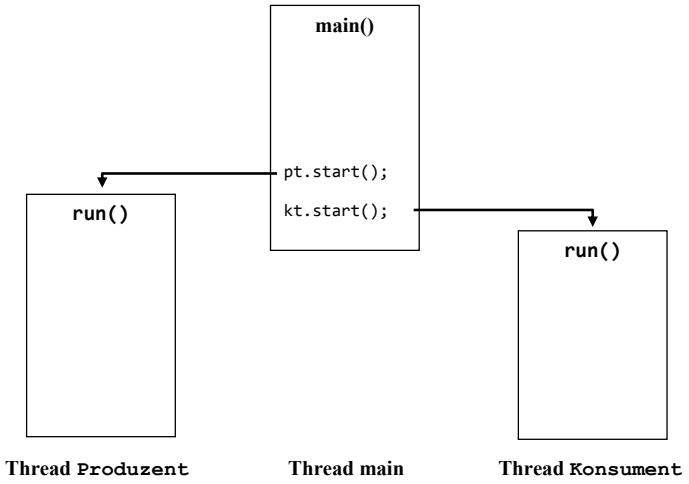
```
class KonThread extends Thread {
 private Lager pl;

 KonThread(Lager pl) {
  super("Konsument");
  this.pl = pl;
 }

 public void run() {
  while (pl.offen()) {
   pl.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 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 `offen()`, 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 Benutzer-Threads sind in einem Java-Programm auch sogenannte *Daemon-Threads* aktiv, die ein Programm *nicht* am Leben erhalten können.

```
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 Multithread - Programmablauf.

In Abschnitt 16.2 werden Techniken zur Koordination bzw. Synchronisation von Threads vorgestellt, mit denen man fehlerhafte Anzeigen und Schlimmeres verhindern kann.

16.1.2 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, was übrigens auch die Klasse **Thread** tut. Dieses Interface verlangt eine Methode ...

- namens **run()**
- ohne Parameter
- mit Rückgabetyp **void**,
- die keine deklarationspflichtige Ausnahmen wirft.

Weil Java bekanntlich *keine* Mehrfachvererbung unterstützt, musste unbedingt die Thread-Fähigkeit unabhängig von der Basisklasse ermöglicht werden.

Wir verwenden weiterhin das Produzenten-Konsumenten - Beispiel aus Abschnitt 16.1.1, ersetzen allerdings die **Thread**-Ableitung `ProThread`

```
class ProThread extends Thread {
    . . .
}
```

durch die Klasse `Produzent`, die das Interface **Runnable** implementiert:

```
class Produzent implements Runnable {
 private Lager pl;

 Produzent(Lager pl) {
  this.pl = pl;
 }

 public void run() {
  while (pl.offen()) {
   pl.ergaenze((int) (5 + Math.random()*100));
   try {
    Thread.sleep((int) (1000 + Math.random()*3000));
   } catch(InterruptedException ie) {Thread.currentThread().interrupt();}
  }
 }
}
```

Solange `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 der aktive Thread, an den der **interrupt()** - Appell zu richten ist, 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 Anzeigen und keine negativen Lagerbestände mehr auftreten.

16.2 Threads koordinieren

In diesem Abschnitt werden Techniken zur Koordination von einzelnen, explizit erstellten Threads vorgestellt. Obwohl in der Java-Weiterentwicklung starke Trends darauf abzielen, durch Nutzung von Frameworks die manuelle Erstellung und Verwaltung von Threads zu reduzieren und gleichzeitig das Potential der Multithreading-Technik besser auszuschöpfen (siehe Abschnitte 16.5 und 16.6), sind die im aktuellen Abschnitt vorgestellten Begriffe und Verfahren 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.

16.2.1 Monitore und synchronisierte Bereiche

Am Anfang des in Abschnitt 16.1.1 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 `offen()` - 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**, 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 eines **long**- oder **double**-Werts (64 Bit groß) unterbrochen wird, und dass schlussendlich die 64 Bits einer Variablen von zwei verschiedenen Threads geschrieben werden (siehe Abschnitt 16.7.2).

Offenbar muss im Beispiel verhindert werden, dass zwei Threads simultan auf das Lager zugreifen. Das ist mit dem von Java unterstützten **Monitor**-Konzept leicht zu realisieren. Zu einem Monitor kann jedes Objekt werden, wenn eine seiner Methoden als **synchronized** deklariert 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 zu den synchronisierten Bereichen des Monitors an sich nimmt. In der englischen Literatur wird der Vorgang als *obtaining the lock* beschrieben. Versucht ein anderer Thread, eine der synchronisierten Methoden desselben Monitors aufzurufen, wird er in den Wartezustand versetzt (**waiting for monitor**, vgl. Abschnitt 16.4.2). 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 beteiligt ist, welches die Klasse in der JVM repräsentiert (siehe Abschnitt 16.2.6).

Weil bei einem Konstruktor der Modifikator **synchronized** *nicht* erlaubt ist, 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 Beispiel sollten die `Lager`-Methoden `offen()`, `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 die Ausflüge in negative Lagerzustände finden nach wie vor statt:

```
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 perfekt geeignet ist, ersetzen wir ihn zu Demonstrationszwecken bei der Methode `ergaenze()`:

```
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**-Methodenmodifikators 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.

In einer per **sleep()** - Methode ausgelösten Ruhephase werden im Besitz eines Threads befindliche Monitore *nicht* zurückgegeben. Folglich ist die **sleep()** - Methode in synchronisierten Bereichen zu vermeiden.

16.2.2 Koordination per wait(), notify() und notifyAll()

Mit Hilfe der **Object**-Methoden **wait()** und **notify()** können in unserem Produzenten-Lager-Konsumenten - 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 16.4.2). 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) {interrupt();}

 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, so dass der Produzenten-Thread freie Bahn hat, den synchronisierten Block zu betreten und das Lager aufzufüllen.

Mit den **Object**-Methoden **notify()** bzw. **notifyAll()** kann ein Thread aus dem Zustand **waiting** in den Zustand **ready** versetzt werden:

- **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-Implementation 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.

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 16.3). Im Beispiel prüft die Methode `liefere()` nach einer Protokollausgabe im **catch**-Block den aktuellen Lagerzustand und begibt sich ggf. erneut in den Wartezustand. In anderen Fällen kann es

sinnvoll sein, den bisherigen Handlungsplan aufzugeben und 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.

16.2.3 Explizite Lock-Objekte

Das Interface **Lock** aus dem Namensraum **java.util.concurrent.locks** und die implementierenden Klassen (z.B. **ReentrantLock**) kommen bei ausgeprägt individuellen Aufgabenstellungen als Alternative zu den synchronisierten Methoden und Blöcken in Frage, die in Abschnitt 16.2.1 behandelt wurden. Man gewinnt an Flexibilität, muss aber auch mehr Verantwortung übernehmen.

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 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* bestehen 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. Dasselbe passiert beim Versuch, über die Methode **lock()** einen expliziten Lock zu erwerben. Allerdings bietet das Interface **Lock** wichtige Alternativen zum endlos blockierenden, nicht unterbrechbaren Lock-Erwerb:

Maximale Wartezeit	Wartezustand unterbrechbar	
	Nein	Ja
Unbegrenzt	**lock()**	**lockInterruptibly()**
Begrenzt		**tryLock(***time***,** *unit***)**

Mit dem Unterbrechen von Threads werden wir uns in Abschnitt 16.3 beschäftigen.

Außerdem sind 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 16.2.4).
- Über die implementierende Klasse **ReentrantReadWriteLock**) kann man *einen* Writer-Thread, aber beliebig viele Reader-Threads zulassen und auf diese Weise unnötige Blockaden vermeiden.

In der folgenden, endlos aktiven **run()** - Methode eines Threads wird ein Warenbestand überwacht, den ein parallel laufender Kunden-Thread laufend reduziert. Sobald ein kritisches Niveau unterschritten wird, erwirbt der Nachfüll-Thread ein **ReentrantLock**-Objekt, welches den simultanen Zugriff auf den Warenbestand durch mehrere Threads verhindert, und ergänzt den Bestand in mehreren simulierten Arbeitsschritten. Hat der Nachfüller sein Ziel erreicht, gibt er das **ReentrantLock**-Objekt wieder frei:

```
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();
 }
}

class Produzent extends Thread {
 Produzent () {
  super("Produzent");
 }

 public void run() {
  while (true) {
   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() +
                         " ergänzt\t" + add + " Stand: " + Lager.bestand);
      try {Thread.sleep(100);} catch(InterruptedException ie) {interrupt();}
     }
   } finally {
    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).

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\BspUeb\Multithreading\Lock

16.2.4 Koordination per await(), signal() und signalAll()

Mit einem **Lock**-Objekt lassen sich über die Methode **newCondition()** beliebig viele **Condition**-Objekte verbinden und deren Methoden **await()**, **signal()** und **signalAll()** erlauben eine Verfeinerung der in Abschnitt 16.2.2 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 ...

- den Lock explizit zurückgeben
- und eine Schlafpause einlegen, wobei deren Dauer schlecht an den Arbeitsbedarf anzupassen wäre.

Sehr 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");
 }

 public void run() {
  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()+
                        " ergänzt\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 {
   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();
```

[1] Im Beispiel sind der Einfachheit halber das **Lock**-Objekt und die **Condition**-Objekte statisch und im gesamten Standardpaket sichtbar definiert, so dass von der wünschenswerten Datenkapselung keine Rede sein kann.

Sobald der Nachfüller seine Arbeit erledigt hat, signalisiert er dies an einen Thread, der auf die `filled`-Bedingung wartet:

```
Lager.filled.signal();
```

Um weiterarbeiten zu können, muss sich dieser Thread um den Lock bewerben. Mit der Methode **signalAll()** spricht man *alle* Threads an, die auf eine Bedingung warten.

Während die **Object**-Methode Methode **notify()** einen Threads 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();
```

Wie die **Object**-Methoden **wait()**, **notify()** und **notifyAll()** dürfen auch 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 mit **await()** und **notify()** interessiert, findet sie z.B. in der API-Klasse **ArrayBlockingQueue<E>**, die in Abschnitt 16.2.7.1 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.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\BspUeb\Multithreading\Condition

16.2.5 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 {
 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 Arbeitsende eines Kollegen, den er per Konstruktor kennen lernt:

```
class Thread2 extends Thread {
 private Thread1 t1;

 Thread2(Thread1 t1) {
  this.t1 = t1;
 }

 public void run() {
  try {
   t1.join();
  } catch (InterruptedException ie) {interrupt();}
  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);
```

16.2.6 Deadlock

Wer sich beim Einsatz von Monitoren oder **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 {
 public void run() {
  Deadlock.lock1.lock();
  System.out.println("Thread 1 besitzt Lock 1.");
  try {Thread.sleep(100);} catch (Exception e) {}
  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 {
 public void run() {
  Deadlock.lock2.lock();
  System.out.println("Thread 2 besitzt Lock 2.");
  try {Thread.sleep(100);} catch (Exception e) {}
  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 1 erwerben.
Thread 2 besitzt Lock 1.
```

16.2.7 Automatisierte Thread-Koordination für Produzenten-Konsumenten - Konstellationen

Die im bisherigen Verlauf von Abschnitt 16.2 beschriebenen Techniken zur individuellen 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**

16.2.7.1 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 benutzt 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, so dass 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, wobei ein Objekt der Klasse **LinkedBlockingQueue<Integer>** als gemeinsam verwendete Warteschlange dient.

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;
 }

 private void wait(int ms) {
  try {
   sleep(ms);
  } catch(InterruptedException ie) {interrupt();}
 }
```

```
  public void run() {
   for (int i = 0; i < produkte.length; i++) {
    System.out.println("\nProduzent liefert gleich.");
    depot.add(produkte[i]);
    wait(10);
    System.out.println("Kurz nach der Lieferung ist der Konsument "
                       + konsument.getState());
    wait(5000);
    System.out.println("Produzent wacht auf, Konsument ist " + konsument.getState());
   }
  }
 }
```

verfährt in seiner **run()** - Methode folgendermaßen:

- 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 behandlungs- bzw. deklarationspflichtigen Ausnahmen zu erwarten sind. Weil der Container 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;
 }

 public void run() {
  for (int i = 0; i < 3; i++) {
   try {
    System.out.println("Konsument hat bezogen: " + depot.take());
   } catch (InterruptedException ie) {interrupt();}
   for (int j = 0; j < 3_000_000; j++)
    Math.random();
  }
 }
}
```

tut in seiner **run()** - Methode Folgendes:

- Er holt per **take()** ein Element aus der Warteschlange und protokolliert sein Verhalten. Weil von **take()** (wie von **wait()**, vgl. Abschnitt 16.2.2) eine **InterruptedException** zu erwarten ist, erfolgt der Aufruf in einer **try**-Anweisung.
- Dann simuliert der Konsument ein geschäftiges Treiben und zieht 3 Millionen Zufallszahlen.

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, ...

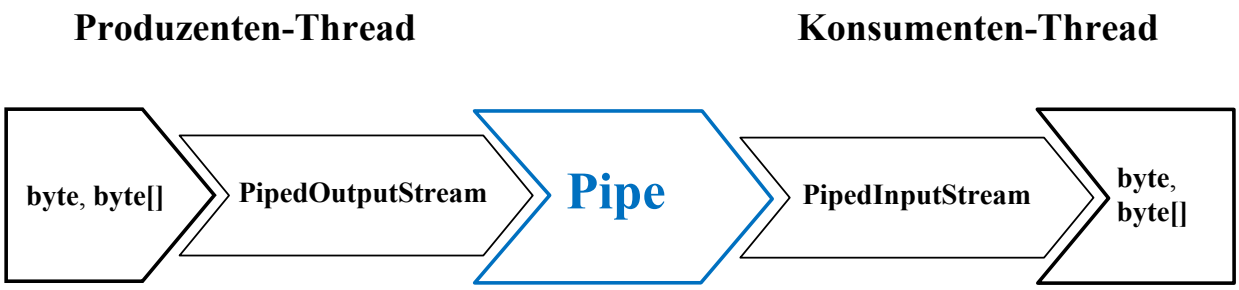
- dass der Konsumenten-Thread im Zustand **RUNNABLE** ist, bis die Methode **take()** auf Datenmangel trifft, was zum Zustand **WAITING** führt,
- dass sich ein Thread nach dem Ende seine **run()** - Methode im Zustand **TERMINATED** befindet.

Die **Thread**-Methode **getState()** meldet den Zustand **RUNNABLE**, wenn ein Thread aus der Sicht der JVM ausgeführt werden kann. Ob er tatsächlich Zugang zu einem Prozessor erhält (Zustand **running**) oder warten muss (Zustand **ready**), hängt auch vom Wirtsbetriebssystem ab.

Auf einem Rechner mit abweichender CPU-Leistung müssen eventuell Einstellungen angepasst werden, um dasselbe Muster beobachten zu können.

16.2.7.2 PipedOutputStream und PipedInputStream

Durch ein Tandem aus einem **PipedOutputStream** (Basisklasse: **OutputStream**, vgl. Abschnitt 15.3.1) und einem verbundenen **PipedInputStream** (Basisklasse: **InputStream**, vgl. Abschnitt 15.3.2) 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, so dass 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 existiert auch eine zeichenorientierte Variante mit Objekten aus den Klassen **PipedWriter** und **PipedReader** am Ein- bzw. Ausgang.

Im Beispielprogramm zur Produzenten-Konsumenten-Kooperation mit Pipe-Lösung ist die Startklasse ähnlich aufgebaut wie bei der **BlockingQueue<E>** - Lösung (vgl. Abschnitt 16.2.7.1):

```
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:[2]

- Es darf nur *ein* Thread in die Pipe schreiben bzw. aus der Pipe lesen.
- Auf gar keinen Fall darf derselbe Thread schreiben und lesen.
- Bevor der schreibende Thread endet, muss er den **PipedOutputStream** schließen, z.B.:

  ```
  pipedOutStream.close();
  ```

  Der lesende Thread kann weiterhin auf die bereits geschriebenen Daten zugreifen.

16.3 Threads unterbrechen, fortsetzen oder beenden

16.3.1 Unterbrechen und Reaktivieren

Zum Unterbrechen und Reaktivieren eines Threads sollten die früher vorgesehenen **Thread**-Methoden **suspend()** und **resume()** *nicht* mehr verwendet werden. Ein **suspend()** - Aufruf kann leicht zu einem Deadlock (deutsch: zu einer *Systemverklemmung*) führen, weil ein suspendierter Thread die in seinem Besitz befindlichen Monitore und Lock-Objekte behält.

---

[1] Das zeigt ein Blick in den Quellcode der Methode **PipedInputStrem.read()** im Java 8 - API:

```
try {
    wait(1000);
} catch (InterruptedException ex) {
    throw new java.io.InterruptedIOException();
}
```

[2] Einige Regeln stammen von der folgenden Webseite von Daniel Ferber (besucht am 09.02.2015):
https://techtavern.wordpress.com/2008/07/16/whats-this-ioexception-write-end-dead/

Stattdessen wird eine Inter-Thread - Kommunikation über die **Object**-Methoden **wait()** und **notify()** empfohlen, die wir schon in Abschnitt 16.2.2 kennen gelernt haben. Dabei wird dem Ziel-Thread z.B. über eine von ihm regelmäßig zu beobachtende Variable signalisiert, dass er sich durch einen **wait()** - Aufruf in den Zustand **waiting** begeben sollte. In seinem Besitz befindliche Monitore und Lock-Objekte werden dabei automatisch zurückgegeben, so dass kein Deadlock droht.[1] Im Produzenten-Lager-Konsumenten - Beispiel spielt der vom Produzenten beeinflusste Lagerbestand die Rolle des vom Konsumenten zu beachtenden Signals.

16.3.2 Beenden

Auf reguläre Weise 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 16.3.1) besteht *keine* Deadlock-Gefahr, weil die vom gestoppten Thread belegten Monitore bzw. Lock-Objekte frei gegeben werden. Aber die abrupt unterbrochene Tätigkeit kann bearbeitete Objekte in einem inkonsistenten Zustand hinterlassen, so dass 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 in Abschnitt 16.3.1 beschrieben Verfahren für das Unterbrechen eines Threads) eine Instanzvariable 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**.
- 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 im Exception Handler wird über das weitere Vorgehen entschieden (siehe unten).

Wir greifen auf das Produzenten-Lager-Konsumenten - Beispiel im ursprünglichen Zustand (siehe Abschnitt 16.1) 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:

---

[1] Das beschriebene Verfahren wird von Oracle auf der folgenden Seite empfohlen (besucht am 26.03.2016): http://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html

```
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 && konsument.isAlive()) {
  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 mit Protest) seine Tätigkeit per **return**:

```
public void run() {
 while (pl.offen()) {
  if (isInterrupted()) {
   System.out.println(
      "Als Kunde muss ich mir so etwas nicht gefallen lassen!\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) unterbrochen zu werden:

```
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
Als Kunde muss ich mir so etwas nicht gefallen lassen!

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.

In der Konsumenten-Methode **run()** befindet sich eine **catch**-Klausel zur Behandlung der **InterruptedException**, in der die **Thread**-Methode **interrupt()** aufgerufen wird. Diese schon in einigen Beispielprogrammen verwendete Konstruktion soll nun endlich erläutert werden. Angewandt auf einen durch die Methoden **sleep()**, **wait()** oder **join()** aus der Klasse **Object** oder durch die Methode **await()** aus der Klasse **Condition**, also durch einen so genannten *blockierenden Me-*

thodenaufruf, in den Wartezustand versetzten Thread hat ein **interrupt()** - Aufruf (wie z.B. in der obigen `Lager`-Methode `ergaenze()`) die folgenden Effekte:

- Der Thread wird sofort in den Zustand **ready** versetzt.
- Die für den Wartezustand verantwortliche Methode endet mit einer **InterruptedException**, und ein bestehendes Interrupt-Signal wird *aufgehoben* (auf **false** gesetzt). Es ist daher oft sinnvoll, **interrupt()** in der **catch**-Klausel der **InterruptedException**-Behandlung erneut aufzurufen, um das Interrupt-Signal wieder auf **true** zu setzen, damit die **run()** - Methode bei nächster Gelegenheit passend reagiert.

Falls beim angesprochenen Thread kein Wartezustand aus den eben genannten Gründen vorliegt, wird das Interrupt-Signal gesetzt. Das kann also 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 (2006b) zu finden.

16.4 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.

16.4.1 Scheduling und Prioritäten

Den Bestandteil der virtuellen Maschine, der die verfügbare Rechenzeit auf die arbeitswilligen und -fähigen 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 Plattform ab, wie viele Prioritätsstufen wirklich unterschieden werden. Der in einer Java-Anwendung automatisch gestartete Thread **main** hat z.B. 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 (z.B. **main**) 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.

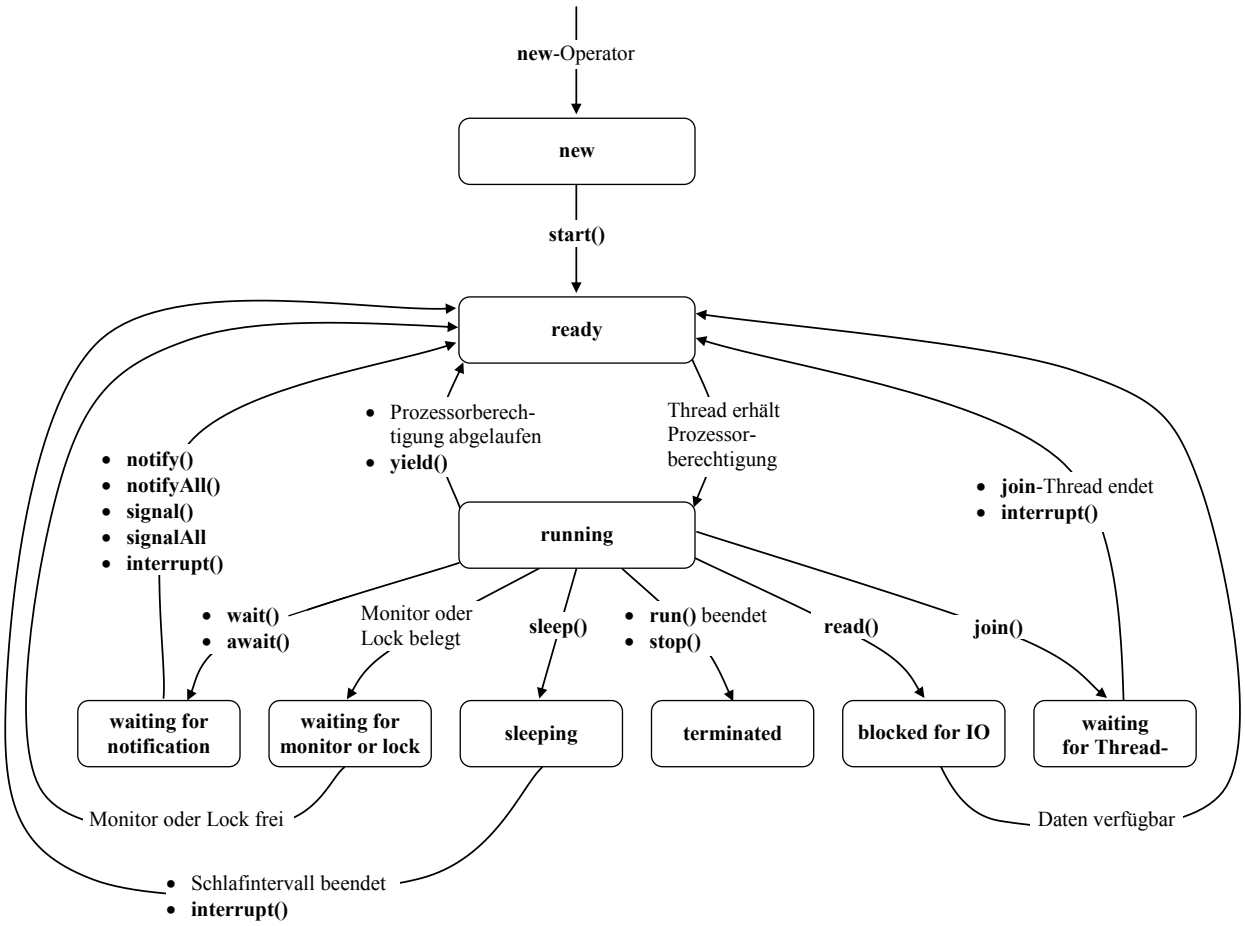
Weil der JVM-Scheduler eng mit dem Wirtsbetriebssystem zusammenarbeiten muss, besteht bei der Verteilung von Rechenzeit auf mehrere Threads mit *gleicher* Priorität *keine vollständige* Plattformunabhängigkeit. Auf einigen Plattformen (z.B. Windows) kommt das **preemtive Zeitscheibenverfahren** zum Einsatz:

- 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.

16.4.2 Zustände von Threads

In der folgenden Abbildung werden für einen explizit kreierten Thread, der nicht von einem Thread-Pool (siehe Abschnitt 16.5) 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.

16.5 Threadpools

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 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.

Die im aktuellen Abschnitt vorgestellten Klassen und Schnittstellen sind nicht etwa für besonders komplexe oder umfangreiche Anwendungsfälle gedacht, sondern für die bequeme und leistungsfähige Multithreading-Routine.

16.5.1 Standardlösung

Ein bequemer und empfohlener Weg zum Threadpool 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(lagerist, i));
```

Findet sich für einen ruhenden Thread binnen 60 Sekunden keine neue Verwendung, wird er terminiert und aus dem Pool entfernt, so dass sich der Ressourcenverbrauch stets am Bedarf orientiert.

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 (siehe Abschnitte 16.1.1 und 16.2.2) wird ein Threadpool verwendet, um eine größere Anzahl von Konsumenten zu bedienen:

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

class ProKonDemo {
 public static void main(String[] args) {
  Lager lagerist = new Lager(1000);
  ProThread pt = new ProThread(lagerist);
  pt.start();

  ExecutorService es = Executors.newCachedThreadPool();
  for (int i = 1; i <= 5; i++) {
   es.execute(new KonThread(lagerist, i));
   try {Thread.sleep(3000);
   } catch (Exception ignored) {}
  }
 }
}
```

Ein Objekt der leicht modifizierten Konsumentenklasse beendet seine Einkaufstour nach zwei Zugriffen:

```
class KonThread extends Thread {
 private Lager pl;
 private int custID;
 final static private int MANZ = 2;
 private int nr;

 KonThread(Lager pl, int custID) {
  super ("Kunde Nr " + custID);
  this.custID = custID;
  this.pl = pl;
 }

 @Override
 public void run() {
  while (nr++ < MANZ) {
   pl.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.
```

Den vollständigen Quellcode finden Sie im Ordner

...\BspUeb\Multithreading\ProKon\ProKon 7 (Threadpool)

16.5.2 Verbesserte Inter-Thread - Kommunikation über das Interface Callable<V>

Um die Kommunikation zwischen Threads 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 16.1.2), 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 11.5) werfen, was der Methode **run()** aufgrund der **Runnable**-Definition verboten ist.

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);
    .  .  .
}
```

Ein solches Objekt verschafft man sich in der Regel mit einer statischen Methode der Klasse **Executors** , die wir im Abschnitt 16.5.1 kennen gelernt 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** implementierende Beispielklasse vorzustellen:

```
class RandomNumberCruncher implements Callable<Double> {
 public Double call() throws TimeoutException {
  final int anz = 100_000_000;
  double d = 0.0;
  long start = System.currentTimeMillis();
  for (int i = 0; i < anz; i++) {
   d += Math.random();
  if (System.currentTimeMillis() - start > 5000)
   throw new TimeoutException();
  }
  return new Double(d/anz);
 }
}
```

Objekte dieser Klasse berechnen in ihrer **call()** - Methode das Mittel von reichlich vielen **double**-Werten 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:

```
class CallableDemo {
 public static void main(String[] args) {
  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()) + ")");
   try {
    Thread.sleep(1000);
   } catch(InterruptedException ie) {Thread.currentThread().interrupt();}
  }
  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 ein von **call()** geworfenes Ausnahmeobjekt weiter. Bei einem gelungenen Aufruf (ohne **TimeoutException**) liefert das Beispielprogramm die Ausgabe:

```
Warten auf call() - Ende (26.03.2016 16:55:09)
Warten auf call() - Ende (26.03.2016 16:55:13)

Mittelwert der Zufallszahlen: 0.5000373898427098
```

Der vom Exekutor 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" prio=6 tid=0x02b2a400 nid=0x91c waiting on condition [0x02e0f000]
   java.lang.Thread.State: WAITING (parking)
```

Im Beispiel wird der überflüssig gewordene Exekutor über seine **shutdown()**-Methode gestoppt:

```
es.shutdown();
```

Beim folgenden Programmablauf hat sich die **call()** - Methode mit dem Werfen einer **TimeoutException** verabschiedet:

```
Warten auf call() - Ende (26.03.2016 16:55:54)
Warten auf call() - Ende (26.03.2016 16:55:57)

Exception beim get() - Aufruf:
  java.util.concurrent.ExecutionException: java.util.concurrent.TimeoutException
```

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\BspUeb\Multithreading\Callable

16.5.3 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. Für eine einzelne periodisch auszuführende Aufgabe genügt also die Klasse **Timer**, die nur einen Thread verwendet.

Sobald jedoch mehrere Aufgaben wiederholt und mit individuellen Zeitplänen auszuführen sind, bietet sich die Klasse **ScheduledThreadPoolExecutor** im Paket **java.util.concurrent** an, die einen Threadpool mit festem Umfang verwendet. Die Pool-Threads können flexibel für unterschiedliche Aufgaben eingesetzt und damit effizient genutzt werden.

Im folgenden Beispiel wird ein **ScheduledThreadPoolExecutor**-Objekt, das maximal zwei Threads einsetzen darf, über die statische **Executors**-Methode **newScheduledThreadPool()** erzeugt:

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

class ScheduledThreadPoolExecutorDemo {
 public static void main(String[] args) {
  ScheduledThreadPoolExecutor es =
     (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(2);
  es.scheduleAtFixedRate(new ScheduledRunner(1,es),
                         0,1000,TimeUnit.MILLISECONDS);
  es.scheduleAtFixedRate(new ScheduledRunner(2,es),
                         2000,2000,TimeUnit.MILLISECONDS);
  es.scheduleAtFixedRate(new ScheduledRunner(3,es),
                         3000,3000,TimeUnit.MILLISECONDS);
  try {Thread.sleep(5000);
  } catch (InterruptedException ie) {Thread.currentThread().interrupt();}
  es.shutdown();
 }
}
```

Statt den für **newScheduledThreadPool()** deklarierten Interface-Rückgabetyp **ScheduledExecutorService** zu verwenden, wird im Beispiel eine Wandlung in die (per **getClass()** ermittelte) Klasse des gelieferten Objekts (**ScheduledThreadPoolExecutor**) vorgenommen. So werden zusätzliche Methoden verfügbar, die für das zeitplangesteuerte Multithreading nicht unbedingt benötigt werden, aber interessante Einblicke in die Arbeitsweise eines **SheduledThreadPool**s ermöglichen, z.B. **getActiveCount()** mit der Anzahl der aktiven Threads als Rückgabe.

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:

```
class ScheduledRunner implements Runnable {
 private int nr;
 private ScheduledThreadPoolExecutor es;

 public ScheduledRunner(int nr, ScheduledThreadPoolExecutor es) {
  this.nr = nr;
  this.es = es;
 }

 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: " + es.getActiveCount());
  try {Thread.sleep(500*nr);
  } catch (InterruptedException ie) {Thread.currentThread().interrupt();}
 }
}
```

Wie das folgende Ablaufprotokoll zeigt, variiert die Anzahl der aktiven Threads, wobei das Maximum 2 nicht überschritten wird:

```
ScheduledRunner 1, Zeit: 15:40:24, Aktive Pool-Threads: 1
ScheduledRunner 1, Zeit: 15:40:25, Aktive Pool-Threads: 1
ScheduledRunner 1, Zeit: 15:40:26, Aktive Pool-Threads: 2
ScheduledRunner 2, Zeit: 15:40:26, Aktive Pool-Threads: 2
ScheduledRunner 1, Zeit: 15:40:27, Aktive Pool-Threads: 2
ScheduledRunner 3, Zeit: 15:40:27, Aktive Pool-Threads: 2
ScheduledRunner 1, Zeit: 15:40:28, Aktive Pool-Threads: 2
ScheduledRunner 2, Zeit: 15:40:29, Aktive Pool-Threads: 2
ScheduledRunner 1, Zeit: 15:40:29, Aktive Pool-Threads: 2
```

Die Methode **main()** fordert den Exekutor nach fünf Sekunden per **shutdown()** - Methode auf, seine Tätigkeit einzustellen. Daraufhin werden keine neuen Ausführungen mehr begonnen, so dass die Pool-Threads nach einiger Zeit enden. Laufende Ausführungen werden auch nach dem Ende des Exekutors 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.

Die im Paket **javax.swing** enthaltene Klasse **Timer** unterscheidet sich von den oben vorgestellten Lösungen in folgenden Punkten:

- Einfache Integration in Swing-Anwendungen
- Ereignisgesteuerte Ausführung im Event Dispatch Thread (EDT)
  Weil die einmalig oder regelmäßig zu einer vorbestimmten Zeit auszuführenden Aufgaben im Event Dispatch Thread (EDT, vgl. Abschnitt 16.9.1) ablaufen, sind nur Aufgaben mit sehr geringem Zeitaufwand sinnvoll (z.B. Aktualisieren einer Zeitanzeige), weil ansonsten die Bedienoberfläche zäh reagiert.

16.6 Moderne Multithreading-Frameworks in Java

16.6.1 Fork-Join

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, so dass mehrere Threads *ohne nennenswerten Koordinierungsbedarf* jeweils eine Teilaufgabe erledigen können. Trotz der recht 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 durch den folgenden Pseudo-Code beschrieben wird:

```
Wenn (Umfang der Aufgabe unterhalb einer Schwelle)
 Erledige die Aufgabe mit einer sequentiellen Technik.
Sonst
 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 folgende 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 Teilaufgaben *kein* Ergebnis melden (z.B. beim Zerlegen einer Bitmap-Filterung auf einzelne Kacheln), 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 auf die Fertigstellung einer Teilaufgabe warten muss, 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 in 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. 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).
- 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 16.2.5).
- 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.*;

public class FacForkJoin {
   static private ForkJoinPool pool = new ForkJoinPool();

   private FacForkJoin() {}

   static public BigDecimal factorial(int argument, int schwelle) {
      FacTask task = new FacTask(1, argument, schwelle);
      pool.invoke(task);
      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 Klasse `FacForkJoin` ist ein statisches Mitgliedobjekt der Klasse **ForkJoinPool** deklariert, wobei durch die Wahl des parameterfreien Konstruktors die Anzahl der Pool-Threads mit der Anzahl der verfügbaren CPU-Kerne gleich gesetzt wird. Es wird nur *ein* **ForkJoinPool**-Objekt benötigt, das beim Laden der Klasse entsteht. In der statischen `FacForkJoin`-Methode `factorial()` wird ...

- ein Objekt der Aufgabenklasse `FacTask` mit der kompletten Aufgabe erstellt,
- die Bearbeitung durch einen Aufruf der **ForkJoinPool**-Methode **invoke()** gestartet, die als Parameter das `FacTask`-Objekt erhält,
- durch einen Aufruf der **ForkJoinTask<BigDecimal>** - Methode **join()** das Ergebnis ermittelt und an den Aufrufer gemeldet.

Damit ist die Fork-Join - Technik zur Bestimmung der Fakultät leicht anzuwenden, z.B.:

```
import java.math.BigDecimal;

class FJuPS {
 public static void main(String[] args) {
  int argument = 50_000;
  long zeit;
  BigDecimal ergebnis;

   zeit = System.currentTimeMillis();
   ergebnis = FacForkJoin.factorial(argument);
   System.out.printf("Ergebnis: %d, Laufzeit: %e",
      ergebnis, (System.currentTimeMillis()-zeit));
 }
}
```

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: 249   (Ergebnis: 3,347321e+213236)
Laufzeit mit Fork-Join in Millisekunden: 174   (Ergebnis: 3,347321e+213236)
Laufzeit mit Fork-Join in Millisekunden: 161   (Ergebnis: 3,347321e+213236)
Laufzeit mit Fork-Join in Millisekunden: 151   (Ergebnis: 3,347321e+213236)
Laufzeit mit Fork-Join in Millisekunden: 74    (Ergebnis: 3,347321e+213236)

Laufzeit Single-Thread in Millisekunden: 1243  (Ergebnis: 3,347321e+213236)
Laufzeit Single-Thread in Millisekunden: 1205  (Ergebnis: 3,347321e+213236)
Laufzeit Single-Thread in Millisekunden: 1251  (Ergebnis: 3,347321e+213236)
Laufzeit Single-Thread in Millisekunden: 1146  (Ergebnis: 3,347321e+213236)
Laufzeit Single-Thread in Millisekunden: 1158  (Ergebnis: 3,347321e+213236)
```

Die Überlegenheit der Multithreading-Lösung ist erheblich größer, als es bei 2 realen CPU-Kernen plus Hyperthreading zu erwarten war.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\BspUeb\Multithreading\Fork Join und Parallel Stream

16.6.2 Parallelverarbeitung bei Java 8 - Strömen

Im Abschnitt 12.2.5.4.2 wurde ein Verfahren zur Fakultätsberechnung unter Verwendung der in Java 8 eingeführten Ströme mit seriellen bzw. parallelen Aggregatoperationen vorgestellt. Aktuell interessieren wir uns vor allem für das automatisierte, auf der Fork-Join-Technik basierende Multithreading bei parallelen Strömen. Das folgende Programm

```
import java.math.BigDecimal;
import java.util.stream.IntStream;

class Prog {
 public static void main(String[] args) {
  int argument = 50_000;
  long zeit;

  System.out.println("\n");
  for (int i = 0; i < 15; i++) {
   zeit = System.currentTimeMillis();
   BigDecimal ergebnis = IntStream
      .rangeClosed(1, argument)
      .parallel()
      .mapToObj(BigDecimal::new)
      .reduce(BigDecimal.ONE, (c,x) -> c.multiply(x));
   System.out.printf(
    "\nLaufzeit mit parallelem Strom in Millisekunden:\t%d \t(Ergebnis: %e)",
    System.currentTimeMillis()-zeit, ergebnis);
  }
 }
}
```

zeigt im Vergleich zu der im letzten Abschnitt vorgestellten expliziten Fork-Join-Lösung vor allem eine enorme Vereinfachung.

Für die schrumpfenden Laufzeiten, die auch schon in Abschnitt 16.6.1 bei einer expliziten Fork-Join - Lösung zu beobachten waren, sorgen wohl dynamische Optimierungsmaßnahmen in der JVM und im Betriebssystem:

```
Laufzeit mit parallelem Strom in Millisekunden: 273  (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 158  (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 254  (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 168  (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 161  (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 136  (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 112  (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 111  (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 99   (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 76   (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 68   (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 71   (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 79   (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 70   (Ergebnis: 3,347321e+213236)
Laufzeit mit parallelem Strom in Millisekunden: 66   (Ergebnis: 3,347321e+213236)
```

Zwischen der expliziten Fork-Join-Lösung und der Parallelstrom-Lösung zeigen sich im Beispiel keine relevanten Laufzeitunterschiede. Damit sollte nach Möglichkeit die Strom-Technik aus Java 8 genutzt werden, um mit minimalem Aufwand effektive Multithreading-Lösungen zu entwickeln.

16.7 Sonstige Thread-Themen

16.7.1 Daemon-Threads

Neben den bisher behandelten Benutzer-Threads kennt Java noch so genannte *Daemon-Threads*, die zur Unterstützung anderer Threads tätig sind und dabei nur aktiv werden, wenn ungenutzte Rechenzeit vorhanden ist. Auch der mittlerweile sicher wohlbekannte Garbage Collector arbeitet im Rahmen eines Daemon-Threads.

Um das Terminieren von Daemon-Threads braucht man sich in der Regel nicht zu kümmern, denn ein Java-Programm endet, sobald ausschließlich Daemon-Threads vorhanden sind.

Mit der Thread-Methode **setDeamon()** lässt sich auch 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.

16.7.2 Der 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 (Ullenboom 2012b, Abschnitt 2.7). Dies wird durch den Modifikator **volatile** verhindert, z.B.:

```
private volatile long counter;
```

Außerdem verhindert die **volatile**-Deklaration eventuelle Speicherzugriffsoptimierungen der Laufzeitumgebung durch Zwischenspeicherung (Cache-Strategien). Somit ist sichergestellt, dass jede Wertänderung sofort allen Threads zur Verfügung steht. Dieser Effekt des Modifikators ist potentiell bei Feldern von beliebigem Typ von Bedeutung.

Weitere Details zum Modifikator **volatile** und seiner Rolle bei der Thread-Synchronisation finden sich bei Kreft & Langer (2008).

16.7.3 Thread-Gruppen

Bezüglich der 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.

16.8 Threads und JavaFX

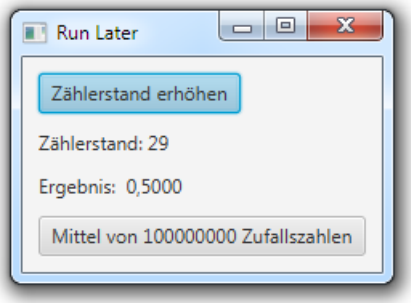
Wie die meisten modernen GUI-Frameworks ist auch JavaFX aus Performanzgründen nach dem Single-Thread-Prinzip konzipiert. Daher muss der Zugriff auf die GUI-Komponenten von JavaFX dem **JavaFX Application Thread** (alias: *UI-Thread*) vorbehalten bleiben.

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 Hintergrund-Thread. In der Regel müssen aber irgendwann Ergebnisse der Hintergrundtätigkeit an der Oberfläche sichtbar werden, wobei wegen der eingangs genannten Regel aus dem Hintergrund-Thread keine direkten Zugriffe auf Steuerelemente möglich sind.

16.8.1 JavaFX-Komponenten aus Hintergrund-Threads modifizieren

Wir konstruieren eine Variante des Mehrzweckzählprogramms aus Abschnitt 13.5.1, um zu demonstrieren, dass die Berechnung des Mittelwerts aus 100 Millionen pseudozufälligen **double**-Zahlen in einem Hintergrund-Thread die Bedienbarkeit eines Programms nicht beeinträchtigt:



Um die Mittelwertberechnung aus dem UI-Thread heraus zu halten, definieren wir eine Klasse, die das Interface **Runnable** erfüllt und die Rechnung in ihrer **run()** - Methode erledigt:

---

[1] Diese Empfehlung stammt von der Webseite (besucht am 11.02.2015):
http://www.oracle.com/technetwork/articles/javase/swingworker-137249.html

```
private 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:

```
final 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 16.7.1) stattfindet, der beim Schließen des Programmfensters automatisch beendet wird. Ohne diese Maßnahme würde ein aktiver Hintergrund-Thread seine Arbeit auch nach dem Verschwinden des Programmfensters fortsetzen.

Nach Vollendung seiner Arbeiten möchte der Hintergrund-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 Hintergrund-Thread führt allerdings zu einem Laufzeitfehler.

Mit Hilfe der statischen Methode **runLater()** aus der Klasse **Platform** kann der Hintergrund-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.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\BspUeb\Multithreading\JavaFX\RunLater

16.8.2 Das JavaFX-Multithreading-API

Mit den Typen im Paket **javafx.concurrent** kann man Aufgaben in Hintergrund-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 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.

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.

Alle in der Schnittstelle **Worker<V>** vorgeschriebenen JavaFX-Properties sind vom ReadOnly-Typ (vgl. Abschnitt 13.4.1.2).

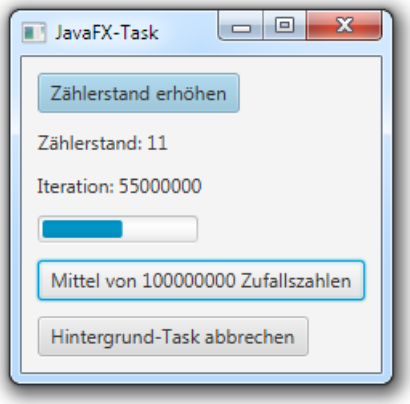
Name	Klasse	Beschreibung
title	**ReadOnlyStringProperty**	Beschreibt die Aufgabe
state	**ReadOnlyObjectProperty <Worker.State>**	Siehe obige Auflistung
running	**ReadOnlyBooleanProperty**	Diese Eigenschaft hat genau dann den Wert **true**, wann der Status **SCHEDULED** oder **RUNNING** ist.
message	**ReadOnlyStringProperty**	Damit kann eine Aufgabe darüber informieren, was sie gerade tut
totalWork	**ReadOnlyDoubleProperty**	Werte von 0 bis **Long.MAX_VALUE** stehen für das gesamte Arbeitsvolumen, -1 steht für einen undefinierten Wert.

Name	Klasse	Beschreibung
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 **FALIED**), dann ist das Ausnahmeobjekt über die Eigenschaft **exception** abrufbar.

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).

16.8.3 Die Klasse Task<V>

Das empfehlenswerte Verfahren zur Realisation einer 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 16.8.1, 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 in Abschnitt 16.8.1 vorgestellten Technik:



U.a. wird es möglich, ...

- die in Abschnitt 16.8.2 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>**:

```
private 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 new Double(d/anz);
   }

   @Override
   protected void succeeded() {
      super.succeeded();
      this.updateMessage("Ergebnis: " + this.getValue());
   }
}
```

In **call()** werden zwei **Task** - Methoden benutzt, um JavaFX-Properties des Hintergrundauftrags (siehe Liste in Abschnitt 16.8.2) 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 13.4.2) mit der **text**-Property eines Labels bzw. mit der **progress**-Property eines **ProgressBar**-Steuerelements verbunden und so sichtbar gemacht:

```
final Label lblMessage = new Label();
final ProgressBar progBar = new ProgressBar();
final 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 diese sinnvollen Bindungen korrekt arbeiten, dürfen die Task-Properties nur im UI-Thread geändert werden. Es ist generell zu unterlassen, aus einem Hintergrund-Thread Schreibzugriffe auf Objekte vorzunehmen, an die GUI-Eigenschaften gebunden sind, weil damit gegen die zu Beginn von Abschnitt 16.8 formulierte Single-Thread-Regel verstoßen wird.

Das letzte Quellcodesegment zeigt, wie im Beispielprogramm ein Objekt der **Task<Double>** - Ableitung `RandomNumberCruncher` ins Spiel kommt: Bei jedem Aufruf der Klickbehandlungsme-

thode zum oberen Befehlsschalter wird ein frisches Objekt erzeugt, das nach Erledigung seiner Arbeit nicht mehr zu gebrauchen ist.

Der Einfachheit halber wird auf den sinnvollen 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:

```
final 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 16.3), 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 Annullierung 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 JDK-Dokumentation.[1]

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\BspUeb\Multithreading\JavaFX\Task

16.9 Threads und Swing

Swing ist zwar seit Java SE 8 nicht mehr die Standardtechnik zur Entwicklung von grafischen Bedienoberflächen, muss aber aufgrund der zahllos vorhandenen Programme mit Swing-Bedienoberfläche als weiterhin relevante Technik eingestuft werden. Daher müssen wir uns auch um die Probleme und Aufgaben im Zusammenhang mit dem Multithreading kümmern.

---

[1] https://docs.oracle.com/javase/8/javafx/api/javafx/concurrent/Task.html

16.9.1 Ereignisverteilungs-Thread und Single-Thread - Regel

Greifen mehrere Threads simultan auf eine Swing-Komponente zu, kann es zu irregulärem Verhalten (z.B. bei der Bildschirmaktualisierung) kommen, weil die meisten Swing-Methoden *nicht* Thread-sicher sind. Man hat (wie bei JavaFX) auf die Thread-Sicherheit verzichtet, weil Performanzprobleme bis hin zur blockierten Bedienoberfläche auftreten könnten.

Daher muss (bis auf einige Ausnahmen, siehe unten) der Zugriff auf die Swing-Komponenten dem bei Swing-Anwendungen mit dem Erscheinen des ersten Fensters vorhandenen **Ereignisverteilungs-Thread** (engl.: *Event Dispatch - Thread*, kurz: *EDT*) vorbehalten bleiben. In diesem Thread laufen ab:

- **paint()** - Aufrufe zur Fensterrenovierung
- Aufrufe von Ereignisbehandlungsmethoden
- vom Programm per **invokeLater()** zur asynchronen Ausführung im EDT veranlasste Methodenaufrufe

Bei allen im EDT ausgeführten Methodenaufrufen muss sich der Zeitaufwand in Grenzen halten (maximal 100 Millisekunden[1]), weil sonst die Bedienoberfläche zäh reagiert. Aufwändige Arbeiten gehören in einen Hintergrund-Thread, der in der Regel irgendwann Ergebnisse seiner Tätigkeit an der Oberfläche sichtbar machen möchte (siehe Abschnitte 16.9.3 und 16.9.4).

Bei Verwendung des Swing-Toolkits ist also unbedingt die folgende **Single-Thread - Regel** zu beachten (Muller & Walrath 2000):

> Methoden, die eine bereits *realisierte* Swing-Komponente modifizieren oder von ihrem Zustand abhängen, dürfen nur im EDT ausgeführt werden. Als *realisiert* gilt eine Swing-Komponente dann, wenn für das zugehörige Top-Level-Fenster eine der Methoden **setVisible(true)**, **show()** oder **pack()** aufgerufen worden ist.

16.9.2 Thread-sichere Swing-Initialisierung

Wenn man es ganz genau nimmt, müssen schon die realisierenden Methoden **setVisible(true)**, **show()** und **pack()** im EDT ausgeführt werden, denn:

- Beim Realisieren nimmt der EDT seinen Betrieb auf, so dass Ereignisbehandlungsmethoden mit Wirkung auf Swing-Komponenten aufgerufen werden können.
- Die realisierenden Methoden sind aber zu diesem Zweitpunkt noch nicht beendet. Absolvieren sie ihre Restlaufzeit *nicht* im EDT (sondern im Thread **main**), kann es zu Multithreading-Problemen kommen.

Im Kapitel 14 haben wir eine Swing-Initialisierung wie im folgenden Beispiel benutzt:

---

[1] Diese Empfehlung stammt von der Webseite (besucht am 11.02.2015):
http://www.oracle.com/technetwork/articles/javase/swingworker-137249.html

```
import javax.swing.*;
import java.awt.*;

class SwingInitNotThreadSafe {
 private JFrame frame;
 private static final String LAB = "Swing-Initialisierung nicht Thread-sicher";
 SwingInitNotThreadSafe() {
  frame = new JFrame(LAB);
  JLabel lblText = new JLabel(LAB);
  lblText.setHorizontalAlignment(SwingConstants.CENTER);
  frame.getContentPane().add(lblText, BorderLayout.CENTER);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(360, 100);
  frame.setVisible(true);
 }

 public static void main(String args[]) {
  new SwingInitNotThreadSafe();
 }
}
```

Streng genommen muss aber durch ein **Runnable**-Objekt dafür gesorgt werden, dass die Methode **setVisible(true)** im EDT abläuft, z.B.:

```
import javax.swing.*;
import java.awt.*;

class SwingIniThreadSafe {
 private JFrame frame;
 private static final String LAB = "Swing-Initialisierung Thread-sicher";
 SwingIniThreadSafe() {
  frame = new JFrame(LAB);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JLabel lblText = new JLabel(LAB);
  lblText.setHorizontalAlignment(SwingConstants.CENTER);
  frame.getContentPane().add(lblText, BorderLayout.CENTER);
  frame.setSize(360, 100);
  frame.setVisible(true);
 }

 public static void main(String args[]) {
  EventQueue.invokeLater(new Runnable() {
   public void run() {
    new SwingIniThreadSafe();
   }
  });
 }
}
```

Über die statische Methode **invokeLater()** der Klasse **EventQueue** mit einem Parameter vom Typ **Runnable** wird dafür gesorgt, dass die **run()** - Methode der anonymen Klasse und damit die Fensterkonstruktion baldmöglichst im EDT abläuft.[1]

[1] Alternativ kann auch die äquivalente statische Methode **invokeLater()** der Klasse **SwingUtilities** verwendet werden.

Erwartungsgemäß vorbildlich verhält sich der im Eclipse-Plugin *WindowBuilder* enthaltene Quellcode-Generator beim Starten einer Swing-Anwendung, z.B.:

```
public static void main(String[] args) {
 EventQueue.invokeLater(new Runnable() {
  public void run() {
   try {
    final BK window = new BK();
    window.frmKrzenVonBrchen.setVisible(true);
   } catch (final Exception e) {
    e.printStackTrace();
   }
  }
 });
}
```

Mit den in Kapitel 12 vorgestellten Neuerungen von Java 8 lässt sich die Thread-sichere Swing-Initialisierung über eine Konstruktor-Referenz (vgl. Abschnitt 12.1.2.2) sehr kompakt vornehmen:

```
public static void main(String args[]) {
 EventQueue.invokeLater(SwingIniThreadSafe::new);
}
```

Nach Muller & Walrath (2000) ist die zu Beginn des aktuellen Abschnitts skizzierte Konfliktkonstellation sehr unwahrscheinlich, und die Thread-Sicherheit beim Starten von Swing-Programmen wird in vielen Lehrbüchern ebenso nachlässig gehandhabt wie im Kapitel 14 dieses Manuskripts (z.B. in Deitel & Deitel 2005, Krüger & Hansen 2014, Mössenböck 2005, Ullenboom 2012a).

Wir haben nun genügend Wissen zur Verfügung, um die bei Swing-Anwendungen systemseitig im Einsatz befindlichen Threads (z.B. **main**, EDT) zu verstehen, und um aufwändige Algorithmen in eigene Benutzer- bzw. Hintergrund-Threads auszulagern.

16.9.3 Swing-Komponenten aus Hintergrund-Threads modifizieren

Im Swing-API können Änderungen von Komponenten auf folgende Weise risikofrei aus beliebigen Threads veranlasst werden:

- **Thread-sichere Methoden**
  Als Ausnahme von der generellen Regel sind einige Methoden im Swing-API Thread-sicher:
  - Mit der **repaint()** - Methode der Klasse **java.awt.Component** wird eine Komponente gebeten, sich baldmöglichst neu zu zeichnen.
  - Mit den Methoden **add*EventClass*Listener()** (z.B. **addActionListener()**) und **remove*EventClass*Listener()** (z.B. **removeActionListener()**) lassen sich Ereignisempfängerlisten ändern (vgl. Abschnitt 14.5.3).
  - Einige Swing-Komponenten bieten spezielle Thread-sichere Methoden, z.B.: **JTextComponent.setText()**.

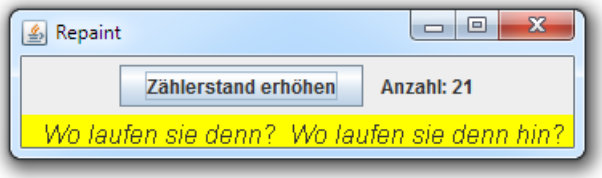
- **invokeLater()**, **invokeAndWait()**
  Mit diesen statischen Methoden, die äquivalent von den Klassen **EventQueue** und **SwingUtilities** angeboten werden, kann die **run()** - Methode eines per Parameter übergebenen **Runnable**-Objekts asynchron im Ereignisverteilungs-Thread ausgeführt werden. Die Methode kommt zum Zug, wenn alle aktuell anstehenden Ereignisse abgearbeitet sind. In Abschnitt 16.9.2 war schon die Thread-sichere Swing-Initialisierung über die Methode **invokeLater()** zu sehen.

Im folgenden Beispielprogramm wird die **repaint()** - Methode einer GUI-Komponente aus einem Hintergrund-Thread aufgerufen:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Laufschrift implements Runnable {
 private JFrame frame;
 private final String TXT = "Wo laufen sie denn?  Wo laufen sie denn hin?";
 private final int pause  = 20;
 private final int sprung = 1;
 private GraphicsPanel gp;
 private int x, tb;
 private final String LABELPREFIX = "  Anzahl der Klicks: ";
 private JLabel lab;
 private int numClicks = 0;

 public Laufschrift() {
  frame = new JFrame("Repaint");
  lab = new JLabel(LABELPREFIX + "0    ");
  JButton butt = new JButton("Klicker");
  JPanel pan = new JPanel();
  pan.add(butt); pan.add(lab);
  frame.add(pan, BorderLayout.CENTER);
  butt.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    numClicks++;
    lab.setText(LABELPREFIX + numClicks);
   }
  });

  gp = new GraphicsPanel();
  gp.setBackground(Color.YELLOW);
  gp.setFont(new Font("SansSerif", Font.ITALIC, 16));
  tb = gp.getFontMetrics(gp.getFont()).stringWidth(TXT);
  gp.setPreferredSize(new Dimension(0, 20));
  frame.add(gp, BorderLayout.SOUTH);

  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.pack();
  frame.setVisible(true);
 }
```

```
    public void run() {
     while (true) {
      try {
       Thread.sleep(pause);
      } catch(InterruptedException ie) {Thread.currentThread().interrupt();}
      if (x-sprung + tb < 0)
       x = frame.getSize().width;
      else
       x -= sprung;
      gp.repaint();
     }
    }

    public static void main(String args[]) {
     EventQueue.invokeLater(new Runnable() {
      public void run() {
       Laufschrift app = new Laufschrift();
       (new Thread(app)).start();
      }
     });
    }

    @SuppressWarnings("serial")
    private class GraphicsPanel extends JPanel {
     @Override
     public void paintComponent(Graphics g){
      super.paintComponent(g);
      g.drawString(TXT, x, 18);
     }
    }
   }
```

Während ein Hintergrund-Thread damit beschäftigt ist, einen Text kontinuierlich am unteren Fensterrad laufen zu lassen (in der südlichen Region des **BorderLayout**-Managers, vgl. Abschnitt 14.4.1), interagiert der Ereignisverteilungs-Thread mit dem Benutzer, der z.B. vorbeifahrende Autos per Mausklick zählt:[1]



Der Hintergrund-Thread basiert auf der **run()** - Methode der Klasse `Laufschrift`, die das Interface **Runnable** implementiert. Im Hintergrund-Thread wird in einer Endlosschleife nach einer kurzen Pause die `x`-Koordinate der Schrift-Ausgabeposition neu berechnet und dann die betroffene GUI-Komponente durch einen Aufruf ihrer **repaint()** - Methode gebeten, die Oberfläche unter Berücksichtigung der aktuellen Schriftposition neu zu zeichnen.

[1] Wer sich über den Text des Laufbands wundert und nicht an einen berühmten Zeichentrickfilm von Loriot denkt, kann eine Bildungslücke auf amüsante Weise schließen: https://www.youtube.com/watch?v=7LgWlAUnW9w

Als Zeichenoberfläche für die Textausgabe dient ein Objekt der von **JPanel** abgeleiteten Klasse **GraphicsPanel**. Weil die Klasse **JPanel** das Interface **Serializable** implementiert, also das Serialisieren ihrer Objekte erlaubt (vgl. Abschnitt 15.6), sollte auch die abgeleitete Klasse **GraphicsPanel** eine statische Variable namens **serialVersionUID** mit einer Versionskennung definieren, damit es beim (De-)serialisieren nicht zu Fehlern aufgrund von Weiterentwicklungen der Klassendefinition kommen kann. Wenn eine solche Variable fehlt, warnt Eclipse. Weil ein Serialisieren *nicht* in Frage kommt, wird die Warnung im Beispielprogramm durch eine Marker-Annotation unterdrückt:

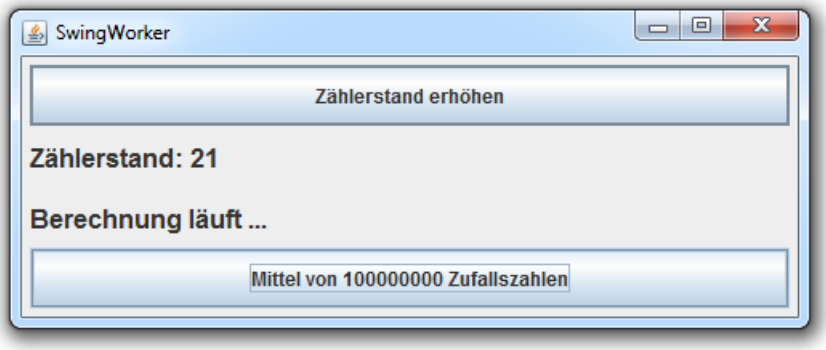
```
@SuppressWarnings("serial")
```

16.9.4 Die Klasse SwingWorker<T, V>

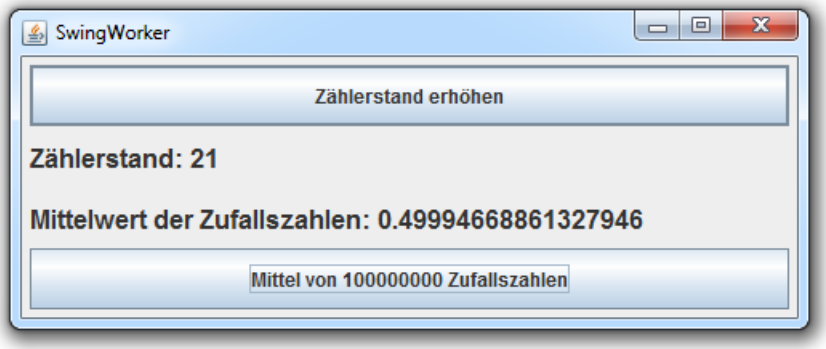
Seit Java 6 vereinfacht es die generische Klasse **SwingWorker<T,V>** im Paket **javax.swing**, …

- aufwändige Arbeiten in Hintergrund-Threads zu verlagern
- und deren Ergebnisse im Swing-GUI zu präsentieren.

Als Beispiel erstellen wir ein Zählprogramm, das auch bei intensiver Rechentätigkeit eines Hintergrund-Threads bedienbar bleibt:



Nachdem der Hintergrund-Thread den Mittelwert aus 100 Millionen pseudozufälligen **double**-Zahlen (im Intervall von 0,0 bis 1,0) berechnet hat, lässt er den EDT (Event Dispatch Thread) die Beschriftung einer **JLabel**-Komponente entsprechend ändern:



Für das Beispielprogramm wird die folgende innere Klasse `RandomNumberCruncher` definiert:

```
private class RandomNumberCruncher extends SwingWorker<Double, Void> {
 public Double doInBackground() {
  double d = 0.0;
  for (int i = 0; i < anz; i++)
   d += Math.random();
  return new Double(d/anz);
 }

 public void done() {
  try {
   lblRandom.setText("Mittelwert der Zufallszahlen: "+get());
  } catch (Exception e) {
   JOptionPane.showMessageDialog(null, "Berechnung gescheitert");
  }
 }
}
```

Die im Hintergrund-Thread auszuführende Methode **doInBackground()** berechnet mit einem enormen Aufwand ein Zufallsergebnis und liefert dieses als Rückgabewert an das Framework ab. Nach Rückkehr der Methode **doInBackground()** wird vom Framework die `RandomNumberCruncher`-Methode **done()** im EDT ausgeführt. Hier können also gefahrlos Swing-Komponenten modifiziert werden, wobei per **get()** der **doInBackground()** -Rückgabewert verfügbar ist.

In der **ActionEvent**-Behandlungsmethode zum unteren Schalter des Beispielprogramms wird ein Objekt der Klasse `RandomNumberCruncher` erzeugt und per **execute()** - Aufruf dazu gebracht, die Rechenarbeit in einem Hintergrund-Thread zu erledigen:

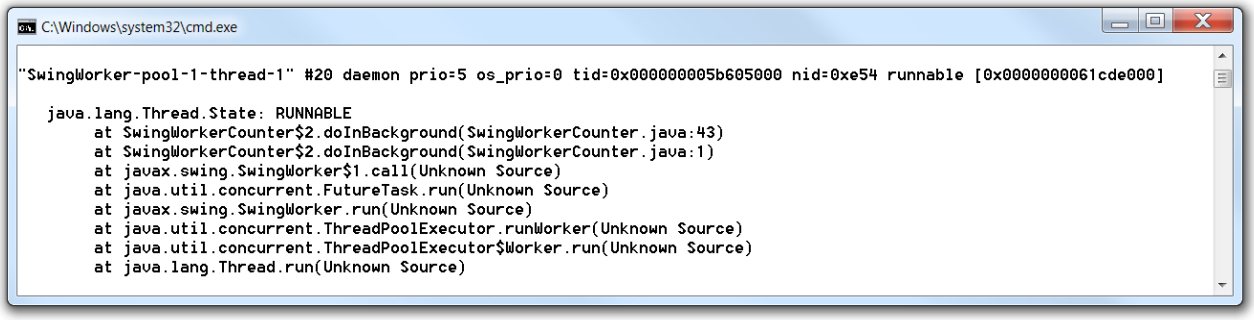
```
cbWorker = new JButton("Mittel von "+ANZ+" Zufallszahlen");
cbWorker.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
  lblRandom.setText("Berechnung läuft ...");
  (new RandomNumberCruncher()).execute();
 }
});
```

Eine **SwingWorker<T,V>** - Instanz kann nur *einmal* in einem Hintergrund-Thread ausgeführt werden, so dass im Beispiel bei jedem Aufruf der Ereignisbehandlungsmethode eine neue Instanz erzeugt werden muss.

Wird das Programm aus einem Konsolenfenster gestartet, die Hintergrundtätigkeit angefordert und dann mit der Tastenkombination **Strg+Pause** eine Liste der aktiven Threads verlangt, dann erscheint u.a. der Thread **SwingWorker-pool-1-thread-1**:[1]

[1] In der ersten sichtbaren Protokollzeile klingen Themen an, die wir schon behandelt haben:
- Der Thread SwingWorker-Thread gehört zu einem Threadpool (vgl. Abschnitt 16.5).
- Der Thread hat die Priorität 5 (vgl. Abschnitt 16.4.1).

Mit Daemon-Threads werden wir uns in Abschnitt 16.7.1 beschäftigen.



Über die in **doInBackground()** aufzurufende **SwingWorker**-Methode **publish()** und die im EDT aufzurufende **SwingWorker**-Methode **process()** können auch Zwischenergebnisse vom Hintergrund-Thread in den EDT übernommen werden.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\BspUeb\Multithreading\Swing\SwingWorker

16.10 Übungsaufgaben zu Kapitel 16

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Ein Java-Programm endet zusammen mit seinem Thread **main**.
2. Ereignisbehandlungsmethoden laufen im UI-Thread (JavaFX) bzw. im Event Dispatcher Thread (Swing) ab 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 wieder 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);
  }
 }
}
```

Außerdem wird demonstriert, dass man auch den Thread **main** per **sleep()** in den vorübergehenden Ruhezustand schicken kann. Um eine **try-catch**-Konstruktion zu vermeiden, wird die von **sleep()** potentiell zu erwartende **InterruptedException** in **main()** - Methodenkopf per **throws**-Klausel deklariert.

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 {
   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.

Operator	**Bedeutung**
[]	Array-Index
.	Komponentenzugriff
!	Negation
++, --	Prä- oder Postinkrement bzw. -dekrement
-	Vorzeichenumkehr
(*Typ*)	Typumwandlung
new	Objekterzeugung
*, /	Punktrechnung
%	Modulo
+, -	Strichrechnung
+	String-Verkettung
<<, >>	Links- bzw. Rechts-Shift
>, <, >=, <=	Vergleichsoperatoren
instanceof	Typprüfung
==, !=	Gleichheit, Ungleichheit
&	Bitweises UND
&	Logisches UND (mit unbedingter Auswertung)
^	Exklusives logisches ODER

Operator	Bedeutung
|	Bitweises ODER
|	Logisches ODER (mit unbedingter Auswertung)
&&	Logisches UND (mit bedingter Auswertung)
||	Logisches ODER (mit bedingter Auswertung)
? :	Konditionaloperator
=	Wertzuweisung
+=, -=, *=, /=, %=	Wertzuweisung mit Aktualisierung

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 Koppelung 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**
5. **Falsch**
   Während der Datentyp **String[]** des **main()**-Parameters in der Tat zwingend vorgeschrieben ist, kann man den 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[] argz) { … }` Der Modifikator **static** fehlt.
Dieser Aufruf scheitert:	`public static void main() { … }` Falsche Parameterliste
Dieser Aufruf klappt:	`static public void main(String[] argz) { … }` Die Modifikatoren **static** und **public** müssen vor dem Rückgabetyp stehen, wobei ihre Reihenfolge irrelevant ist.
Dieser Aufruf scheitert:	`public static void Main(String[] argz) { … }` Der Anfangsbuchstabe im Methodennamen **main()** muss klein geschrieben werden.

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$-10.3f", Math.PI);
  System.out.println();
  System.out.printf("%1$-10.4f%1$-10.5f%1$-10.6f", Math.PI);
 }
}
```

Aufgabe 2

Der im **println()** - Parameter unmittelbar auf die Zeichenkette folgende Plus-Operator 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.

```
"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, so dass die Zeichenfolgen „3.3“ und „2“ nacheinander an die Zeichenfolge „3.3 + 2 = “ angehängt werden.

Durch Klammerung muss dafür gesorgt werden, dass der *rechte* Plus-Operator *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.

```
3.3 + 2
```

ergibt

```
5.3
```

Anschließend bewirkt der linke Plus-Operator eine Zeichenfolgenverkettung.

```
"3.3 + 2 = " + 5.3
```

ergibt

```
"3.3 + 2 = 5.3"
```

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 (auf dem Stack), als Instanzvariablen von Objekten (auf dem Heap) und als Klassenvariablen (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

```
class Prog {
  public static void main(String[] args) {
    System.out.println("Dies ist ein Java-Zeichenkettenliteral:\n   \"Hallo\"");
  }
}
```

Aufgabe 5

```
class Prog {
  public static void main(String[] args) {

    float PI = 3.141593f;

    double radius = 2.0;

    System.out.println("Der Flaecheninhalt betraegt: "+PI*radius*radius);
  }
}
```

Abschnitt 3.4 (Eingabe bei Konsolen)

Aufgabe 1

In folgendem 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`**`*2.0` **`1*2.0`**
`(int)6/4.0*3`	**double**	`4.5`	Der Typumwandlungsoperator hat die höchste Priorität und bezieht sich daher (ohne Wirkung) auf die 6. Abarbeitung mit Zwischenergebnissen: **`(int)6`**`/4.0*3` **`6/4.0`**`*3` **`1.5*3`**
`(int)(6/4.0*3)`	**int**	`4`	Abarbeitung mit Zwischenergebnissen: `(int)(`**`6/4.0`**`*3)` `(int)(`**`1.5*3`**`)` **`(int)4.5`**
`3*5+8/3%4*5`	**int**	`25`	Abarbeitung mit Zwischenergebnissen: **`3*5`**`+8/3%4*5` `15+`**`8/3`**`%4*5` `15+`**`2%4`**`*5` `15+`**`2*5`** **`15+10`**

Aufgabe 2

Nach der Tabelle mit den Ergebnistypen der Ganzzahlarithmetik in 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`, so dass 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 Priorität als die logischen Operatoren und der Zuweisungsoperator, so dass 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 Priorität vor dem Operator **&&** ausgewertet.

`la2` erhält den Wert **true**, weil die runden Klammern dafür sorgen, dass der Operator ^ zuletzt ausgewertet 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** ist im Vergleich zu primitiven Datentypen mit einem erheblich höheren Speicher- und Zeitaufwand zu rechnen.

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, so dass 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

Zur Lösung dieser Aufgabe sollte an Stelle einer Schleife die folgende Formel verwendet werden:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Beim Kodieren der Formel sind Probleme durch die begrenzte Kapazität bzw. Genauigkeit der primitiven Datentypen zu vermeiden:

- Wenn Sie mit einer Variablen vom Typ **long** arbeiten, kommt es zu einem Ganzzahlüberlauf (vgl. Abschnitt 3.6.1).
- Wenn Sie eine Variable vom Typ **double** verwenden, kommt es zu einem Präzisionsverlust (vgl. Abschnitt 3.5.7).

Mit der in Abschnitt 3.6.1 vorgestellten Klasse **BigInteger** kann man beide Probleme vermeiden, wie das folgende Programm zeigt:

```
import java.math.*;
public class Prog {
 public static void main(String[] args) {
  BigInteger bigi = new BigInteger("1000000000001"), sum;
  System.out.printf("Summationsgrenze =     " + bigi);

  long start = System.currentTimeMillis();
  sum = bigi.multiply(bigi.add(new BigInteger("1")));
  sum = sum.divide(new BigInteger("2"));
  System.out.printf("\nSumme via BigInteger = %d (Zeit = %d)",
    sum, System.currentTimeMillis()-start);

  double d = bigi.doubleValue();
  start = System.currentTimeMillis();
  System.out.printf("\nSumme via double =     %24.0f (Zeit = %d)",
    d*(d+1)/2, System.currentTimeMillis()-start);

  long lo = bigi.longValue();
  start = System.currentTimeMillis();
  System.out.printf("\nSumme via long =       %24d (Zeit = %d)",
    lo*(lo+1)/2, System.currentTimeMillis()-start);
 }
}
```

Es liefert folgende Ausgaben:

```
Summationsgrenze =     1000000000001
Summe via BigInteger = 500000000001500000000001 (Zeit = 0)
Summe via double =     500000000001500000000000 (Zeit = 0)
Summe via long =            1001883602603448321 (Zeit = 0)
```

Unter Verwendung der Klasse **BigInteger** erhält man das korrekte Ergebnis. Die Gleitkommaarithmetik liegt knapp daneben, und bei der Ganzzahlarithmetik führt der Überlauf zu Unsinn.

Durch Verwendung der geschlossenen Formel an Stelle einer Schleife wird die vorgegebene Zeitschranke (maximal 1 Millisekunde) bei allen Berechnungen locker eingehalten, während eine Schleife „ewig“ laufen würde.

Aufgabe 8

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

„mühelos“ 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, d.h. die Aktualparametertypen müssen sich erweiternd (vgl. Abschnitt 3.5.7) in die Formalparametertypen konvertieren lassen.
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 irrelevant.

Aufgabe 6

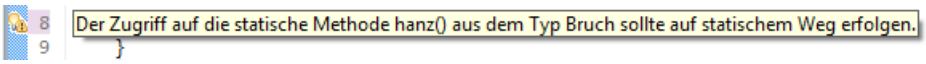
1. **Falsch**
   Der Standardkonstruktor hat dieselbe Schutzstufe wie die Klasse.
2. **Richtig**
3. **Falsch**
   Eine Methode kann eine Referenz auf ein von ihr erzeugtes Objekt an die Außenwelt übergeben (z.B. per Rückgabewert). Ein Objekt wird erst dann überflüssig und ein potentielles Opfer des Garbage Collectors, wenn im gesamten Programm keine Referenz mehr auf das Objekt vorhanden ist.
4. **Leider falsch**
   Bedauerlicherweise gelingt in Methoden berechtigter Klassen der Aufruf statischer Methoden auch über eine Objektreferenz vom Typ der angesprochenen Klasse. Die sinnvolle Anweisung

   ```
   System.out.println(Bruch.hanz() + " Brueche erzeugt");
   ```

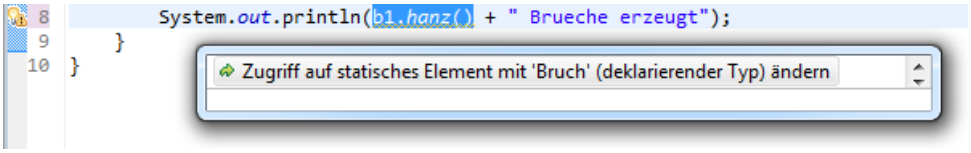
   aus der Startklasse `Bruchrechnung` kann also durch die folgende irritierende Variante (mit der lokalen `Bruch`-Referenzvariablen `b1`) ersetzt werden:

   ```
   System.out.println(b1.hanz() + " Brueche erzeugt");
   ```

   Die Denk- und Arbeitsweise der objektorientierten Programmierung aufzuweichen, schafft letztlich Unsicherheit und erhöhte Fehlerrisiken. Eclipse erkennt den schlechten Programmierstil und ermahnt:

   Nach einem Mausklick auf die Warnleuchte wird sogar ein sinnvoller Verbesserungsvorschlage zur direkten Übernahme gemacht:

Aufgabe 7

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\TimeDuration

Aufgabe 8

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 9

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\FakulRek

Das Programm eignet sich übrigens dazu, einen Stapelüberlauf (**StackOverflowError**) zu provozieren. Allerdings hat bei der Wahl eines passend großen Arguments (ca. 10000) die resultierende Fakultät den **double**-Wertebereich längst in Richtung Infinity verlassen.

Aufgabe 10

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\R2Vec

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\Marry

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<T>** (vgl. Abschnitt 3.7.3.2).
2. **Richtig**
3. **Falsch**
   Verwenden Sie stattdessen die **String**-Methode **charAt()**.
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 angetroffen wird.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Zeichenketten\PerZuf

Aufgabe 4

Die Klasse **StringBuilder** hat die von **java.lang.Object** geerbte **equals()**-Methode *nicht* überschrieben, so dass *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\Zeichenketten\StringUtil

Abschnitt 5.3 (Verpackungsklassen für primitive Datentypen)

Aufgabe 1

Lösungsvorschlag:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.println("Min. byte-Zahl:\n " + Byte.MIN_VALUE);` `  }` `}`	`Min. byte-Zahl:` `  -128`

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\Wrapper\MaxFakul

Durch Vergleiche mit dem **Double**-Grenzwert **Double.POSITIVE_INFINITY** ermittelt man, dass die Fakultät von 170 ($\approx 7{,}26 \cdot 10^{306}$) gerade noch in einer **double**-Variablen unterkommt, während die Fakultät von 171 ($\approx 1{,}24 \cdot 10^{309}$) den maximalen **double**-Wert ($1{,}7976931348623157 \cdot 10^{308}$) übertrifft.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Wrapper\Mint

Abschnitt 5.4 (Aufzählungstypen)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Enumerationen\Wochentage

Kapitel 6 (*Pakete*)

Aufgabe 1

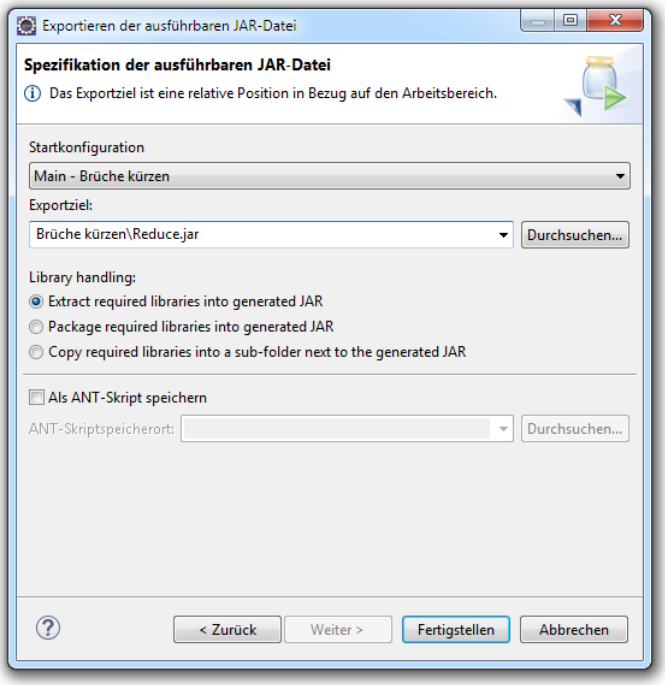
Die Klassen gehören beide zum Standardpaket und können sich somit gegenseitig „sehen“. Weil die `Worker`-Methode ohne Zugriffsmodifikator definiert wurde, besitzt sie die voreingestellte Schutzstufe *package* und kann von allen Klassen im selben Paket verwendet werden. Auch der Standardkonstruktor der Klasse `Worker` ist für alle Klassen im selben Paket verfügbar, weil er dieselbe Schutzstufe besitzt wie seine Klasse (nämlich *package*).

Aufgabe 2

Öffnen Sie das Eclipse-Projekt in

...\BspUeb\JavaFX\Brüche kürzen

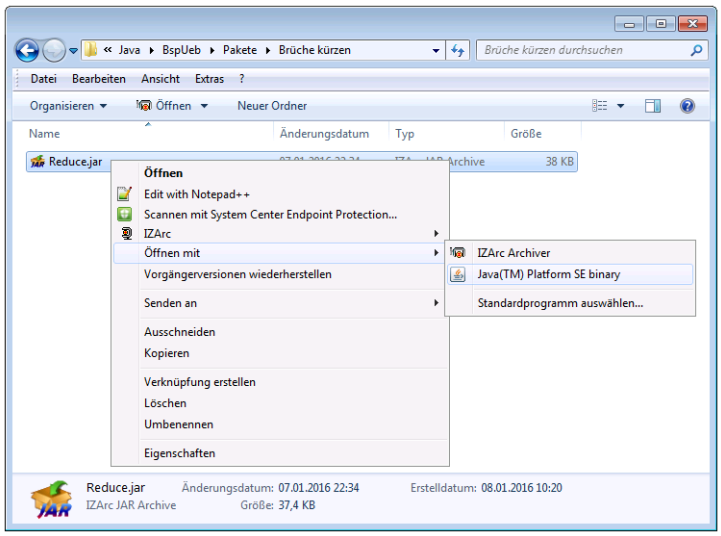
und verwenden Sie die Export-Option von Eclipse gemäß Beschreibung in Abschnitt 6.4.5:



Zu Starten des verpackten Programms kann das folgende Kommando verwendet werden, sofern sich die JAR-Datei im aktuellen Verzeichnis befindet:

```
> java -jar reduce.jar
```

Ist auf einem Rechner die aktuelle JRE der Firma Oracle installiert, sollte auch der Start per Doppelklick klappen. Wenn die Dateinamenserweiterung **.jar** von einem anderen Programm gekapert worden ist, erlaubt das Kontextmenüitem **Öffnen mit > Java™ Platform SE binary** den Maus-initiierten Programmstart, z.B.:



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, gibt es einen Dualismus zwischen Typformalparametern und Wildcard-Datentypen, so dass viele Methoden funktionsäquivalent mit beiden Techniken realisiert werden können (Bloch 2008, S. 140f).

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

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Pakete\Generische Klassen und Methoden\GenMax

Aufgabe 4

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 in Abschnitt 9.2 vorgeführt API-Interface **Serializable** enthält keine Methoden. Es ist ein so genanntes *Marker-Interface*.
4. **Richtig**
5. **Richtig**

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Interfaces\Bruch implements Comparable

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 Collection Framework*)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Pakete\Kollektionen\Baumlotto

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Pakete\Kollektionen\StringUtil

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Pakete\Kollektionen\Mengenlehre

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Pakete\Kollektionen\DataMat

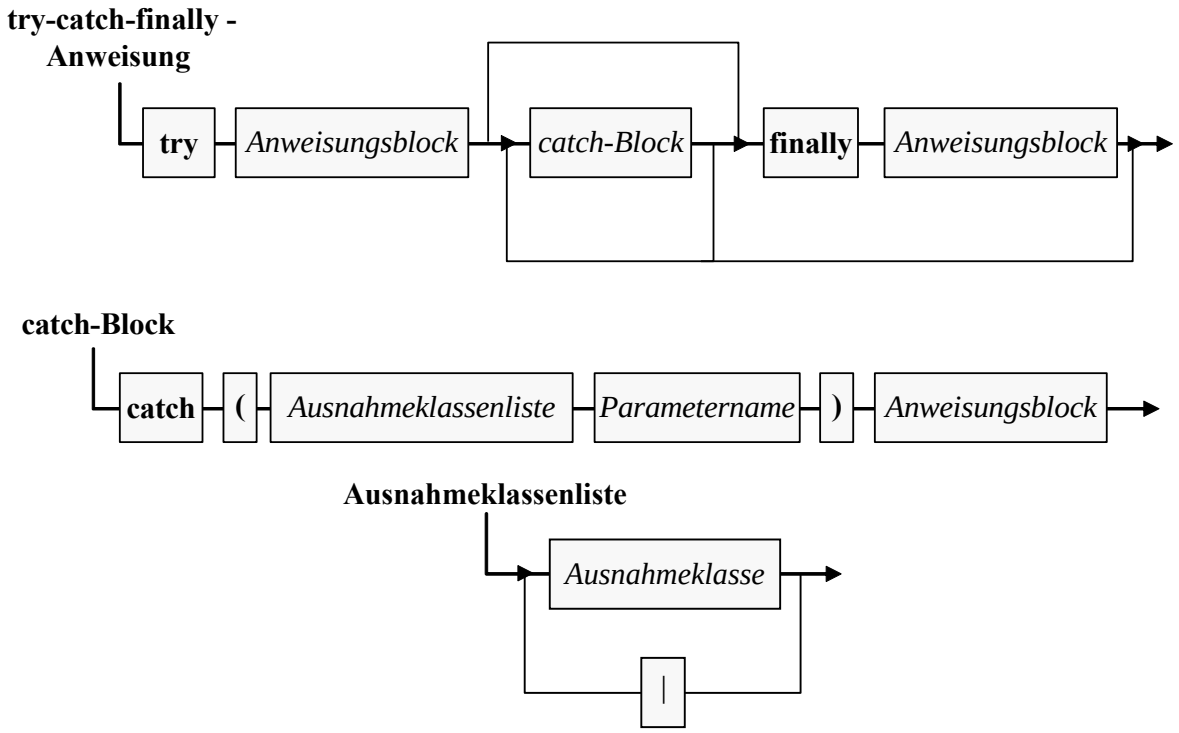
Kapitel 11 (Ausnahmebehandlung)

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 (wie bei jeder Ausnahmeklasse) 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:



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\LambdaVsAK

Mit einem Lambda-Ausdruck ist das Problem sehr elegant zu lösen:

```
int s100even = IntStream.rangeClosed(1,100)
                        .map(n -> n*2)
                        .sum();
```

Bei Verwendung einer anonymen Klasse steigt der Aufwand, während sich die Lesbarkeit des Quellcodes verschlechtert:

```
int s100even = IntStream.rangeClosed(1,100)
                        .map(new IntUnaryOperator() {
                           @Override
                           public int applyAsInt(int n) {
                             return n*2;
                           }
                         })
                        .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.:

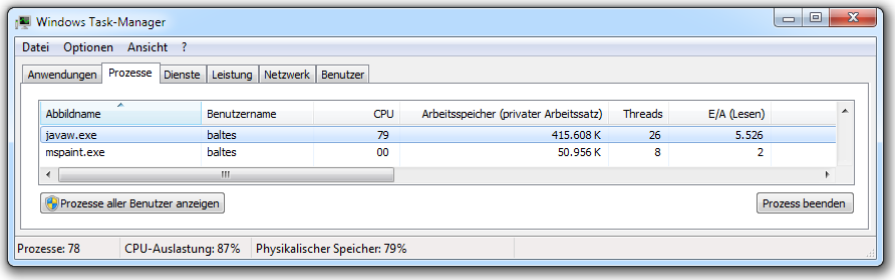
```
Predicate<String> ps = s -> s.length() >= 5;
Object obj = ps;
```

Aufgabe 3

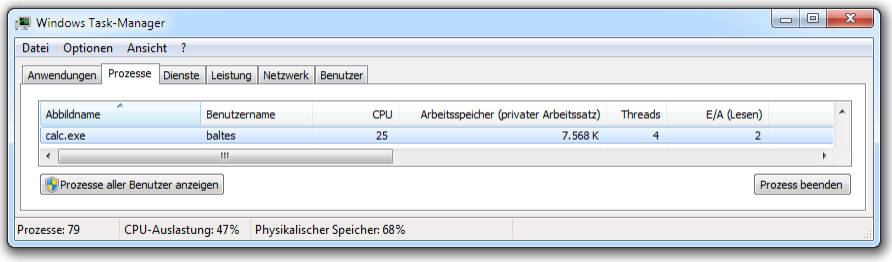
Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Funktionales Programmieren\Fakultät

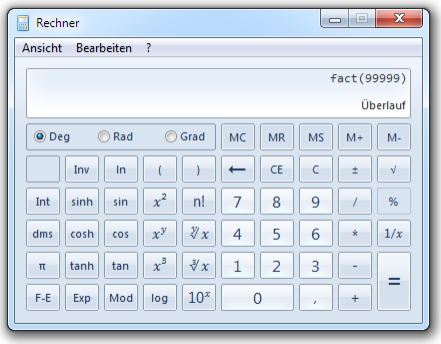
Der Lösungsvorschlag muss übrigens einen Vergleich mit dem Taschenrechner von Windows 7 nicht scheuen. Das Java-Programm liefert z.B. zum Argument 99.999 bei Verwendung eines parallelen Stroms nach wenigen Sekunden eine Lösung und arbeitet dabei parallel mit mehreren CPU-Kernen, was an der Nutzung von mehr als 25% CPU-Zeit auf einem Rechner mit 4 virtuellen Kernen zu erkennen ist:



Der Windows-Taschenrechner lässt sich erheblich mehr Zeit und benutzt dabei nur *einen* CPU-Kern, was an der Nutzung von maximal 25% CPU-Zeit auf einem Rechner mit 4 virtuellen Kernen zu erkennen ist:



Außerdem beendet der Windows-Taschenrechner seine Tätigkeit mit einer Fehlermeldung:



Abschnitt 13 (*GUI-Programmierung mit JavaFX*)

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\JavaFX\Steuerelemente\Euro-Konverter

Abschnitt 14 (*GUI-Programmierung mit Swing*)

Aufgabe 1

1. **Falsch**
   Siehe Abschnitt 14.5.2 (Ereignisarten und Ereignisklassen)
2. **Richtig**
3. **Richtig**
4. **Falsch**

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\EventID

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\KeyModifiers

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\AnonClass

Eine alternative Technik besteht darin, ein Objekt einer anonymen Klasse zu erstellen und seine Adresse in einer Referenzvariablen vom Typ der zugehörigen Ereignisschnittstelle abzulegen:

```
final ActionListener acList = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
   info.setText(" "+ ((JButton) e.getSource()).getText()+" gedrückt");
  }
 };
```

Anschließend kann das Objekt beliebig oft als Ereignisempfänger registriert werden, z.B.:

```
butt1.addActionListener(acList);
butt2.addActionListener(acList);
```

Aufgabe 5

Einige Unterschiede zwischen Ausnahme- und Ereignisbehandlung in Java:

- Eine GUI-Anwendung kann als Ansammlung von Ereignisbehandlungsmethoden betrachtet werden. Ereignisse sind also unverzichtbare Bestandteile des normalen Ablaufs und bringen durch den Aufruf der zugehörigen Behandlungsmethoden das Geschehen voran. Eine Ausnahme stellt hingegen eine Störung des regulären Ablaufs dar und führt zum Abbruch einer Methodenausführung.
- Eine Ausnahme *muss* behandelt werden, um die Beendigung des Programms (genauer: des Threads) zu verhindern. Ein Ereignis hat nur dann Konsequenzen, wenn zuvor ein Empfänger registriert wurde.
- Ein Event Handler ist für eine spezielle Event Id zuständig, z.B. für Mausklicks:
  ```
  public void mouseClicked(MouseEvent e) { . . . }
  ```
  Ein exception handler kann für eine beliebig breite Klasse von Ausnahmen zuständig sein,
  ```
  catch (Exception e) { . . . }
  ```

Aufgabe 6

Swing-Container-Klasse	**Voreingestellter Layout-Manager**
javax.swing.JFrame (Genau genommen geht es um die Inhaltsschicht eines **JFrame**-Objekts.)	**java.awt.BorderLayout**
javax.swing.JPanel	**java.awt.FlowLayout**
javax.swing.Box	**javax.swing.BoxLayout**

Aufgabe 7

Adapterklassen erleichtern das Implementieren von Ereignisempfängerklassen, indem sie die zugehörigen Interface-Methoden leer implementieren, so dass man in einer eigenen Adapterklassen-Ableitung nur die tatsächlich benötigten (zu bestimmten Event IDs gehörigen) Ereignisbehandlungsmethoden implementieren (überschreiben) muss. Das Interface **ActionListener** schreibt mit **actionPerformed()** nur eine einzige Methode vor, die ein (sinnvoller) Ereignisempfänger auf jeden Fall implementieren muss, so dass eine Adapterklasse nutzlos wäre.

Aufgabe 8

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\MPC Professional

Aufgabe 9

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\E2 (mit Hintergrundfarbe)

Aufgabe 10

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\Euro-Konverter

Aufgabe 11

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\Zwischenablage

Abschnitt 15 (Ein-/Ausgabe über Datenströme)

Aufgabe 1

1. **Falsch**
   Es stimmt, dass die aus Java 1.0 stammende 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. **Falsch**
   Es entscheidet die Systemeigenschaft **file.encoding**, deren Wert mit der **System**-Methode **getProperty()** ermittelt werden kann.
5. **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\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 **PrintWriter** (bzw. durch den eingebundenen **OutputStreamWriter**, vgl. Abschnitt 15.4.1.5) wird abgeschaltet. Für das Programm ist der folgende Konstruktoraufruf besser geeignet:

```
PrintWriter pw = null;
  . . .
pw = new PrintWriter(fos);
```

Aufgabe 5

Beim Ausführen der Methode **readShort()** fordert das **DataInputStream**-Objekt den angekoppelten **InputStream** auf, zwei Bytes zu beschaffen. Schickt der Benutzer unter Windows z.B. eine eingetippte „0“ per **Enter**-Taste ab, dann gelangen folgende Bytes in den **InputStream**:

- 00110000 (0x30, 8-Bit-Code der *Ziffer* „0“)
- 00001101 (0x0D, 8-Bit-Code für Wagenrücklauf)
- 00001010 (0x0A, 8-Bit-Code für Zeilenvorschub)

Anschließend entnimmt **readShort()** dem **InputStream** die beiden ersten Bytes und interpretiert sie als 16-Bit-Ganzzahl, was im Beispiel den dezimalen Wert 12301 ergibt:

$$0011000000001101_2 = 12301_{10}$$

Schickt der Benutzer eine leere Eingabe mit **Enter** ab, resultiert der **short**-Wert:

$$0000110100001010_2 = 3338_{10}$$

Während der Filterstrom korrekt arbeitet, ist die Eingabe passender Bytes via Tastatur (Standardeingabestrom **System.in**) äußerst umständlich bis unmöglich.

Um numerische Daten von der Konsole zu lesen, sollte man die in Abschnitt 15.5 beschriebene Klasse **Scanner** verwenden.

Die Transformationsklasse **DataInputStream** ist dafür konstruiert, aus einem *binären* Eingabestrom Werte primitiver Typen zu lesen. Oft sind diese Werte zuvor von einem Objekt der Klasse **DataOutputStream** geschrieben worden.

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\IO\MS-DOS

Aufgabe 7

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\IO\DataMatrix

Abschnitt 16 (Multithreading)

Aufgabe 1

1. **Falsch**
2. **Richtig**
   Werden im UI-Thread bzw. im EDT 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 `Schnarcher`-Thread wird fast immer schlafend angetroffen. In diesem Fall wird eine **InterruptedException** geworfen und das Interrupt-Signal wieder gelöscht. Damit die **run()** - Methode

plangemäß reagieren kann, muss bei der Ausnahmebehandlung **interrupt()** erneut aufgerufen werden:

```
public class Schnarcher extends Thread {
 public void run() {
  while (true) {
   if(isInterrupted())
    return;
   try {
    sleep(100);
   } catch(InterruptedException ie) {
    interrupt();
   }
   System.out.print("*");
  }
 }
}
```

Aufgabe 3

Weil jeder Thread seinen eigenen Stapelspeicher für Methodenaufrufe besitzt, sind bei lokalen Variablen konkurrierende Zugriffe durch mehrere Threads unmöglich.

Literatur

Ball, J., Carson,D. B., Evans, I., Haase, K & Jendrock, E. (2006). *The Java EE 5 Tutorial*. Santa Clara, CA: Sun Microsystems.

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. (2011). *Einführung in das Programmieren mit C# 4.0*. Online-Dokument: http://www.uni-trier.de/index.php?id=22777

Baltes-Götz, B. (2016). *Einführung in die Entwicklung von Android-Apps*. Online-Dokument: http://www.uni-trier.de/index.php?id=60390

Balzert, H. (1999). *Lehrbuch der Objektmodellierung: Analyse und Entwurf*. Heidelberg: Spektrum.

Bloch, J. (2008). *Effective Java* (2nd. ed.). Upper Saddle River, NJ: Addison-Wesley.

Bracha, G. (2004). *Generics in the Java Programming Language*. Online-Dokument: http://www.cs.rice.edu/~cork/312/Readings/GenericsTutorial.pdf (abgerufen: 05.01.2014)

Burd, B. (2005). Eclipse for Dummies. Hoboken, HJ: Wiley.

Deitel, H. M. & Deitel, P. J. (1999). *Java: how to program* (3rd ed.). Upper Saddle River, NJ: Prentice Hall.

Deitel, H. M. & Deitel, P. J. (2005). *Java: how to program* (6th ed.). Upper Saddle River, NJ: Prentice Hall.

Echtle, K. & Goedicke, M. (2000). *Lehrbuch der Programmierung mit Java*. Heidelberg: dpunkt.

Ebner, M. (2000). *Delphi 5 Datenbankprogrammierung*. München: Addison-Wesley.

Eckel, B. (2002). *Thinking in Java* (3rd ed.). New Jersey: Prentice Hall. Online-Dokument: http://www.mindview.net/Books/TIJ/

Eidenberger, H. & Divotkey, R. (2004). *Medienverarbeitung in Java*. Heidelberg: dpunkt.verlag.

Erlenkötter, H. (2001). *Java. Programmieren von Anfang an*. Reinbek: Rowohlt.

Evans, B.J. & Flanagan, D. (2015). *Java in a Nutshell* (6th ed.). Sebastopol, CA: O'Reilly.

Flanagan, D. (2005). *Java in a Nutshell* (5th ed.). Sebastopol, CA: O'Reilly.

Fowler, A. (2003). *Painting in AWT and Swing*. Online-Dokument: http://java.sun.com/products/jfc/tsc/articles/painting/index.html

Fuchs, D. (2012). *Connecting Scene Builder edited FXML to Java Code*. Online-Dokument (abgerufen: 08.02.2016): https://blogs.oracle.com/jmxetc/entry/connecting_scenebuilder_edited_fxml_to

Goetz, B. (2006a). *Java Concurrency in Practice*. Addison-Wesley. Upper Saddle River, NJ: Addison-Wesley

Goetz, B. (2006b). *Dealing with InterruptedException*. Online-Dokument (abgerufen am 27.03.2016): http://www.ibm.com/developerworks/library/j-jtp05236/j-jtp05236-pdf.pdf

Goetz, B. (2007). *Java Theory and Practice. Stick a Fork in it, Part 1*. Online-Dokument (abgerufen am 08.04.2012): 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.

Gosling, J., Joy, B., Steele, G., Bracha, G. & Buckley, A. (2015). *The Java Language Specification. Java SE 8 Edition* (Ausgabe 2015-02-13). Online-Dokument (abgerufen: 05.12.2015): http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

Grossmann, D. (2012). Beginner's Introduction to Java's ForkJoin Framework. Online-Dokument: http://www.cs.washington.edu/homes/djg/teachingMaterials/grossmanSPAC_forkJoinFramework.html (abrefufen: 08.04.2012)

Hennebrüder, S. (2007). *Hibernate. Das Praxisbuch für Entwickler*. Bonn: Galileo.

Hommel, S. (2014). Oracle JvaFX. Implementing JavaFX Best Practices. Online-Dokument: http://docs.oracle.com/javafx/2/best_practices/jfxpub-best_practices.pdf (abgerufen: 07.02.2016)

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 (abgerufen: 26.12.2014): http://www.drdobbs.com/jvm/lambda-expressions-in-java-8/240166764?pgno=1

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.

Kreft, K & Langer, A. (2008). *Regeln für die Verwendung von volatile*. Online-Dokument (abgerufen: 16.02.2015): http://www.angelikalanger.com/Articles/EffectiveJava/42.JMM-volatileIdioms/42.JMM-volatileIdioms.html

Kreft, K & Langer, A. (2014). *Java 8. Default-Methoden und statische Methoden in Interfaces*. Online-Dokument (abgerufen: 18.12.2014): http://www.angelikalanger.com/Articles/EffectiveJava/72.Java8.DefaultMethods/72.Java8.DefaultMethods.html

Kröckertskothen, T. (2001). *Java 2. Grundlagen und Einführung*. Hannover: RRZN.

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

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.

Middendorf, S. & Singer, R. (1999). *Java. Programmierhandbuch und Referenz für die Java-2-Plattform*. Heidelberg: dpunkt.

Minhorst, A. & Schäfer, U. (2006). Fadenspiel. Objektrelationales Mapping in Java und .NET. *c't Magazin für Computertechnik*. 2006, Heft 9, 236-243.

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.

Mössenböck, H. (2005). *Sprechen Sie Java? Einführung in das systematische Programmieren*. Heidelberg (3. Aufl.). Heidelberg, dpunkt.Verlag.

Müller, N. (2004). *Rechner-Arithmetik*. Online-Dokument: http://www.informatik.uni-trier.de/~mueller/Lehre/2005-arith/arith-2003-folien.pdf

Muller, H. & Walrath, K. (2000). *Threads and Swing*. Online-Dokument: http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html

SELFHTML (2015). *SELFHTML-Wiki*. Online-Dokument: http://wiki.selfhtml.org/wiki/Startseite

MySQL AB (2008). *MySQL 5.0 Reference Manual*. Online-Dokument http://dev.mysql.com/doc/

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.

Oracle (2015). *The Java Tutorials*. Online-Dokument: http://download.oracle.com/javase/tutorial/.

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.

Rittmeyer, W. (2005). *JSP-Tutorial*. Online-Dokument: http://www.jsptutorial.org/content

RRZN (1999). *Grundlagen der Programmierung mit Beispielen in C++ und Java*. Hannover.

RRZN (2004). *MySQL Administration*. Hannover.

RRZN (2005). *Eclipse 3*. Hannover.

Sharan, K. (2015). *Learn JavaFX 8*. New York: Apress.

Simons, R. (2004). *Hardcore Java*. Sebastopol, CA: O'Reilly.

Spurgeon, C. E. (2000). *Ethernet. The Definitive Guide*. Sebastopol, CA: O'Reilly.

Strey, A. (2003). Computer-Arithmetik. Online-Dokument: http://www.informatik.uni-ulm.de/ni/Lehre/WS02/CA/CompArith.html

Sun Microsystems (2009). JSR 317. Java(TM) Persistence API, Version 2.0. Online-Dokument: http://jcp.org/en/jsr/detail?id=317

Ullenboom, C. (2012a). *Java ist auch eine Insel* (10. Aufl.). Bonn: Galileo. OpenBook: http://openbook.galileocomputing.de/javainsel/

Ullenboom, C. (2012b). *Java 7 - Mehr als eine Insel*. Bonn: Galileo. OpenBook: http://openbook.galileocomputing.de/java7/index.html

Urma, R.-G. (2014). *Processing Data with Java SE 8 Streams, Part 1*. Online-Dokument (abgerufen am 06.01.2014): http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

Vogel, L. (2012). *JPA 2.0 with EclipseLink - Tutorial*. Online-Dokument (abgerufen am 23.09.2012): http://www.vogella.com/articles/JavaPersistenceAPI/article.html.

Vorontsov, M. (2014). *Java Performance Tuning Guide. String.intern in Java 6, 7 and 8 – string pooling*. Webseite (abgerufen am 13.12.2014): 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.

Index