B. Baltes-Götz

# Einführung in das Programmieren mit Java 6


2010.04.05

Herausgeber: Universitäts-Rechenzentrum Trier
Universitätsring 15
D-54286 Trier
WWW: http://www.urt.uni-trier.de/
E-Mail: urt@uni-trier.de
Tel.: (0651) 201-3417, Fax.: (0651) 3921
Autor: Bernhard Baltes-Götz (E-Mail: baltes@uni-trier.de)

# Vorwort

Dieses Manuskript entstand als Begleitlektüre zum Java - Einführungskurs, den das Universitäts-Rechenzentrum Trier (URT) im Wintersemester 2009/2010 angeboten hat, sollte aber auch für das Selbststudium geeignet sein.

### Lerninhalte und -ziele

Die von der Firma **Sun Microsystems** 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 wohl den größten Verbreitungsgrad, und dieses 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 aktuellen Version 6 als Open Source unter die GPL **(***General Public License)* zu stellen, ist in der Entwicklerszene positiv aufgenommen worden und trägt sich zum wachsenden 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 2009). Beide Konkurrenten inspirieren sich gegenseitig und treiben so den Fortschritt voran.

Der Kurs beschränkt sich auf die *Java Standard Edition* (JSE) zur Entwicklung von Anwendersoftware, auf die viele weltweit populäre Softwarepakete setzen (z.B. Matlab, SPSS). 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). Nach derzeitiger Planung findet im Sommersemester 2010 ein Java-Aufbaukurs statt, der sich auch mit der Java Enterprise Edition beschäftigen wird und zu einer Erweiterung des vorliegenden Manuskripts führen wird.

Im Kurs geht es nicht um Kochrezepte zur Erstellung lebendiger Webseiten, 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. Jedoch wird auch die Entwicklung von Java-Applets für den Einsatz im Internet berücksichtigt.

### Voraussetzungen bei den Leser(innen)

- **EDV-Allgemeinbildung**
  Dass die Kursteilnehmer (innen)[1] wenigstens durchschnittliche Erfahrungen bei der *Anwendung* von Computerprogrammen haben sollten, versteht sich von selbst.
- **Programmierkenntnisse**
  Programmierkenntnisse werden *nicht* vorausgesetzt. Kursteilnehmer(innen) *mit* Programmiererfahrung werden sich bei den ersten Abschnitten eventuell etwas langweilen.
- **Motivation**
  Generell ist mit einem erheblichen Zeitaufwand von ca. 3-5 Stunden pro Woche bei der Lektüre und bei der aktiven Auseinandersetzung mit dem Stoff (z.B. durch das Lösen von Übungsaufgaben) zu rechnen.
- **EDV-Plattform**
  Wir arbeiten im Kurs mit PCs unter Microsoft Windows. Weil dabei ausschließlich Java-Software zum Einsatz kommt, ist die Plattformunabhängigkeit jedoch garantiert.

[1] Zugunsten einer guten Lesbarkeit beschränkt sich das Manuskript meist auf die männliche Form.

### Software zum Üben

Für die unverzichtbaren Übungen sollte ein Rechner zur Verfügung stehen, auf dem das **Java SE Development Kit 6.0** der Firma Sun Microsystems mitsamt der zugehörigen Dokumentation installiert ist. Als integrierte Entwicklungsumgebung zum komfortablen Erstellen von Java-Software wird **Eclipse** in einer Version ab 3.0 empfohlen. Die genannte Software ist kostenlos für alle signifikanten Plattformen (z.B. Linux, MacOS, UNIX, Windows) verfügbar und wird für die Kursteilnehmer auf einer Begleit-DVD zusammengestellt. Nähere Hinweise zum Bezug, zur Installation und zur Verwendung folgen in Abschnitt 2.1.

### Dateien zum Manuskript

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:

Rechenzentrum > Studierende > EDV-Dokumentationen >
Programmierung > Einführung in das Programmieren mit Java

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

baltes@uni-trier.de

werden dankbar entgegen genommen.

Trier, im April 2010 Bernhard Baltes-Götz

# Inhaltsverzeichnis

## 3 ELEMENTARE SPRACHELEMENTE 47

# 1 Einleitung

Im ersten Kapitel geht es zunächst um die Denk- und Arbeitsweise (leicht übertrieben: die *Weltanschauung*) 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 Bildern, Klängen 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 Gegenstandsbereich *modellieren*
  Beispiel: In einem Programm zur Verwaltung einer Spedition sind z.B. Fahrer, Fahrzeuge, Servicestationen, Zielpunkte etc. und die kommunikativen Prozesse 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 Fahrt 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 allerdings 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 aufwändige, Speditionsverwaltungsprogramm jetzt nicht im Detail vorstellen.

Wir analysieren stattdessen ein Beispielprogramm, das trotz angestrebter Einfachheit nicht auf objektorientiertes Programmieren (OOP) verzichtet. Seine Aufgabe besteht darin, elementare Operationen mit Brüchen auszuführen (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 **Klassen** und deren Kommunikation untereinander. Man identifiziert die beteiligten **Objektsorten** und definiert für sie jeweils eine **Klasse**. Eine solche Klasse ist gekennzeichnet durch:

- **Eigenschaften** (in Java repräsentiert durch **Felder**)
  Viele Eigenschaften gehören zu den *Objekten* bzw. *Instanzen* der Klasse (z.B. Zähler und Nenner eines Bruchs), manche gehören zur Klasse selbst (z.B. Anzahl der in 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** (in Java repräsentiert durch **Methoden**)
  Analog zu den Eigenschaften sind auch die Handlungskompetenzen entweder individuellen Objekten bzw. Instanzen zugeordnet (z.B. das Kürzen bei Brüchen) oder der Klasse selbst (z.B. das Erstellen neuer Objekte). 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).

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 ausnahmsweise *nicht* zum Instantiieren (Erzeugen von Objekten) gedacht ist sondern als Akteur.

In unserem Bruchrechnungsbeispiel kann man sich bei der objektorientierten Analyse vorläufig wohl auf die Klasse der *Brüche* beschränken. Beim möglichen Ausbau des Programms zu einem Bruchrechnungstrainer kommen jedoch sicher weitere Klassen hinzu (z.B. Aufgabe, Übungsaufgabe, Testaufgabe).

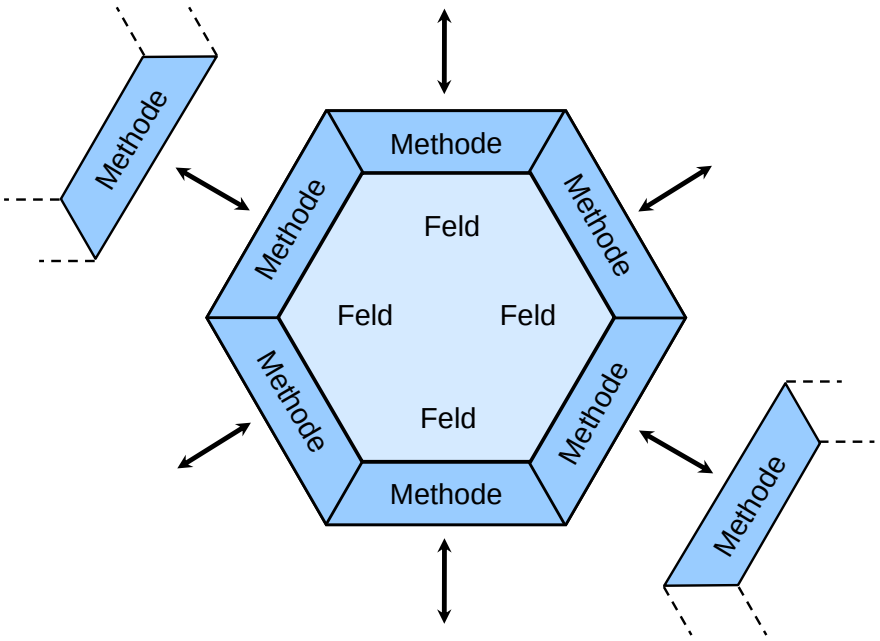
Dass Zähler und Nenner die zentralen **Eigenschaften** eines Bruchs sind, bedarf keiner Begründung. Sie werden in der Klassendefinition durch ganzzahlige Felder (Java-Datentyp **int**) repräsentiert:

- `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 direktem Zugriff durch fremde Klassen geschützt sein. So ist sichergestellt, dass nur sinnvolle Änderungen der Eigenschaften möglich sind. Außerdem wird aus später zu erläuternden Gründen die Produktivität der Softwareentwicklung durch die Datenkapselung gefördert.

Wie die folgende, an Goll et al. (2000) angelehnte, Abbildung zeigt, bilden die **Handlungskompetenzen** (Methoden) einer Klasse demgegenüber ihre öffentlich zugängliche **Schnittstelle** zur Kommunikation mit anderen Klassen:




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 Gegenstandbereiches ist nicht unbedingt eine direkte Abbildung, sondern eine *Rekonstruktion*. Einerseits soll der Gegenstandsbereich im Modell gut repräsentiert sein, andererseits soll eine möglichst stabile, gut erweiterbare und wieder verwendbare 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 wohl über Methoden zum Verändern von Zähler und Nenner verfügen. Bei einer geschützten Eigenschaft ist auch der direkte *Lese*zugriff ausgeschlossen, so dass im `Bruch`-Beispiel auch noch Methoden zum Ermitteln von Zähler und Nenner ratsam sind. 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 entworfenen Methoden möglich sind. Treten doch Fehler auf, sind diese leichter 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 1.2.2), von denen alle Java-Programmierer regen Gebrauch machen.

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

- `setzeZaehler(int zpar), setzeNenner(int npar)`
  Das 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 Null 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 (z.B. `zpar`) und einen Datentyp. Im Beispiel erlaubt der Datentyp **int** ganze Zahlen als Werte.
- `gibZaehler(), gibNenner()`
  Das `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()`
  Das Objekt wird beauftragt, `zaehler` und `nenner` zu kürzen. Welcher Algorithmus dazu benutzt wird, bleibt dem Objekt bzw. dem Klassendesigner überlassen.
- `addiere(Bruch b)`
  Das 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()`
  Das Objekt wird beauftragt, `zaehler` und `nenner` beim Anwender via Konsole (Eingabeaufforderung, „DOS-Fenster“) zu erfragen.
- `zeige()`
  Das 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. Welche Methoden eine Klasse benötigt, hängt von der geplanten Verwendung ab. Beim Eigenschaftsbegriff ist eine (ungefährliche) Zweideutigkeit festzustellen, die je nach Anwendungsbeispiel mehr oder spürbar wird (beim Bruchrechnungsbeispiel überhaupt nicht). Man kann unterscheiden:

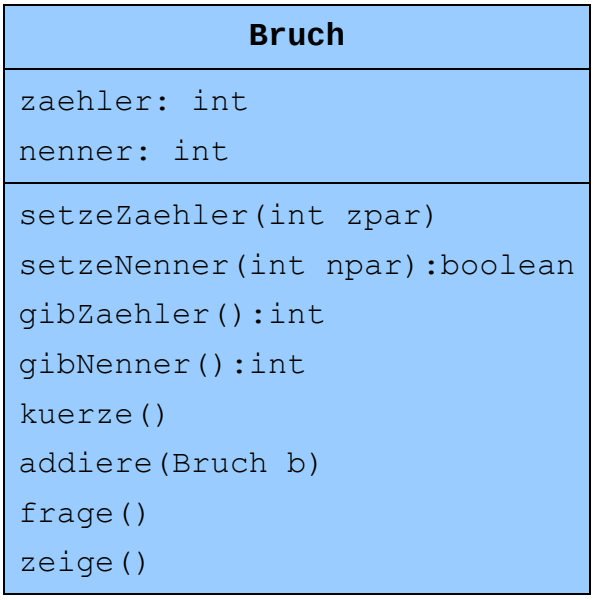
- real definierte, meist gekapselte Felder
  Diese sind für die Außenwelt (für anders Klassen) irrelevant und unbekannt. In diesem Sinn wurde der Begriff oben eingeführt.
- nach Außen dargestellte Eigenschaften
  Eine solche Eigenschaft ist über Methoden zugänglich und *nicht* unbedingt durch ein einzelnes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

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

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

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

Am Anfang von Abschnitt 1.1 wurden mit der Modellierung des Gegenstandbereichs 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 Kurses 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

[1] Für die im Kurs bevorzugte Java-Entwicklungsumgebung Eclipse (siehe Abschnitt 2) sind etliche, teilweise kostenlose UML-Werkzeuge verfügbar (siehe z.B. http://www.eclipse.org/modeling/mdt/).

Java sowie der zugehörigen Standardbibliothek beschäftigen und dabei mit möglichst einfachen Beispielprogrammen (Algorithmen) arbeiten.

Auch im Einführungsbeispiel treten überwiegend einfache Methoden auf, bei denen man kaum von *Algorithmen* sprechen wird. Eine Ausnahme stellt die Methode `kuerze()` dar, die über den **euklidischen Algorithmus** den größten gemeinsamen Teiler (ggT) von Zähler und Nenner eines Bruchs bestimmt, 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 $u$ und $v$ ($u > v > 0$) der ggT gleich dem ggT von $v$ und ($u$ - $v$) ist:

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

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

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

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

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

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

Folglich ist $t$ auch ein Teiler von $v$:

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

Weil die Paare ($u$, $v$) und ($u$, $u$ - $v$) dieselben Mengen gemeinsamer Teiler besitzen, sind auch die größten gemeinsamen Teiler identisch. Weil die Zahl Eins als trivialer Teiler zugelassen ist, existiert übrigens 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 werden Sie eine erheblich effizientere Variante implementieren.

### 1.1.3 Objektorientierte Programmierung

In unserem einfachen Beispielprojekt soll nun die `Bruch`-Klasse 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. Demgegenüber werden die Methoden über den Modifikator **public** für die Verwendung in klassenfremden Methoden frei gegeben. 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 zpar) {zaehler = zpar;}

   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 : ");
         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 Benutzeroberfläche, Netzwerkverbindungen, Störungen des normalen Programmablaufs).

### 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 `Bruch`-Klasse in einer Konsolenanwendung zur Addition von zwei Brüchen. Dabei bringen wir einen Akteur ins Spiel, der in einem einfachen sequentiellen Handlungsplan `Bruch`-Objekte anfordert 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 seine Startklasse ausfindig gemacht und aufgefordert, ihre **main()**-Methode auszuführen.

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 `Bruch`-Klasse 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 andere Klasse zum Starten verwenden, wird u.a. gleich demonstriert, wie leicht das Hauptergebnis unserer Arbeit (die `Bruch`-Klasse) für verschiedene Projekte genutzt werden kann.

In der `BruchAddition`–Methode **main()** werden zwei Objekte (Instanzen) aus der Klasse `Bruch` angefordert und mit der Ausführung verschiedener Methoden beauftragt. Beim Erstellen 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 einfache Operationen mit 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 Handlungskompetenzen (Methoden) **und** alle 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 Abschnitt 7) 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 Klass `BruchAddition` ebenfalls in der Datei **Bruch.java** unterzubringen.

Wie aus den beiden vorgestellten Klassen ein ausführbares Programm entsteht, erfahren Sie in Abschnitt 2. Wer sich schon jetzt von der Nützlichkeit unseres Bruchrechnungsprogramms überzeugen möchte, findet eine ausführbare Version an der im Vorwort angegebenen Stelle im Ordner

**...\BspUeb\Einleitung\Bruch\Konsole**

und kann die beiden Dateien **Bruch.class** und **BruchAddition.class** mit ausführbarem Java-Bytecode (siehe unten) auf einen eigenen Datenträger kopieren. Weil die Klasse `Bruch` wie viele andere im Kurs verwendete Beispielklassen mit konsolenorientierter Benutzerinteraktion die (nicht zur Java-Standardbibliothek gehörige) Klasse `Simput` verwendet, muss auch die Klassendatei **Simput.class** übernommen werden. Sobald Sie die zur Vereinfachung der Konsoleneingabe (*Simple Input*) für den Kurs entworfene Klasse `Simput` in eigenen Programmen einsetzen sollen, wird sie näher vorgestellt. 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\Einleitung\Bruch\Konsole
```

- Starten Sie das Programm unter Beachtung der Groß/Kleinschreibung mit

  ```
  java BruchAddition
  ```

  Zum Starten eines Programms ruft man eine **Java Runtime Environment** (JRE, vgl. Abschnitt 1.2.1) auf und gibt als Kommandozeilenoption die Startklasse an.
- Eine noch fehlende JRE besorgt man sich in der Regel bei der Firma **Sun Microsystems** über die Webseite

  http://java.sun.com/javase/downloads/index.jsp

  Bei der in Abschnitt 2.1 beschriebenen und für Programmierer empfohlenen JDK-Installation (*Java Development Kit*) landet auch eine JRE auf der Festplatte, so dass keine zusätzliche JRE-Installation erforderlich ist.

Ab jetzt sind Bruchadditionen kein Problem mehr:


### 1.1.5 Ausblick auf Anwendungen mit graphischer Benutzerschnittstelle

Das obige Beispielprogramm arbeitet der Einfachheit halber mit einer konsolenorientierten Ein- und Ausgabe. Nachdem wir in dieser übersichtlichen Umgebung grundlegende Sprachelemente kennen gelernt haben, werden wir uns natürlich auch mit der Programmierung von graphischen Benutzerschnittstellen beschäftigen. In folgendem Programm zur Addition von Brüchen wird die oben definierte Klasse `Bruch` verwendet, ihre Methoden `frage()` und `zeige()` sind jedoch durch grafikorientierte Varianten ersetzt worden:


Mit dem Quellcode zur Gestaltung der graphischen Oberfläche könnten Sie im Moment noch nicht allzu viel anfangen. Am Ende des Kurses werden Sie derartige Anwendungen aber mit Leichtigkeit

erstellen, zumal die im Kurs bevorzugte Entwicklungsumgebung *Eclipse* (siehe Abschnitt 2.3) die Erstellung graphischer Benutzeroberflächen durch einen visuellen Editor erleichtert.

Zum Ausprobieren startet man mit Hilfe der Java Runtime Environment (JRE), vgl. Abschnitt 1.2.1) aus dem Ordner

**...\BspUeb\Einleitung\Bruch\GUI**

die Klasse `BruchAdditionGui`:

```
java BruchAdditionGui
```

`BruchAdditionGui` stützt sich auf etliche andere Klassen, die im selben Ordner anwesend sein müssen.

### 1.1.6 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 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
  Die zum Erlernen elementarer Sprachelemente geeigneten Beispielprogramme werden mit einer einzigen Klasse auskommen. Praxisgerechte Programme bestehen in der Regel aus zahlreichen Klassen.
- Eine Klasse ist charakterisiert durch Eigenschaften (Felder) und 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, in der Regel unter Verwendung von zahlreichen vordefinierten 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 Die Java-Plattform*

Weil auf der indonesischen Insel Java eine auch bei Programmierern hoch geschätzte Kaffee-Sorte wächst, kam die in diesem Kurs vorzustellende Software-Plattform zu ihrem Namen.

### 1.2.1 Quellcode, Bytecode und Maschinencode

Eben 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 Vier 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, heutzutage immerhin mehrere Milliarden Befehle pro Sekunde (*Instructions Per Second*, IPS). 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 Editor verfasste) Quellcode wird vom **Compiler** in einen maschinen-unabhängigen **Bytecode** übersetzt. Dieser besteht aus den Befehlen einer von der Firma Sun Microsystems definierten **virtuellen Maschine**, die sich durch ihren vergleichsweise einfachen Aufbau gut auf aktuelle Hardware-Architekturen abbilden lässt. Wenngleich der Bytecode von den heute üblichen Prozessoren noch nicht direkt ausgeführt werden kann, hat er doch bereits die meisten Verarbeitungsschritte auf dem Weg vom Quell- zum Maschinencode durchlaufen. Sein Name geht darauf zurück, dass die Instruktionen der virtuellen Maschine jeweils genau ein Byte (= 8 Bit) lang sind. Weil Bytecode kompakter ist als Maschinencode, eignet er sich gut für die Übertragung via Internet.

Den im kostenlosen **Java Development Kit** (**JDK**) der Firma Sun Microsystems (siehe Abschnitt 2) enthaltenen Compiler **javac.exe** setzen auch viele Java-Entwicklungsumgebungen im Hintergrund ein (z.B. der JCreator). Demgegenüber setzt die im Kurs bevorzugte Entwicklungsumgebung Eclipse auf 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**

Für jede Betriebssystem-Plattform mit Java-Unterstützung muss ein (naturgemäß plattformabhängiger) **Interpreter** erstellt werden, der den Bytecode zur Laufzeit in die jeweilige Maschinensprache übersetzt. Dabei findet auch eine Bytecode-**Verifikation** statt, um potentiell gefährliche Aktionen zu verhindern. Die eben erwähnte Bezeichnung *virtuelle Maschine* (engl.: ***J**ava **V**irtual **M**achine*, JVM) verwendet man auch für die an der Ausführung von Java-Programmen beteiligte Software. Man benötigt also für jede reale Maschine eine vom jeweiligen Wirtsbetriebssystem abhängige JVM, um den Java-Bytecode auszuführen.

Für alle relevanten Betriebssysteme liefert die Firma Sun Microsystems über die folgende Adresse

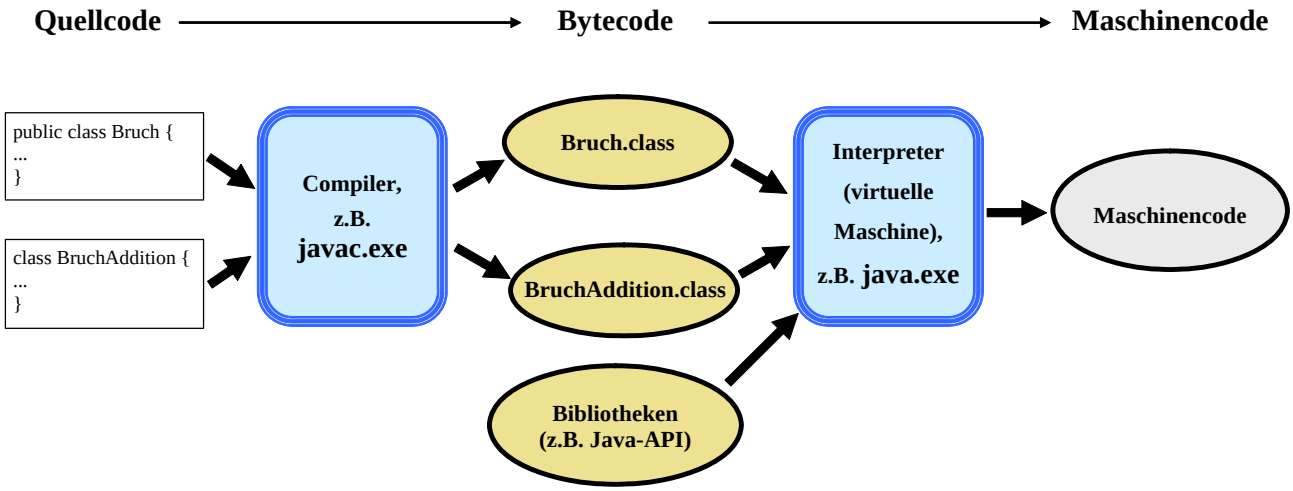
http://java.sun.com/javase/downloads/index.jsp

kostenlos eine **Java Runtime Environment** (**JRE**), die neben dem Bytecode-Interpreter (unter Windows mit dem Namen **java.exe**) auch die Java-Standardbibliothek mit Klassen für alle Routineaufgaben enthält. Diese JRE oder eine äquivalente Software muss auf einem Rechner installiert werden, damit dort Java-Programme ablaufen können.

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 Ma-

schinencode zu erzeugen. So wird wohl keine Verwirrung gestiftet, wenn in diesem Manuskript weiterhin vom *Interpreter* die Rede ist.

In der folgenden Abbildung sind die Dateinamen zum `Bruch`-Beispiel und außerdem die Namen des Compilers **javac.exe** (aus dem JDK) und des Interpreters **java.exe** (aus der JRE) eingetragen:




### 1.2.2 Java als Programmiersprache und als Klassenbibliothek

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 *Java-**API*** (***A**pplication **P**rogram **I**nterface*) bezeichnet wird. Im Abschnitt über Pakete werden die wichtigsten API-Bestandteile grob skizziert. An eine systematische Behandlung ist wegen des enormen Umfangs nicht zu denken.

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. 2005), 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.

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 Microsystems entschied, die noch an vielen signifikanten Stellen (z.B. im Namen der Datei mit dem JDK) präsente Versionsnummer 1.5 durch die „fortschrittliche“ Nummer 5.0 zu ergänzen. Derzeit ist bei der Java Standard Edition (JSE) die Version 6.0 bzw. 1.6.0 im Einsatz. Für die Java Enterprise Edition (JEE) wird eine eigenständige Versionierung verwendet, die sich aber aktuell ebenfalls auf dem Stand 6.0 befindet. Meist besitzt die JSE eine etwas höhere Versionsnummer, was auch bald (im Frühjahr 2010) mit dem Erscheinen der JSE-Version 7.0 wieder der Fall sein wird.

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 steht natürlich 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, Multimedianwendungen, Applets).

### 1.2.3 Herkunft und zentrale Merkmale der Java-Technologie

Die Programmiersprache Java wurde ab 1990 von einem Team der Firma Sun Microsystems unter Leitung von James Gosling entwickelt (siehe z.B. Gosling et al. 2005). Nachdem erste Pläne zum Einsatz in Geräten aus dem Bereich der Unterhaltungselektronik (z.B. Set-Top-Boxen) 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 -Applikationen frei verfügbar.

Mittlerweile hat sich Java als moderne, objektorientierte und für vielfältige Zwecke einsetzbare Programmiersprache etabliert, die als de-facto – Standard für die plattformunabhängige Entwicklung gelten kann und wohl von allen objektorientierten Programmiersprachen den größten Verbreitungsgrad besitzt.

Weil die Java-Plattform so mächtig und vielgestaltig geworden ist, hat die Firma Sun Microsystems drei Editionen für spezielle Einsatzfelder definiert:

- **Java Standard Edition (JSE)** zur Entwicklung von Anwendersoftware
  Darauf wird sich unser Kurs im Wintersemester 2009/2010 beschränken.
- **Java Enterprise Edition (JEE)** für unternehmensweite oder serverorientierte Lösungen
  Nach derzeitiger Planung findet im Sommersemester 2010 ein Fortsetzungskurs statt, der sich auch mit der JEE beschäftigt.
- **Java Micro Edition (JME)** für Kommunikationsgeräte mit beschränkter Leistung (z.B. Mobiltelefone)

Die Java-Entwickler haben sich stark an der Programmiersprache **C++** orientiert, so dass sich Umsteiger von dieser sowohl im Windows- als auch im Linux/UNIX - Bereich weit verbreiteten Sprache schnell in Java einarbeiten können. Wesentliche Ziele bei der Weiterentwicklung waren Einfachheit, Robustheit, Sicherheit und Portabilität. Auf den Aufwand einer systematischen Einordnung von Java im Ensemble der verschiedenen Programmiersprachen bzw. Softwaretechnologien wird hier verzichtet (siehe z.B. RRZN 1999, 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 Teilnahme am Kurs:

#### Objektorientierung

Java wurde als objektorientierte Sprache konzipiert und erlaubt im Unterschied zu hybriden Sprachen wie C++ und Delphi außerhalb von Klassendefinitionen praktisch 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 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. Deren Interaktion miteinander und mit dem Anwender sorgt für den Programmablauf.

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. Im Abschnitt 3 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.

**Portabilität**

Die in Abschnitt 1.2.1 beschriebene Übersetzungsprozedur führt zusammen mit der Tatsache, dass sich Bytecode-Interpreter für aktuelle Rechner-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 verfügbare Standardbibliothek mit ihren insgesamt ca. 3000 Klassen weit reichende Fähigkeiten für die Gestaltung graphischer Benutzerschnittstellen, für Datenbank- und Netzwerkzugriffe usw., so dass sich plattformunabhängige Anwendungen mit modernem Funktionsumfang und Design realisieren lassen.

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

**Sicherheit**

Auch der Sicherheit dient die oben beschriebene Übersetzungsprozedur, weil ein als Bytecode übergebenes Programm durch den beim Empfänger installierten Interpreter vor der Ausführung recht effektiv auf unerwünschte Aktivitäten geprüft werden kann.

**Robustheit**

Zur Robustheit von Java trägt u.a. der Verzicht auf Merkmale von C++ bei, die erfahrungsgemäß zu Fehlern verleiten, z.B.:

- Pointerarithmetik
- Überladen von Operatoren
- Mehrfachvererbung

Außerdem wird der Programmierer zu einer systematischen Behandlung der bei einem Methodenaufruf potentiell zu erwartenden Ausnahmefehler gezwungen. Von den sonstigen Maßnahmen zur

---

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

Förderung der Stabilität ist vor allem noch die Feldgrenzenüberwachung bei Arrays (siehe unten) zu erwähnen.

**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 und Präprozessor-Anweisungen benötigt.

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++ deutlich einfacher zu beherrschen und damit für Einsteiger eher zu empfehlen.

Mittlerweile gibt es etliche **Java-Entwicklungsumgebungen**, die bei vielen Routineaufgaben (z.B. Gestaltung von Benutzeroberflächen, Datenbankzugriffe, Web-Anwendungen) das Erstellen des Quellcodes erleichtern, so dass Java-Entwickler auf diese Bequemlichkeit nicht verzichten müssen. Wir verwenden im Kurs die Entwicklungsumgebung *Eclipse* mit dem Plugin *Visual Editor* und verfügen daher über ein gutes Werkzeug zur Gestaltung visueller Klassen (mit Auftritt auf dem Bildschirm). Im Fortsetzungskurs mit dem Schwerpunkt *Java Enterprise Edition* werden wir das Eclipse-Plugin *Web Tools Platform* verwenden. Einen ähnlichen Funktionsumfang wie Eclipse bieten auch andere Entwicklungsumgebungen, die (wie z.B. Suns's *NetBeans* meist kostenlos verfügbar sind.

**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. Durch die zunehmende Verbreitung von Mehrkern- bzw. Mehrprozessor-Systemen wird für Programmierer die Beherrschung der Multithreaded-Architektur immer wichtiger.

**Verteilte Anwendungen**

Java ist besonders kommunikationsfreudig. Neben den zum Herunterladen via Internet bestimmten Applets gibt es weitere Möglichkeiten, verteilte Anwendungen auf Basis des TCP/IP – Protokolls zu realisieren. Die Kommunikation kann über Sockets oder über höhere Protokolle wie z.B. **RMI** (*Remote Method Invocation*) oder **SOAP** (*Simple Object Access Protocol*) laufen. Selbstverständlich unterstützt Java moderne Internet-Techniken wie **Webservices** und **AJAX**. Wer die genannten Begriffe noch nicht (alle) kennt, muss keinesfalls besorgt sein. Wenn ein Begriff später im Kurs relevant wird, folgt eine Erklärung.

**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 (z.B. Spiele) geht. Mit unserer Entwick-

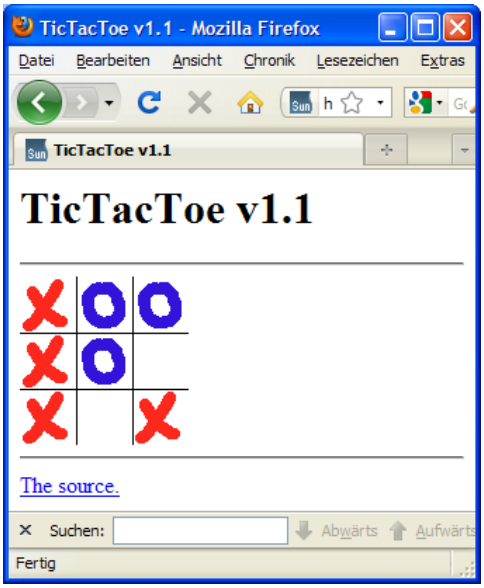
lungsumgebung Eclipse werden Sie eine komplett in Java erstellte, recht komplexe und dabei flott agierende Anwendung kennen lernen.

### 1.2.4 Ausblick auf die Erstellung von Java-Applets

Wir werden uns die Grundlagen der objektorientierten Java-Programmierung im Rahmen von eigenständigen Java-*Anwendungen* erarbeiten. Diese unterscheiden sich in einigen (z.B. sicherheitsrelevanten) Merkmalen von den Java-*Applets*, die durch einen WWW-Browser von einem Server geholt und dann von der JVM innerhalb des Browser-Fensters ausgeführt werden. Wenn Sie möglichst schnell ein attraktives Java-Applet zur Aufwertung einer WWW-Seite schreiben möchten, müssen Sie also in der ersten Phase dieses Kurses eine Durststrecke überstehen. Da hilft vielleicht ein Blick auf die im JDK 6.0 enthaltenen Applet-Demos. Wurde das JDK in den Ordner **C:\Programme\Java\jdk1.6.0_16** installiert, sind die Demos im Ordner

**C:\Programme\Java\jdk1.6.0_16\demo\applets**

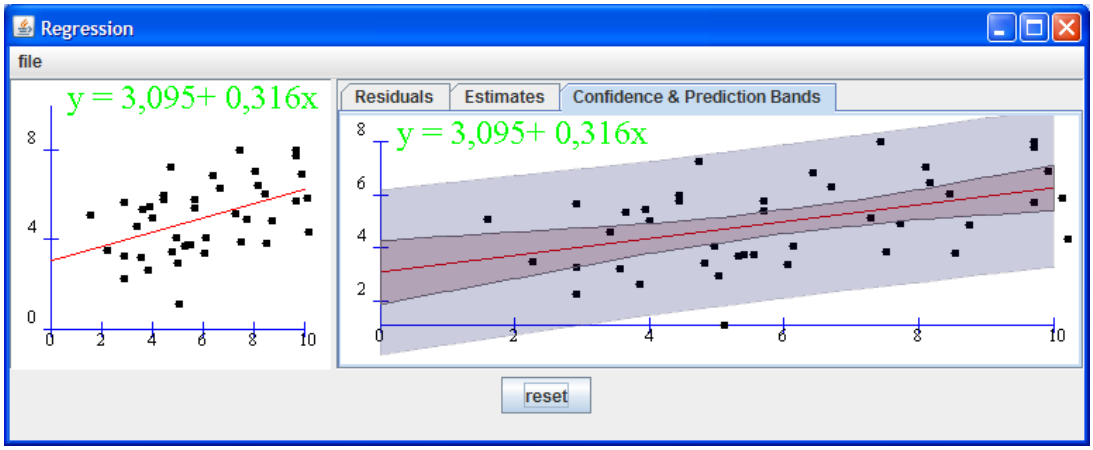
zu finden und per Doppelklick auf die zugehörige HTML-Datei zu starten, z.B.:




Wer sich nicht nur für Java, sondern auch für Wahrscheinlichkeitstheorie und Statistik interessiert, wird auf der folgenden Webseite fündig:

http://www.math.csusb.edu/faculty/stanton/m262/index.html

Hier wird demonstriert, wie mit Java interaktive Webseiten (E-Learning!) erstellt werden können, z.B. im Applet zur linearen Regression:




Dieses Applet startet in einem eigenen Fenster.

## *1.3 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. In der objektorientierten Programmierung dient eine Klasse entweder als Bauplan für Objekte oder kann selbst aktiv werden (Methoden ausführen).
2. Die Programmiersprache Java ist relativ leicht zu erlernen, weil beim Design Einfachheit angestrebt wurde.
3. In Java muss jede Klasse eine Methode namens **main()** enthalten.
4. Die meisten aktuellen CPUs können Java-Bytecode direkt ausführen.
5. Java eignet sich für eine sehr breite Palette von Anwendungen, vom Handy-Spiel über Anwendungsprogramme für Arbeitsplatzrechner bis zur unternehmenskritischen Server-Software.

# 2 Werkzeuge zum Entwickeln von Java-Programmen

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

Anschließend gönnen wir uns aber doch erheblich mehr Luxus in Form der kostenlos verfügbaren Entwicklungsumgebung **Eclipse**, die neben einem guten Editor (z.B. mit Syntaxhervorhebung und -vervollständigung) sehr viele Arbeitserleichterungen bietet und sich vielfältig erweitern lässt, z.B. durch einen visuellen Designer zur Gestaltung der Benutzeroberfläche. Eclipse hat unter den zahlreich vorhandenen Java-Entwicklungsumgebungen den größten Verbreitungsgrad gefunden, obwohl es mit NetBeans eine ebenfalls kostenlose und leistungsfähige Alternative gibt.

## *2.1 Das JDK und Eclipse installieren*

Auf der Begleit-DVD zum Kurs finden Sie einige Programm- und Informationspakete, die unter Windows das Programmieren mit Java ermöglichen bzw. erleichtern. Alle Pakete sind auf den unten genannten Web-Seiten kostenlos für alle relevanten Betriebssysteme verfügbar.

Installieren Sie die Pakete in der folgenden Reihenfolge:

### a) JDK 6.0 (alias 1.6.0) Update 16

Das JDK der Firma Sun Microsystems enthält u.a. …

- den Java-Compiler **javac.exe**
- zahlreiche Werkzeuge (z.B. den Dokumentationsgenerator **javadoc.exe** und den Archivgenerator **jar.exe**)
- etliche Demos
- den Quellcode der Klassen im Kern-API
- eine Java Runtime Environment für die interne Verwendung durch die Entwicklungswerkzeuge

Das JDK-Installationsprogramm kann auch eine öffentliche, durch beliebige Java-Programme und -Applets zu verwendende Java Runtime Environment einrichten, sodass man kein separates JRE-Paket benötigt.

Gleich werden wir noch ein separat von Sun Microsystems angebotenes Dokumentationspaket zum JDK und zum Java-API auf den Rechner befördern.

Das später zu installierende Eclipse SDK 3.5 bringt einen eigenen Compiler mit, der kompatibel mit der Sprachdefinition von Java 6.0 ist. Es ist aber trotzdem sinnvoll, das JDK zu installieren, damit die zusätzlichen Entwicklungswerkzeuge und der Quellcode zu den API-Klassen verfügbar sind.

Voraussetzungen:

- Windows 2000 oder aktueller
- ca. 300 - 400 MB Festplattenspeicher (je nach Installationsumfang)

URL zum Herunterladen:

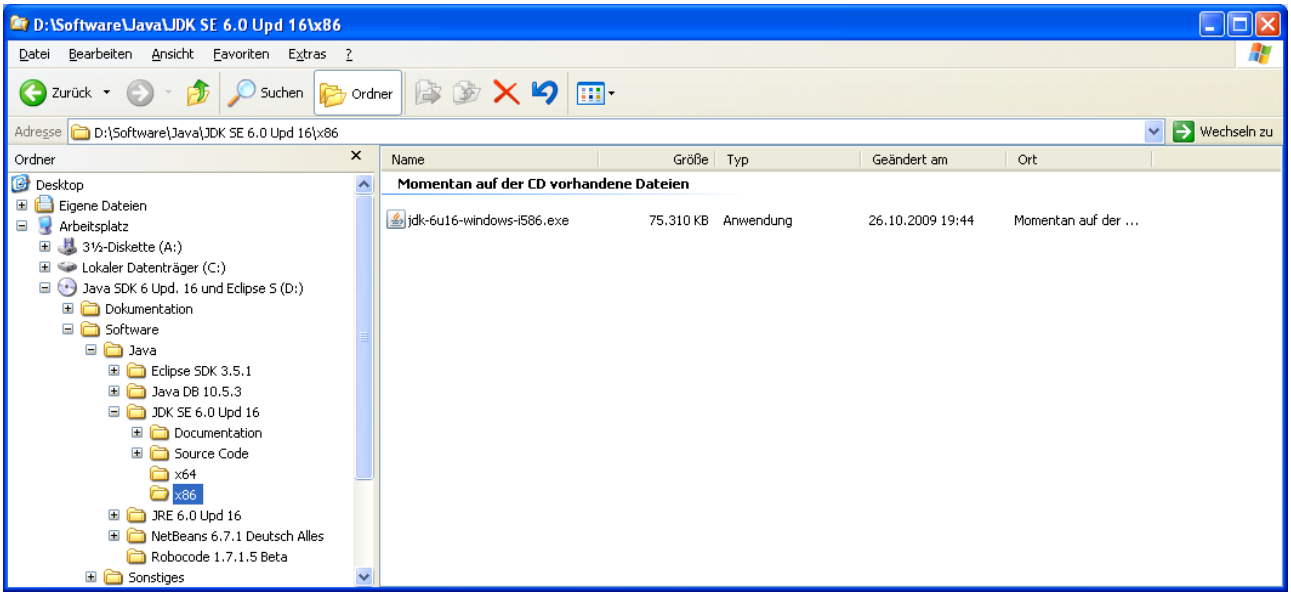
http://java.sun.com/javase/downloads/index.jsp

Auf der Begleit-DVD zum Kurs finden Sie für Windows-Betriebssysteme mit 32- bzw. 64-Bit-Architektur in den folgenden Ordnern die passende Variante:

Win32: **\Software\Java\JDK SE 6.0 Upd 16\x86**

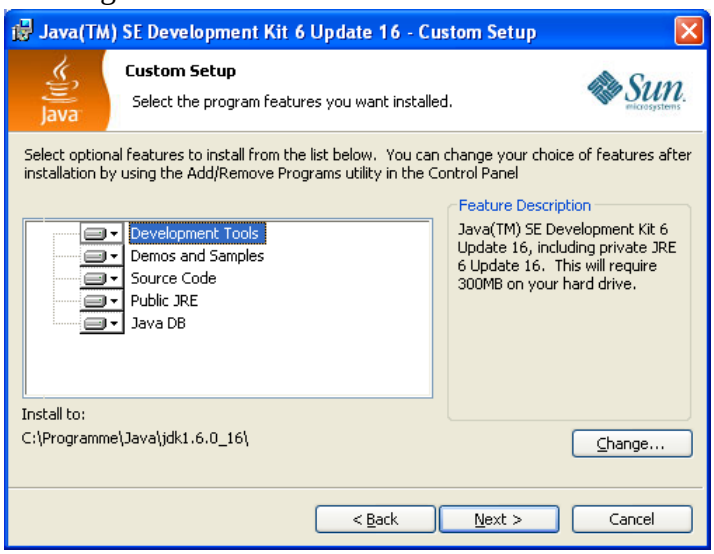
Win64: **\Software\Java\JDK SE 6.0 Upd 16\x64**

Es wird die von Microsoft vorgeschlagene, etwas inkonsistent wirkende Notation verwendet, wobei zur Bezeichnung der 32-Bit-Variante auf die Prozessorarchitektur Bezug genommen wird.




Die mit Administratorrechten durchzuführende Installation wird anschießend exemplarisch für Windows XP Professional (32 Bit) beschrieben:

- Doppelklick auf **jdk-6u16-windows-i586-p.exe**
- Lizenz akzeptieren
- Im Dialog **Custom Setup** wollen Sie eventuell über den Schalter **Change** einen alternativen Installationsordner wählen (statt **C:\Programme\Java\jdk1.6.0_16**). Beim voreingestellten Installationsumfang




landet unter Win32 auch die in Java 6 erstmals enthaltene Datenbank-Software **JavaDB** auf der Festplatte (in der Version 10.5.3). Diese auch unter dem Namen *Apache Derby* bekannte Software kommt erst im Fortsetzungskurs (Sommersemester 2010) zum Einsatz, muss also jetzt nicht unbedingt installiert werden. Das Programm ist auch separat bei Sun Microsystems erhältlich und samt umfangreicher Dokumentation auf der Kurs-DVD vorhanden.

Außerdem wird angeboten, eine *öffentliche* JRE mit einer Komplettausstattung an Sprachen, Schriftarten, Medien und Browser-Plugins zu installieren (per Voreinstellung in den Ordner **C:\Programme\Java\jre6**). Während die im JDK enthaltene JRE normalerweise nur für interne Zwecke verwendet wird, ist die separate öffentliche JRE für *alle* Java-Anwendungen und -Applets sichtbar. In der Regel ist es sinnvoll, die öffentliche JRE (zusätzlicher Festplattenspeicherbedarf ca. 100 MB) zu installieren.

- Alle weiteren Vorschläge können in der Regel übernommen werden.

Erfolgsmeldung nach Abschluss aller Installationsarbeiten:


Später erfahren Sie noch, …

- wie man den API-Quellcode und die API-Dokumentation aus ZIP-Dateien in den JDK-Installationsordner befördert,
- wie man den **bin**-Unterordner der JDK-Installation in die Definition der Umgebungsvariablen PATH einträgt, damit die JDK-Werkzeuge (samt Compiler **javac.exe**) bequem aufgerufen werden können.

### b) 7-Zip 4.65

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

Pfad auf der Begleit-DVD:

Win32: **\Software\Sonstiges\7-ZIP 4.65\x86**

Win64: **\Software\Sonstiges\7-ZIP 4.65\x64**

URL zum Herunterladen:

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

Installation unter Windows XP (32 Bit):

- Doppelklick auf **7z465.exe**
- Ggf. den Zielordner ändern

7-Zip steht ohne Neustart samt Integration in den Windows-Explorer sofort zur Verfügung.

### c) Dokumentation zum JDK 6.0 (alias 1.6.0)

Diese systematische Dokumentation zu den JDK-Werkzeugen und zu allen Klassen im Java-API ist bei der Erstellung von Java-Software unverzichtbar. 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 kann, wenn eine schnelle und zuverlässige Internet-Anbindung verfügbar ist.

Auf der Festplatte belegt die Dokumentation im ausgepackten Zustand (siehe unten) ca. 300 MB.

Pfad auf der Begleit-CD:

**\Software\Java\JDK SE 6.0 Upd 16\Documentation**

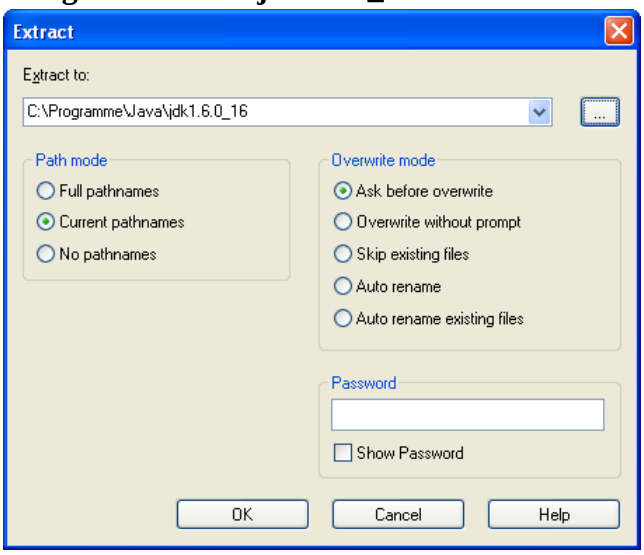
URL zum Herunterladen:

http://java.sun.com/javase/downloads/index.jsp

Installation:

Extrahieren Sie das Archiv **jdk-6u10-docs.zip** in den JDK-Installationsordner, so dass z.B. der Ordner **C:\Programme\Java\jdk1.6.0_16\docs** entsteht.
Vorgehen mit 7-Zip:

- Kontextmenü zur ZIP-Datei öffnen
- **7-Zip > Extract files**
- Ziel wählen, z.B. **C:\Programme\Java\jdk1.6.0_16**




Weil wir gelegentlich einen neugierigen Blick auf den Quellcode zu API-Klassen werfen, sollten Sie die zum JDK gehörige Quellcodedatei **src.zip** analog zur API-Dokumentation behandeln und in den JDK-Unterordner **src** auspacken.

### d) Entwicklungsumgebung Eclipse SDK 3.5

Zur aktuellen Eclipse-Version 3.5 (*Galileo*) werden auf der Webseite

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

etliche Pakete angeboten. Wir begnügen uns mit der für Java-Einsteiger angemessenen Standardvariante *Eclipse Classic*, die meist als *Eclipse SDK* bezeichnet wird.[1]

Auf der Begleit-DVD zum Kurs finden Sie (mit identischem Funktionsumfang für x86 und x64) eine Eclipse-Zusammenstellung, die zum Installieren nur auf die Festplatte Ihres Rechners kopiert werden muss und folgende Bestandteile enthält:

- **Eclipse SDK 3.5.1**
- **Visual Editor 1.4.0**
- **Deutsche Sprachpakete (Babel-Projekt, Version 0.7)**

Voraussetzungen:

- Java Runtime Environment ab 1.5 (5.0)
  Als Java-Anwendung benötigt Eclipse zur Ausführung eine JRE. Weil wir bereits das JDK 6.0 (alias 1.6.0) installiert haben, ist auch eine passende JRE vorhanden.

---

[1] Beim Fortsetzungskurs im Sommersemester werden wir das Paket

Eclipse IDE for Java EE Developers

bevorzugen, das leider derzeit im Unterschied zum SDK-Paket noch nicht für Windows-Betriebssysteme mit 64-Bit - Technik verfügbar ist. Jetzt sind Sie sicher irritiert, weil Eclipse oben als Java-Anwendung bezeichnet wurde und daher perfekt portabel sein sollte. Vermutlich ist die mit Eclipse eingeführte GUI-Bibliothek SWT (mit Klassen zur Gestaltung einer graphischen Benutzeroberfläche), dafür verantwortlich, dass die ein oder andere Datei mit Maschinen-Code bei Eclipse beteiligt ist, so dass sich die Versionen für x86 und x64 unterscheiden. Nach meinen Erfahrungen lässt sich die auch 32-Bit-Version von Eclipse 3.5 unter der 64-Bit-Version von Windows 7 problemlos verwenden.

- 512 MB Arbeitsspeicher
- ca. 240 MB Festplattenspeicher (für die Eclipse-Zusammenstellung auf der Begleit-DVD)

Pfad auf der Begleit-DVD:

Win32: **\Software\Java\Eclipse SDK 3.5.1\x86**

Win64: **\Software\Java\Eclipse SDK 3.5.1\x64**

Installation:

- Ordner von der DVD auf die Festplatte kopieren
- Bei Bedarf eine Verknüpfung mit der Datei **eclipse.exe** an passender Stelle anlegen:
  - Dateinamen mit gedrückter rechter Maustaste auf den Desktop ziehen
  - Aus dem PopUp-Menü wählen: **Verknüpfung hier erstellen**

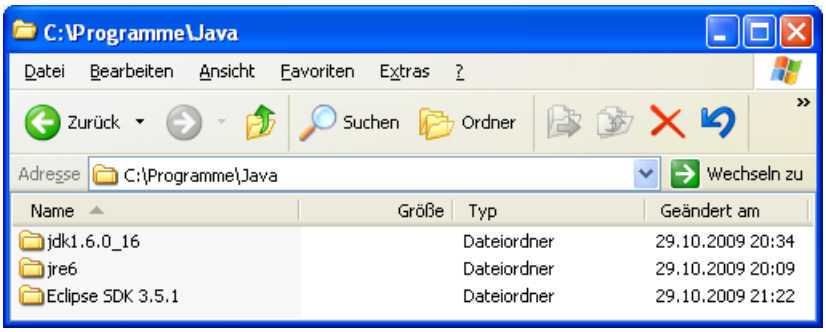
Wenn Sie auf Ihrem Heim-PC Schreibrechte im Programmordner besitzen, wird das Eclipse SDK seine **Konfiguration** dort verwalten, z.B. in

**C:\Programme\Java\Eclipse SDK 3.5.1\configuration**)

Anderenfalls legt das Eclipse SDK einen **configuration**-Ordner im Benutzprofil an, auf Ihrem Heim-PC unter Windows XP z.B. unter

**C:\Dokumente und Einstellungen\<*user*>\.eclipse\ org.eclipse.platform_3.5.0_2032111477**

Nach Abschluss der Installationsarbeiten sollte sich auf der Festplatte Ihres Heim-PCs etwa folgendes Bild einstellen:




Gedulden Sie sich bitte noch einen Moment bis zum ersten Eclipse-Einsatz. Wir beschreiten in Abschnitt 2.2 einmalig einen unbequemen Weg zum Java-Programm, um die in Abschnitt 1.2.1 beschriebenen Strukturen und Abläufe unverstellt beobachten zu können.

## *2.2 Java-Entwicklung mit Texteditor und JDK*

### 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 oben beschriebenen POO-Typ ist (*pseudo*-objektorientiert):

| Quellcode | Ausgabe |
|---|---|
| `class Hallo {`<br>`  public static void main(String[] args) {`<br>`  System.out.println("Hallo Allerseits!");`<br>`  }`<br>`}` | `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**. Die mit dem Starten der Anwendung beauftragte Klasse

`Hallo` erzeugt allerdings 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 betraut 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 ist zwar nicht „vorbildlich“, eignet sich aber aufgrund seiner Kürze zum Erläutern wichtiger Regeln, an die Sie sich so langsam gewöhnen sollten. Alle Themen werden aber später noch einmal systematischer und ausführlicher 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!");
      }
  }
  ```

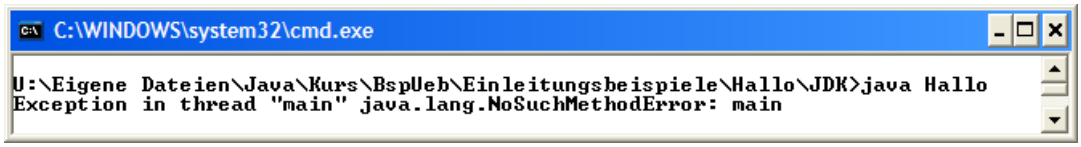
  Bei der Wahl einer Regel für das vorliegende Manuskript habe ich mich am Verhalten der Java-Urheber orientiert: Gosling et at. (2005) lassen bei Startklassen (, die nur von der JRE angesprochen werden,) den Modifikator **public** systematisch weg. Wir werden später klare und unvermeidbare Gründe für die Verwendung des Klassen-Modifikators **public** kennen lernen.
- Dem Kopf der Klassendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf.
- Weil die `Hallo`-Klasse startfähig sein soll, muss sie eine Methode namens **main()** besitzen. Diese wird vom Interpreter beim Programmstart ausgeführt und dient bei 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 vom Interpreter gesucht und ausgeführt. Dazu muss (zumindest ab Java 1.4.x) die Methode **main()** den Zugriffsmodifikator **public** erhalten. Anderenfalls reklamiert der Interpreter beim Starten:

    
    
  - **static**
    Mit diesem Modifikator wird **main()** als **Klassenmethode** gekennzeichnet. Im Unterschied zu den *Instanz*methoden der *Objekte* werden die Klassenmethoden, oft auch als *statische Methoden* bezeichnet, 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 unsere Klasse `BruchAddition` auf Seite 8).

- **void**
  Die Methode **main()** erhält den Typ **void**, weil sie keinen Rückgabewert liefert.

- In der **Parameterliste** einer Methode, die ihrem Namen zwischen runden Klammern folgt, 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 ein Feld 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 der Interpreter ansonsten die Methode beim Programmstart nicht erkennt und sich beim Starten ungefähr so äußert:




    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** (**Editor**) und speichern Sie Ihr Quellprogramm unter dem Namen **Hallo.java** in einem geeigneten Verzeichnis, z.B. in

**U:\Eigene Dateien\Java\Kurs\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*. Bei einer Klasse mit dem Zugriffsmodifikator **public** (siehe unten) besteht der Compiler darauf, dass der Dateinamensstamm mit dem Klassennamen übereinstimmt.
- 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.

### 2.2.2 Kompilieren

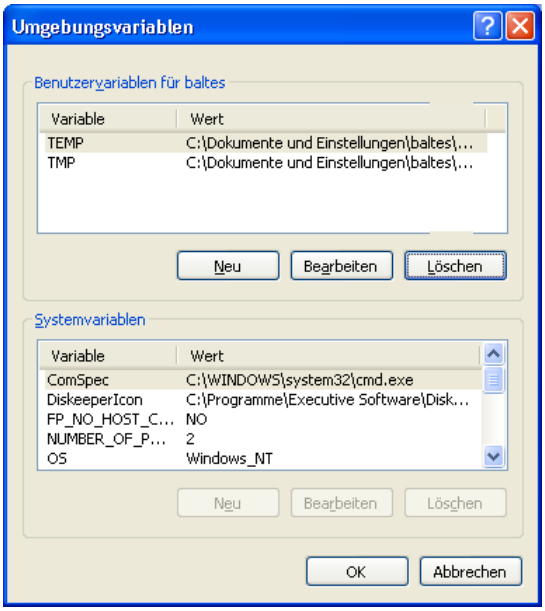
Öffnen Sie ein Konsolenfenster, und wechseln Sie in das Verzeichnis mit dem neu erstellten Quellprogramm **Hallo.java**.

Lassen Sie das Programm vom JDK-Compiler **javac** übersetzen:

```
javac Hallo.java
```

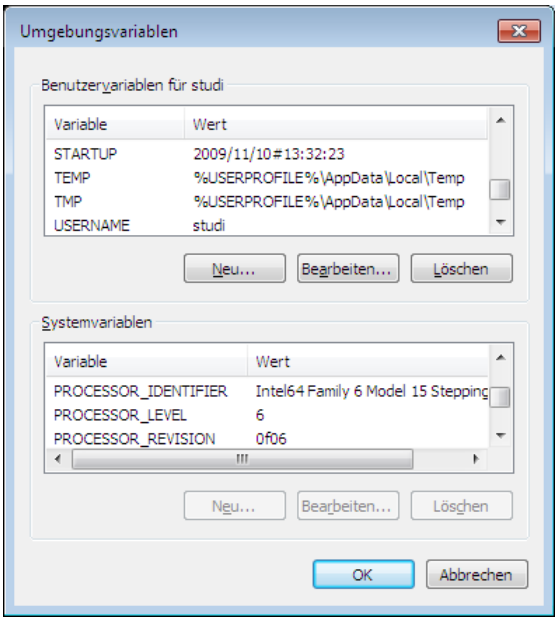
Damit dieser Aufruf von jedem Verzeichnis aus klappt, muss nach der JDK-Installation das **bin**-Unterverzeichnis (mit dem Compiler **javac.exe**) in die Definition der Umgebungsvariablen PATH aufgenommen werden. Dies kann unter Windows XP z.B. so geschehen:

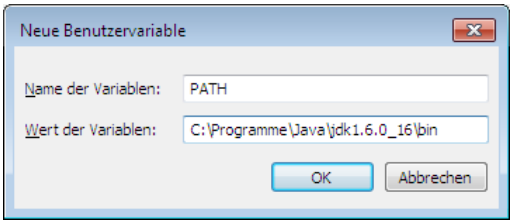
- Setzen Sie im Startmenü einen Rechtsklick auf den Eintrag **Arbeitsplatz**, um sein Kontextmenü zu öffnen, und wählen Sie daraus den Eintrag **Eigenschaften**.
- Betätigen Sie auf der Registerkarte **Erweitert** den Schalter **Umgebungsvariablen**.
- In der Dialogbox **Umgebungsvariablen** können Sie die **Benutzervariable** PATH anlegen oder erweitern. Bei vorhandenen Administratorrechten lässt sich auch die in der Regel vorhandene **Systemvariable** gleichen Namens erweitern, z.B.:




- Hier wird eine neue **Benutzervariable** angelegt:

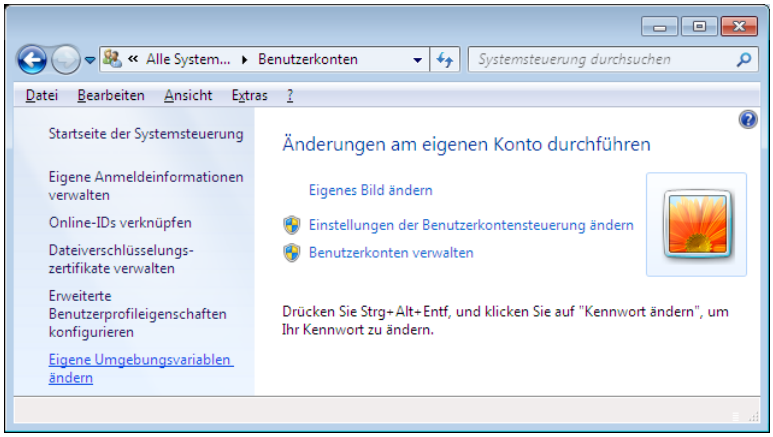

Windows 7 bietet im neuen Design dieselben Dialogboxen:



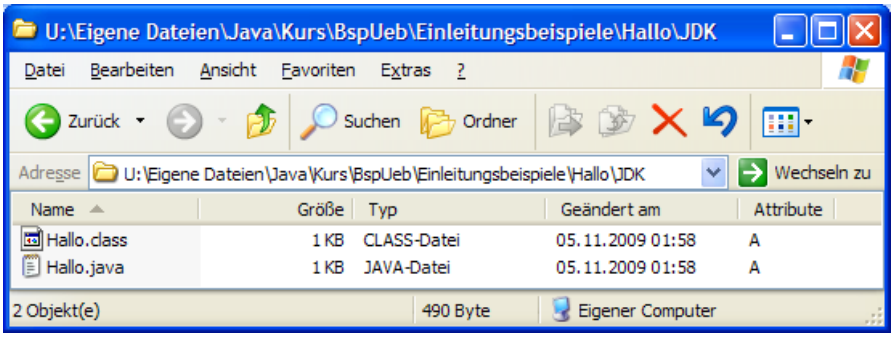

Allerdings sind die vertrauten Dialoge im neuen System nicht ganz leicht zu finden:

- **Start > Systemsteuerung**
- Bei **Anzeige** als **Kategorie** entweder **große** oder **kleine Symbole** wählen
- Erst jetzt die **Benutzerkonten** anklicken, so dass der folgende Dialog erscheint:




- Nach einem Klick auf **Eigene Umgebungsvariablen ändern** erscheint endlich die gesuchte Dialogbox.

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




Beim Kompilieren einer Quellcodedatei werden auch alle darin benutzten fremden Klassen neu übersetzt, falls deren Bytecode-Datei fehlt oder älter als die zugehörige Quellcodedatei ist. Sind

etwa im Bruchrechnungsbeispiel die Quellcodedateien **Bruch.java** *und* **BruchAddition.java** geändert worden, dann genügt folgender Compileraufruf, um beide neu zu übersetzen:

```
javac BruchAddition.java
```

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 natürlich scheitern.

### 2.2.3 Ausführen

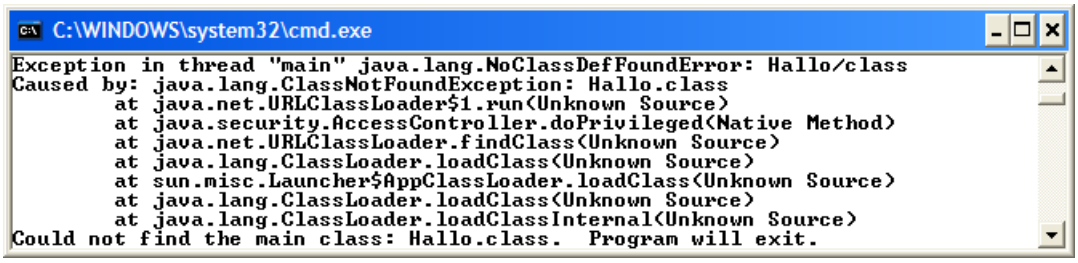
Lassen Sie das Programm (bzw. die Klasse) **Hallo.class** von der JVM ausführen. Der Aufruf

```
java Hallo
```

sollte zum folgenden Ergebnis führen:


Beim Programmstart ist zu beachten:

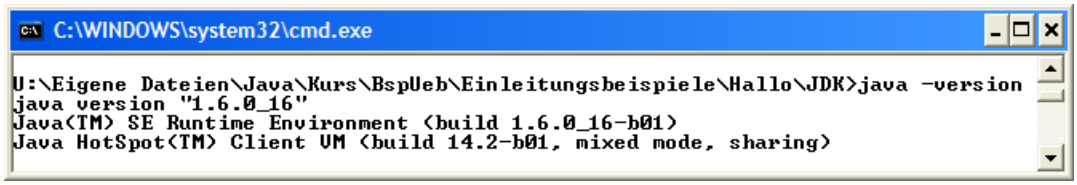
- Die Namenserweiterung **.class** wird **nicht** angegeben. Wer es doch tut, erhält keinen Fleißpunkt, sondern eine Fehlermeldung:




- Beim Aufruf des Interpreters wird der Name der auszuführenden Klasse als Argument angegeben. Weil es sich dabei um einen Java-Bezeichner handelt, muss auch 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.
- Die Version des installierten Interpreters kann mit dem Kommando

  ```
  java -version
  ```

  ermittelt werden. In diesem Beispiel



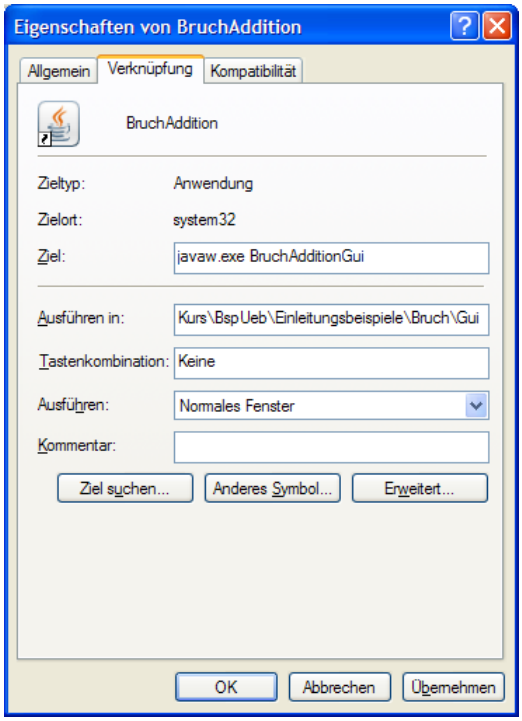

  ist zu erfahren, dass Suns **HotSpot - Client VM** im Einsatz ist. Sie stellt sich beim Übersetzen des Java-Bytecodes in Maschinencode besonders geschickt an (Analyse des Laufzeitverhaltens) und ist älteren JVMs, die mit einem **JIT**-Compiler (***Just-in-Time***) arbeiten, deutlich überlegen.

Zum *Ausführen* eines Java-Programms wird *nicht* das vollständige JDK (mit Entwicklerwerkzeugen, Dokumentation etc.) benötigt, sondern lediglich die **Java Runtime Environment** (JRE) mit dem Interpreter **java.exe** und der Standardbibliothek. Dieses Produkt ist bei Sun Microsystems auch separat über folgende Adresse erhältlich:

http://java.sun.com/javase/downloads/index.jsp

Man darf die JRE zusammen mit eigenen Programmen weiter gegeben, um diese auf beliebigen Rechnern lauffähig zu machen. Das JDK enthält auch eine Laufzeitumgebung, so dass diese nicht unbedingt zusätzlich installiert werden muss (vgl. Abschnitt 2.1).

Das beim Einsatz von **java.exe** unvermeidliche Konsolenfenster kann beim Starten einer Java-Anwendung mit graphischer Benutzeroberfläche störend wirken. Unter Windows bietet sich als Alternative der Starter **javaw.exe** an, der auf ein Konsolenfenster verzichtet. Über eine Verknüpfung mit den folgenden Eigenschaften

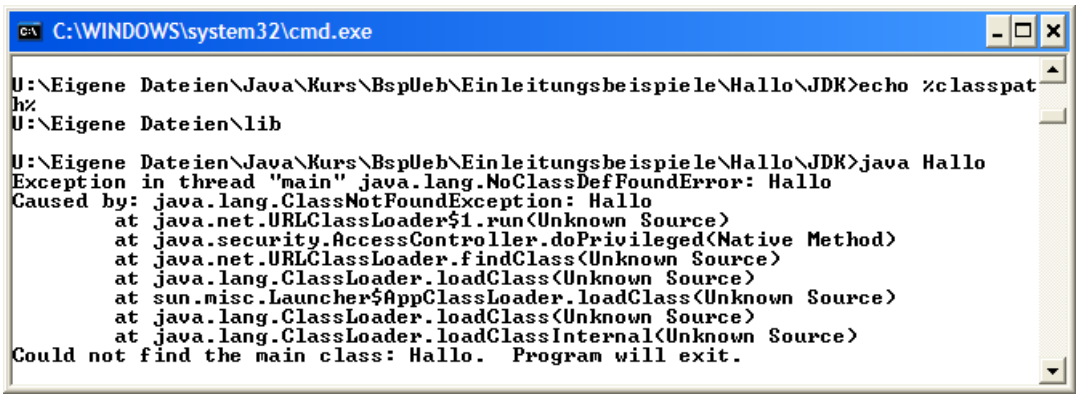



kann das in Abschnitt 1.1 vorgestellte graphische Bruchrechnungsprogramm per Doppelklick gestartet werden (ohne Auftritt eines Konsolenfensters).

### 2.2.4 Pfad für class-Dateien setzen

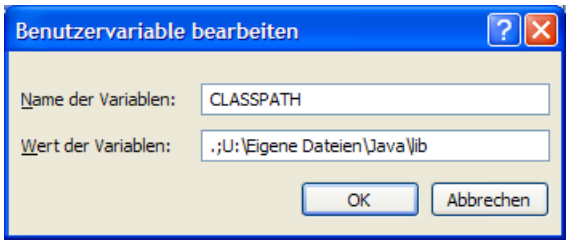
Compiler und Interpreter benötigen Zugriff auf die Bytecode-Dateien zu allen Klassen, die im zu übersetzenden Quellcode bzw. im auszuführenden Programm angesprochen werden. Mit Hilfe der Umgebungsvariablen CLASSPATH kann man eine Liste von Pfaden, JAR-Archiven (siehe Abschnitt 7.4) oder ZIP-Archiven spezifizieren, die nach **class**-Dateien durchsucht werden sollen, z.B.:


Befinden sich alle benötigten Klassen entweder in der Java-API-Bibliothek (siehe Abschnitt 7.5) 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 API-Bibliothek nur die angegebenen Pfade berücksichtigt. Dies führt zu Problemen bei CLASSPATH-Variablen, die das aktuelle Verzeichnis *nicht* enthalten, z.B.:




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

Weil in vielen konsolenorientierten Beispielprogrammen des Kurses die nicht zum Java-API gehörige Klasse **Simput.class** (siehe unten) zum Einsatz kommt, sollte die Umgebungsvariable CLASSPATH so gesetzt werden, dass der Compiler und der Interpreter die Klasse **Simput.class** finden. Wie man eine benutzereigene Umgebungsvariable mit den Bordmitteln von Windows XP anlegt, wurde schon im Abschnitt 2.2.2 beschrieben, z.B.:




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

Unsere Entwicklungsumgebung Eclipse ignoriert die CLASSPATH-Umgebungsvariable, bietet aber eine alternative Möglichkeit zur Definition eines Klassenpfads (siehe unten).

Wenn sich nicht alle 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 für Compiler und Interpreter über die **classpath**-Option (abzukürzen durch **cp**) angegeben werden, z.B.:

```
javac -cp ".;U:\Eigene Dateien\java\lib" Bruch.java
java -cp ".;U:\EigeneDateien\java\lib" BruchAddition
```

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

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

Mit dem Verwenden der **cp**-Kommandozeilenoption wird eine eventuell vorhandene CLASSPATH-Umgebungsvariable für den gestarteten Compiler- oder Interpreterlauf desaktiviert.

### 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.

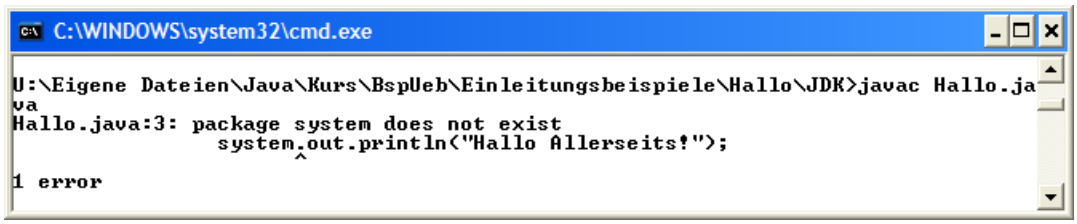
- **Semantikfehler**
  Hier liegt kein Syntaxfehler vor, aber das Programm verhält sich anders als erwartet, wiederholt z.B. ständig eine nutzlose Aktion („Endlosschleife“).

Die Java-Entwickler haben dafür gesorgt (z.B. durch strenge Typisierung, Beschränkung der impliziten Typanpassung, Zwang zur Behandlung von Ausnahmen), dass möglichst viele Fehler vom Compiler aufgedeckt werden können.

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.

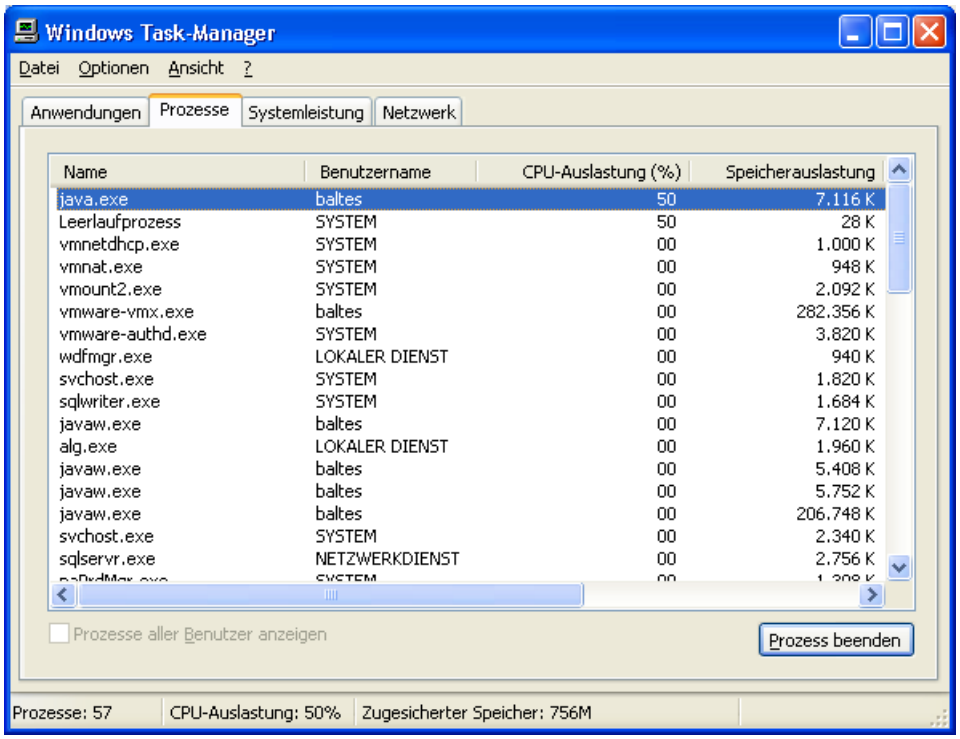
Manchmal wird dem Compiler aber erst in einiger Distanz zur Schadstelle klar, dass ein Regelverstoß vorliegt, so dass statt der kritisierten Stelle eine frühere Passage zu korrigieren ist.

Im Beispiel fällt auch 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 *Semantikfehler* einbauen lässt, betrachten wir das Bruchrechnungsprogramm aus Abschnitt 1.1. 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;
}
```

Wenn ein „Bruch“ aufgrund der untauglichen Absicherung den verbotenen Nennerwert Null erhält und anschließend zum Kürzen aufgefordert wird, zeigt das Programm ein unerwünschtes Verhalten. Es gerät in eine Endlosschleife (siehe unten) und verbraucht dabei reichlich Rechenzeit, wie der Windows-Taskmanager (auf einem PC mit Hyper-Threading-CPU, also mit zwei logischen Kernen) belegt:



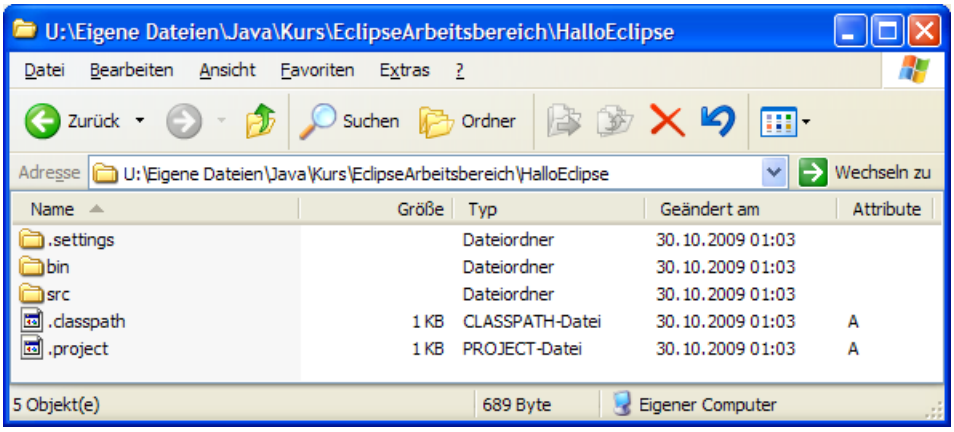

Ein derart außer Kontrolle geratenes Konsolenprogramm kann man unter Windows z.B. mit der Tastenkombination **Strg+C** beenden.

## *2.3 Java-Entwicklung mit Eclipse*

Zu Eclipse sind umfangreiche Bücher entstanden mit einer Beschreibung der zahlreichen Profiwerkzeuge zur Softwareentwicklung. Wir beschränken uns anschließend auf elementare Informationen für Einsteiger, die später nach Bedarf ergänzt werden. Weiterführende Informationen bieten das Eclipse-Hilfesystem und diverse Bücher (z.B. Künneth 2009, RRZN 2005).

### 2.3.1 Arbeitsbereich und Projekte

Eclipse legt für jedes Projekt (z.B. zum Erstellen eines Programms oder einer Bibliothek) einen **Projektordner** an, der Quellcode-, Bytecode-, Konfigurations- und Hilfsdateien (z.B. zur Änderungsverfolgung) aufnimmt, z.B.:




Jede Eclipse-Version ist mit fest mit einem **Arbeitsbereichsordner** verbunden, der beliebig viele Projekte zusammenfasst. In der Regel enthält ein Arbeitsbereich einen recht umfangreichen Konfigurationsordner namens **.metadata** (mit einleitendem Punkt) sowie die Ordner der Projekte. Allerdings können die Projekte auch unabhängig vom Arbeitsbereichsordner aufbewahrt werden, was sich im Kurskontext als sinnvoll erweisen wird.

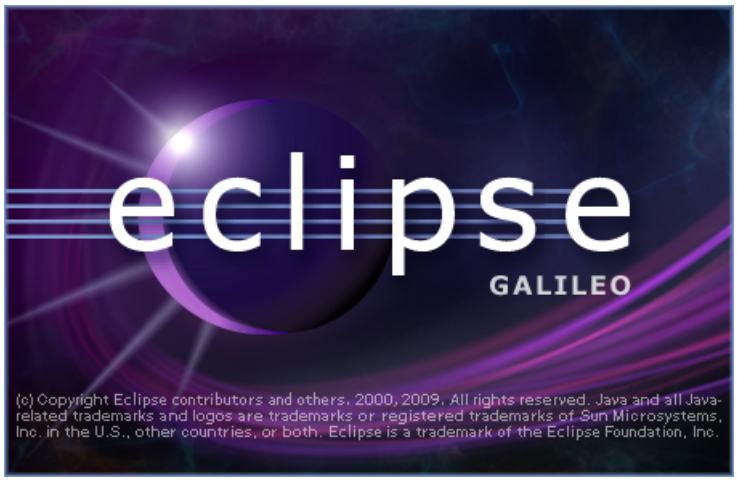
### 2.3.2 Eclipse starten

Auf den Pool-PCs an der Universität Trier können Sie das Eclipse SDK 3.5 folgendermaßen starten:

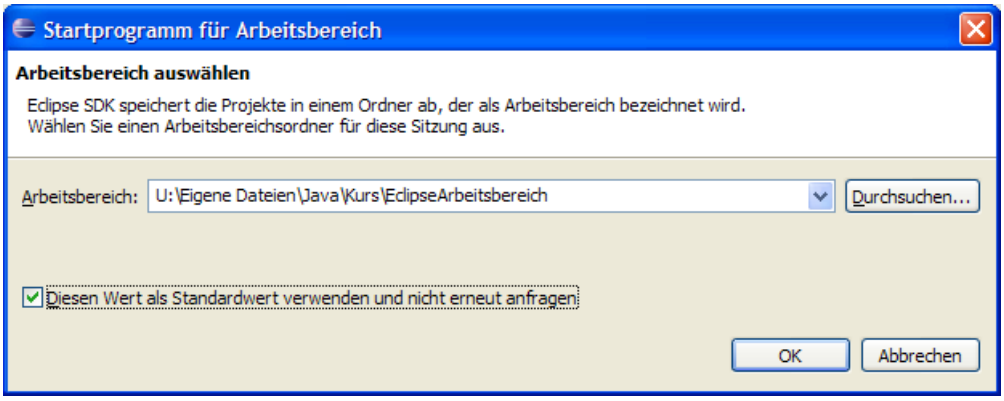
**Start > Alle Programme > Programmentwicklung > Java > Eclipse 3.5> Eclipse SDK 3.5**

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

Eclipse erkundigt sich beim imposant eingeleiteten Start




nach dem gewünschten Arbeitsbereich, wobei Sie an einem Pool-PC der Universität Trier einen Ordner auf dem Laufwerk U: wählen sollten, z.B.:




Nachdem Sie Eclipse veranlasst haben, einen Standardwert ohne Anfrage zu verwenden, können Sie Ihre Festlegung auf folgende Weise modifizieren:

- Für einen spontanen Wechsel steht in Eclipse der Menübefehl

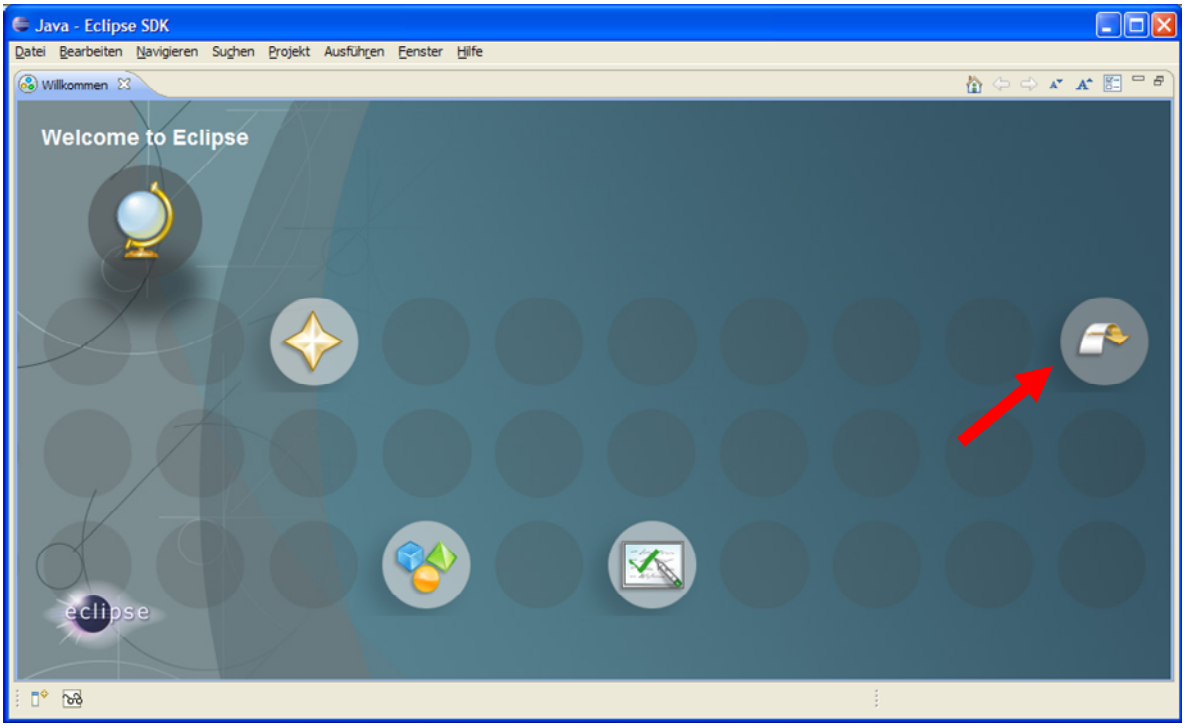
  **Datei > Arbeitsbereich wechseln > Andere**

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

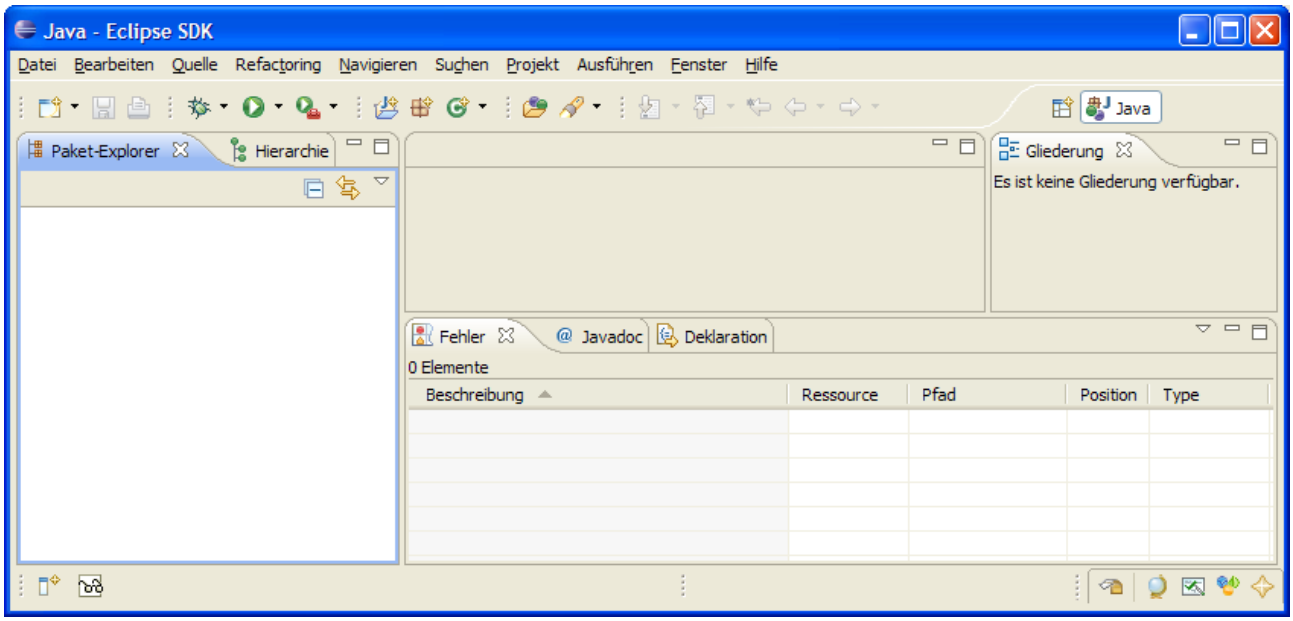
  **Fenster > Benutzervorgaben > Allgemein > Start und Beendigung > Arbeitsbereiche > Arbeitsbereich bei Start anfordern**

  reaktivieren Sie die routinemäßige Arbeitsbereichsanfrage beim Start.

Beim ersten Eclipse-Start werden Sie recht eindrucksvoll begrüßt:




Mit einem Mausklick auf den Pfeil am rechten Rand gelangen Sie zur **Werkbank** (*Workbench*) mit zahlreichen Werkzeugen für eine komfortable und erfolgreiche Softwareentwicklung:

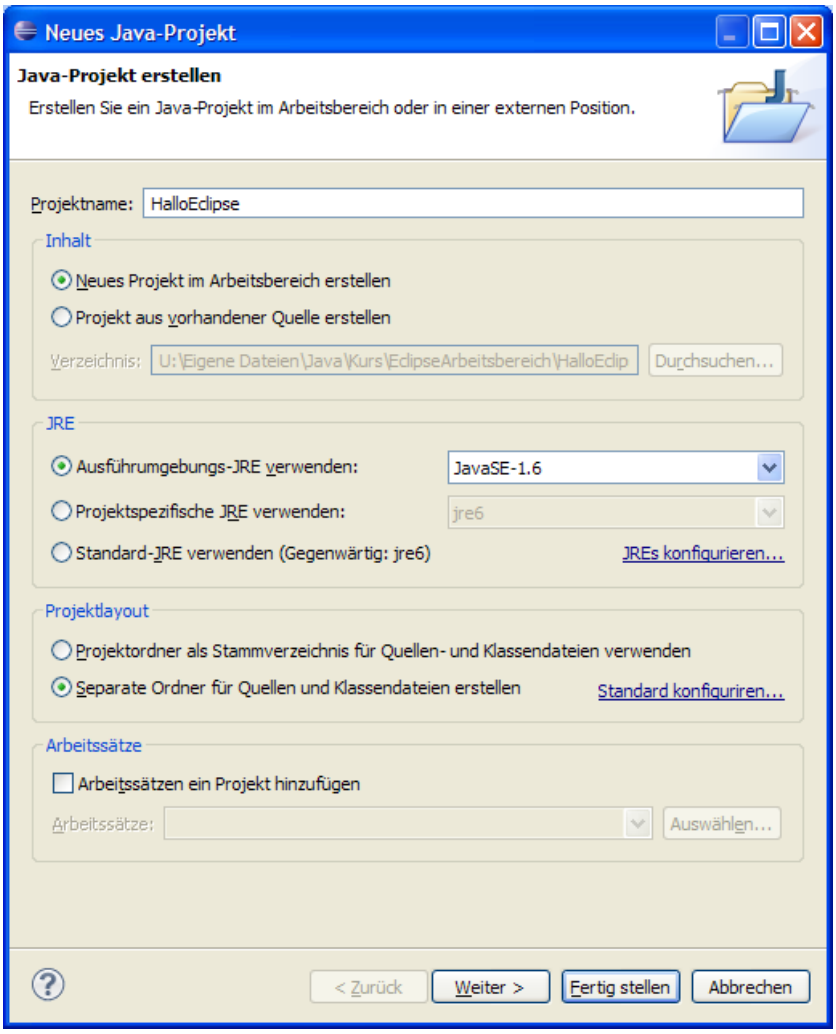



### 2.3.3 Neues Projekt anlegen

Wir starten mit

**Datei > Neu > Java-Projekt**

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




Bei der voreingestellten **Ausführungsumgebungs-JRE** unterstützt Eclipse den Sprachumfang von Java 6.0 (alias 1.6.0), wobei die entstehenden Programme bei der ausführenden JRE auf dem Computern der Anwender mindestens dieses Niveau benötigen. Es sind aber keine ernsthaften Kompatibilitätsprobleme zu erwarten, denn …

- die Anwender können kostenlos die passende JRE-Version installieren,
- Sie können speziell für Ihr Programm (ohne Nebenwirkungen auf andere Java-Programme auf demselben Rechner) eine JRE mitliefern.

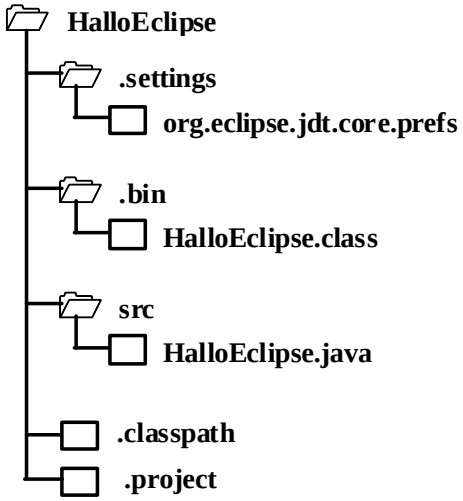
Im Rahmen **Projektlayout** geht es um die Struktur des Projektordners, der im Arbeitsbereichsordner angelegt wird, im Beispiel also in:

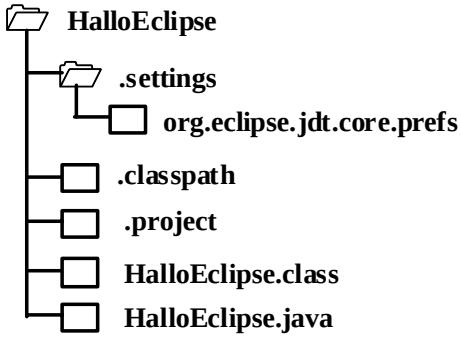
**U:\Eigene Dateien\Java\Kurs\EclipseArbeitsbereich**

Aus der Voreinstellung

**Separate Ordner für Quellen- und Klassendateien erstellen**

resultiert das linke Projektlayout:




Aus der alternativen Option

**Projektordner als Stammverzeichnis für Quellen- und Klassendateien verwenden**

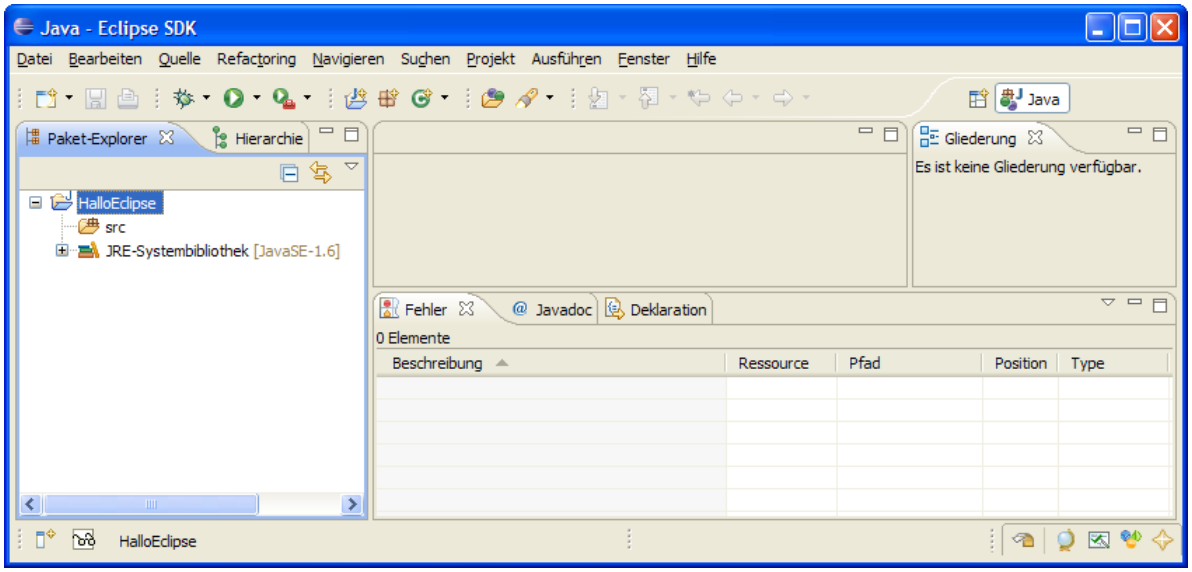
resultiert das rechte Projektlayout. Beim aktuell entstehenden Projekt verwenden wir das voreingestellte Layout mit Unterordnern für die Quellcode- und Klassendateien, das aufgrund der insgesamt sehr geringen Anzahl von Dateien etwas übertrieben zergliedert wird. Bei vielen anderen, ähnlich simplem Beispielprojekten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

Für das Einstiegsprojekt können Sie den Assistenten jetzt mit **Fertigstellen** beenden und damit alle weiteren Dialoge ignorieren. Viele Einstellungen eines Projektes sind später über den Menübefehl

**Projekt > Eigenschaften**

zu ändern.

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



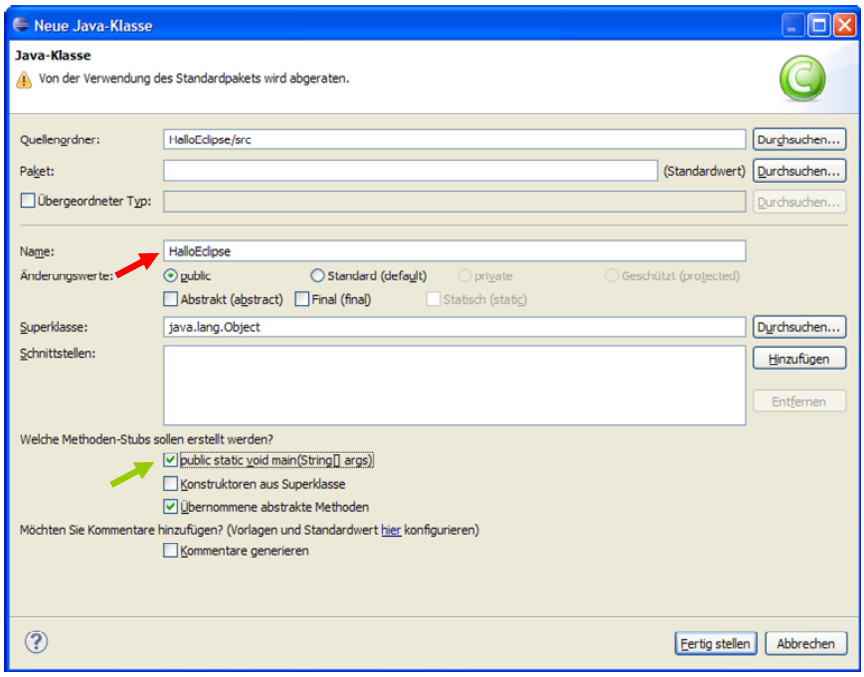

Man kann den Dialog **Neues Java-Projekt** auch über den Schalter  erreichen.

### 2.3.4 Klasse hinzufügen

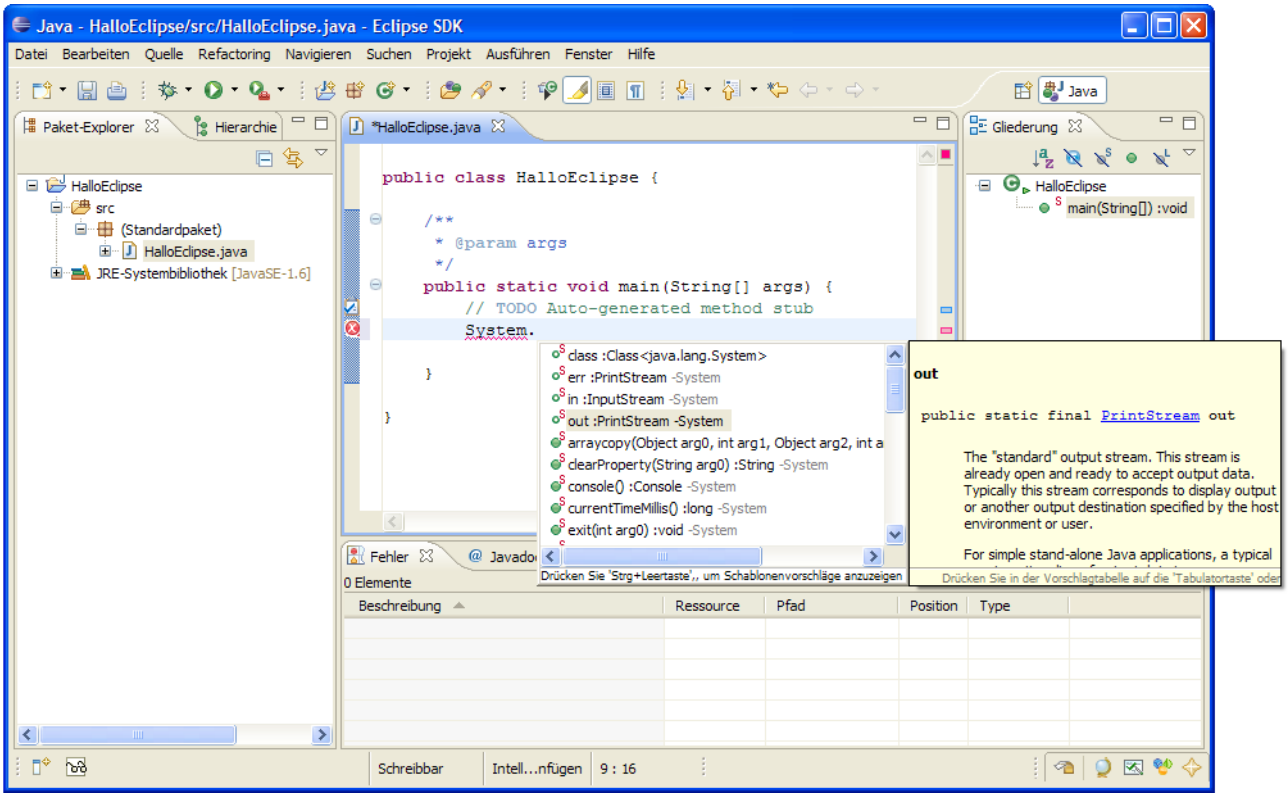
Wir starten über den Menübefehl

**Datei > Neu > Klasse**

oder den Symbolschalter die Definition einer neuen Klasse, legen im folgenden Dialog deren Namen fest (roter Pfeil), lassen einen **main()**-Methoden-Rohling automatisch anlegen (grüner Pfeil) und ignorieren die (bei großen Projekten sehr berechtigte) Kritik an der Verwendung des Standard-pakets:[1]




Nach dem **Fertigstellen** befindet sich auf der Werkbank ein fast komplettes POO - Hallo-Programm, und wir müssen im Editor nur noch die unvermeidliche Ausgabeanweisung verfassen, wobei die Syntaxvervollständigung eine große Hilfe ist:



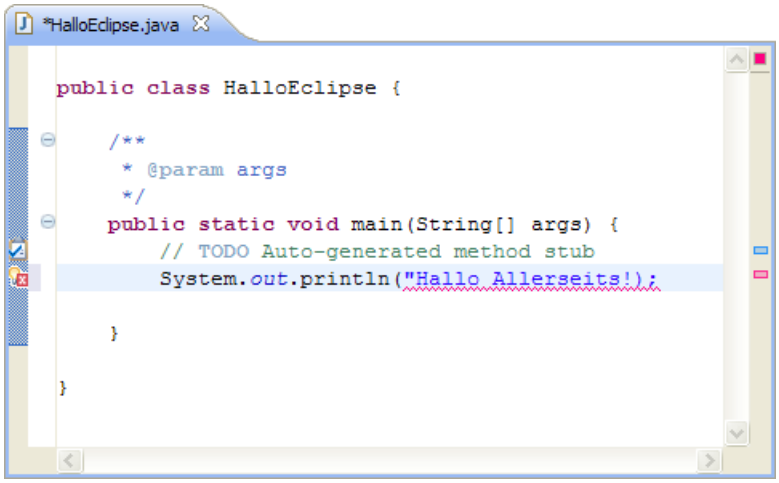

[1] Mit Paketen werden wir uns später ausführlich beschäftigen.

Sobald wir einen Punkt hinter den Klassennamen **System** 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. Nach dem nächsten Punkt werden u.a. die Instanzmethoden der Klasse **PrintStream** aufgelistet, und wir komplettieren die Nachricht an das Objekt **System.out** folgendermaßen:

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

### 2.3.5 Übersetzen und Ausführen

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

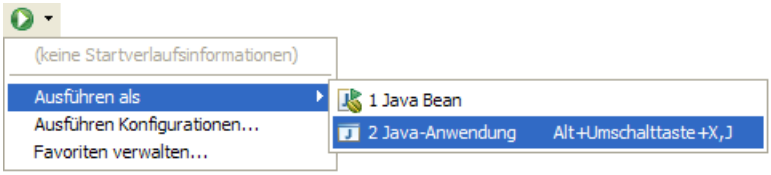



Hier ist ein inkrementeller Compiler am Werk.

Den ersten Start unserer Anwendung veranlassen wir mit

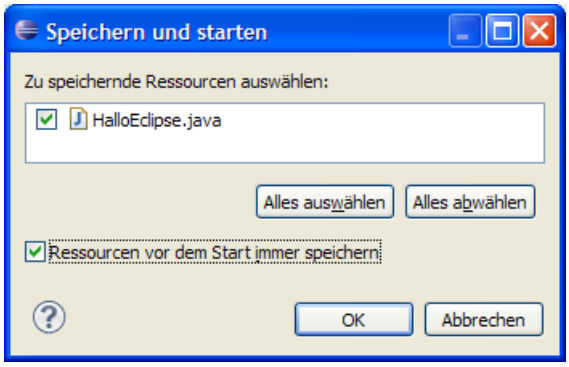
**Ausführen > Ausführen als > Java-Anwendung**

oder äquivalent mit dem Klappmenü zur Schaltfläche :

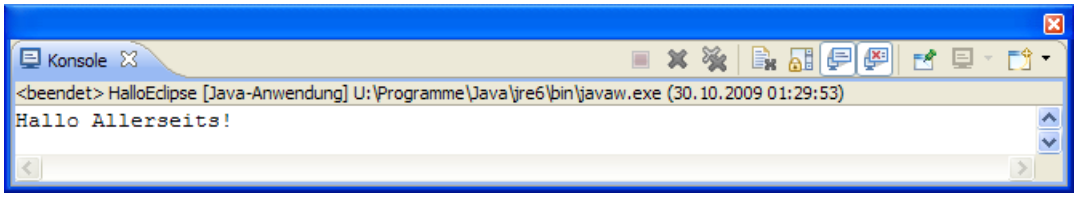



Weil sich Eclipse die letzte Startkonfiguration merkt, genügt später zum Starten desselben Programms ein Mausklick auf den Schalter oder die Tastenkombination **Strg+F11**.

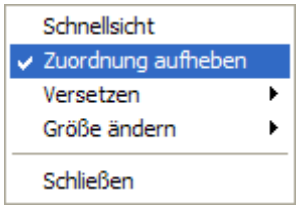
Falls Sie Ihr Projekt noch nicht gespeichert haben, können Sie dies jetzt tun und auch veranlassen, dass in Zukunft geänderter Quellecode grundsätzlich vor dem Programmstart gesichert wird:



Die Ausgabe des Programms erscheint im Konsolenfenster, 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. Alternativ kann man über das Kontextmenü zu Fensterbeschriftung die **Zuordnung** wiederherstellen:




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

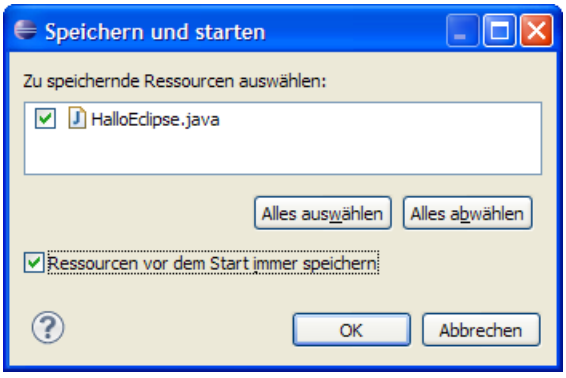
**Fenster > Sicht anzeigen > Konsole**

Nach der Korrektur des Ausgabetexts (z.B. „Hallo Eclipse“) und erneuter Ausführung (z.B. mit **Strg+F11**) wird die Quelle ohne Nachfrage gesichert.

### 2.3.6 Einstellungen ändern

#### *2.3.6.1 Automatische Quellcodesicherung beim Ausführen*

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




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

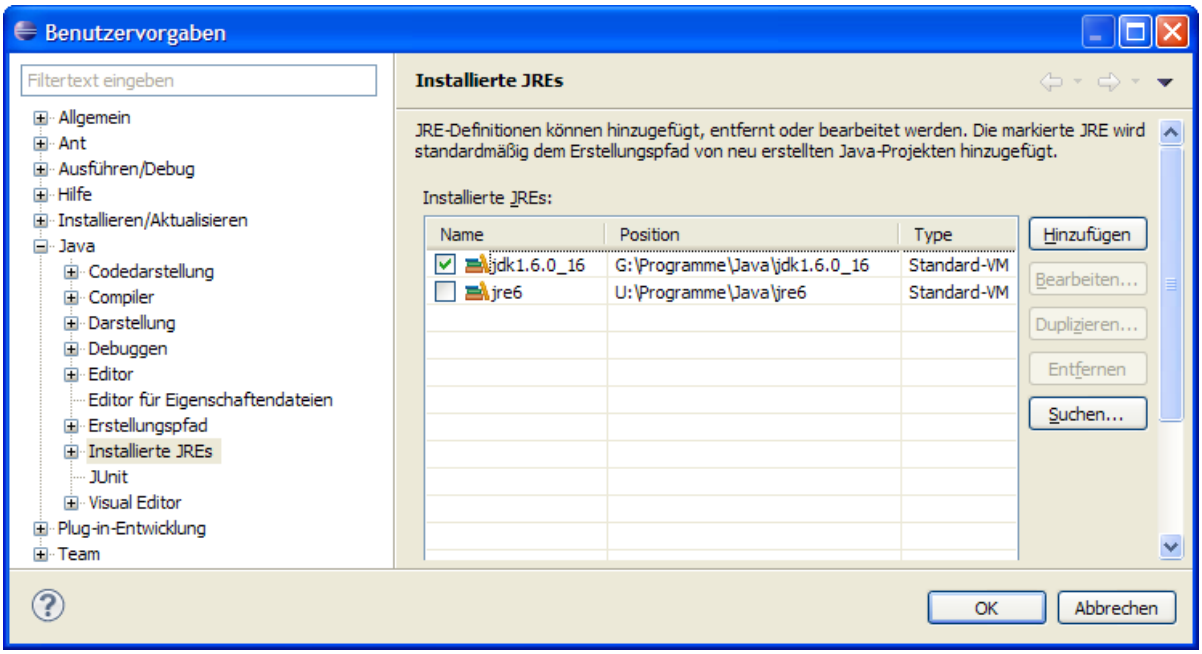
- **Fenster > Benutzervorgaben > Ausführen/Debug > Startvorgang**
- Wählen Sie im Optionsfeld **Erforderliche Befehlseditoren vor dem Starten speichern** den gewünschten Wert:


### *2.3.6.2 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** und festlegen, welche JRE bei neuen Projekten per Voreinstellung verwendet werden soll, z.B.:




Wenn Sie ein JDK als **Standard-VM** angeben, kann bei einem Fehler die Unfallstelle im API-Quellcode lokalisiert werden, z.B.:

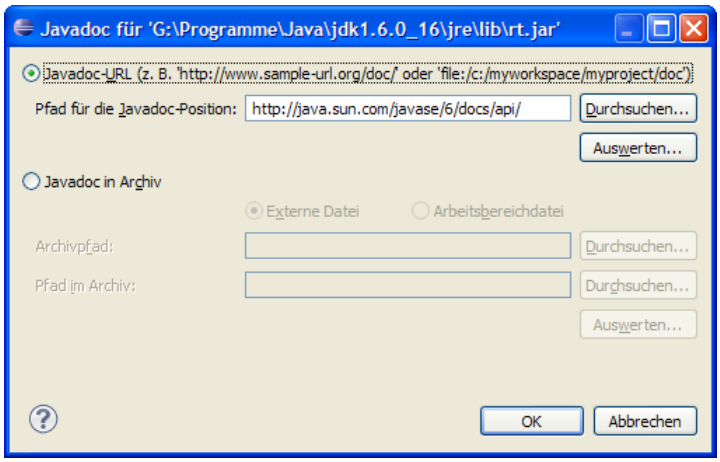
```
java.lang.NumberFormatException: For input string: "vier"
   at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
   at java.lang.Integer.parseInt(Integer.java:449)
   at java.lang.Integer.parseInt(Integer.java:499)
   at Fakul.getFakulArg(Fakul.java:5)
   at Fakul.main(Fakul.java:25)
```

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

```
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.getFakulArg(Fakul.java:5)
   at Fakul.main(Fakul.java:25)
```

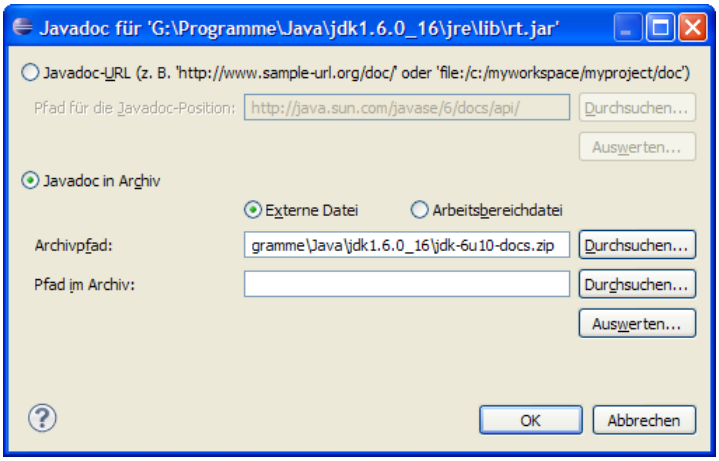
#### *2.3.6.3 Alternative Javadoc-Quelle zu den Klassen der Standardbibliothek wählen*

Per Voreinstellung bezieht Eclipse die Dokumentation zu den in Ihrem Programm verwendeten API-Klassen aus dem Internet:




So lässt sich zu einer JRE-Systembibliothek (d.h. zu einer JAR-Datei mit API-Klassen) eine alternative Dokumentationsquelle eintragen:

- Betroffene installierte JRE (siehe Abschnitt 2.3.6.2) markieren
- Klick auf **Bearbeiten**
- JRE-Systembibliothek wählen
  Es genügt, eine Installationsquelle für die Datei **rt.jar** festzulegen, weil sich die meisten wichtigen Klassen hier befinden.
- Klick auf **Javdoc-Position**
- Quelle wählen, z.B. eine (unausgepackte!) ZIP-Datei:




### 2.3.7 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, so dass die erwünschte Strukturierung der recht umfangreichen Projektsammlung durch mehrere Arbeitsbereiche unökonomisch ist.
- 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 des Bruchadditionsprojekts mit graphischer Benutzeroberfläche, das sich im folgenden Ordner befindet:

**…\BspUeb\Einleitung\Bruch\Gui**

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

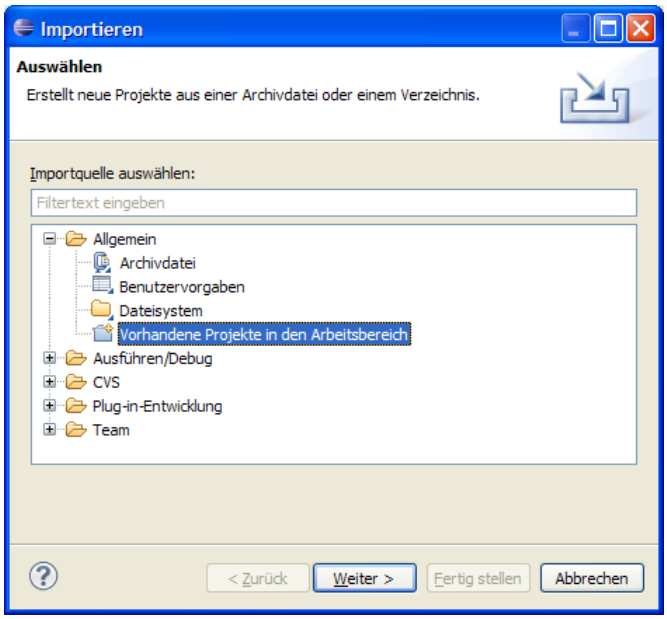
**U:\Eigene Dateien\Java\Kurs\BspUeb\Einleitung\Bruch\Gui**

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

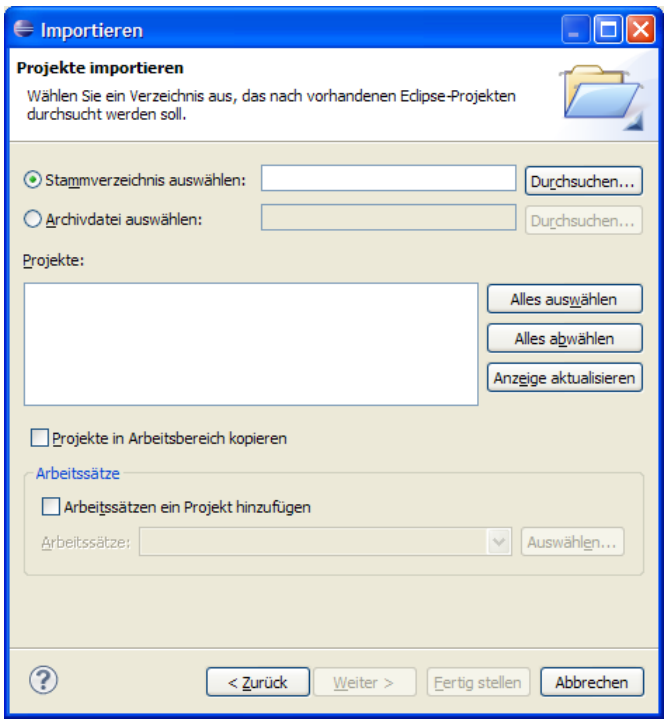
**U:\Eigene Dateien\Java\Kurs\EclipseArbeitsbereich**

Initiieren Sie den Import mit

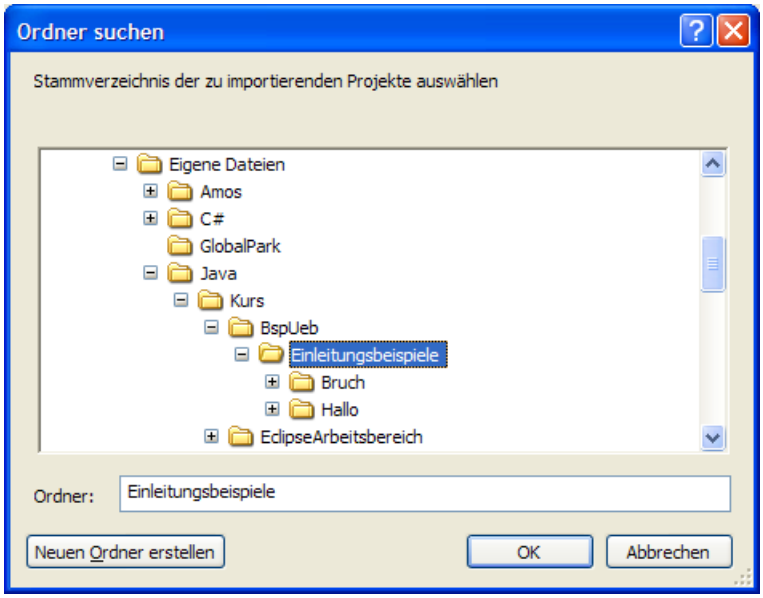
**Datei > Importieren > Allgemein > Vorhandene Projekte in den Arbeitsbereich > Weiter**



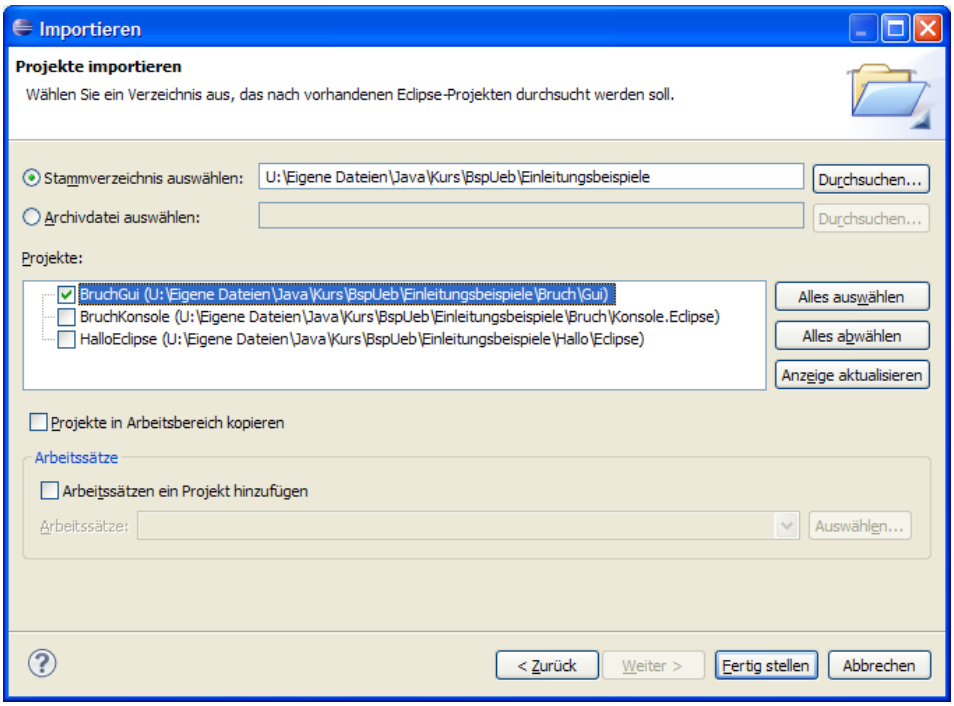

Klicken Sie in im folgenden Dialog auf den Schalter **Durchsuchen**,



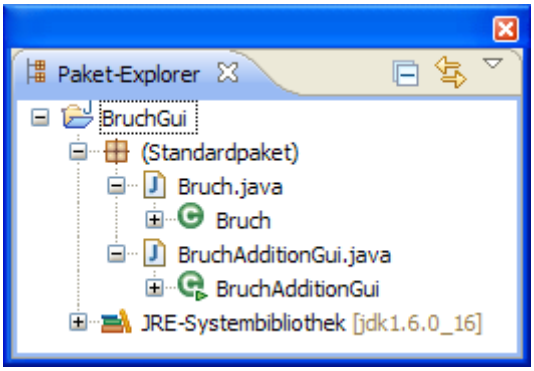
und wählen Sie im Verzeichnisbaum einen Knoten oberhalb des zu importierenden Projektordners, z.B.:




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:



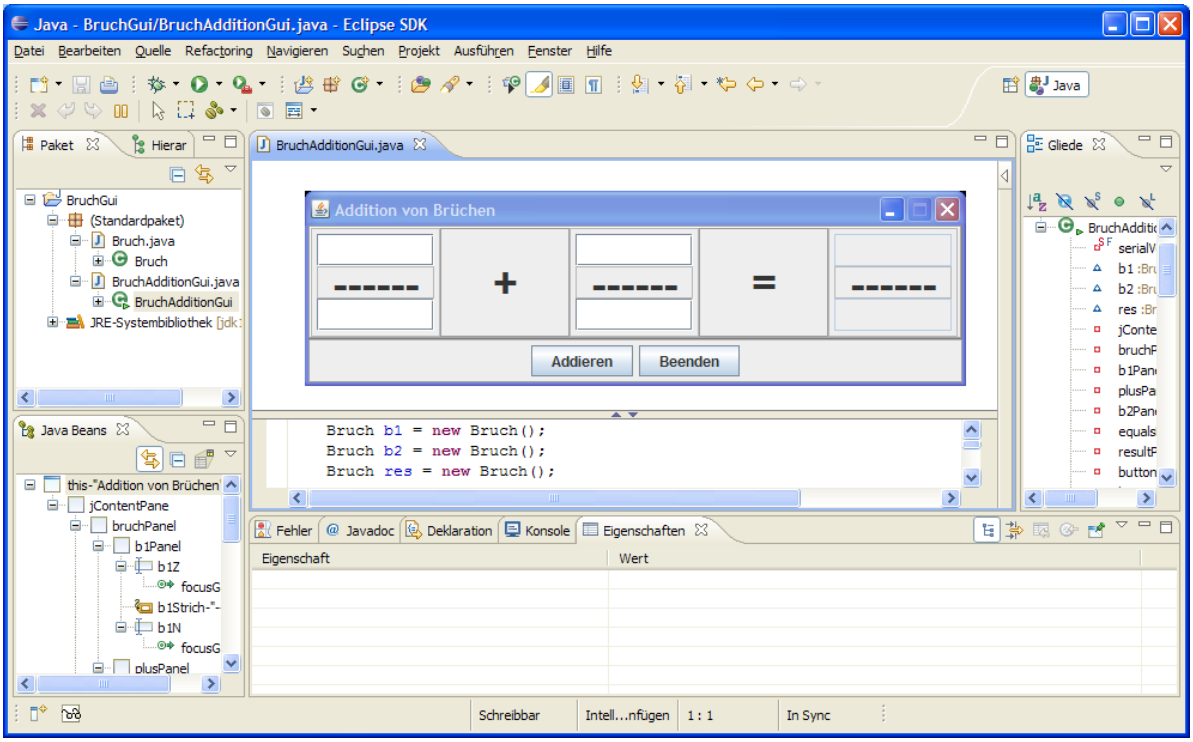

Anschließend sollten die importierten Projekte im Paket-Explorer auftauchen, z.B.:



Wenn Sie das Projekt mit der grafischen Variante des Bruchadditionsprogramms (vgl. Abschnitt 1.1.5) importiert haben, können Sie über die Kontextmenüoption

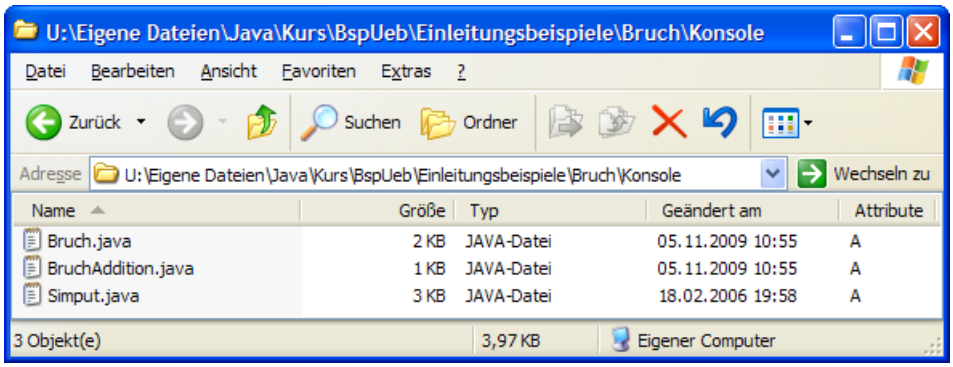
**Öffnen mit > Visual Editor**

zur Datei **BruchAdditionGui.java** das Anwendungsfenster des importierten Programms im visuellen Editor öffnen:




### 2.3.8 Projekt aus vorhandenen Quellen erstellen

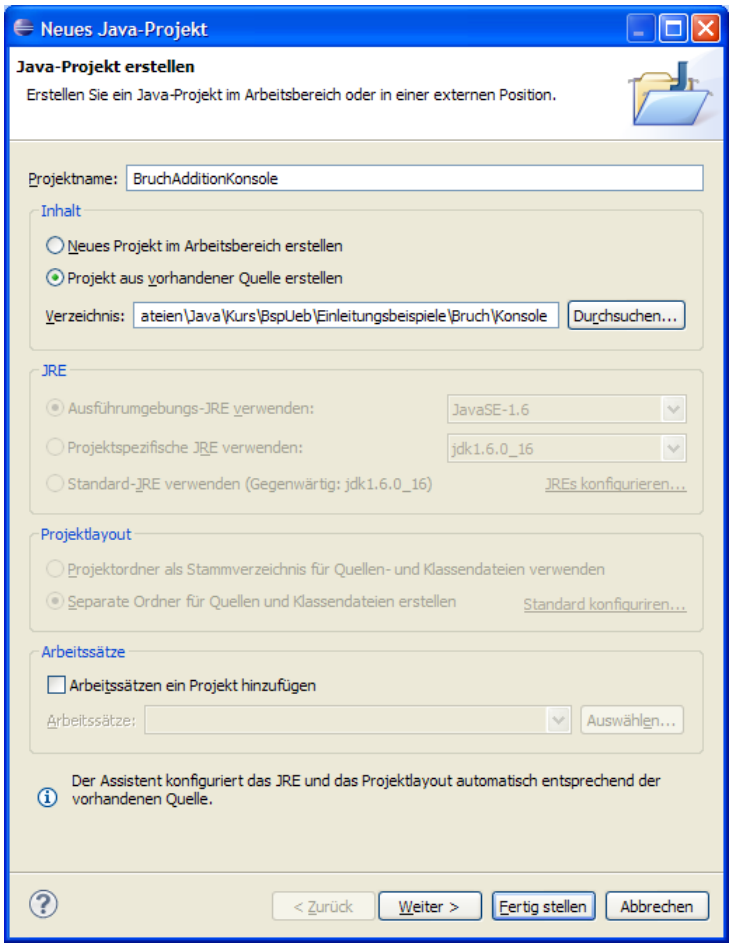
Gelegentlich soll ein neues Projekt unter Verwendung bereits existierender Quelldateien erstellt werden, z.B.:



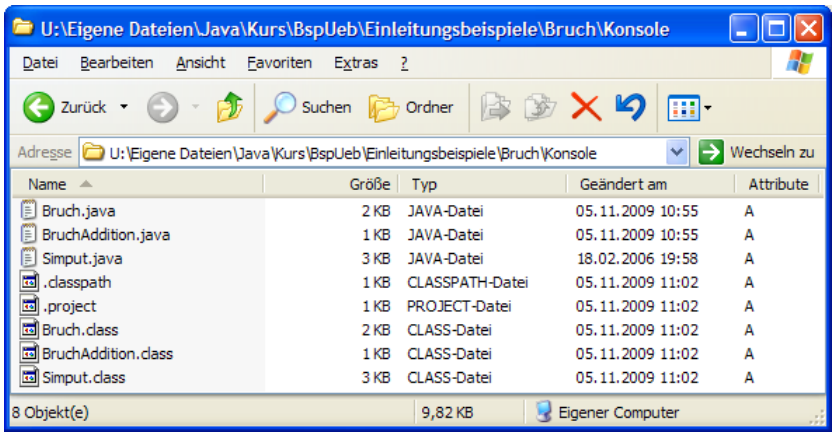

In diesem Fall wählt man in Eclipse nach

**Datei > Neu > Java-Projekt > Weiter**

in der Dialogbox **Neues Java-Projekt** die Option **Projekt aus vorhandener Quelle erstellen**:




Nach dem **Fertigstellen** übernimmt Eclipse den Ordner und ergänzt dort seine Projektdateien, z.B.:




## *2.4 Übungsaufgaben zu Kapitel 2*

1) Experimentieren Sie mit dem `Hallo`-Beispielprogramm, 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 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 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:\Programme\Java\jre6\lib\rt.jar**).

4) Führen Sie nach Möglichkeit die in Abschnitt 2.1 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 Bruchrechnungsprogramms. Alternativ können Sie auch die Java-Archivdatei **Simput.jar** kopieren und in den Klassenpfad aufnehmen. Mit Java-Archivdateien werden wir uns später noch ausführlich beschäftigen.

# 3 Elementare Sprachelemente

In Abschnitt 1 wurde anhand eines halbwegs realistischen Beispiels versucht, einen ersten Eindruck von der objektorientierten Softwareentwicklung mit Java zu vermitteln. 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**.
  Unser Bruchrechnungsbeispiel in Abschnitt 1.1 besteht aus den beiden Klassen `Bruch` und `BruchAddition`. Meist verwendet man für den Quellcode einer Klasse jeweils eine eigene Datei. Der Compiler erzeugt auf jeden Fall für jede Klasse eine eigene Bytecodedatei.
- Eine **Klassendefinition** besteht aus …
  - dem **Kopf** mit dem einleitenden Schlüsselwort **class** und dem Namen der Klasse
    In Abschnitt 2.2.1 finden sich im Zusammenhang mit dem `Hallo`-Beispiel nähere Erläuterungen.
  - 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 Beispiel in Abschnitt 2.2.1 ). All diese Bestandteile werden noch ausführlich diskutiert.
  - und dem **Rumpf**
    Begrenzt durch ein Paar geschweifte Klammern befinden sich hier beliebig viele **Anweisungen**, mit denen z.B. lokale Variablen deklariert oder verändert werden. Der Unterschied zwischen Instanzvariablen (Eigenschaften von Objekten) und lokalen Variablen von Methoden wird in Abschnitt 3.3 erläutert.
- Von den Klassen eines Programms muss mindestens 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 3 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.

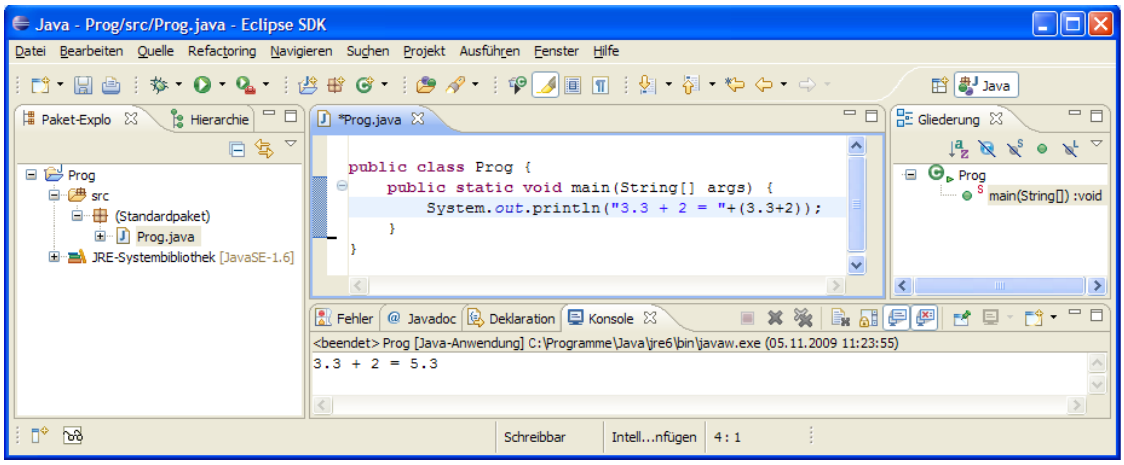
[1] Hier ist ausdrücklich von Java-*Programmen* (alias -*Applikationen*) die Rede. Bei den später vorzustellenden Java-*Applets* ergeben sich einige Abweichungen.

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. Es wird nur *eine* Klasse definiert, und diese erhält nur eine einzige Methodendefinition. Weil die Klasse startfähig sein muss, liegt **main()** als Name der Methode fest, und wir erhalten die folgende Programmstruktur:

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

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

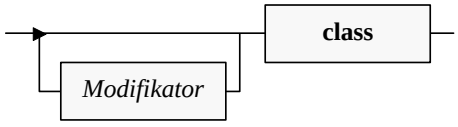
Für die meist kurzzeitige Beschäftigung mit bestimmten elementaren Sprachelementen lohnt sich selten ein spezielles Eclipse-Projekt. Legen Sie für solche Zwecke ein Projekt namens `Prog` mit gleichnamiger Startklasse an, und tauschen Sie den Inhalt der **main()**-Methode nach Bedarf aus, z.B.:




### 3.1.2 Syntaxdiagramme

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

- Man bewegt sich vorwärts in Pfeilrichtung durch das Pfaddiagramm 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.
- 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.

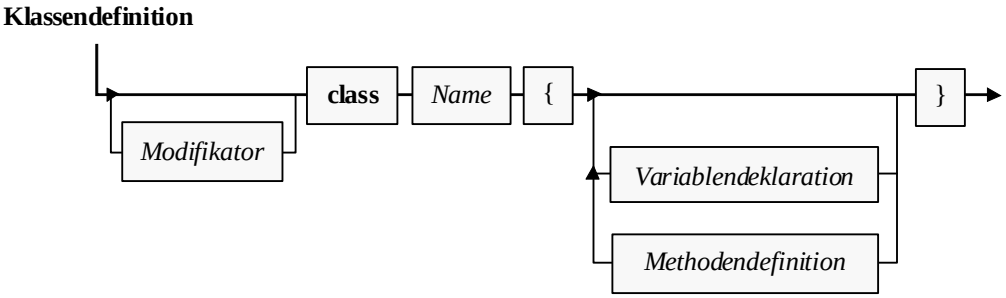
Als Beispiele betrachten wir die Syntaxdiagramme zur Definition von Klassen und Methoden. Aus didaktischen Gründen zeigen die Diagramme nur solche Sprachbestandteile, die im Beispielpro-

gramm von Abschnitt 1.1 (Bruchrechnung) verwendet wurden. Durch den engen Bezug zum Beispielprogramm 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.

### *3.1.2.1 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 (eventuell an anderer Stelle erläuterte) Weise ersetzt, sollte eine syntaktisch korrekte Klassendefinition entstehen.

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

Modifikator Name

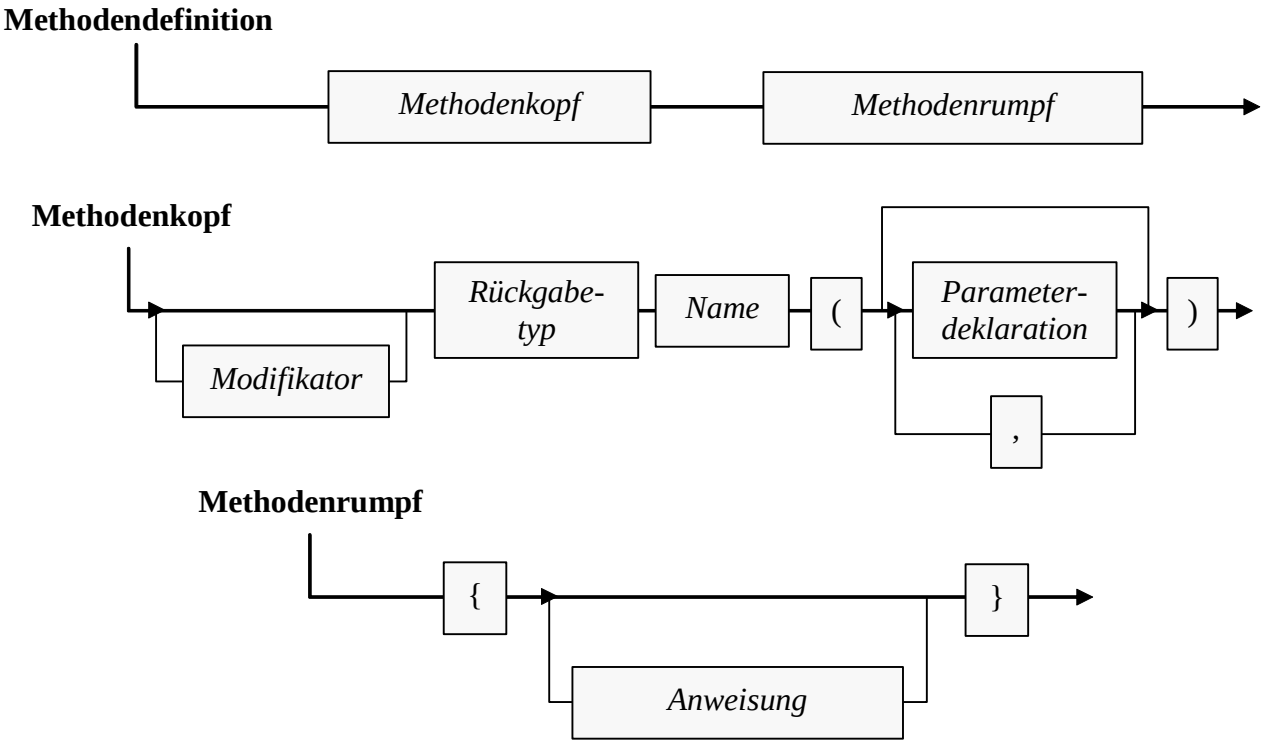
Variablen-deklarationen

Methoden-definitionen

```
public class Bruch {

   int zaehler;
   int nenner = 1;

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

   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() {
      . . .
   }

   public void addiere(Bruch b) {
      . . .
   }

   public void frage() {
      . . .
   }

   public void zeige() {
      . . .
   }

}
```

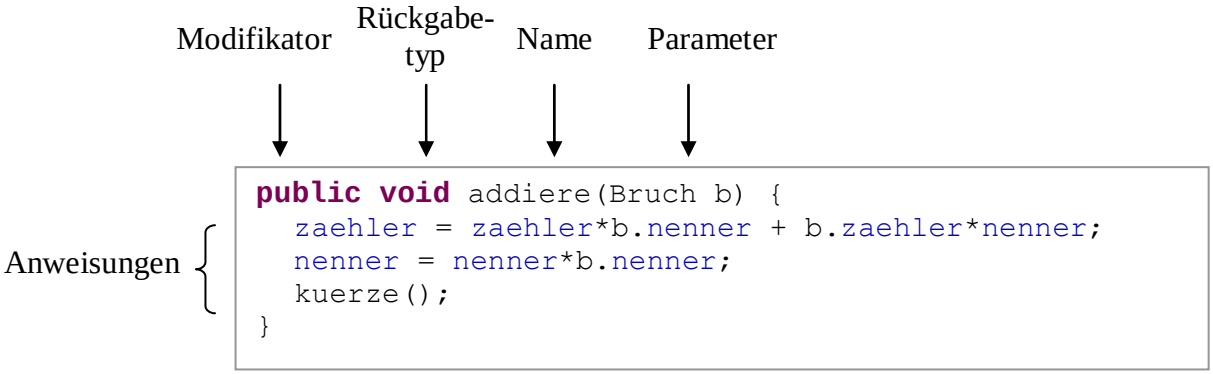
### *3.1.2.2 Methodendefinition*

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




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

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




### 3.1.3 Hinweise zur Gestaltung des Quellcodes

Zur Formatierung von Java-Programmen haben sich Konventionen entwickelt, die wir bei passender Gelegenheit besprechen werden. Der Compiler ist hinsichtlich der Formatierung 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 *selbst begrenzend*, d.h. davor und danach sind keine Trennzeichen nötig (aber erlaubt).

Wer dieses Manuskript am Bildschirm liest oder an einen Farbdrucker geschickt hat, profitiert hoffentlich von der Syntaxgestaltung durch Farben und Textattribute, die von Eclipse 3 stammt.

Ob man beim Rumpf einer Klassen- oder Methodendefinition die öffnende geschweifte Klammer an das Ende der Kopfzeile setzt oder an den Anfang der Folgezeile, ist Geschmacksache, z.B.:

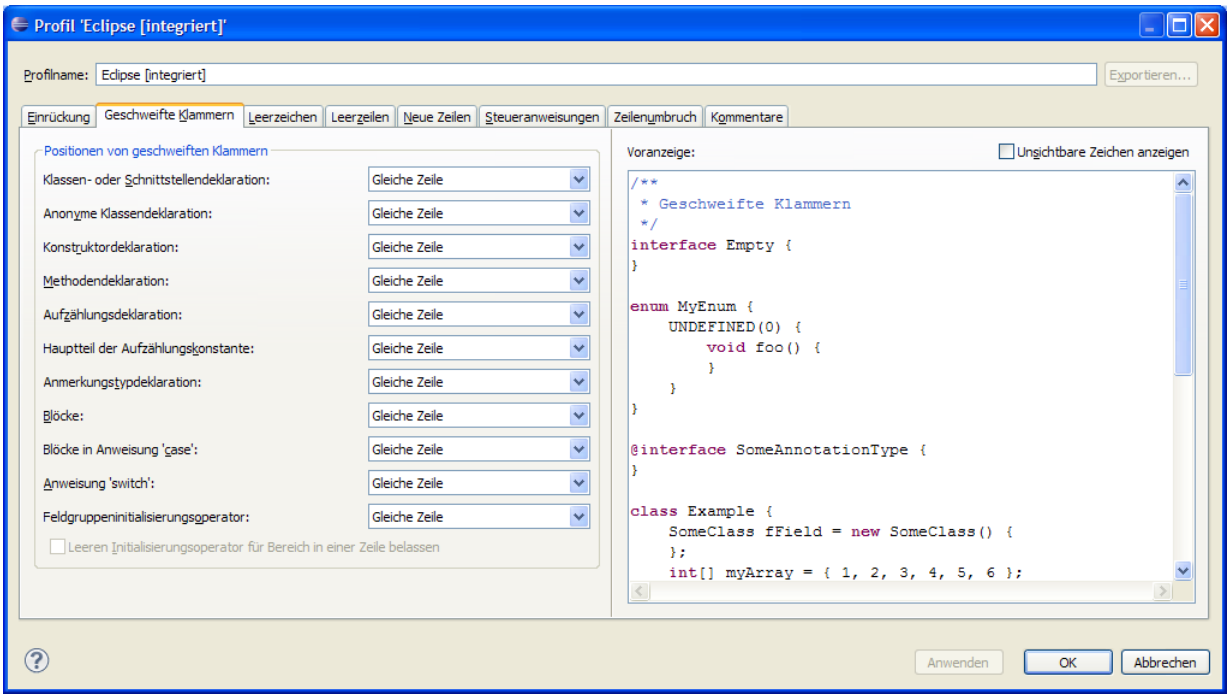
```
class Hallo {
   public static void main(String[] par) {
      System.out.print("Hallo");
   }
}
```

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

Eclipse 3 bevorzugt die linke Variante, könnte aber nach

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

in der folgenden Dialogbox umgestimmt werden:




### 3.1.4 Kommentare

Java bietet drei Möglichkeiten, den Quelltext zu kommentieren:

- **Zeilenrestkommentar**
  Alle Zeichen von // bis zum Ende der Zeile gelten als Kommentar, wobei kein Terminierungszeichen erforderlich ist, z.B.:

  ```
  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 Null
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 desaktivieren, ohne ihn löschen zu müssen.

  Weil der Mehrzeilenkommentar ohne farbliche Hervorhebung der auskommentierten Passage unübersichtlich ist, wird er selten verwendet. Wenn Sie z.B. in Eclipse mehrere markierte Quellcodezeilen mit dem Menübefehl

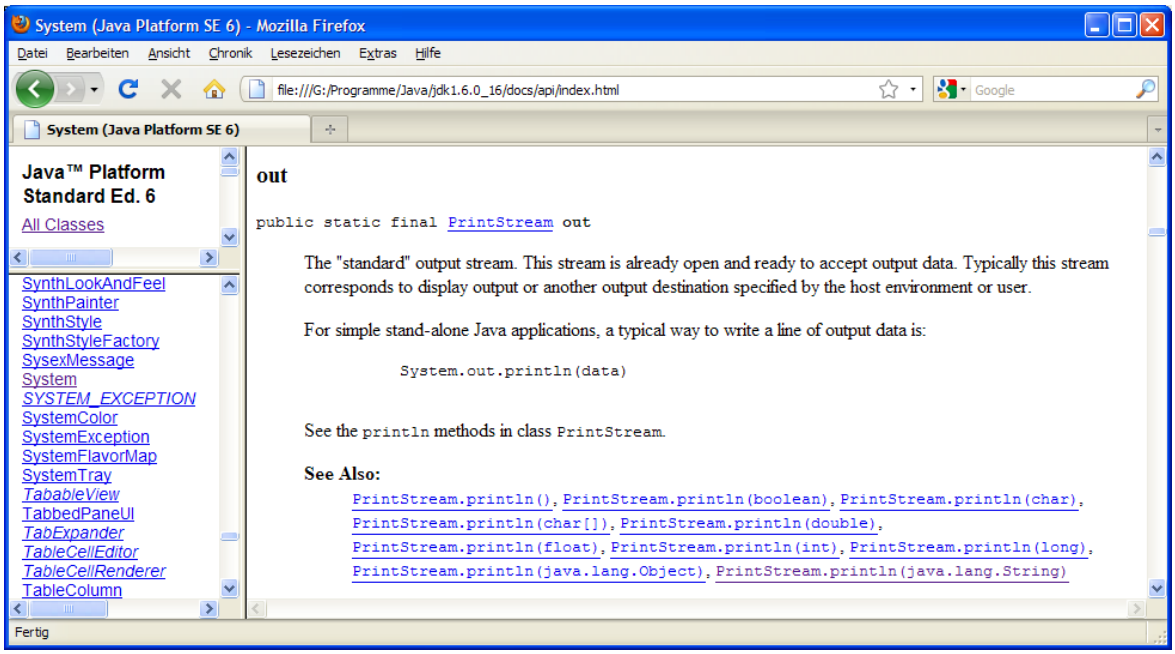
  **Quelle > Kommentar umschalten**

  bzw. mit der Tastenkombination **Strg+/** 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 etc. unterstützt. Nähere Informationen finden Sie z.B. in der JDK 6 - Dokumentation auf folgendem Weg

  **Javadoc Tool > Javadoc Tool Reference Page**

  Hinter der folgenden API-Dokumentation




  zum Ausgabeobjekt **out** in der Klasse **System**, das Sie schon kennen gelernt haben, steckt der folgende Dokumentationskommentar:

```
/**
 * 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 = nullPrintStream();
```

Man findet ihn in der Quellcodedatei **System.java** zur Klasse **System**, die Bestandteil des API-Quellcodearchivs **src.zip** ist. In Abschnitt 2.1 wurde empfohlen, das Quellcodearchiv als JDK-Bestandteil zu installieren und anschließend in den Unterordner **src** der JDK-Installation auszupacken.

### 3.1.5 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.
- Java-Programme werden intern im **Unicode**-Zeichensatz dargestellt. Daher erlaubt Java im Unterschied zu vielen anderen Programmiersprachen 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[7] | continue | default | do | double |
| else | enum | extends | false | final | finally | float |
| for | goto[1] | 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 | | | |

- Namen müssen innerhalb ihres Kontexts (s. u.) eindeutig sein.

[7] Diese Schlüsselwörter sind reserviert, werden aber derzeit nicht unterstützt. Im Falle von **goto** wird sich an diesem Zustand wohl auch nichts ändern.

### 3.1.6 Pakete in eine Quellcodedatei importieren

Jede Java-Klasse gehört zu einem sogenannten **Paket** (siehe Abschnitt 7), und dem Namen der Klasse ist grundsätzlich der jeweilige Paketname voranzustellen. Dies gilt natürlich auch für die Klassen aus dem Java-API, die wir häufig verwenden, z.B. die Klasse **Random** aus dem Paket **java.util** zum Erzeugen von Pseudozufallszahlen:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  java.util.Random zuf = new java.util.Random();`<br>`  System.out.println(zuf.nextInt());`<br>`  }`<br>`}` | `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 Paket-Präfix ansprechbar. Die zuständigen **import**-Anweisungen sind an den Anfang der Quellcodedatei zu setzen, z.B. zum Importieren der Klasse **java.util.Random**:

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

Um *alle* Klassen in einem Paket zu importieren, gibt man einen Stern an Stelle des Klassennamens an, z.B.:

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

## *3.2 Ausgabe bei Konsolenanwendungen*

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

### 3.2.1 Ausgabe einer (zusammengesetzten) Zeichenfolge

Um eine einfache Konsolenausgabe in Java zu realisieren, bittet man das Objekt **System.out**, seine **print()** - oder seine **println()** -Methode auszuführen. erzeugen.[8] 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, …

[8] 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 Klas-

- wenn eine Benutzereingabe unmittelbar hinter einer Ausgabe in derselben Zeile ermöglicht werden soll (s. u.).
- 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 (s. u.).

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 Schlüsselwort **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.

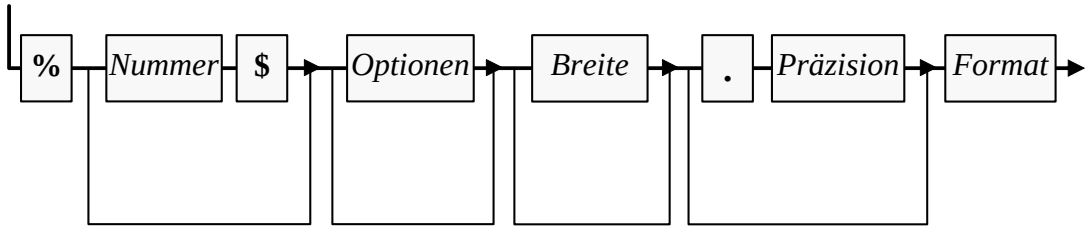
### 3.2.2 Formatierte Ausgabe

Der Methodenaufruf **System.out.printf()**[9] erlaubt eine formatierte Ausgabe von mehreren Ausdrücken, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  System.out.printf("Pi = %5.2f  e = %12.7f",`<br>`  Math.PI, Math.E);`<br>`  }`<br>`}` | `Pi =  3,14  e =  2,7182818` |

Als erster Parameter wird eine Zeichenfolge übergeben, die Platzhalter mit Formatierungsangaben für die restlichen Parameter enthält. Für einen Platzhalter kann folgende Syntax verwendet werden:

**Platzhalter für die formatierte Ausgabe**




se **System** zu erzeugen. U.a. befindet sich unter den **System**-Klassenvariablen 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.

9 Es handelt sich um eine Instanzmethode der Klasse **PrintStream** (siehe Fußnote in Abschnitt 3.2.1).

Darin bedeuten:

| | |
|---|---|
| *Nummer* | Fortlaufende Nummer des auszugebenden Arguments, bei 1$ beginnend |
| *Optionen* | Formatierungsoptionen, u.a. sind erlaubt<br>- bewirkt eine linksbündige Ausgabe<br>, 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** | für Zeichen | `// x ist eine char-`<br>`// Variable (s.u.)`<br>`("Inhalt von x %c", x)` | `h` |
| **d** | für ganze Zahlen im Dezimalsystem | `("%7d", 4711)`<br>`("%-7d", 4711)`<br>`("%1$d %1$,d", 4711)` | `  4711`<br>`4711`<br>`4711 4.711` |
| **f** | für Dezimalzahlen<br>Präzision: Anzahl der Nachkommastellen | `("%5.2f", 4.711)` | ` 4,71` |
| **e** | für Dezimalzahlen in wissenschaftlicher Notation<br>Präzision: Anzahl Stellen in der Mantisse | `("%e", 47.11)`<br>`("%.2e", 47.11)`<br>`("%12.2e", 47.11)` | `4,711000e+01`<br>`4,71e+01`<br>`  4.71e+01` |
| **g** | für Dezimalzahlen in variabler Notation<br>Präzision: Anzahl der Ziffern nach dem Runden | `("%.4g", 47.11)`<br>`("%.4g", 44444447.11)` | `47.11`<br>`4.444e+07` |

Eben wurde nur eine kleine Teilmenge der Syntax einer Java-Formatierungszeichenfolge vorgestellt. Die komplette Information findet sich in der JDK-Dokumentation zur Klasse **Formatter** (im Paket **java.util**[10]), die folgendermaßen zu erreichen ist:

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

  **C:\Program Files\Java\jdk1.6.0_16\docs\index.html**

- Wechseln Sie zur **Java Platform API Specification** durch einen Mausklick auf den zugehörigen Link.
- 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**.

[10] 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.

### 3.2.3 Schönheitsfehler bei der Konsolenausgabe unter Windows

Java-Konsolenanwendungen haben unter Windows einen Schönheitsfehler, von dem die erheblich relevanteren GUI-Anwendungen *nicht* betroffen sind:

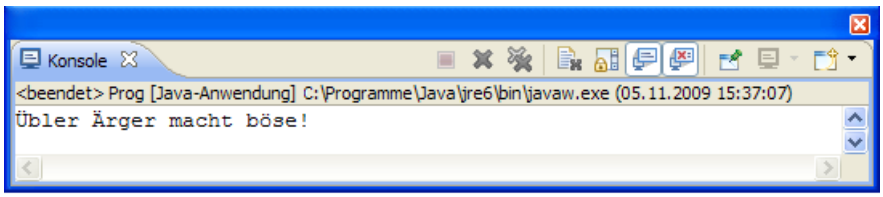
| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  System.out.println("Übler Ärger macht böse!");`<br>`  }`<br>`}` | ■bler ─rger macht b÷se! |

Die schlechte Behandlung von Umlauten bei Konsolenanwendungen unter Windows geht auf folgende Ursachen zurück:

- Windows arbeitet in Konsolenfenstern DOS-konform mit dem ASCII-Zeichensatz, in grafikorientierten Anwendungen hingegen mit dem ANSI-Zeichensatz.
- Die virtuelle Java-Maschine arbeitet unter Windows grundsätzlich mit dem ANSI-Zeichensatz.

Java bietet durchaus eine Lösung für das Problem, die im Wesentlichen aus einer Ausgabestromkonvertierungsklasse besteht und später im passenden Kontext dargestellt wird. Da wir Konsolenanwendungen nur als relativ einfache Umgebung zum Erlernen grundlegender Techniken und Konzepte verwenden und letztlich grafikorientierte Anwendungen entwickeln wollen, werden wir den Schönheitsfehler vorübergehend tolerieren.

Außerdem tritt das Problem *nicht* auf, wenn eine Konsolenanwendung innerhalb von Eclipse abläuft:




## *3.3 Variablen und Datentypen*

Während ein Programm läuft, müssen zahlreiche Informationen mehr oder weniger lange im Arbeitsspeicher des Rechners aufbewahrt und natürlich auch modifiziert werden, z.B.:

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

Zum Speichern eines Werts (z.B. einer Zahl) wird eine so genannte **Variable** verwendet, worunter Sie sich einen **benannten Speicherplatz von bestimmtem 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.:

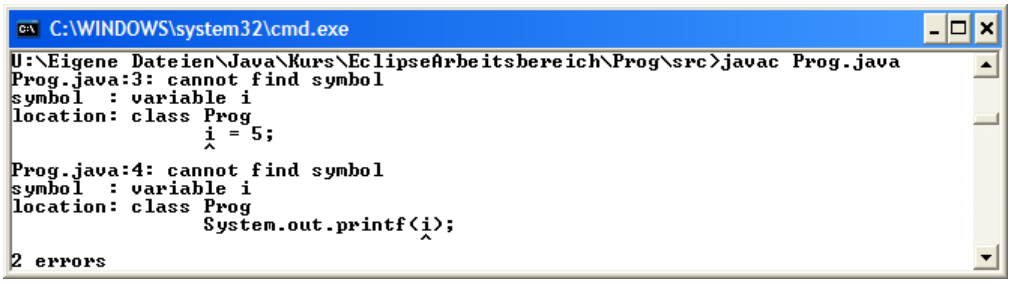
```
class Prog {
  public static void main(String[] args) {
    int ivar = 4711;        //schreibender Zugriff auf ivar
    System.out.println(ivar); //lesender Zugriff auf ivar
  }
}
```

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 Deklarieren von Variablen im Vergleich zu anderen Programmier- oder Skriptsprachen einige Sorgfalt, letztlich mit dem Ziel, Fehler zu vermeiden:

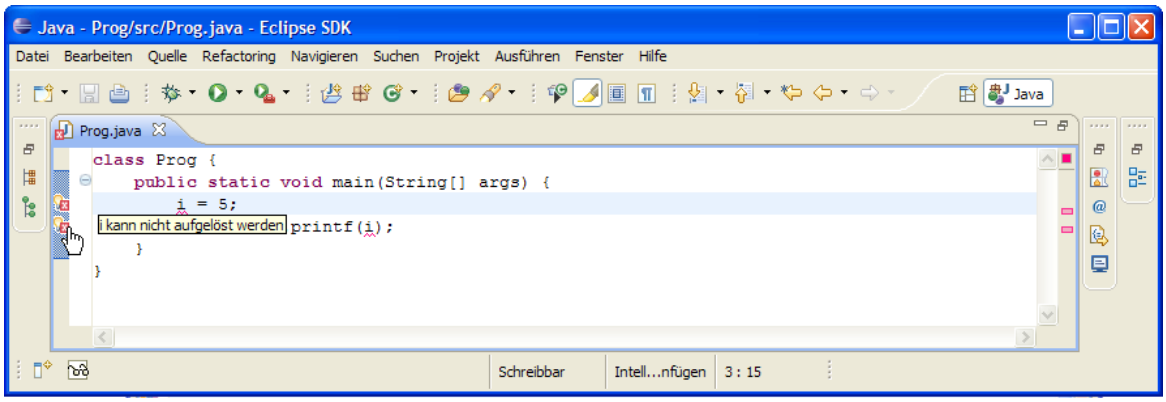
- Variablen müssen **explizit deklariert** werden, z.B.:

  ```
  int ivar = 4711;
  ```

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

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

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

- Variablen sind **statisch typisiert**.
  Jede Variable besitzt einen **Datentyp**, und dieser legt fest, welche Informationen (z.B. ganze Zahlen, rationale Zahlen, Zeichen) in der Variablen gespeichert werden können. In obigem Beispiel 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. Der Compiler kennt zu jeder Variablen den Datentyp und kann daher **Typsicherheit** garantieren, d.h. die Zuweisung von Werten mit ungeeignetem Datentyp verhindern.

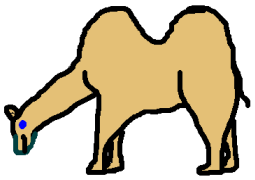
Die Variable `ivar` erhält in der Anweisung

```
int ivar = 4711;
```

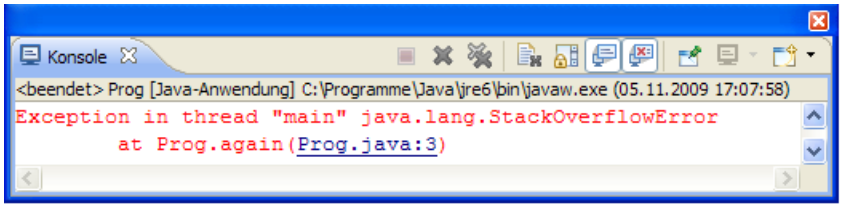
beim Deklarieren gleich den **Initialisierungswert** 4711. Auf diese oder andere Weise müssen Sie jeder *lokalen*, d.h. innerhalb einer Methode deklarierten, Variablen einen Wert zuweisen, bevor Sie zum ersten Mal lesend darauf zugreifen (vgl. Abschnitt 3.3.5).

Als wichtige Eigenschaften einer Java-Variablen halten wir fest:

- **Name**
  Es sind beliebige Bezeichner gemäß Abschnitt 3.1.5 erlaubt. Eine Beachtung der folgenden Konventionen verbessert die Lesbarkeit des Quellcodes, insbesondere auch für *andere* Programmierer:
  - 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*, siehe z.B. Gosling et al. 2005, S. 150). Das zur Vermeidung von Urheberrechtsproblemen handgemalte Tier kann hoffentlich trotz ästhetischer Mängel zur Begriffsklärung beitragen:



- **Datentyp**
  Damit sind festgelegt: Wertebereich, Speicherplatzbedarf und zulässige Operationen.
- **Aktueller Wert**
- **Ort im Hauptspeicher**
  Im Unterschied zu anderen Programmiersprachen (z.B. C++) belastet Java die Programmierer *nicht* mit der Verwaltung von Speicher*adressen*. Wir werden jedoch (nicht nur zur Förderung unserer EDV-Allgemeinbildung) zwei wichtige Speicher*regionen* unterscheiden (*Stack* und *Heap*), in denen Java-Programme ihre Variablen bzw. Objekte ablegen. Dieses Hintergrundwissen hilft z.B., wenn vom Laufzeitsystem (von der virtuellen Java-Maschine) ein **StackOverflowError** gemeldet wird:

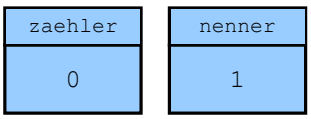



### 3.3.1 Primitive Typen und Referenztypen

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.

- **Primitive Datentypen**
  Die Variablen mit primitivem Datentyp sind auch in Java unverzichtbar, obwohl sie „nur“ zur Verwaltung ihres Inhalts dienen und keine Rolle bei Kommunikation mit Objekten spielen.
  Beispiele (aus der `Bruch`-Klassendefinition, siehe Abschnitt 1.1):

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



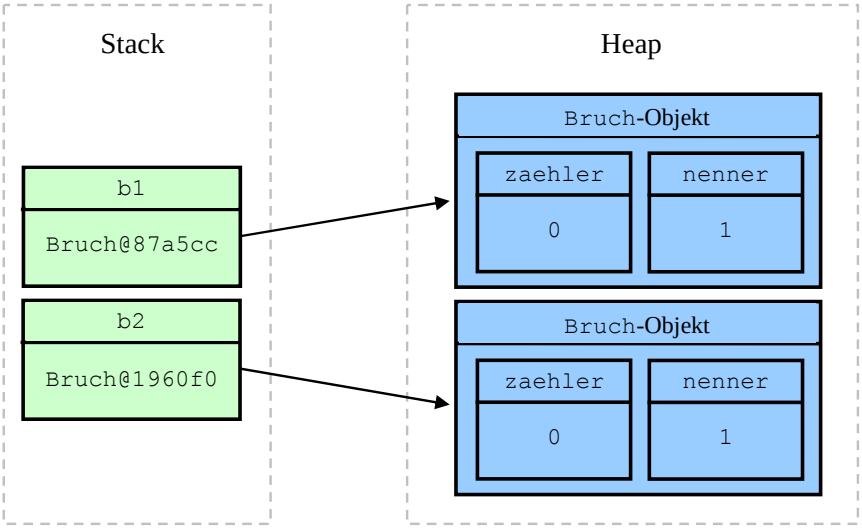

  Eine Variable vom Typ **int** kann ganze Zahlen im Bereich von –2147483648 bis 2147483647 als Werte aufnehmen. In Abschnitt 3.3.3 werden zahlreiche weitere primitive Datentypen vorgestellt.

- **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 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) werden z.B. die Referenzvariablen `b1` und `b2` mit dem Datentyp `Bruch` deklariert:

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

  Sie erhalten als Initialisierungswert jeweils eine Referenz auf ein neu erzeugtes `Bruch`-Objekt. Daraus resultiert im programmeigenen Speicher 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, wie z.B. in der folgenden Anweisung aus der **main()**-Methode der Klasse `BruchAddition`:

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

  Jedes `Bruch`-Objekt enthält die Felder (Instanzvariablen) `zaehler` und `nenner` vom primitiven Typ **int**. Es befindet sich in der als **Heap** (*Haufen*) bezeichneten Speicherregion, während im Beispiel die Referenzvariablen zu einer Methode gehören und daher in der als **Stack** (*Stapel*, *Keller*) bezeichneten Speicherregion abgelegt sind.

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, wenn im gesamten Programm keine einzige Referenz (Kommunikationsmöglichkeit) mehr vorhanden ist.

Wir werden im Abschnitt 3 überwiegend mit Variablen von elementarem 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.

### 3.3.2 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 3.3.6).

  Solange eine Methode ausgeführt wird, befinden sich ihre Variablen in einem Speicherbereich, den man als *Stack* (*Stapel*, *Keller*) bezeichnet. Die Abbildung in Abschnitt 3.3.1 zeigt die lokalen Variablen `b1` und `b2` aus der **main()**-Methode der Klasse `BruchAddition`, die als Referenzvariablen auf Objekte der Klasse `Bruch` zeigen.

- **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 seine Instanzvariablen in einem Speicherbereich, den man als *Heap* (*Haufen*) bezeichnet.

- **Klassenvariablen (statische Felder)**
  Diese Variablen beziehen sich auf eine Klasse, nicht auf einzelne Instanzen. Z.B. hält man oft in einer Klassenvariablen fest, wie viele Objekte der Klasse bereits bei einem Programmeinsatz erzeugt worden sind. In unserem Bruchrechnungs-Beispielprojekt haben wir der Einfachheit halber auf statische Felder verzichtet, allerdings sind uns schon statische Felder aus anderen Klasen begegnet:
  - Aus der Klasse **System** kennen wir schon die statische Variable **out**. Sie zeigt auf ein Objekt, das wir häufig mit Konsolenausgaben beauftragen.
  - Im Abschnitt 3.2.2 über die formatierte Ausgabe haben wir die Zahl π aus der statischen Variablen **PI** der Klasse **Math** entnommen.

  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 programmeigenen Speichers abgelegt.

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.

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

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

### 3.3.3 Primitive Datentypen

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 | Speicher-bedarf in Bit |
|---|---|---|---|
| **byte** | Diese Variablentypen speichern ganze Zahlen.<br>Beispiel:<br>`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 mit einer Genauigkeit von mindestens 7 Dezimalstellen.<br>Beispiel:<br>`float pi = 3.141593f;`<br>**float**-Literale (s. u.) benötigen den Suffix **f** (oder **F**). | Minimum:<br>$-3{,}4028235 \cdot 10^{38}$<br>Maximum:<br>$3{,}4028235 \cdot 10^{38}$<br>Kleinster Betrag:<br>$1{,}4 \cdot 10^{-45}$<br>(Norm: IEEE-754) | 32<br>1 für das Vorz.,<br>8 für den Expon.,<br>23 für die Mantisse |
| **double** | Variablen vom Typ **double** speichern Gleitkommazahlen mit einer Genauigkeit von mindestens 15 Dezimalstellen.<br>Beispiel:<br>`double pi = 3.1415926535898;` | Minimum:<br>$-1{,}7976931348623157 \cdot 10^{308}$<br>Maximum:<br>$1{,}7976931348623157 \cdot 10^{308}$<br>Kleinster Betrag:<br>$4{,}9406564584124654 \cdot 10^{-324}$<br>(Norm: IEEE-754) | 64<br>1 für das Vorz.,<br>11 für den Expon.,<br>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.<br>Beispiel:<br>`char zeichen = 'j';`<br>**char**-Literale (s. u.) sind durch *einfache* Anführungszeichen zu begrenzen. | Unicode-Zeichen<br>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.<br>Beispiel:<br>`boolean cond = false;` | **true**, **false** | 1 |

Als *Gleitkommazahl* (synonym: *Gleitpunkt-* oder *Fließkommazahl*, engl.: *floating point number* bezeichnet man eine EDV-freundlich notierte rationale Zahl (Dezimalzahl). Während die Mantisse

mit einer festen Anzahl von Ziffern für Genauigkeit sorgt, speichert man zusätzlich per Exponentialfaktor die Position des Dezimalkommas, sodass der Mantissenwert anwendungsgerecht für z.B. für Lichtjahre oder Nanometer stehen kann. Weil mit dem Speicheraufwand auch der Wertebereich begrenzt ist, bilden die Gleitkommazahlen natürlich nur eine endliche Teilmenge der rationalen Zahlen (im mathematischen Sinn). Zur Verarbeitung von Gleitkommazahlen wurde die *Gleitkommaarithmetik* entwickelt, normiert und zur Verbesserung der Verarbeitungsgeschwindigkeit teilweise sogar in Computer-Hardware realisiert. Nähere Informationen über die Darstellung von rationalen Zahlen im Arbeitsspeicher eines Computers folgen für speziell interessierte Leser gleich in Abschnitt 3.3.4.

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* Datentypen 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 als Bausteine für Klassen und als lokale Variablen in Methoden unverzichtbar.

### 3.3.4 Vertiefung: Darstellung rationaler Zahlen 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. Im Kurskontext dienen sie auch als Zeitvertreib für Teilnehmer mit Vorkenntnissen, die sich eventuell (z.B. im Abschnitt über elementare Sprachelemente) etwas langweilen.

#### *3.3.4.1 Binäre Gleitkommadarstellung*

Bei den binären Gleitkommatypen **float** und **double** werden auch „relativ glatte“ rationale Zahlen im Allgemeinen nur approximativ gespeichert, wie das folgende Programm zeigt:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  double d130 = 1.3f;`<br>`  double d125 = 1.25f;`<br>`  System.out.printf("%9.7f", d130);`<br>`  System.out.println();`<br>`  System.out.printf("%10.8f", d130);`<br>`  System.out.println();`<br>`  System.out.printf("%20.18f", d125);`<br>`  }`<br>`}` | `1,3000000`<br>`1,29999995`<br>`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 zugrunde liegende **IEEE-754** – Format für die Darstellung von rationalen Zahlen zu erklären. Es handelt sich um ein **binäres Gleitkommaformat**, wobei jede Zahl als Produkt aus drei getrennt zu speichernden Faktoren dargestellt wird:

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

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. Allerdings sind im Exponenten die Werte 0 und 255 (**float**) bzw. 0 und 2047 (**double**) für Spezialfälle (z.B. denormalisierte Darstellung, +/-Unendlich) reserviert (siehe Abschnitt 3.6.2). Um auch die für Zahlen mit einem Betrag kleiner Eins benötigten *negativen* Exponenten darstellen zu können, werden Exponenten mit einer Verschiebung (*Bias*) um den Wert 127 (**float**) bzw. 1023 (**double**) abgespeichert und interpretiert. Besitzt z.B. eine **float**-Zahl den Exponenten Null, landet der Wert

$$01111111_2 = 127$$

im Speicher, und bei negativen Exponenten resultieren dort Werte kleiner als 127.

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 Eins 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 Eins beginnend nummeriert) hat die Wertigkeit $2^{-i}$, so dass sich ihr *dezimaler* Gesamtwert folgendermaßen ergibt:

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

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

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

In der folgenden Tabelle finden Sie einige normalisierte **float**-Werte:

| **Wert** | **float-Darstellung (normalisiert)** | | |
|---|---|---|---|
| | **Vorz.** | **Exponent** | **Mantisse** |
| 0,75 = $(-1)^0 \cdot 2^{(126-127)} \cdot (1+0{,}5)$ | 0 | 01111110 | 10000000000000000000000 |
| 1,0 = $(-1)^0 \cdot 2^{(127-127)} \cdot (1+0{,}0)$ | 0 | 01111111 | 00000000000000000000000 |
| 1,25 = $(-1)^0 \cdot 2^{(127-127)} \cdot (1+0{,}25)$ | 0 | 01111111 | 01000000000000000000000 |
| -2,0 = $(-1)^1 \cdot 2^{(128-127)} \cdot (1+0{,}0)$ | 1 | 10000000 | 00000000000000000000000 |
| 2,75 = $(-1)^0 \cdot 2^{(128-127)} \cdot (1+0{,}25+0{,}125)$ | 0 | 10000000 | 01100000000000000000000 |
| -3,5 = $(-1)^1 \cdot 2^{(128-127)} \cdot (1+0{,}5+0{,}25)$ | 1 | 10000000 | 11000000000000000000000 |

Nun kommen wir endlich zur Erklärung der eingangs dargestellten Genauigkeitsunterschiede beim Speichern der Zahlen 1,25 und 1,3. Während die Restmantisse

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

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

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

Sehr aufmerksame Leser werden sich darüber wundern, wieso die Tabelle mit den elementaren Datentypen in Abschnitt 3.3.3 z.B.

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

als betragsmäßig kleinsten **float**-Wert nennt, obwohl der minimale Exponent nach obigen Überlegungen -126 beträgt, was zum (gerundeten) dezimalen Exponentialfaktor

$$1{,}2 \cdot 10^{-38}$$

führt. Dahinter steckt die *denormalisierte* Gleitkommadarstellung, die als Ergänzung zur bisher beschriebenen normalisierten Darstellung eingeführt wurde, um eine bessere Annäherung an die Zahl Null zu erreichen. Alle Exponenten-Bits sind auf Null 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* (Null oder Eins), mit komplett auf Null 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**

Hier ist das Programm **DoubleBits** zu sehen:

```
class DoubleBits {
  public static void main(String[] args) {
    double d;
    System.out.print("double: ");
    d = Simput.gdouble();
    System.out.println("\nBits: ");
    long bits = Double.doubleToLongBits(d);
    System.out.println(
      "1 12345678901 12345678901234567890123456789012345678901234567890123456789012");
    for (int i = 63; i >= 0; i--) {
      if (i == 62 || i == 51)
        System.out.print(' ');
      if ((1L << i & bits) != 0)
        System.out.print('1');
      else
        System.out.print('0');
    }
  }
}
```

Die statische **Double**-Methode **doubleToLongBits()** liefert zur interessierenden **double**-Zahl einen **long**-Wert mit identischen Bits, die mit Hilfe von bitorientierten Operatoren (siehe Abschnitt 3.5.6) identifiziert werden. So erfährt man z.B. das Speicherabbild der **double**-Zahl -3,5:

```
double: -3,5

Bits:
1 12345678901 1234567890123456789012345678901234567890123456789012
1 10000000000 1100000000000000000000000000000000000000000000000000
```

### *3.3.4.2 Dezimale Gleitkommadarstellung*

Wenn die Speicher- und Rechengenauigkeit der binären Gleitkommatypen nicht reicht, kommt die Klasse **BigDecimal** aus dem Paket **java.math** in Frage (siehe JDK-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 (*uw*)
- Eine Ganzzahl mit 32 Bit für die Anzahl der Nachkommastellen (*an*)

Bei der Zahl

$$1{,}3 = 13 \cdot 10^{-1}$$

gelingt eine verlustfreie Speicherung mit:

$$uw = 13,\ an = 1$$

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 3.3.4.1, dass auch eine **double**-Variable den Wert 1,3 nur approximativ speichern kann:

```
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, speziell bei arithmetischen Operationen

Möglicherweise bietet Java irgendwann einen dezimalen Gleitkommatyp als elementaren Datentyp, was zu einer vereinfachten Syntax führen würde.

Bei der Aufgabe,

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

zu berechnen, ergeben sich für die Datentypen **double** und **BigDecimal** folgende Genauigkeits- und Laufzeitunterschiede:

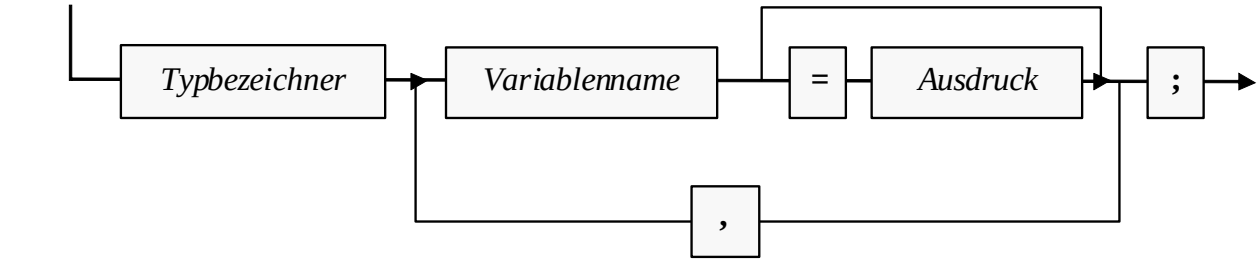
```
double:
 Abweichung:            -29.96745276451111
 Zeit in Millisekunden: 1688

BigDecimal:
 Abweichung:            0.0
 Zeit in Millisekunden: 51125
```

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 Sielcasino) wären heilfroh, wenn nach einem Spiel mit 1,7 Milliarden Euro nur 30 Euro fehlen würden. Generell sind im Finanzsektor solche Fehlbeträge aber unerwünscht, so dass man bei finanzmathematischen Aufgaben trotz des erhöhten Zeitaufwands (im Beispiel: Faktor 30) die Klasse **BigDecimal** verwenden sollte.

### 3.3.5 Variablendeklaration, Initialisierung und Wertzuweisung

In Java-Programmen muss jede Variable vor ihrer ersten Verwendung deklariert[11] werden. Dabei sind auf jeden Fall der Name und der Datentyp anzugeben, wie das Syntaxdiagramm zur **Variablendeklarationsanweisung** zeigt:

**Deklaration einer lokalen Variablen**




Als Datentypen kommen in Frage (vgl. Abschnitt 3.3.1):

- Primitive Datentypen, z.B.
  ```
  int i;
  ```
- Referenztypen, also Klassen (aus dem Java-API oder selbst definiert), z.B.
  ```
  Bruch b1;
  ```

Wir betrachten vorläufig nur *lokale* Variablen, die innerhalb einer Methode existieren. Ihre Deklaration darf im Methodenquellcode an beliebiger Stelle *vor* der ersten Verwendung erscheinen.

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

```
int i = 4711;
Bruch b1 = new Bruch();
```

---

[11] Während in C++ die beiden Begriffe *Deklaration* und *Definition* verschiedene Bedeutungen haben, werden sie im Zusammenhang mit den meisten anderen Programmiersprachen (so auch bei Java) synonym verwendet. In diesem Manuskript wird im Zusammenhang mit Variablen bevorzugt von *Deklarationen*, im Zusammenhang mit Klassen und Methoden hingegen von *Definitionen* gesprochen.

Im zweiten Beispiel wird per **new**-Operator ein `Bruch`-Objekt erzeugt und dessen Adresse in die neue Referenzvariable `b1` geschrieben. Mit der Objektkreation und auch mit der Konstruktion von gültigen *Ausdrücken*, die einen Wert von passendem Datentyp liefern müssen, werden wir uns noch ausführlich beschäftigen.

Es hat sich eingebürgert, Variablennamen mit einem Kleinbuchstaben beginnen zu lassen, um sie im Quelltext gut von den Bezeichnern für Klassen oder Konstanten (s. u.) unterscheiden zu können.

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);
  }
}
```

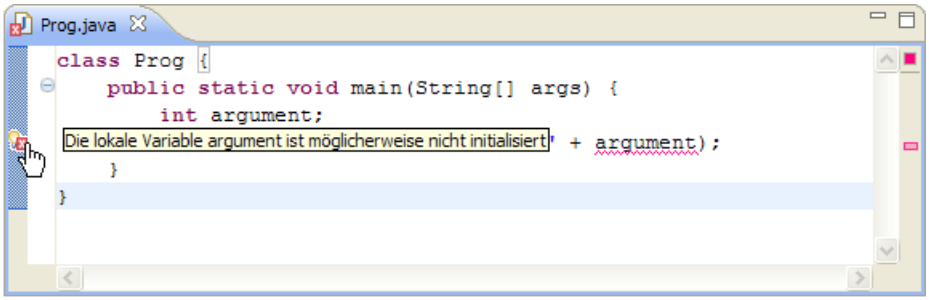
Der JDK-Compiler meint dazu:

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

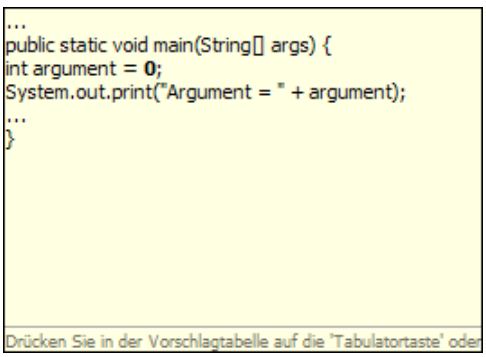
Ähnlich äußert sich auch der Eclipse-Compiler:

```
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 linken Mausklick auf das Fehlersymbol neben der betroffenen Zeile

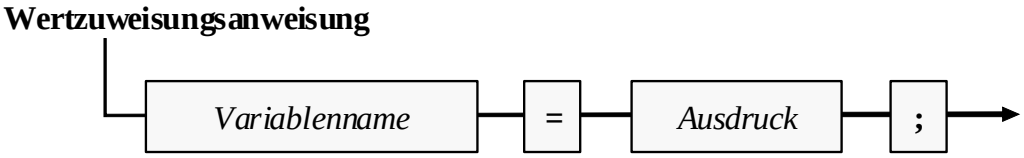



sogar konkrete Hinweise zur Verbesserung des Quellcodes:




Weil Instanz- und Klassenvariablen automatisch mit dem Standardwert ihres Typs initialisiert werden (s. u.), kann in Java-Programmen kein Zugriff auf undefinierte Werte stattfinden.

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




Beispiel: `ggt = az;`
Durch diese Wertzuweisungsanweisung aus der `kuerze()`-Methode unserer `Bruch`-Klasse (siehe Abschnitt 1.1) erhält die **int**-Variable `ggt` den Wert der **int**-Variablen `az`.

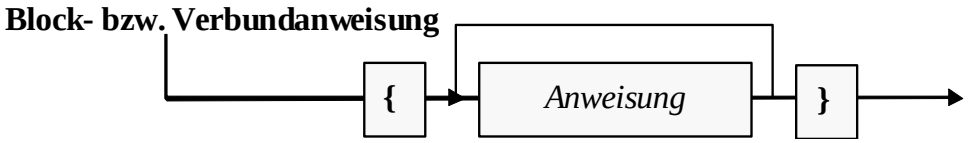
Es wird sich bald herausstellen, dass auch ein Ausdruck stets einen Datentyp hat. Bei der Wertzuweisung muss dieser Typ natürlich kompatibel zum Datentyp der Variablen sein.

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

- Variablendeklaration
- Wertzuweisung

### 3.3.6 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 weitere 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.

Oft treten Blöcke als Bestandteil von Bedingungen oder Schleifen (s. u.) auf, z.B. in der Methode `setzeNenner()` der Klasse `Bruch`:

```
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 Verlassen des Blocks wird der belegte Speicher frei gegeben, so dass der Eclipse-Compiler das Übersetzen des folgenden (weitgehend sinnfreien) Beispielprogramms

```
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);
   }
}
```

mit einer Fehlermeldung ablehnt:

```
Exception in thread "main" java.lang.Error: Unaufgelöstes Kompilierungsproblem:
   wert2 kann nicht aufgelöst werden

   at Prog.main(Prog.java:11)
```

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);
}
```

Bei Eclipse kann ein markierter Block aus mehreren Zeilen mit

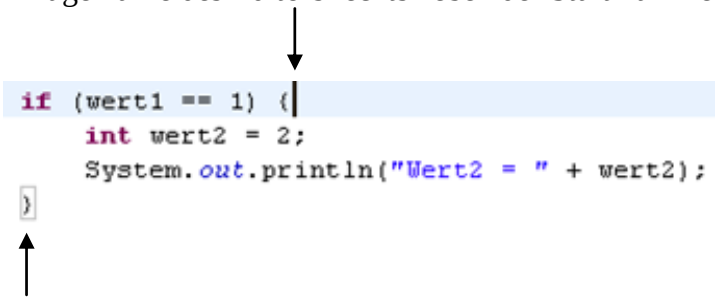
**Tab** komplett nach rechts eingerückt

und mit

**Umschalt + Tab** komplett nach links ausgerückt

werden. Außerdem kann man sich zu einer Blockklammer das Gegenstück anzeigen lassen:

Einfügemarke des Editors rechts neben der Startklammer




hervorgehobene Endklammer

### 3.3.7 Finalisierte Variablen (Konstanten)

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:

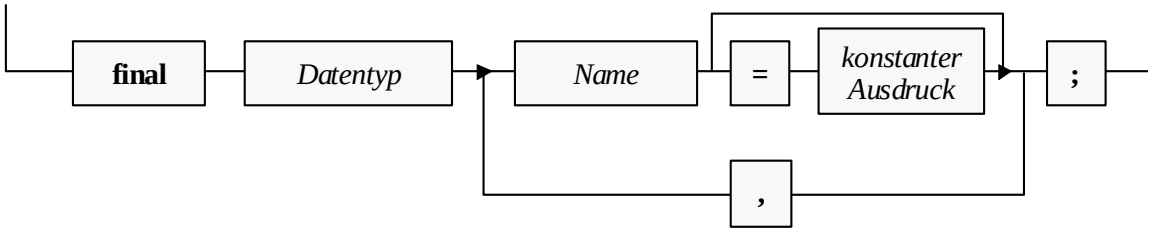
- Bei einer späteren Änderung des Wertes ist nur die Quellcodezeile mit der Variablendeklaration und -initialisierung betroffen.
- Der Quellcode ist leichter zu lesen.

Beispiel:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  final double MWST = 1.19;`<br>`  double netto = 100.0, brutto;`<br>`  brutto = netto * MWST;`<br>`  System.out.println("Brutto: " + brutto);`<br>`  }`<br>`}` | `Brutto: 119.0` |

Variablen, die nach ihrer Initialisierung im gesamten Programmverlauf 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 Unterschiede zur gewöhnlichen Variablendeklaration steht der Modifikator **final**. Das Initialisieren einer finalisierten Variablen kann bei der Deklaration oder in einer späteren Wertzuweisung erfolgen. Dabei ist ein *konstanter* Ausdruck zu verwenden, zu dessen Berechnung bereits der Compiler alle benötigten Informationen besitzt.

Man verwendet üblicherweise im Namen einer finalisierten Variablen (Konstanten) ausschließlich Großbuchstaben. Besteht ein Name aus mehreren Wörtern, trennt man sie der Lesbarkeit halber durch einen Unterstrich (z.B.: `MAXIMALE_QUOTE`).

Neben lokalen Variablen können auch (statische) Felder einer Klasse finalisiert werden (siehe Abschnitte 4.2.5 und 4.5.1).

### 3.3.8 Literale

Die im Quellcode auftauchenden expliziten Werte bezeichnet man als *Literale*. Wie Sie aus dem Abschnitt 3.3.7 wissen, sollten Literale vorzugsweise bei der Initialisierung von finalen Variablen (Konstanten) verwendet werden, z.B.:

```
final double MWST = 1.19;
```

Auch die Literale besitzen in Java stets einen Datentyp, wobei einige Regeln zu beachten sind.

In diesem Abschnitt haben manche Passagen Nachschlage-Charakter, so dass man beim ersten Lesen nicht jedes Detail aufnehmen muss bzw. kann.

#### *3.3.8.1 Ganzzahlliterale*

Für ein Ganzzahlliteral wird meist das dezimale Zahlensystem verwendet. Java unterstützt aber auch das hexadezimale (mit der Basis 16 und den Ziffern 0, 1, …, 9, A, B, C, D, E, F) und das oktale Zahlensystem (mit der Basis 8 und den Ziffern 0, 1, 2, …, 7), wobei ein Präfix zu setzen ist:

| **Zahlensystem** | **Präfix** | **Beispiele** | |
|---|---|---|---|
| | | **println()-Aufruf** | **Ausgabe** |
| dezimal | | `System.out.println(11);` | `11` |
| hexedezimal | 0x, 0X | `System.out.println(0x11);` | `17` |
| oktal | 0 | `System.out.println(011);` | `9` |

Für das Ganzzahlliteral `0x11` ergibt sich der dezimale Wert 17 aufgrund der Stellenwertigkeiten im Hexadezimalsystem folgendermaßen:

$$11_{Hex} = 1 \cdot 16 + 1 \cdot 1 = 17$$

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:

$$11_{Oktal} = 1 \cdot 8 + 1 \cdot 1 = 9$$

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**-Wert außerhalb des zulässigen Bereichs resultiert:

```
Exception in thread "main" java.lang.Error: Unaufgelöstes Kompilierungsproblem:
  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.

#### *3.3.8.2 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 F = 9.78f;
```

Mit dem (kaum jemals erforderlichen) Suffix **D** oder **d** wird der Datentyp **double** „optisch betont“.

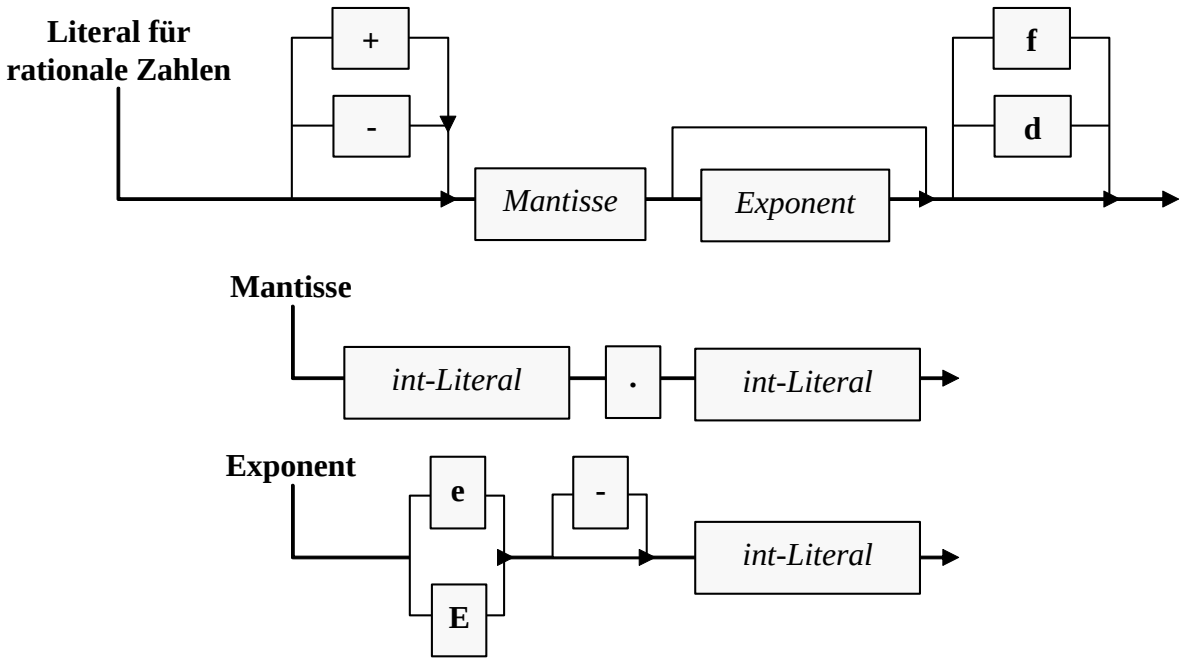
Die Java-Compiler achten bei Wertzuweisungen streng auf die Typkompatibilität. Z.B. führt die folgende Zeile:

```
final float MWST = 1.19;
```

zur folgenden Fehlermeldung des Eclipse-Compilers:

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

Hinsichtlich der Schreibweise von Gleitkommaliteralen bietet Java etliche Möglichkeiten, von denen die wichtigsten in den folgenden Syntaxdiagrammen dargestellt werden:




Die in der Mantisse und im Exponenten auftretenden Ganzzahlliterale müssen das dezimale Zahlensystem verwenden und den Datentyp **int** besitzen, so dass die in Abschnitt 3.3.8.1 beschriebenen Präfixe (0, 0x, 0X) und Suffixe (L, l) verboten sind. Die Exponenten werden natürlich zur Basis Zehn verstanden.

Beispiele:

```
-9.78f
107.45875e-20
```

#### *3.3.8.3 char-Literale*

**char**-Literale werden in Java durch *einfache* Hochkommata begrenzt. Es sind erlaubt:

- **Einfache Zeichen**
  Beispiel:
  ```
  char bst = 'a';
  ```
  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**
  Hier dürfen einem einleitenden Rückwärts-Schrägstrich u.a. folgen:
  - Ein Steuerzeichen, z.B.:

    | | |
    |---|---|
    | Neue Zeile | `\n` |
    | Horizontaler Tabulator | `\t` |

  - Einfaches oder doppeltes Hochkomma sowie der Rückwärts-Schrägstrich:

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

  Beispiel:
  ```
  final char RS = '\\';
  ```

- **Unicode-Escape-Sequenzen**
  Eine Unicode-Escape-Sequenz enthält eine Unicode-Zeichennummer (vorzeichenlose Ganzzahl mit 16 Bits, also im Bereich von 0 bis $2^{16}$-1 = 65535) in hexadezimaler, vierstelliger Schreibeweise (ggf. links mit Nullen aufgefüllt) 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.

  Weil die ASCII-Zeichen mit ihren gewohnten Nummer in den Unicode-Zeichensatz übernommen wurden, kann z.B. auf folgende Weise bei Konsolenanwendungen ein Ton (ASCII-Nummer 7) erzeugt werden:[12]

```
System.out.println('\u0007');
```

#### *3.3.8.4 Zeichenkettenliterale*

Zeichenkettenliterale werden (im Unterschied zu **char**-Literalen) durch *doppelte* Hochkommata begrenzt. Hinsichtlich der erlaubten Zeichen und der Escape-Sequenzen gelten die Regeln für **char**-Literale analog, wobei das einfache und das doppelte Hochkomma ihre Rollen tauschen, 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 der Standardbibliothek handelt.

#### *3.3.8.5 boolean-Literale*

Als Literale vom Typ **boolean** sind nur die beiden reservierten Wörter **true** und **false** erlaubt, z.B.:

```
boolean cond = true;
```

## *3.4 Eingabe bei Konsolenanwendungen*

### 3.4.1 Die Klassen Scanner und `Simput`

Für die Übernahme von Tastatureingaben ist kann die API-Klasse **Scanner** (aus dem Paket **java.util**) verwendet werden.[13] Im folgenden Beispielprogramm zur Berechnung der Fakultät wird ein **Scanner**-Objekt per **nextInt()**-Methodenaufruf gebeten, vom Benutzer eine **int**-Ganzzahl entgegen zu nehmen:

---

[12] Beim Start aus der Entwicklungsumgebung Eclipse 3 misslingt die Tonausgabe allerdings. Der folgende Methodenaufruf bewirkt zuverlässing (auch bei GUI-Anwendungen) ein wohlgeformtes akustisches Signal:

```
java.awt.Toolkit.getDefaultToolkit().beep();
```

[13] 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.

```
import java.util.*;
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, wird dieses Paket importiert.
- 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 **nextInt()** mit einer so genannten Ausnahme, und das Programm „stürzt ab“, z.B.:

```
Argument:  vier
Exception in thread "main" java.util.InputMismatchException
  at java.util.Scanner.throwFor(Scanner.java:840)
  at java.util.Scanner.next(Scanner.java:1461)
  at java.util.Scanner.nextInt(Scanner.java:2091)
  at java.util.Scanner.nextInt(Scanner.java:2050)
  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 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) sowie im folgenden Ordner auf dem Laufwerk P: im Campusnetz der Universität Trier:

**P:\Prog\JavaSoft\JDK\Erg\class**

Mit Hilfe der Klassenmethode `Simput.gint()` lässt sich das Fakultätsprogramm einfacher und zugleich robust gegenüber Eingabefehlern realisieren:

```
class Fakul {
  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 3.1.6). In Klassen anderer Pakete steht `Simput` (wie alle anderen Klassen des Standardpakets) jedoch *nicht* zur Verfügung. Im Kurs verwenden wir in der Regel das Standardpaket, um unserer Aufmerksamkeit voll auf das jeweilige Thema konzentrieren zu können, so dass die Klasse `Simput` als bequemes Hilfsmittel zur Verfügung steht. Bei ernsthaften Projekten werden Sie jedoch eigene Pakete benutzen (siehe Kapitel 7), so dass die Klasse `Simput` dort nicht verwendbar ist. Mit Hilfe des Quellcodes in der Datei **Simput.java** (Verzeichnis **…\BspUeb\Simput**, weitere Ortsangaben im Vorwort) lässt sich die Klasse aber leicht in ein Paket aufnehmen.

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

```
Argument:  vier
Falsche Eingabe!

Fakultaet: 1.0
```

Bei `gint()` oder anderen `Simput`-Methoden, die auf Eingabefehler *nicht* mit einer Ausnahme reagieren (vgl. Abschnitt 9), kann man sich durch einen Aufruf der `Simput`-Klassenmethode `checkError()` mit Rückgabetyp **boolean** nachträglich 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 Fakul {`<br>`  public static void main(String args[]) {`<br>`  int i, argument;`<br>`  double fakul = 1.0;`<br><br>`  System.out.print("Argument:  ");`<br>`  argument = Simput.gint();`<br><br>`  if (!Simput.checkError()) {`<br>`  for (i = 1; i <= argument; i += 1)`<br>`  fakul = fakul * i;`<br>`  System.out.println("Fakultaet: " + fakul);`<br>`  }`<br>`  else`<br>`  System.out.println(`<br>`  Simput.getErrorDescription());`<br>`  }`<br>`}` | `Argument:  vier`<br>`Falsche Eingabe!`<br><br>`Eingabe konnte nicht konvertiert werden.` |

Neben `gint()` besitzt die Klasse `Simput` noch analoge Methoden für andere Datentypen, u.a.:

- `static public char gchar()`
  Liest ein Zeichen von der Konsole
- `static public 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.

Außerdem sind Methoden mit einer Fehlerbehandlung über die Ausnahmetechnik (vgl. Abschnitt 9) vorhanden (wie bei der Klasse **Scanner**).

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

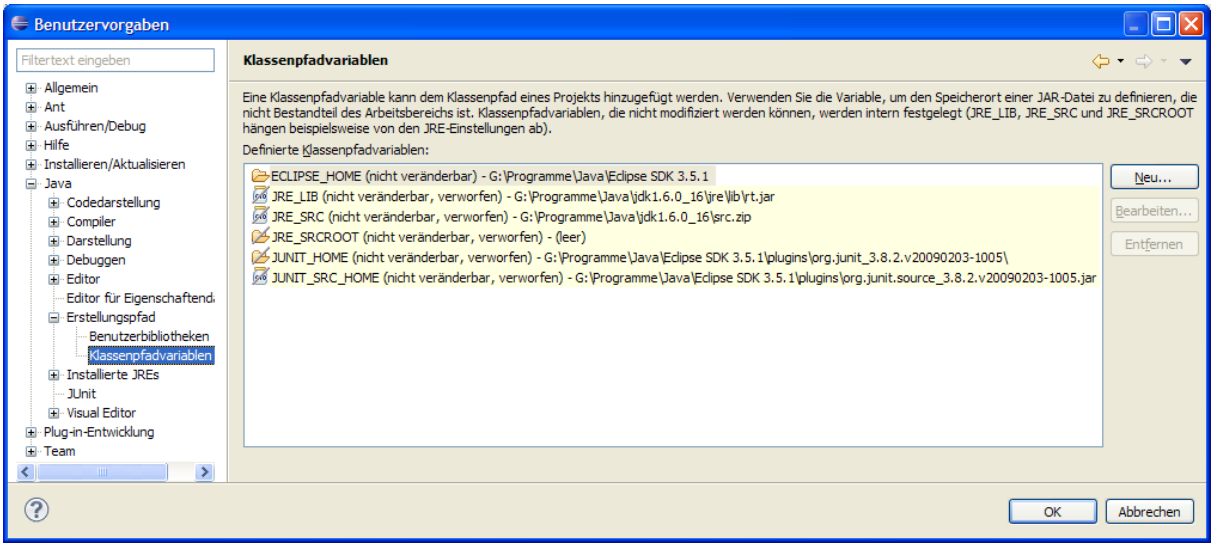
Damit beim Übersetzen durch den JDK-Compiler (**javac.exe**) und/oder beim Ausführen durch die JRE (**java.exe**) die `Simput`-Klasse mit ihren Methoden verfügbar ist, muss die Datei **Simput.class** entweder im aktuellen Verzeichnis liegen oder über die CLASSPATH-Umgebungsvariable (vgl. Abschnitt 2.2.4) auffindbar sein, wenn sie nicht bei jedem Compiler- oder Interpreteraufruf per **classpath**-Kommandozeilenoption 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 Abschnitt 7 mit Java-Archivdateien ausführlich beschäftigen.

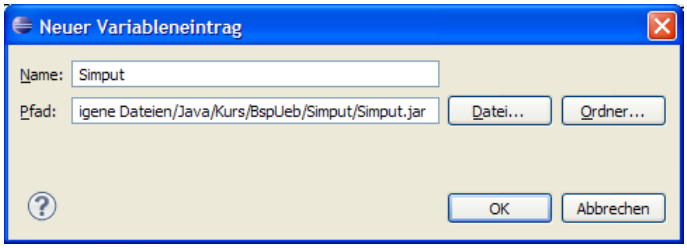
Namen und Pfad einer Archivdatei hinterlegt man am besten in einer **Klassenpfadvariablen** auf Arbeitsbereichsebene, damit das Archiv in einzelnen Projekten ohne Pfadangaben angesprochen werden kann. Nach dem Menübefehl

**Fenster > Benutzervorgaben > Java > Erstellungspfad > Klassenpfadvariablen**

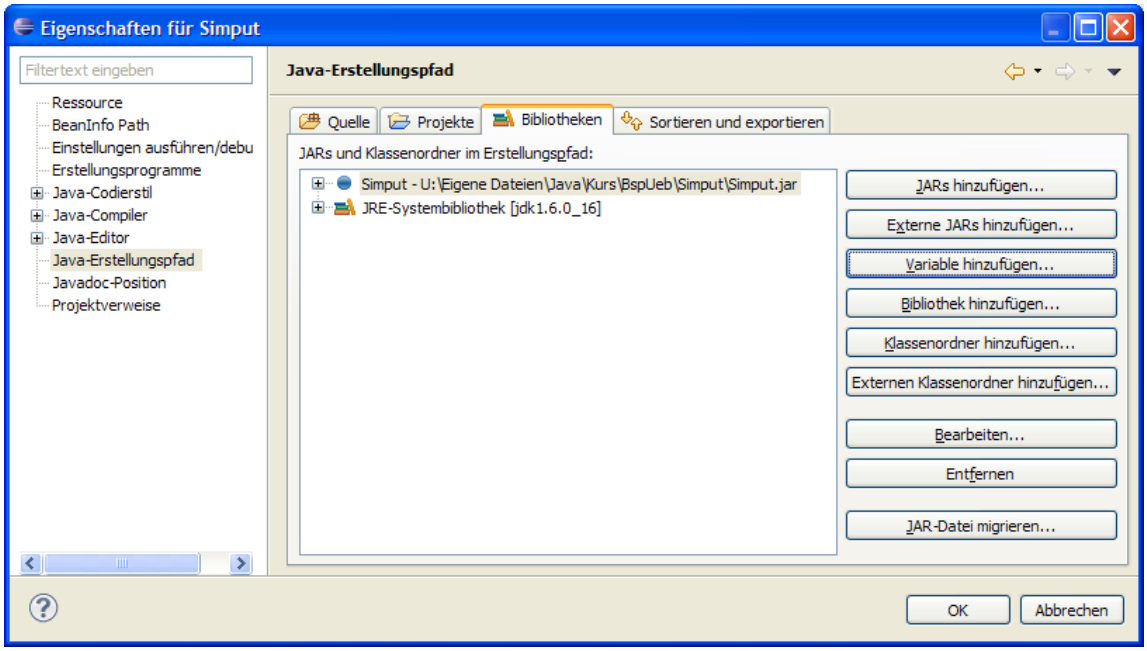
kann man in der folgenden Dialogbox




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



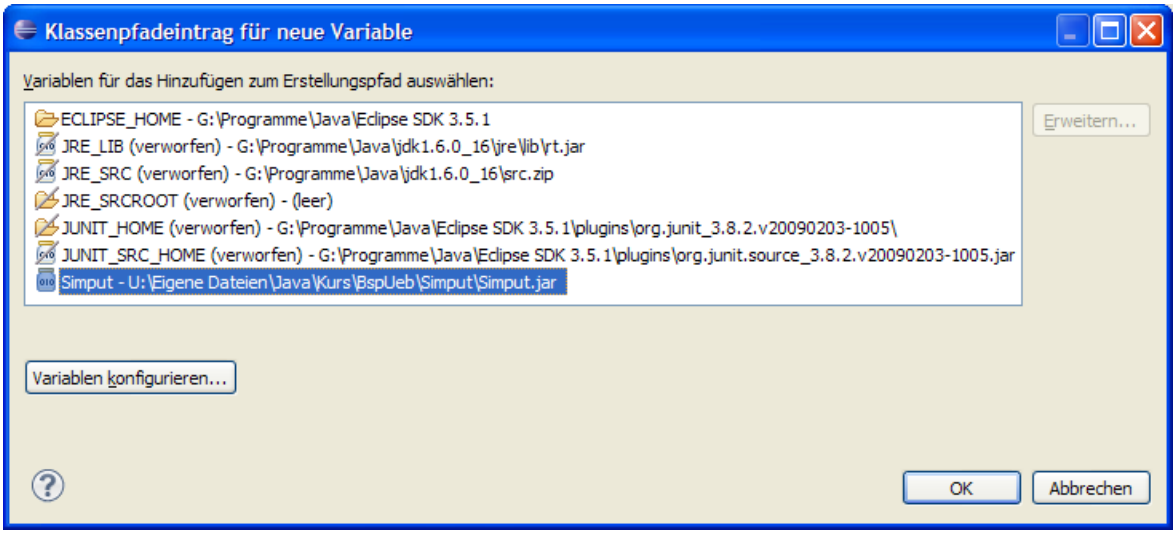
Soll ein konkretes Projekt die Klassenpfadvariable nutzen, muss diese im Eigenschaftsdialog des Projekts (z.B. erreichbar via Kontextmenü zum Projekteintrag im Paket-Explorer)




über

**Java-Erstellungspfad > Bibliotheken > Variable hinzufügen**

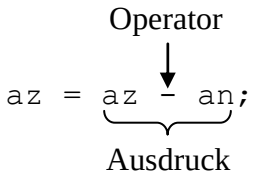
hinzugefügt werden, z.B.:




## *3.5 Operatoren und Ausdrücke*

Im Zusammenhang mit der Variablendeklaration und der Wertzuweisung haben wir das Sprachelement *Ausdruck* ohne Erklärung benutzt, und diese soll nun nachgeliefert werden. Im aktuellen Abschnitt 3.5 werden wir Ausdrücke als wichtige Bestandteile von Java-Anweisungen recht detailliert untersuchen. 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 oder Objekten realisiert.

Während die Variablen zur *Speicherung* von Werten dienen, geht es bei den **Operatoren** darum, aus vorhandenen Variableninhalten und/oder anderen Argumenten neue Werte zu berechnen. Den zur Berechnung eines Werts geeigneten, aus Operatoren und zugehörigen Argumenten aufgebauten Teil einer Anweisung, bezeichnet man als **Ausdruck**, z.B. in folgender Wertzuweisung:




Durch diese Anweisung aus der `kuerze()`-Methode unserer `Bruch`-Klasse (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.

Man kann einen Ausdruck als eine *temporäre* Variable mit einem **Datentyp** und einem **Wert** auffassen.

Schon bei einem Literal, einer Variablen oder einem Methodenaufruf haben wir es mit einem Ausdruck zu tun. Besteht ein Ausdruck aus einem Methodenaufruf mit dem Pseudorückgabetyp **void**, dann liegt allerdings *kein* Wert vor.

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.

Mit Hilfe der Operatoren entstehen komplexere Ausdrücke, wobei Typ und Wert von den Argumenten und den Operatoren 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 haben jedoch zusätzlich einen **Nebeneffekt** auf eine als Argument fungierende Variable.

Beispiel:
```
int i = 12;
int j = i++;
```
Im Beispiel hat der Ausdruck `i++` den Typ **int** und den Wert 12. Außerdem wird die Variable `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** bzw. **binär**.

Beispiel: `a + b`
Der Additionsoperator erwartet zwei numerische Argumente.

Manche Operatoren begnügen sich mit *einem* Argument und heißen daher **einstellig** bzw. **unär**.

Beispiel: `!cond`
Der Negationsoperator wird mit einem „!“ bezeichnet und erwartet *ein* Argument mit dem Typ **boolean**.

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

Weil die arithmetischen Operatoren für die vertrauten Grundrechenarten der Schulmathematik zuständig sind, müssen ihre Operanden (Argumente) einen 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 {`<br>`  public static void main(String[] args) {`<br>`  int i = 2, j = 3;`<br>`  double a = 2.0;`<br>`  System.out.printf("%10d\n", i/j);`<br>`  System.out.printf("%10.5f", a/j);`<br>`  }`<br>`}` | `0`<br>`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;`<br>`System.out.println(-i);` | `-2` |
| *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);`<br>`System.out.println(8 / 5);` | `1.6`<br>`1` |
| *Num1 % Num2* | **Modulo (Divisionsrest)**<br>Sei *GAD* der ganzzahlige Anteil aus dem Ergebnis der Division (*Num1* / *Num2*). Dann ist *Num1* % *Num2* def. durch<br>*Num1 - GAD · Num2* | `System.out.println(19 % 5);`<br>`System.out.printf("%-4.2f",`<br>`  -19 % 5.4);` | `4`<br>`-2,80` |
| *++Var*<br>*--Var* | **Präinkrement bzw. -dekrement**<br>Als Argumente sind hier nur Variablen erlaubt.<br>*++Var* liefert *Var* + 1 und erhöht *Var* um 1<br>*--Var* liefert *Var* - 1 und reduziert *Var* um 1 | `int i = 4;`<br>`double a = 0.2;`<br>`System.out.println(++i +"\n"+`<br>`  --a);` | `5`<br>`-0.8` |
| *Var++*<br>*Var--* | **Postinkrement bzw. -dekrement**<br>Als Argumente sind hier nur Variablen erlaubt.<br>*Var++* liefert *Var* und erhöht *Var* um 1<br>*Var--* liefert *Var* und reduziert *Var* um 1 | `System.out.println(i++ +"\n"+`<br>`  i);` | `4`<br>`5` |

Bei den Inkrement- bzw. Dekrementoperatoren ist zu beachten, dass sie *zwei* Effekte haben:

- Das Argument wird ausgelesen, um den Wert des Ausdrucks zu ermitteln.
- Die als Argument fungierende numerische Variable wird 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:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  int i = 12;`<br>`  i++;`<br>`  System.out.println(i);`<br>`  }`<br>`}` | `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;
```

### 3.5.2 Methodenaufrufe

Mit den arithmetischen Operatoren lassen sich nur elementare mathematische Probleme lösen. Darüber hinaus stellt Java eine große Zahl mathematischer Standardfunktionen (z.B. Potenzfunktion, Logarithmus, Wurzel, trigonometrische und hyperbolische Funktionen) über Methoden der Klasse **Math** im API-Paket **java.lang** zur Verfügung.[14] Im folgenden Programm wird die Methode **pow()** zur Potenzberechnung genutzt:

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

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.

### 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**)

[14] Mit den Paketen der Standardbibliothek werden wir uns später ausführlich beschäftigen. An dieser Stelle dient die Angabe der Paketzugehörigkeit nur 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.

| 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 mehr oder weniger unangenehmer Semantikfehler resultieren, also ein irreguläres Verhalten des Programms (vgl. Abschnitt 2.2.5 zur Unterscheidung von Syntax- und Semantikfehlern). Im ersten **println()**-Aufruf des folgenden Beispielprogramms wird das Ergebnis eines Vergleichs auf die Konsole geschrieben:[15]

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  int i = 1;`<br>`  System.out.println(i == 2);`<br>`  System.out.println(i);`<br>`  }`<br>`}` | `false`<br>`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 {`<br>`  public static void main(String[] args) {`<br>`  int i = 1;`<br>`  System.out.println(i = 2);`<br>`  System.out.println(i);`<br>`  }`<br>`}` | `2`<br>`2` |

Der versehentlich gebildete Ausdruck sorgt nicht nur für eine unerwartete Ausgabe, sondern verändert auch den Wert den 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.:

[15] 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 {`<br>`  public static void main(String[] args) {`<br>`  final double EPSILON = 1.0e-14;`<br>`  double d1 = 10.0 - 9.9;`<br>`  double d2 = 0.1;`<br>`  System.out.println(d1 == d2);`<br>`  System.out.println(Math.abs((d1 - d2)/d1) < EPSILON);`<br>`  }`<br>`}` | `false`<br>`true` |

Der naive Vergleich

```
10.0 - 9.9 == 0.1
```

führt trotz Datentyp **double** (mit mindestens 15 signifikanten Dezimalstellen) zum Ergebnis `false`. Wenn man die in Abschnitt 3.3.4.1 beschriebenen Genauigkeitsprobleme bei der Speicherung von binären Gleitkommazahlen berücksichtigt, ist das Vergleichsergebnis durchaus nicht überraschend.

Mit den Objekten der in Abschnitt 3.3.4 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.*;`<br>`class Prog {`<br>`  public static void main(String[] args) {`<br>`  BigDecimal bd1 = new BigDecimal("10.0");`<br>`  BigDecimal bd2 = new BigDecimal("9.9");`<br>`  BigDecimal bd3 = new BigDecimal("0.1");`<br>`  System.out.println(bd3.equals(bd1.subtract(bd2)));`<br>`  }`<br>`}` | `true` |

Allerdings ist mit einem erhöhten Speicher- und Zeitaufwand zu rechnen.

Sollen **double**-Werte mit möglichst großer Präzision verglichen werden, kann 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 $d_2$ 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 Wahl der Bezugsgröße $d_1$ oder $d_2$ für den Nenner ist beliebig. Um das Verfahren vollständig festzulegen, wird jedoch die Verwendung der betragsmäßig größeren Zahl vorgeschlagen.

Ein Begriff der numerischen Identität 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 **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 enthalten:

$$d_1 = (t + e_1)\ 10^k \text{ und } d_1 = (t + e_2)\ 10^k$$

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:

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

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

$$2^{-52} \approx 2{,}2 \cdot 10^{-16}$$

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 Identitätsprüfung mit Hilfe einer Schwelle für den relativen Abweichungsbetrag ist u.a. zu bemängeln, …

- dass noch Feinarbeit erforderlich ist, um eine Division durch Null zu verhindern,
- dass eine Verallgemeinerung für die mit geringerer Genauigkeit gespeicherten *denormalisierte* Werte (Betrag kleiner als $2^{-1022}$ beim Typ **double**, siehe Abschnitt 3.3.4.1) benötigt wird,
- dass die definierte Indifferenzrelation nicht transitiv ist.

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 Unterschiedschwelle 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<br>*La1* | Negation<br>!*La1* |
|---|---|
| **true** | **false** |
| **false** | **true** |

| Argument 1<br>*La1* | Argument 2<br>*La2* | Logisches UND<br>*La1* **&&** *La2*<br>*La1* **&** *La2* | Logisches ODER<br>*La1* **\|\|** *La2*<br>*La1* **\|** *La2* | Exklusives ODER<br>*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**<br>Der Wahrheitswert wird umgekehrt. | `boolean erg = true;`<br>`System.out.println(!erg);` | `false` |
| *La1* && *La2* | **Logisches UND (mit bedingter Auswertung)**<br>*La1* && *La2* ist genau dann wahr, wenn beide Argumente wahr sind. Ist *La1* falsch, wird *La2* nicht ausgewertet. | `int i = 3;`<br>`boolean erg = false && i++ > 3;`<br>`System.out.println(erg + "\n"+i);`<br><br>`erg = true && i++ > 3;`<br>`System.out.println(erg + "\n"+i);` | `false`<br>`3`<br><br>`false`<br>`4` |
| *La1* & *La2* | **Logisches UND (mit *un*bedingter Auswertung)**<br>*La1* & *La2* ist genau dann wahr, wenn beide Argumente wahr sind. Es werden auf jeden Fall *beide* Ausdrücke ausgewertet. | `int i = 3;`<br>`boolean erg = false & i++ > 3;`<br>`System.out.println(erg + "\n"+i);` | `false`<br>`4` |
| *La1* \|\| *La2* | **Logisches ODER (mit bedingter Auswertung)**<br>*La1* \|\| *La2* ist genau dann wahr, wenn mindestens ein Argument wahr ist. Ist *La1* wahr, wird *La2* nicht ausgewertet. | `int i = 3;`<br>`boolean erg = true \|\| i++ == 3;`<br>`System.out.println(erg + "\n"+i);`<br><br>`erg = false \|\| i++ == 3;`<br>`System.out.println(erg + "\n"+i);` | `true`<br>`3`<br><br>`true`<br>`4` |
| *La1* \| *La2* | **Logisches ODER (mit unbedingter Auswertung)**<br>*La1* \| *La2* ist genau dann wahr, wenn mindestens ein Argument wahr ist. Es werden auf jeden Fall *beide* Ausdrücke ausgewertet. | `int i = 3;`<br>`boolean erg = true \| i++ == 3;`<br>`System.out.println(erg + "\n"+i);` | `true`<br>`4` |

| Operator | Bedeutung | Beispiel | |
|---|---|---|---|
| | | Programmfragment | Ausgabe |
| *La1* ^ *La2* | **Exklusives logisches ODER**<br>*La1* ^ *La2* ist genau dann wahr, wenn genau ein Argument wahr ist, wenn also die Argumente verschiedene Wahrheitswerte haben. | `boolean erg = true ^ true;`<br>`System.out.println(erg);` | `false` |

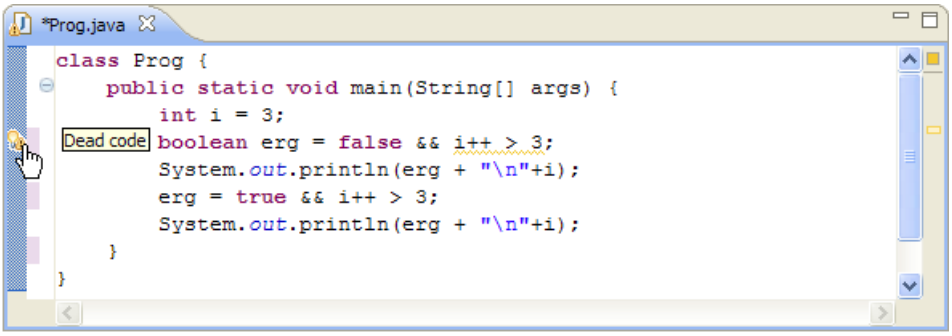
Der Unterschied zwischen den beiden logischen UND-Operatoren **&&** und **&** bzw. zwischen den beiden logischen ODER-Operatoren || und | ist für Einsteiger vielleicht etwas unklar, weil man spontan den nicht ausgewerteten logischen Ausdrücken keine Bedeutung beimisst. Allerdings ist es in Java nicht unüblich, „Nebeneffekte“ in einen logischen Ausdruck einzubauen, z.B.

```
a == b & 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 unterlassen, wenn bereits nach Auswertung des linken UND-Arguments das Gesamtergebnis **false** feststeht:

```
a == b && i++ > 3
```

Das vom Programmierer nicht erwartete Ausbleiben einer Auswertung (z.B. bei „`i++`“) kann erhebliche Auswertungen auf ein Programm haben. Daher warnt Eclipse vor totem (und eventuell tödlichem) Code:




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

Wie der Tabelle auf Seite 94 zu entnehmen ist, unterscheiden sich die beiden UND-Operatoren **&&** und **&** bzw. die beiden ODER-Operatoren || und | auch hinsichtlich der Auswertungspriorität.

Um die Verwirrung noch ein wenig zu steigern, werden die Zeichen **&** und | auch für *bitorientierte* Operatoren verwendet (siehe Abschnitt 3.5.6). Weil diese Operatoren zwei *integrale* Argumente (z.B. Datentyp **int**) erwarten, können Compiler und Programmierer allerdings 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 Manipulation von Variableninhalten. Statt einer systematischen Darstellung der verschiedenen Operatoren (siehe z.B. Java-Tutorial, Sun Microsystems (2009) 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 danach steht, können Sie den aktuellen Abschnitt ohne Sorge um den weiteren Kurserfolg an dieser Stelle verlassen.

Das 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 {`<br>`  public static void main(String[] args) {`<br>`  char cbit;`<br>`  System.out.print("Zeichen:  ");`<br>`  cbit = Simput.gchar();`<br>`  System.out.print("Unicode:  ");`<br>`  for(int i = 15; i >= 0; i--) {`<br>`  if ((1 << i & cbit) != 0)`<br>`  System.out.print("1");`<br>`  else`<br>`  System.out.print("0");`<br>`  }`<br>`  System.out.println("\nint-Wert: " + (int)cbit);`<br>`  }`<br>`}` | `Zeichen:  x`<br>`Unicode:  0000000001111000`<br>`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 auf der rechten Seite Nullen nachrücken. Von den 32 Bit, die ein **int**-Wert insgesamt belegt (siehe Abschnitt 3.3.3), interessieren im Augenblick nur die rechten 16. Bei der Eins 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 *i* eine Eins enthält, wenn *beide* Argumentmuster an dieser Stelle eine Eins 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 im *i*-ten Schleifendurchgang genau dann wahr, wenn das korrespondierende Bit in der Binärdarstellung des untersuchten Zeichens den Wert Eins hat.

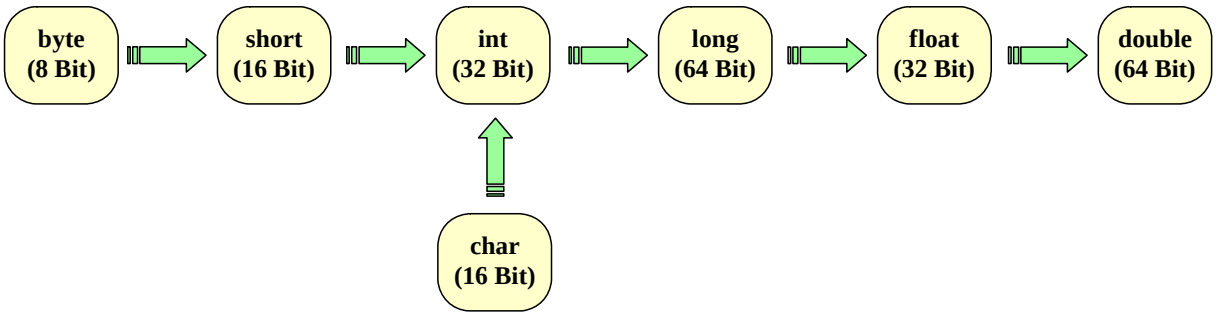
### 3.5.7 Typumwandlung (Casting) bei primitiven Datentypen

Beim Auswerten 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 die Gleitkommaarithmetik zum Einsatz kommt. Dabei wird für das **int**-Argument eine **automatische (implizite) Wandlung** in den Datentyp **double** vorgenommen.

Java nimmt bei Bedarf für primitive Datentypen die folgenden **erweiternden Konvertierungen** automatisch vor:




Bei den Konvertierungen von **int** oder **long** in **float** sowie von **long** in **double** kann es zu einem Verlust an Genauigkeit kommen, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  long i = 9223372036854775313L;`<br>`  double d = i;`<br>`  System.out.println(i);`<br>`  System.out.printf("%.2f\n", d);`<br>`  }`<br>`}` | `9223372036854775313`<br>`9223372036854776000,00` |

Eine Abweichung von 687 (z.B. Euro oder Meter) kann durchaus Proteste von Kunden oder Schlimmeres zur Folge haben.

Weil eine **char**-Variable die Unicode-Nummer eines Zeichens speichert, macht die Konvertierung in numerische Typen kein Problem, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  System.out.printf("x/2  = %5d", 'x'/2);`<br>`  System.out.printf("\nx*0,27 = %5.2f", 'x'*0.27);`<br>`  }`<br>`}` | `x/2  =  60`<br>`x*0,27 = 32,40` |

Gelegentlich gibt es gute Gründe, über den **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 letzte Programm beim „Halbieren“ dieses Zeichens auf den Wert 60 kam:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  double a = 3.14159;`<br>`  System.out.println((int) 'x');`<br>`  System.out.println((int) a);`<br>`  System.out.println((int) 1.45367e50);`<br>`  }`<br>`}` | `120`<br>`3`<br>`2147483647` |

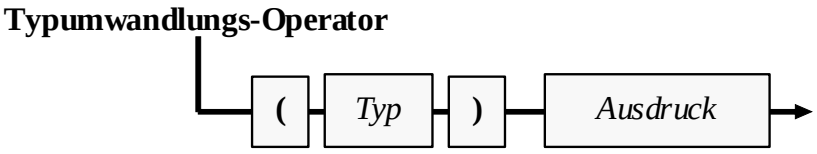
In der zweiten Ausgabeanweisung des letzten Beispielprogramms wird per Casting-Operation der ganzzahlige Anteil eines **double**-Wertes ermittelt, was im Programmieralltag gelegentlich sinnvoll ist. Wie die dritte Ausgabe zeigt, sind bei einer explizit angeforderten **einschränkenden Konvertierung** kapitale Programmierfehler möglich, wenn die Wertebereiche der beteiligten Variablen bzw. Datentypen nicht beachtet werden. So soll die Explosion der europäischen Weltraumrakete 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.

Während sich Java-Compiler grundsätzlich weigern, ein **double**-Literal in einer **float**-Variablen zu speichern, erlauben sie z.B. das Speichern eines **int**-Literals in einer **byte**-Variablen, sofern deren Wertebereich nicht verlassen wird, z.B.:

```
float f = 3.14;
byte b = 25;
```

Die Java-Syntax zur expliziten Typumwandlung:




### 3.5.8 Zuweisungsoperatoren

Bei den ersten Erläuterungen zu Wertzuweisungen (vgl. Abschnitt 3.3.5) 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. Dekrementoperators 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 {`<br>`  public static void main(String[] args) {`<br>`  int ivar = 13;`<br>`  System.out.println(ivar = 4711);`<br>`  System.out.println(ivar);`<br>`  }`<br>`}` | `4711`<br>`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 {`<br>`  public static void main(String[] args) {`<br>`  int i = 2, j = 4;`<br>`  i = j = j * i;`<br>`  System.out.println(i + "\n" + j);`<br>`  }`<br>`}` | `8`<br>`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 Priorität 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.

Ausdrücke der Art

```
i = j = k;
```

stammen übrigens *nicht* aus einem Kuriositätenkabinett, sondern sind in Java - Programmen oft anzutreffen.

Wie wir seit Abschnitt 3.3.5 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** 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 typkompatiblen Ausdruck:

| Operator | Bedeutung | Beispiel | |
|---|---|---|---|
| | | Programmfragment | Neuer Wert von `i` |
| *Var* **+=** *Expr* | *Var* erhält den neuen Wert *Var* + *Expr*. | `int i = 2;`<br>`i += 3;` | 5 |
| *Var* **-=** *Expr* | *Var* erhält den neuen Wert *Var* − *Expr*. | `int i = 10, j = 3;`<br>`i -= j * j;` | 1 |
| *Var* ***=** *Expr* | *Var* erhält den neuen Wert *Var* * *Expr*. | `int i = 2;`<br>`i *= 5;` | 10 |
| *Var* **/=** *Expr* | *Var* erhält den neuen Wert *Var* / *Expr*. | `int i = 10;`<br>`i /= 5;` | 2 |
| *Var* **%=** *Expr* | *Var* erhält den neuen Wert *Var* % *Expr*. | `int i = 10;`<br>`i %= 5;` | 0 |

Während für zwei **byte**-Variablen

```
byte b1 = 1, b2 = 2;
```

die folgende Zuweisung

```
b1 = b1 + b2;
```

verboten ist, weil der Ausdruck `(b1 + b2)` den Typ **int** besitzt (vgl. Tabelle mit den Ergebnistypen der Ganzzahlarithmetik in Abschnitt 3.5.1), akzeptiert der Compiler den äquivalenten Ausdruck mit Aktualisierungsoperator:

```
b1 += b2;
```

### 3.5.9 Konditionaloperator

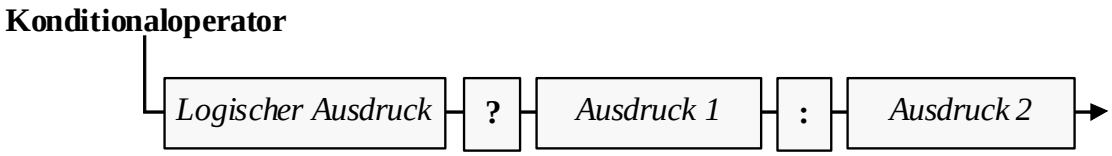
Der **Konditionaloperator** erlaubt eine sehr kompakte Schreibweise, wenn beim neuen Wert einer Zielvariablen bedingungsabhängig zwischen zwei Ausdrücken zu entscheiden ist, z.B.

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

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

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  int i = 2, j = 1, k = 7;`<br>`  i = k>0 ? i+j : j-j;`<br>`  System.out.println(i);`<br>`  }`<br>`}` | 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*.

### 3.5.10 Auswertungsreihenfolge

Bisher haben wir zusammengesetzte Ausdrücke mit *mehreren* Operatoren und das damit verbundene Problem der Auswertungs*reihenfolge* nach Möglichkeit gemieden. Nun werden die Regeln vorgestellt, nach denen ein Java-Compiler komplexe Ausdrücke mit mehreren Operatoren auswertet:

**1) Priorität**
Zunächst entscheidet die Priorität der Operatoren (siehe Tabelle unten) darüber, in welcher Reihenfolge die Auswertung vorgenommen wird. Z.B. hält sich Java bei arithmetischen Ausdrücken an die mathematische Regel

*Punktrechnung geht vor Strichrechnung.*

**2) Assoziativität (Auswertungsrichtung)**
Stehen mehrere Operatoren gleicher Priorität zur Auswertung an, dann entscheidet die Assoziativität der Operatoren über die Reihenfolge der Auswertung:

- Mit Ausnahme der Zuweisungsoperatoren sind alle binären Operatoren *links*-assoziativ; sie werden also von links nach rechts ausgewertet. Z.B. wird

  ```
  x - y - z
  ```

  ausgewertet als

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

- Die Zuweisungsoperatoren und der Konditionaloperator sind *rechts*-assoziativ; sie werden also von rechts nach links ausgewertet. Z.B. wird

  ```
  x = y = z
  ```

  ausgewertet als

  ```
  x = (y = z)
  ```

Für manche Operationen gilt das mathematische Assoziativitätsgesetz, so dass die Reihenfolge der Auswertung irrelevant ist, z.B.:

$(3 + 2) + 1 = 6 \;=\; 3 + (2 + 1)$

Anderen Operationen fehlt diese Eigenschaft, z.B.:

$(3 - 2) - 1 = 0 \neq 3 - (2 - 1) = 2$

**3) Klammern**
Wenn aus obigen Regeln nicht die gewünschte Auswertungsfolge resultiert, greift man mit runden Klammern steuernd ein. Die Auswertung von (eventuell mehrstufig) eingeklammerten Teilausdrücken erfolgt von innen nach außen.

In der folgenden Tabelle sind die bisher behandelten Operatoren in absteigender Priorität aufgelistet. Gruppen von Operatoren mit gleicher Priorität sind durch fette 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 |
| *B* | Ausdruck mit beliebigem kompatiblem Datentyp |
| *S* | **String** (Zeichenfolge) |
| *V* | Variable mit beliebigem kompatiblem Datentyp |
| $V_n$ | Variable mit numerischem Datentyp (**byte**, **short**, **int**, **long**, **char**, **float**, **double**) |

| Operator | Bedeutung | Operanden |
|---|---|---|
| () | Methodenaufruf | |
| ! | Negation | *L* |
| ++, -- | Prä- oder Postinkrement bzw. -dekrement | $V_n$ |
| - | Vorzeichenumkehr | *N* |
| (*Typ*) | Typumwandlung | *B* |
| *, / | Punktrechnung | *N*, *N* |
| % | Modulo | *N*, *N* |
| +, - | Strichrechnung | *N*, *N* |
| + | Stringverkettung | *S*, *B* oder *B*, *S* |
| <<, >> | Links- bzw. Rechts-Shift | *I*, *I* |
| >, <, >=, <= | Vergleichsoperatoren | *N*, *N* |
| ==, != | Gleichheit, Ungleichheit | *B*, *B* |
| & | 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*, *B*, *B* |
| = | Wertzuweisung | *V*, *B* |
| +=, -=, *=, /=, %= | Wertzuweisung mit Aktualisierung | $V_n$, *N* |

Im Anhang finden Sie eine erweiterte Version dieser Tabelle, die zusätzlich alle Operatoren enthält, die im weiteren Verlauf des Kurses noch behandelt werden.

### *3.6 Über- und Unterlauf bei numerischen Variablen*

Wie Sie inzwischen wissen, haben die primitiven Datentypen für ganze und rationale Zahlen jeweils einen bestimmten Wertebereich (siehe Tabelle in Abschnitt 3.3.3). 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. In einem solchen Fall 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 3.3.4.2 demonstriert, die Klasse **BigDecimal** bei finanzmathematischen Anwendungen wegen ihrer beliebigen Genauigkeit zu bevorzugen ist.

### 3.6.1 Überlauf bei Ganzzahltypen

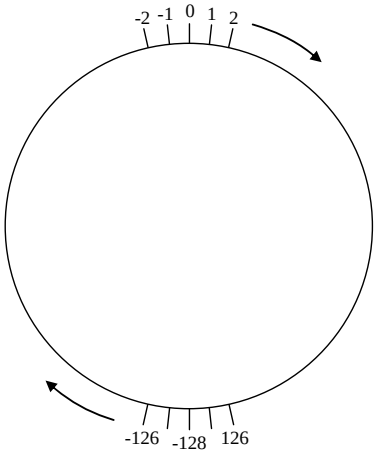
Ohne besondere Vorkehrungen stellt ein Java-Programm im Falle eines Ganzzahlüberlaufs keinesfalls seine Tätigkeit (z.B. mit einem Ausnahmefehler) ein, sondern arbeitet munter weiter. Dieses Verhalten ist beim Programmieren von Zufallszahlgeneratoren willkommen, ansonsten aber eher bedenklich. Das folgende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

Die Werte der Ganzzahltypen sind nach dem *Zweierkomplementprinzip* auf einem Zahlenkreis angeordnet, und nach der größten positiven Zahl beginnt der Bereich der negativen Zahlen (mit abnehmendem Betrag), z.B. beim Typ **byte**:




Speziell bei der Steuerung von Raketenmotoren (vgl. Abschnitt 3.5.7) ist also Vorsicht geboten, weil ansonsten das Kommando „Mr. Spock[16], please push the engine.“ zum heftigen Rückwärtsschub führen könnte.

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:

---

[16] Chefingenieur auf dem Raumschiff Enterprise

```
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 zunächst der Ausdruck `i+j` berechnet.
- 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.

Ein Entsprechung zur **checked**-Option in C# (siehe Baltes-Götz 2009) steht in Java leider noch nicht zur Verfügung.

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

```
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 $2^{63}$ und quadriert diese auch noch mutig:

```
2 hoch 126 = 85070591730234615865843651857942052864
```

### 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 der unterstützte Wertebereich hier weit größer ist. Dabei kommt es aber weder zu einem sinnlosen Zufallswert noch zu einem Ausnahmefehler, 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
```

Mit Hilfe der Unendlich-Werte „gelingt“ offenbar bei der Gleitkommaarithmetik sogar die Division durch Null, während bei der Ganzzahlarithmetik ein solcher Versuch zum Laufzeitfehler führt.

Bei diesen „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 folgende Programm 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 Ausgabe:

```
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

- **isInfinite()**
- **isNaN()**

mit Rückgabetyp **boolean** lässt sich für eine **double**-Variable prüfen, ob sie einen unendlichen oder undefinierten Wert besitzt, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  System.out.println(Double.isInfinite(1.0/0.0));`<br>`  System.out.print(Double.isNaN(0.0/0.0));`<br>`  }`<br>`}` | `true`<br>`true` |

Für besonders neugierige Leser sollen abschließend noch die **float**-Darstellungen der speziellen Gleitkommawerte angegeben werden (vgl. Abschnitt 3.3.4.1):

| 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

```
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);
   }
}
```

Ausgabe

```
Very Big: 1.057066e+3000
```

Ein Überlauf ist bei **BigDecimal**-Objekten nicht zu befürchten. Allerdings sind maximal

$$2^{31} - 1 = 2147483647$$

Dezimalstellen erlaubt, so dass mit einiger Anstrengung ein Laufzeitfehler provoziert werden kann.

### 3.6.3 Unterlauf bei den Gleitkommatypen

Bei den Gleitkommatypen **float** und **double** ist auch ein **Unterlauf** möglich, wobei eine Zahl mit sehr kleinem Betrag nicht mehr dargestellt werden kann. In diesem Fall rechnet ein Java-Programm mit dem Wert 0,0 weiter, was in der Regel akzeptabel ist, z.B.:

Quellcode

```
class Prog {
   public static void main(String[] args) {
      double smalld = Double.MIN_VALUE;
      System.out.println(smalld);
      smalld /= 2.0;
      System.out.println(smalld);
   }
}
```

Ausgabe

```
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 3.3.4.1 zu denormalisierten Werten bei den binären Gleitkommatypen **float** und **double**.

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

```
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);
   }
}
```

Ausgabe

```
9.881312916824932
0.0
```

Das Ergebnis des Ausdrucks

```
a * b * c
```

wird halbwegs korrekt ermittelt (vgl. Abschnitt 3.3.4.1 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, z.B.:

```
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");
      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 ohne Unterlauf und ohne Genauigkeitsproblem (siehe oben). Neben dem leicht zu verschmerzenden Schreibaufwand entsteht durch die Verwendung von **BigDecimal**-Objekten aber auch ein erhöhter Rechenaufwand (siehe Abschnitt 3.3.4.2), so dass die binären Gleitkommatypen in vielen Situationen die erste Wahl bleiben.

### 3.6.4 Der Modifikator strictfp

In der Norm IEEE-754 für 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 (2005, S. 411) 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 betroffenen 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 6.0 ist es mir auf einem Rechner mit Intel-CPU (Pentium 4, 3 GHz) unter Windows XP *nicht* gelungen, einen Effekt des **strictfp**-Modifikators zu beobachten. Das von Sun Microsystems auf einer Webseite für Entwickler[17] angebotene Beispielprogramm

[17] http://java.sun.com/developer/JDCTechTips/2001/tt0410.html

```
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.

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 ein Beispiel für die Verwendung des Methoden-Modifikators **strictfp**:

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

## *3.7 Anweisungen (zur Ablaufsteuerung)*

Wir haben uns im Abschnitt 3 über elementare Sprachelemente zunächst mit (lokalen) **Variablen** und primitiven **Datentypen** vertraut gemacht. Dann haben wir gelernt, aus Variablen, Literalen und Methodenaufrufen mit Hilfe von **Operatoren** mehr oder weniger komplexe **Ausdrücke** zu bilden. Diese wurden entweder mit Hilfe des Objekts **System.out** 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:

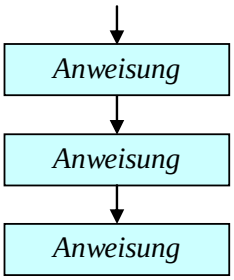
- **Variablendeklarationsanweisung**
  Die Variablendeklarationsanweisung wurde schon in Abschnitt 3.3.5 eingeführt.
  Beispiel: `int i = 1, j = 2, k;`
- **Ausdrucksanweisungen**
  Folgende Ausdrücke werden zu Anweisungen, sobald man ein Semikolon dahinter setzt:
  - **Wertzuweisung** (vgl. Abschnitte 3.3.5 und 3.5.8)
    Beispiel: `k = i + j;`

- **Prä- bzw. Postinkrement- oder -dekrementoperation**
  Beispiel: `i++;`
  Hier ist nur der „Nebeneffekt“ des Ausdrucks `i++` von Bedeutung. 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 **Verbund- bzw. Blockanweisung.** Wir haben uns bereits in Abschnitt 3.3.6 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 ja sehr unpraktisch, dieselbe Bedingung für jede betroffene Anweisung wiederholen zu müssen.

- **Anweisungen zur Ablaufsteuerung**
  Die **main()**-Methoden der bisherigen Beispielprogramme in Abschnitt 3 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.

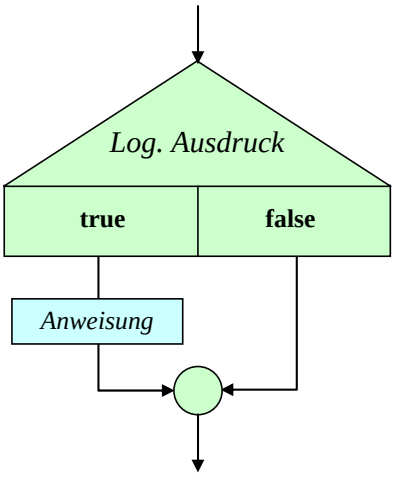
Anweisungen werden durch ein **Semikolon** abgeschlossen, sofern sie nicht mit einer schließenden Blockklammer enden.

### 3.7.2 Bedingte Anweisung und Verzweigung

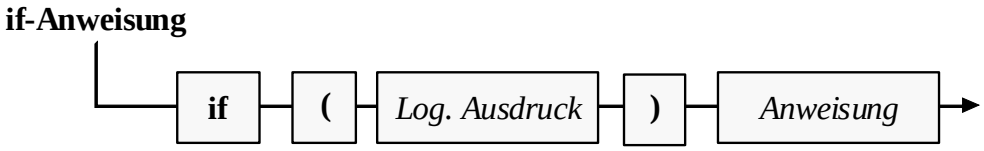
Oft ist es erforderlich, dass eine Anweisung nur unter einer bestimmten Bedingung ausgeführt wird. Etwas allgemeiner formuliert geht es darum, dass viele Algorithmen *Fallunterscheidungen* benötigen, also an bestimmten Stellen in Abhängigkeit vom Wert eines steuernden Ausdrucks in unterschiedliche Pfade verzweigen müssen.

#### *3.7.2.1 if-Anweisung*

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




Zur Realisation verwendet man die **if**-Anweisung mit der folgenden Syntax:




Als Anweisung ist allerdings keine Variablendeklaration erlaubt. Im folgenden Beispiel wird eine Meldung ausgegeben, wenn die Variable `anz` den Wert Null besitzt:

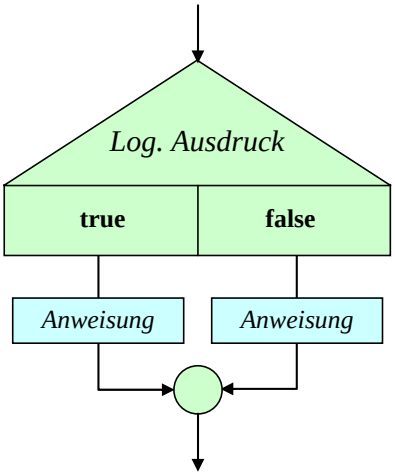
```
if (anz == 0)
  System.out.println("Die Anzahl muss > 0 sein!");
```

Der Zeilenumbruch zwischen dem logischen Ausdruck und der (Unter-)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 durch *kursive* Schrift gekennzeichnet.

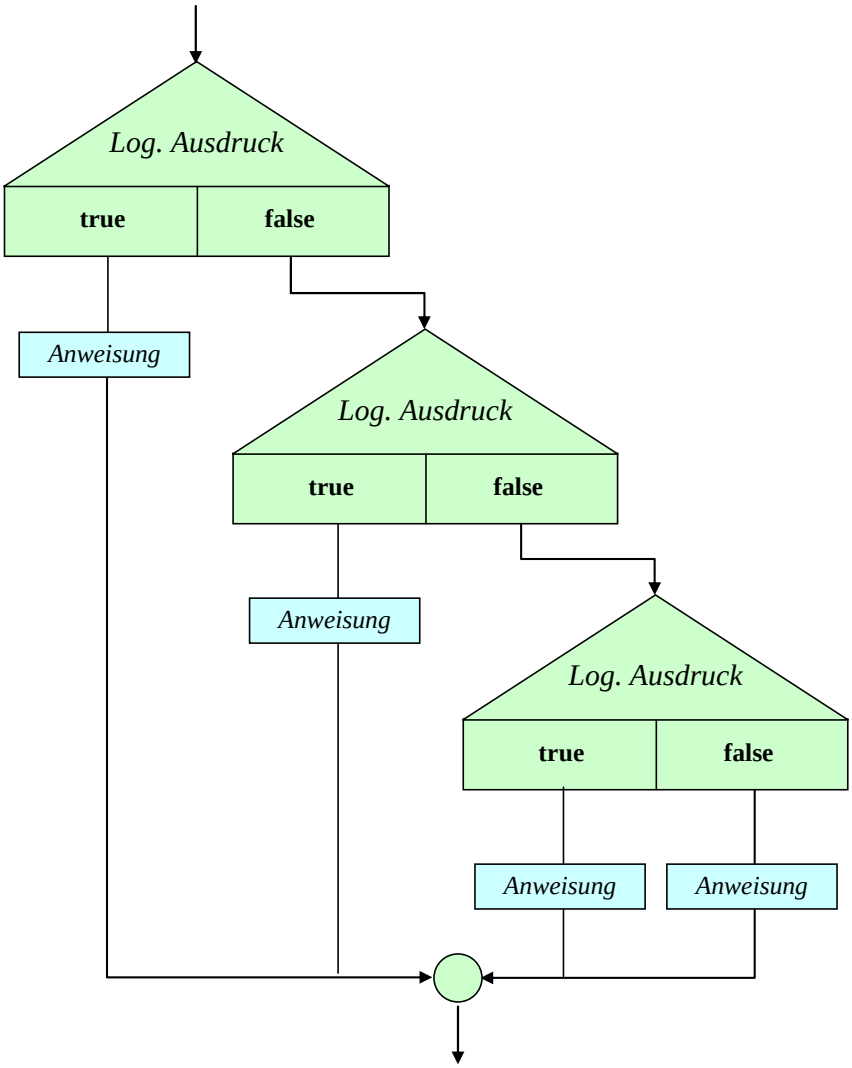
Während die Syntaxbeschreibung im Quellcode-Layout sehr übersichtlich ist, bietet das Syntaxdiagramm den Vorteil, bei komplizierter, variantenreicher Syntax alle zulässigen Formulierungen kompakt und präzise als Pfade durch das Diagramm zu beschreiben.

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 mit Alarmton (Unicode-Escape-Sequenz **\u0007**, vgl. Abschnitt 3.3.8.3). Das Argument wird vom Benutzer über die `Simput`-Methode `gdouble()` erfragt (vgl. Abschnitt 3.4).

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  System.out.print("Argument: ");`<br>`  double arg = Simput.gdouble();`<br>`  if (arg > 0)`<br>`  System.out.println("ln("+arg+") = "`<br>`  +Math.log(arg));`<br>`  else`<br>`  System.out.println("\u0007Argument <= 0!");`<br>`  }`<br>`}` | `Argument: 2`<br>`ln(2.0) = 0.6931471805599453` |

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_t
else
    Anweisung k_f
```

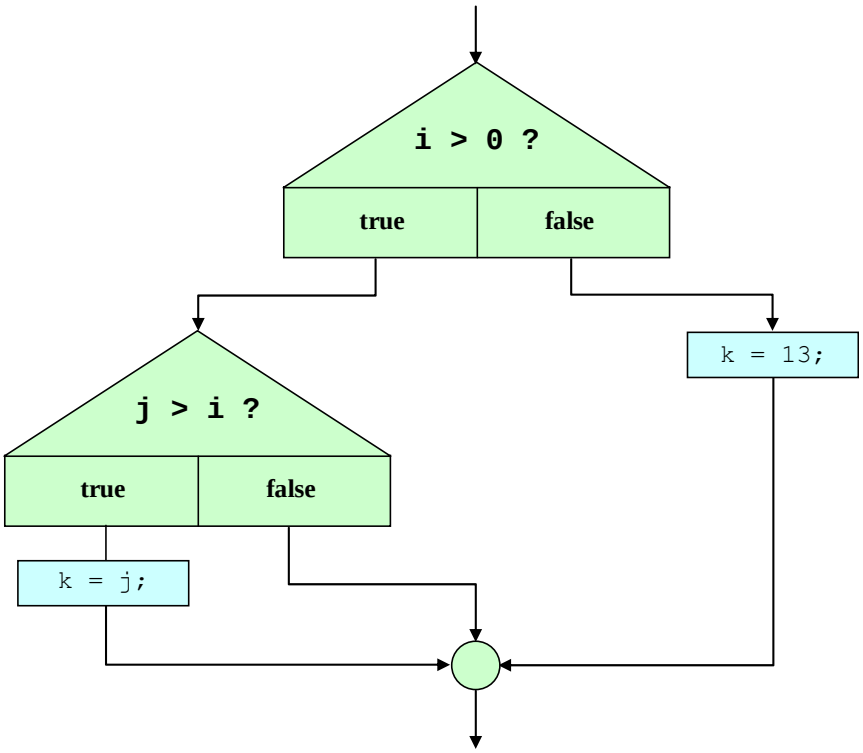
Wenn alle logischen Ausdrücke den Wert **false** annehmen, dann wird die **else**-Klausel zur letzten **if**-Anweisung ausgeführt.

Beim Schachteln von bedingten Anweisungen kann es zum genannten **dangling-else** - Problem[18] kommen, wobei ein Missverständnis zwischen Compiler und Programmierer hinsichtlich der Zuordnung einer **else**-Klausel besteht. Im folgenden Code-Fragment

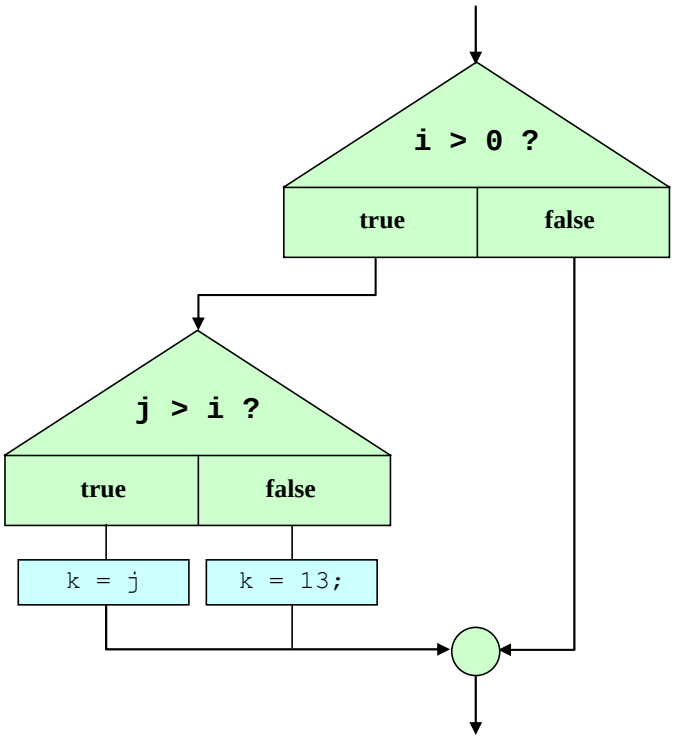
[18] Deutsche Übersetzung von *dangling*: *baumelnd*.

```
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:



Bei `i` ≤ 0 behält die Variable `k` ihren alten Wert, während der Programmierer den neuen Wert 13 erwartet.

Mit Hilfe von Blockklammern kann man die gewünschte Zuordnung erzwingen:

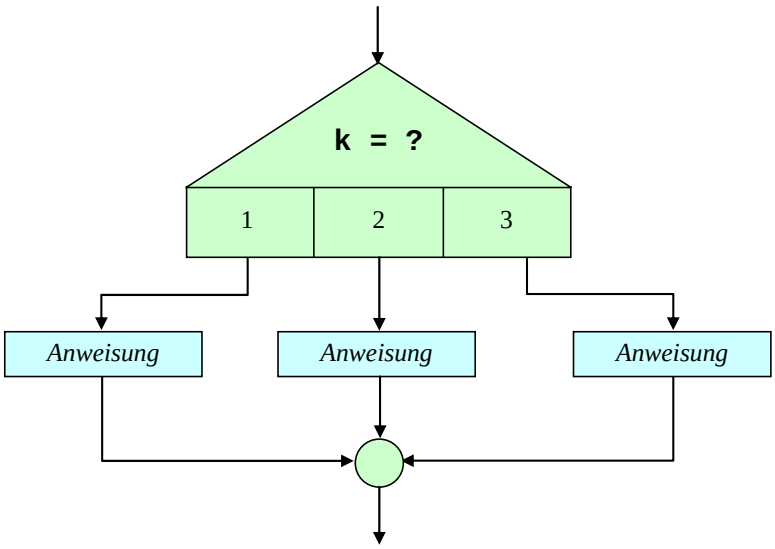
```
if (i > 0)
  {if (j > i)
    k = j;}
else
  k = 13;
```

Alternativ könnte man auch dem zweiten **if** eine **else**-Klausel spendieren und dabei eine leere Anweisung verwenden:

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

### *3.7.2.3 switch-Anweisung*

Wenn eine Fallunterscheidung mit mehr als zwei Alternativen in Abhängigkeit vom Wert *eines* Ausdrucks vorgenommen werden soll,

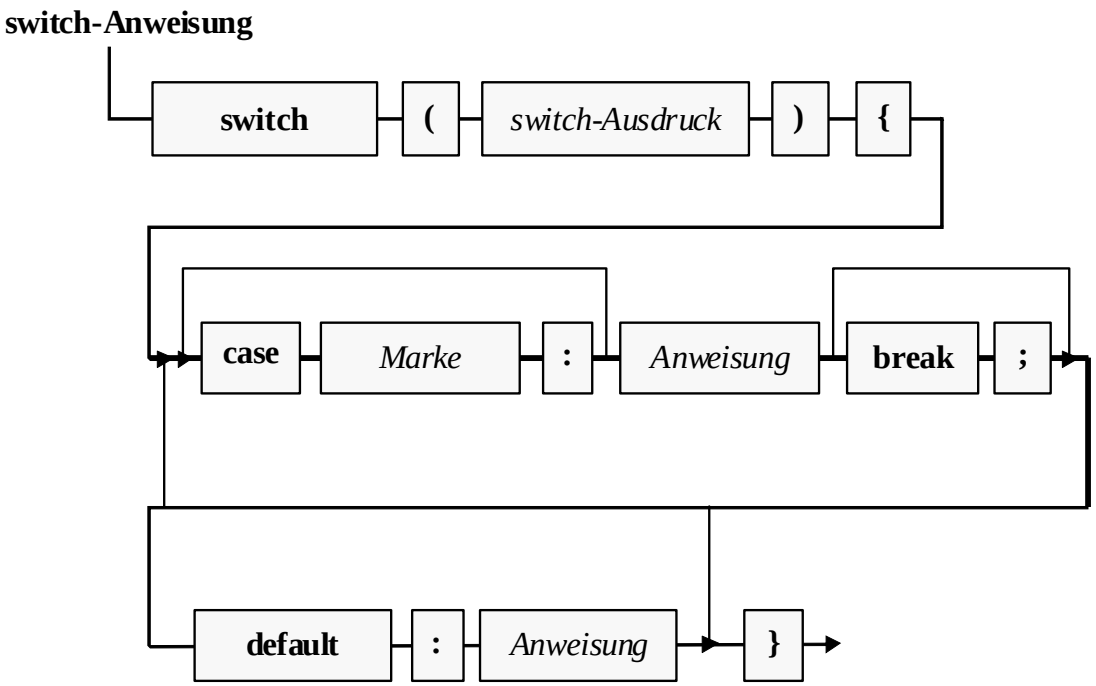



dann ist eine **switch**-Anweisung weitaus handlicher als eine verschachtelte **if-else** - Konstruktion.

Beim steuernden **switch**-Ausdruck erlaubt Java die folgenden Datentypen, wobei es sich letztlich um integrale (ganzzahlige) Datentypen handelt:

- Integrale primitive Datentypen:
  **byte**, **short**, **char** oder **int** (nicht **long**!)
- Verpackungsklassen (siehe unten) für integrale primitive Datentypen:
  **Byte**, **Short**, **Character** oder **Integer** (nicht **Long**!)
- Aufzählungstypen (siehe unten)

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:

```
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: System.out.println("Fall 3");
        break;
      default: System.out.println("Restkategorie");
        break;
    }
  }
}
```

Als **case**-Marken sind *konstante* Ausdrücke erlaubt, deren Wert schon der Compiler ermitteln kann (z.B. Literale, Konstanten oder mit konstanten Argumenten gebildete Ausdrücke). 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 jedoch 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 sehr elegant 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 *z*) 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 fest steht, wird bei den restlichen Farben auch die Lieblingszahl berücksichtigt:

```
class PerST {
   public static void main(String[] args) {
      char farbe = args[0].charAt(0);
      int zahl = Integer.parseInt(args[1]);
      switch (farbe) {
         case 'r': System.out.println("Sie sind ein emotionaler Typ."); break;
         case 'g':
         case 'b': {
                   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.");
                   }
                   break;
         case 's': System.out.println("Nehmen Sie nicht alles so tragisch."); break;
         default:  System.out.println("Offenbar mangelt es Ihnen an Disziplin.");
      }
   }
}
```

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 aus einer Palette mit den drei Grundfarben und Schwarz sowie ihre Lieblingszahl über Programmargumente (Kommandozeilenparameter) angeben, wobei die Farbe durch einen Buchstaben zu kodieren ist:

r für Rot
g für Grün
b für Blau
s für Schwarz

Wer z.B. die Farbe Blau und die Zahl 17 bevorzugt, sollte das Programm also folgendermaßen starten:

```
java PerST b 17
```

Im Programm wird jeweils nur eine Anweisung benötigt, um die Programmargumente in eine **char**- bzw. **int**-Variable zu befördern. Die zugehörigen Erklärungen werden Sie mit Leichtigkeit verstehen, sobald Methodenparameter sowie Arrays und Zeichenketten behandelt worden sind. An dieser Stelle greifen wir späteren Erläuterungen mal wieder etwas vor (hoffentlich mit motivierendem Effekt):

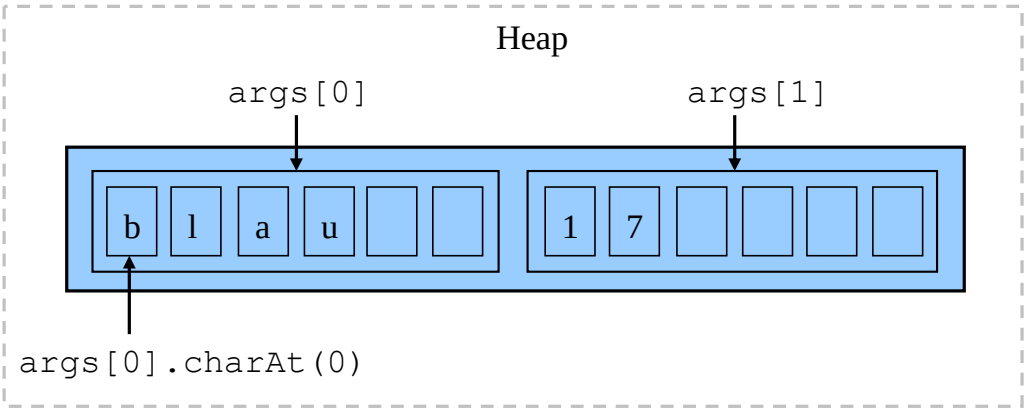
- Bei einem **Array** handelt es sich um ein Objekt, das eine Serie von Elementen desselben Typs aufnimmt, auf die man per Index, d.h. durch die mit eckigen Klammern begrenzte Elementnummer, zugreifen kann.
- In unserem Beispiel kommt ein Array mit Elementen vom Datentyp **String** zum Einsatz, wobei es sich um Zeichenketten 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 Programmnamen, jeweils durch Leerzeichen getrennt, in die Kommandozeile geschrieben hat.
- Das erste Programmargument landet im ersten Element des Zeichenketten-Arrays `args` und wird mit `args[0]` angesprochen, weil Array-Elemente mit Null beginnend nummeriert werden. Als Objekt der Klasse **String** wird `args[0]` aufgefordert, die Methode **charAt()** mit dem Parameterwert Null auszuführen. Diese Methode liefert als Rückgabe das erste Zeichen (mit der Nummer Null), welches der **char**-Variablen `farbe` zugewiesen wird.
- Das zweite Element des Zeichenketten-Arrays `args` (mit der Nummer Eins) 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 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:

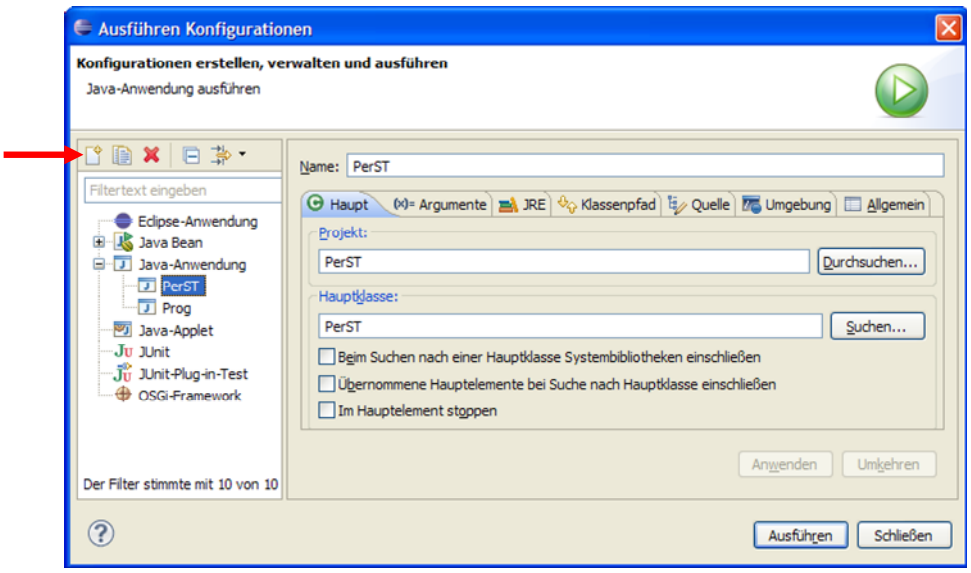



### *3.7.2.4 Eclipse-Startkonfigurationen*

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.

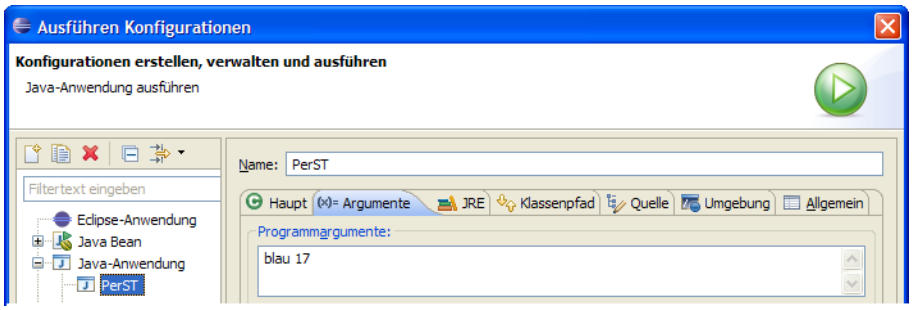
Gehen Sie folgendermaßen vor, nachdem Sie das Java-Projekt `PerST` mit der gleichnamigen Startklasse angelegt (siehe Abschnitt 2.3.3) oder importiert (siehe Abschnitt 2.3.7) haben:

- Öffnen Sie im Editor die Quellcodedatei **PerST.java** (siehe Paket-Explorer, Standardpaket zum Projekt `PerST`).
- Menübefehl **Ausführen > Konfigurationen ausführen**
- 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.

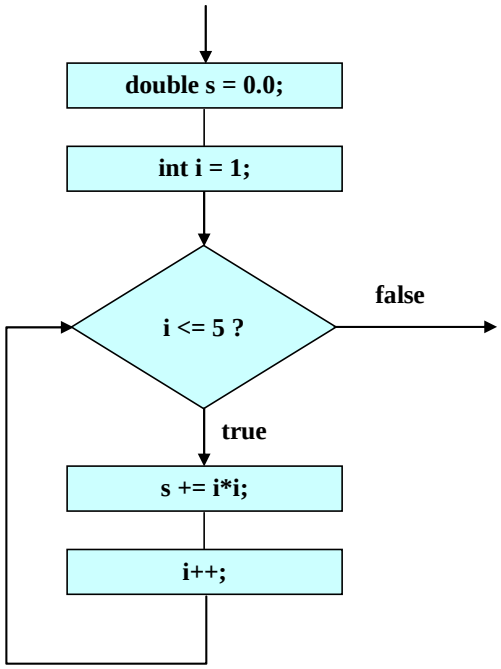
Für spätere Starts genügt bei geöffneter Quellcodedatei **PerST.java** ein Klick auf den Schalter .

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 zum Einsatz, wenn eine (Verbund-)Anweisung *mehrfach* 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 Eins bis Fünf berechnet:




Bei leicht vereinfachender Betrachtung kann man hinsichtlich der Schleifensteuerung unterscheiden:

- **Zählergesteuerte Schleife** (**for**)
  Die Anzahl der Wiederholungen steht typischerweise schon vor Schleifenbeginn fest. Bei der Ablaufsteuerung kommt eine Zählvariable 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, bis die Zählvariable einen festgelegten Grenzwert erreicht hat (siehe obige Abbildung).
- **Iterieren über die Elemente einer Kollektion**
  Seit der Java-Version 5.0 (bzw. 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.
- **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 Indexvariable Bezug nimmt. Anschließend wird die Methode hinter der **for**-Schleife fortgesetzt.

Auf das Schlüsselwort **for** folgt die von runden Klammern umgebene Schleifensteuerung, wo die Vorbereitung der Indexvariablen (nötigenfalls samt Deklaration), die Fortsetzungsbedingung und

die Aktualisierungsvorschrift untergebracht werden können. Am Ende steht 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* Indexvariable 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.
    ```
    int i = 1
    ```

  Im folgenden Programm findet sich die zweite Variante:

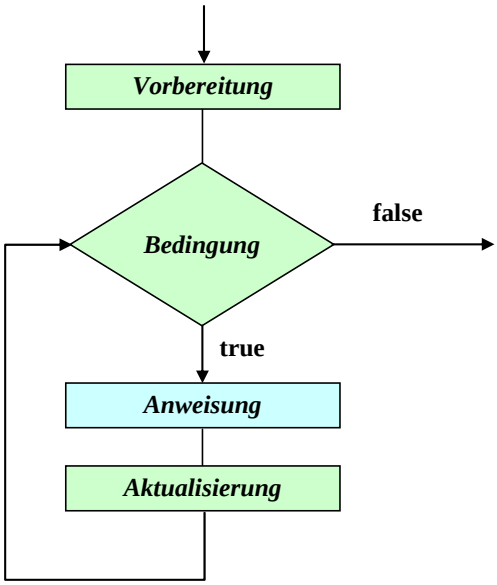
| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  double s = 0.0;`<br>`  for (int i = 1; i <= 5; i++)`<br>`  s += i*i;`<br>`  System.out.println("Quadratsumme = " + s);`<br>`  }`<br>`}` | `Quadratsumme = 55.0` |

  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 Indexvariable gesetzt, doch erlaubt Java beliebige logische Ausdrücke. Die Bedingung wird *vor jedem Schleifendurchgang* geprüft. Resultiert der Wert **true**, so findet eine weitere Wiederholung der Anweisungsteils statt, anderen falls wird die **for**-Anweisung verlassen. Folglich kann es auch passieren, dass überhaupt kein Durchlauf zustande kommt.

- ***Aktualisierung***
  Am Ende jedes Schleifendurchgangs (nach Ausführung der Anweisung) wird der Aktualisierungsteil ausgeführt. Hier wird meist die Indexvariable in- oder dekrementiert.

Im folgenden Flussdiagramm sind die semantischen Regeln zur **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. In solchen Fällen ist die Umlaufzahl einer **for**-Schleife natürlich nicht mehr aus der Schleifensteuerung abzulesen. Dies gelingt auch dann nicht, wenn eine Indexvariable in der Schleifenanweisung modifiziert wird.

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

Obwohl wir uns bisher nur anhand von Beispielen mit Arrays (Objekte, die eine feste Anzahl von Elementen desselben Datentyps enthalten) und mit anderen Kollektionen noch gar nicht beschäftigt haben, sollen die mit Java 5.0 (bzw. 1.5) eingeführten Erweiterungen der **for**-Schleife (siehe z.B. Gosling et al. 2005, S. 387) doch hier im Kontext mit den übrigen Wiederholungsanweisungen behandelt werden. Konzentrieren Sie sich also auf das 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 neuer Bauart jedes Element im **String**-Array `args` mit den Programmargumenten ausgegeben:

| Quellcode | Ausgabe nach einem Start mit<br>`java Prog eins zwei drei` |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  for (String s : args)`<br>`  System.out.println(s);`<br>`  }`<br>`}` | `eins`<br>`zwei`<br>`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 im *i*-ten Schleifendurchgang das *i*-te Element der Kollektion anspricht.

### *3.7.3.3 Bedingungsabhängige Schleifen*

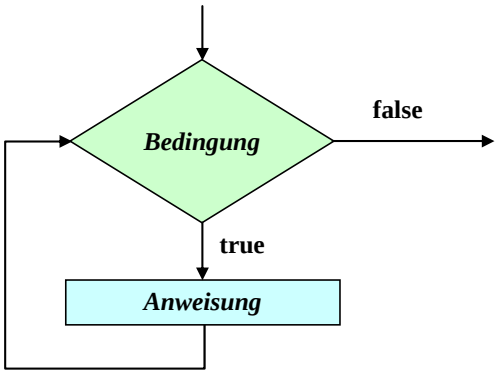
Wie die Erläuterungen zur **for**-Schleife gezeigt haben, ist die Überschrift dieses Abschnitts nicht sehr trennscharf, weil bei der **for**-Schleife ebenfalls eine beliebige Terminierungsbedingung angegeben werden darf. In vielen Fällen ist es eine Frage des persönlichen Geschmacks, welche Wiederholungsanweisung 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 ausgeführt, anderenfalls wird die **while**-Anweisung verlassen, eventuell noch vor dem ersten Durchgang:

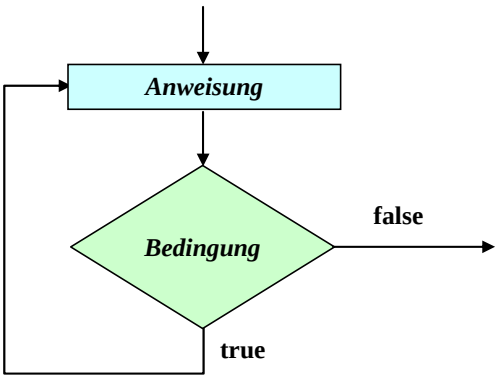



Im obigen Beispielprogramm zur Quadratsummenberechnung (siehe Abschnitt 3.7.3.1) kann man die **for**-Schleife leicht durch eine **while**-Schleife ersetzen:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  int i = 1;`<br>`  double s = 0.0;`<br>`  while (i <= 5) {`<br>`  s += i*i;`<br>`  i++;`<br>`  }`<br>`  System.out.println("Quadratsumme = " + s);`<br>`  }`<br>`}` | `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. In folgendem 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' );
```

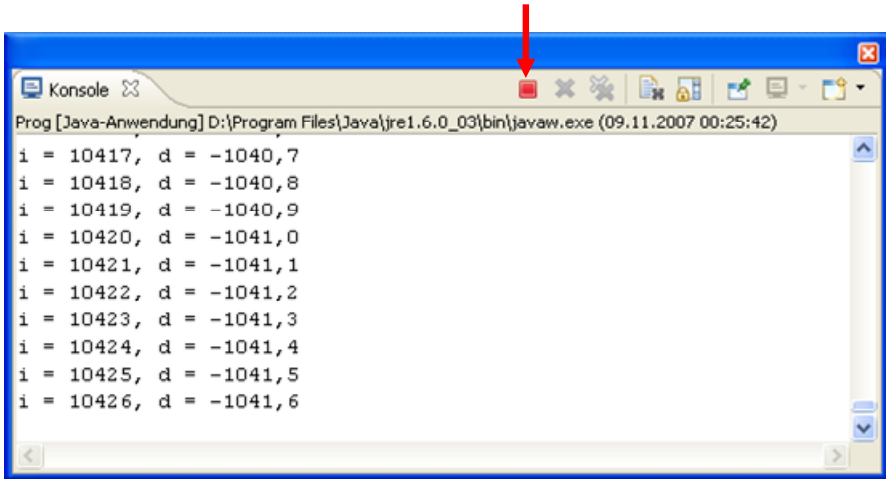
Bei einer **do**-Schleife mit Anweisungs*block* sollte man die **while**-Klausel unmittelbar hinter die schließende Blockklammer setzen (in dieselbe Zeile), um sie optisch von einer selbständigen **while**-Anweisung abzuheben (siehe Beispiel).

### *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 3.5.3):

```
class Prog {
  public static void main(String[] args) {
    int i = 0;
    double d = 1.0;
    // besser: while (Math.abs(d) > 1.0e-14) {
    while (d != 0.0) {
      i++;
      d -= 0.1;
      System.out.printf("i = %d, d = %.1f\n", i, d);
    }
    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 als Bestandteil der **switch**-Anweisung begegnet ist, kann eine **for**-, **while**- oder **do**-Schleife vorzeitig verlassen werden. Mit der **continue**-Anweisung veranlasst man Java, den aktuellen Schleifendurchgang zu beenden und sofort mit dem nächsten zu beginnen. In der Regel kommen **break** und. **continue** im Rahmen einer **if**-Anweisung zum Einsatz, z.B. in folgendem Programm zur (relativ primitiven) Primzahlendiagnose:

```
class Primitiv {
   public static void main(String[] args) {
      boolean tg;
      int i, mtk, zahl;
      System.out.println("Einfacher Primzahlendetektor\n");
      while (true) {
         System.out.print("Zu untersuchende Zahl > 2 oder 0 zum Beenden: ");
         zahl = Simput.gint();
         if (!Simput.checkError() || (zahl <= 2  &&  zahl != 0)) {
            System.out.println("Keine Zahl oder illegaler Wert!\n");
            continue;
         }
         if (zahl == 0)
            break;
         tg = false;
         mtk = (int) 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!");
   }
}
```

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 Null kann das Beispielprogramm 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 124), doch werden bei dem vorgeschlagenen Verfahren lästige Sonderfälle (unzulässige

Werte, Null als Terminierungssignal) auf besonders übersichtliche Weise abgehakt, bevor der Kernalgorithmus startet.

Zum Kernalgorithmus der Primzahlendiagnose sollte vielleicht noch erläutert werden, warum die Suche nach einem Teiler des Primzahlkandidaten bei seiner Wurzel enden kann (genauer: bei der größten ganzen Zahl ≤ Wurzel):

Sei $d$ (≥ 1) 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 $z$ keinen echten Teiler kleiner oder gleich $\sqrt{z}$ hat, kann man auch jenseits dieser Grenze keinen finden, und $z$ ist eine Primzahl.

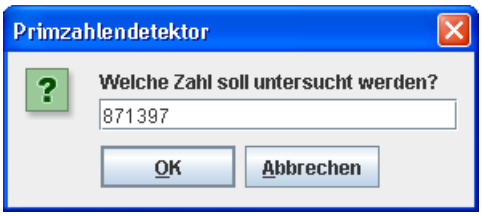
Zur Berechnung der Wurzel verwendet das Beispielprogramm die statische Methode **sqrt()** aus der Klasse **Math**.

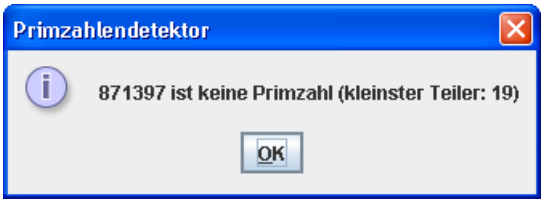
## 3.8 Entspannungs- und Motivationseinschub: GUI-Standarddialoge

Nach etlichen recht anstrengenden Themen, soll dieser Abschnitt zur Entspannung und zur Regeneration Ihrer Motivation beitragen. Sie lernen die 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. Wir erstellen zum Primzahldiagnoseprogramm aus Abschnitt 3.7.3.5 mit erstaunlich geringem Aufwand die folgende Variante

```
import javax.swing.JOptionPane;
class PrimitivGuiJop {
  public static void main(String[] args) {
    String s;
    boolean tg;
    long i, mtk, argumemt;
    while (true) {
      argumemt = Long.parseLong(
           JOptionPane.showInputDialog(null,
             "Welche Zahl soll untersucht werden?",
             "Primzahlendetektor", JOptionPane.QUESTION_MESSAGE));
      tg = false;
      mtk = (int) Math.sqrt(argumemt);   //maximaler Teilerkandidat
      for (i = 2; i <= mtk; i++)
        if (argumemt % i == 0) {
          tg = true;
          break;
        }
      if (tg)
        s = String.valueOf(argumemt) +
          " ist keine Primzahl (kleinster Teiler: "+String.valueOf(i)+")";
      else
        s = String.valueOf(argumemt) + " ist eine Primzahl";
      JOptionPane.showMessageDialog(null,
         s, "Primzahlendetektor", JOptionPane.INFORMATION_MESSAGE);
    }
  }
}
```

mit graphischer Bedienoberfläche:




Die linke Dialogbox zur Erfassung des Primzahlkandidaten geht auf den Aufruf der statischen **JOptionPane**-Methode **showInputDialog()** zurück. 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.[19]

Die rechte Dialogbox mit dem Ergebnis der Primzahlendiagnose produzieren wir mit Hilfe der statischen **JOptionPane**-Methode **showMessageDialog()**, wobei die statische Methode **valueOf()** der Klasse **String** die Zeichenkettendarstellung des übergebenen **long**-Werts liefert.

Weil der Klassenname **JOptionPane** an mehreren Stellen auftaucht, wird er am Anfang des Programms importiert.

Die beiden Dialogboxmethoden kennen etliche Parameter (Argumente zur näheren Bestimmung der Ausführung), deren Verwendung durch das Beispielprogramm hinreichend klar geworden sein sollte. In der folgenden Tabelle werden die Parameter in der Reihenfolge ihres Auftretens beschrieben:

| Name | Erläuterung |
|---|---|
| **parentComponent** | Standarddialoge werden oft von einem anderen Fenster aufgerufen. Die Angabe eines Fensterobjekts (an Stelle der Alternative **null**) hat zur Folge, dass die Dialogbox in der Nähe dieses Fensters erscheint. |
| **message** | Dieser Text erscheint in der Dialogbox. |
| **title** | Dieser Text erscheint in der Titelzeile der Dialogbox. |
| **messageType** | Dieser Parameter legt den Typ der Nachricht fest, der auch über das Icon am linken Rand der Dialogbox entscheidet. Als Werte sind die folgenden statischen und finalisierten Felder der Klasse **JOptionPane** erlaubt:<br>• **JOptionPane.ERROR_MESSAGE**<br>• **JOptionPane.INFORMATION_MESSAGE**<br>• **JOptionPane.WARNING_MESSAGE**<br>• **JOptionPane.QUESTION_MESSAGE**<br>• **JOptionPane.PLAIN_MESSAGE** |

Falls der Benutzer eine ungültige Zeichenfolge an unser Programm sendet, auf die Schaltfläche **Abbrechen** klickt oder die **Esc**-Taste drückt, endet es mit einer unbehandelten Ausnahme, z.B.:

```
Exception in thread "main" java.lang.NumberFormatException: null
  at java.lang.Long.parseLong(Long.java:375)
  at java.lang.Long.parseLong(Long.java:468)
  at PrimitivGuiJop.main(PrimitivGuiJop.java:8)
```

Sie werden im Abschnitt 9 erfahren, wie man solche Ausnahmen abfangen und behandeln kann.

Startet man den Primzahlendetektors konsolenfrei (mit dem JRE-Werkzeug **javaw.exe**), bemerkt der Benutzer nichts von der fehlenden Ausnahmebehandlung:

```
>javaw PrimitivGui
```

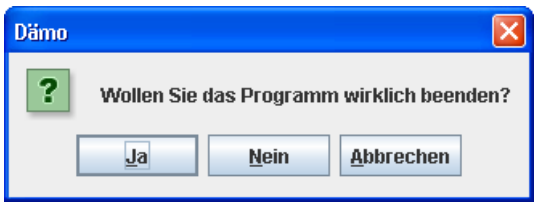
Von den zahlreichen weiteren Möglichkeiten der Klasse **JOptionPane** (siehe API-Dokumentation) soll noch die statische Methode **showConfirmDialog()** erwähnt werden. Sie liefert über einen **int**-Rückgabewert die Antwort des Benutzers auf eine Frage, z.B.:

---

[19] Derartige Konvertierungsmethoden werden in Abschnitt 5.3.2 offiziell behandelt.

```
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 vierten Parameter (Name **optionType)** wählt man die angebotenen Schaltflächen:




Als Werte sind die folgenden statischen und finalisierten Felder der Klasse **JOptionPane** erlaubt:

| **optionType**-Wert | Resultierende Schalter |
|---|---|
| **JOptionPane.DEFAULT_OPTION** | **OK** |
| **JOptionPane.YES_NO_OPTION** | **Ja**, **Nein** |
| **JOptionPane.YES_NO_CANCEL_OPTION** | **Ja**, **Nein**, **Abbrechen** |
| **JOptionPane.OK_CANCEL_OPTION** | **OK**, **Abbrechen** |

Über die Rückgabe der Methode **showConfirmDialog()** erfährt man, welchen Schalter der Benutzer gewählt hat, wobei als Werte die folgenden statischen und finalisierten Felder der Klasse **JOptionPane** auftreten können:

| Vom Benutzer gewählter Schalter | **showConfirmDialog()**-Rückgabewert |
|---|---|
| Schließkreuz in der Titelzeile | **JOptionPane.CLOSED_OPTION** |
| **Ja** | **JOptionPane.YES_OPTION** |
| **Nein** | **JOptionPane.NO_OPTION** |
| **Abbrechen** | **JOptionPane.CANCEL_OPTION** |

Anders als bei der Konsolenausgabe über **System.out** (vgl. Abschnitt 3.2.3) haben wir beim Einsatz von graphischen Benutzeroberflächen keine Probleme mit Umlauten (siehe Titelzeile des Beispielprogramms).

## *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
maiLink
else
Lösung
```

3) Das folgende Programm gibt den Wert der Klassenvariable **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“ in folgendem Programm zu erklären?

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  System.out.println("3.3 + 2 = " + 3.3 + 2);`<br>`  }`<br>`}` | `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 bitte 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 einer **char**-Variablen eine *Zahl* zugewiesen, die sie offenbar unbeschädigt an eine **int**-Variable weitergeben kann, wobei ihr Inhalt von **println()** aber als Buchstabe ausgegeben wird. Wie erklären sich diese Merkwürdigkeiten?

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

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

3) Warum meldet der JDK-Compiler hier einen möglichen Verlust an Genauigkeit?

| Quellcode | Fehlermeldung des JDK-Compilers |
|---|---|
| `class Prog {`<br>`  public static void main(String args[]) {`<br>`  int i = 7l;`<br>`  System.out.println(i);`<br>`  }`<br>`}` | `possible loss of precision` |

Die Aufgabe ist zugegebenermaßen etwas gemein und verlangt vor allem ein gutes Auge.

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

| Quellcode | Fehlermeldung |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {{`<br>`  int i = 2;`<br>`  }`<br>`  System.out.println(i);`<br>`  }`<br>`}` | `i kann nicht aufgelöst werden` |

5) Schreiben Sie bitte ein Java-Programm, das folgende Ausgabe macht:

```
Dies ist ein Java-Zeichenkettenliteral:
   "Hallo"
```

6) Beseitigen Sie bitte alle Fehler in folgendem 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 Konsolenanwendungen)**

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) Welche 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 Wahrheitsweite 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*) = !*La1* || !*La2*
!(*La1* || *La2*) = !*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(1,000) = 2,7182818285`

Hinweise:

- Suchen Sie mit Hilfe der JDK-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 oder ohne Dezimalkomma) 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.

2) Welche Werte und Datentypen besitzen die folgenden Ausdrücke?

```
1/(1/0.0)+5
1/(1/0)+5.0
1/(0/0.0)+5
```

**Abschnitt 3.7 (Anweisungen (zur Ablaufsteuerung))**

1) In einer Lotterie gilt folgender Gewinnplan:

- Durch 13 teilbare Losnummern gewinnen 100 Euro.
- Losnummern, die nicht durch 13 teilbar sind, gewinnen immerhin noch einen Euro, wenn sie durch 7 teilbar sind.

Wird in folgendem Codesegment für Losnummern in der Variablen `losNr` der richtige Gewinn ermittelt?

```
if (losNr % 13 != 0)
  if (losNr % 7 == 0)
    System.out.println("Das Los gewinnt einen Euro!");
else
  System.out.println("Das Los gewinnt 100 Euro!");
```

2) Warum liefert dieses Programm widersprüchliche Auskünfte über die boolesche Variable b?

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  boolean b = true;`<br>`  if (b = false)`<br>`  System.out.println("b ist false");`<br>`  else`<br>`  System.out.println("b ist true");`<br>`  System.out.println("\nKontr.ausg.: b ist "+b);`<br>`  }`<br>`}` | `b ist true`<br><br>`Kontr.ausg.: b ist false` |

3) Das folgende Programm soll Buchstaben mit 1 beginnend nummerieren:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  char bst = 'a';`<br>`  byte nr = 0;`<br><br>`  switch (bst) {`<br>`  case 'a': nr = 1;`<br>`  case 'b': nr = 2;`<br>`  case 'c': nr = 3;`<br>`  }`<br>`  System.out.println("Zu "+bst+`<br>`  " gehoert die Nummer "+nr);`<br>`  }`<br>`}` | `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?

4) Erstellen Sie eine Variante des Primzahlen-Diagnoseprogramms aus Abschnitt 3.7.3.5, die ohne **break** und **continue** auskommt.

5) 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);
    }
  }
}
```

6) 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.

7) Bei einem **double**-Wert sind mindestens 15 signifikante Dezimalstellen garantiert (siehe Abschnitt 3.3.3). 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 Index *i*, für den man noch erhält:

$$1{,}0 + 2^{-i} > 1{,}0$$

In dem (zur freiwilligen Lektüre empfohlenen) Vertiefungsabschnitt 3.3.4.1 findet sich eine Erklärung für das Ergebnis.

8) 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 Performanzunterschiede messen. Verwenden Sie als ersten Kandidaten den im Einführungsbeispiel zum Kürzen von Brüchen (Methode `kuerze()`) benutzten Algorithmus (siehe Abschnitt 1.1.3). Sein offensichtliches 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 1.1.2): Für natürliche Zahlen *u* und *v* mit $u \geq v$ 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 Pentium 4 mit 3 GHz):

```
ggT-Bestimmung mit Euklid (Differenz)
Erste Zahl:   999000999
Zweite Zahl:  36
ggT:          9
Benoet. Zeit: 94 Millisek.
```

```
ggT-Bestimmung mit Euklid (Modulo)
Erste Zahl:   999000999
Zweite Zahl:  36
ggT:          9
Benoet. Zeit: 0 Millisek.
```

9) Wer kann ein Programm erstellen, das zur Berechnung der Summe der natürlichen Zahlen von 1 bis 1 Milliarde

$$\sum_{i=1}^{1000000000} i = 1+2+3+...+1000000000$$

weniger als 1 Millisekunde benötigt?

# 4 Klassen und Objekte

Softwareentwicklung in Java besteht im Wesentlichen aus der Definition von **Klassen**, die aufgrund einer vorangegangenen objektorientierten Analyse …

- als Baupläne für Objekte
- und/oder als Akteure

konzipiert werden können. 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 im 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) vereinbart bzw. entworfen. Man spricht von den *Mitgliedern* einer Klasse (engl. *members*).

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 kann der passende Ansprechpartner zur Erledigung eines Auftrags sein:

- eine Klasse mit passenden Handlungskompetenzen, z.B.:
  ```
  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 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.

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 Abschnitt 1 speziell für größere Projekte empfohlene objektorientierte *Analyse* (z.B. mit Hilfe der UML) ist dabei leider keine Zeit (siehe z.B. Balzert 1999).

## *4.1 Überblick, historische Wurzeln, Beispiel*

### 4.1.1 Einige Kernideen und Vorzüge der OOP

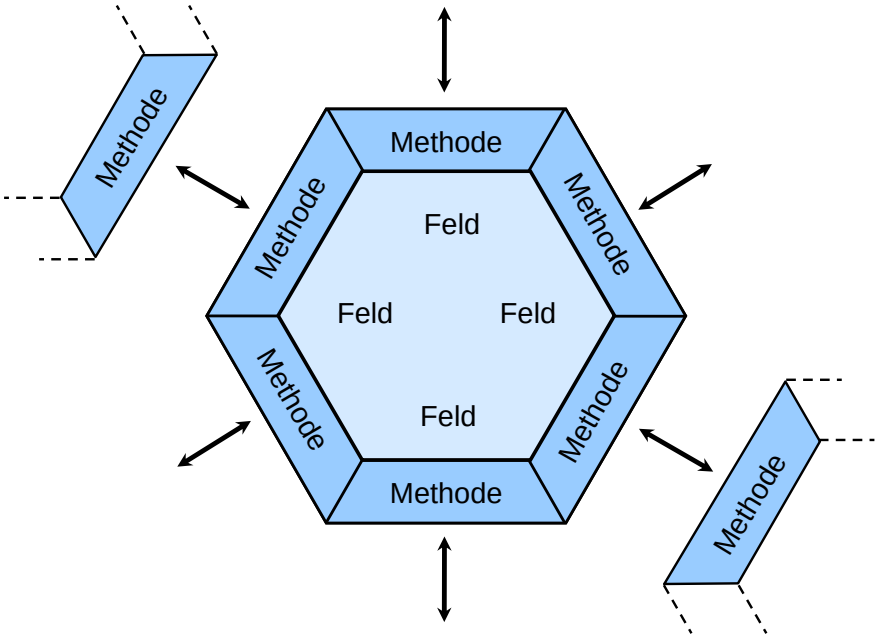
Lahres & Rayman (2006) nennen in ihrem Buch *Praxisbuch Objektorientierung* unter Berufung auf Alan Kay, der den Begriff *Objektorientierte Programmierung* geprägt und die objektorientierte Programmiersprache *Smalltalk* entwickelt hat, als unverzichtbare OOP-Grundelemente:

- **Datenkapselung**
  Mit diesem Thema haben wir uns bereits beschäftigt. Das vorhandene Wissen soll gleich vertieft und gefestigt werden.
- **Vererbung**
  Dieses OOP-Element wird gleich vorgestellt und später ausführlich behandelt.

- **Polymorphie**
  Weil die Vorteile der Polymorphie mit den bisherigen Beispielen nicht plausibel vermittelt werden können, schieben wir die Behandlung dieses Themas noch etwas auf.

#### *4.1.1.1 Datenkapselung und Modularisierung*

In der objektorientierten Programmierung (OOP) wird die traditionelle Trennung von Daten und Operationen aufgegeben. 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 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 3.3.7), also vor Veränderungen geschützt sein. Wir haben mit den statischen, öffentlichen und finalisierten Felder `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**, das schon Julius Cäsar (100 v. Chr. - 44 v. Chr.) bei seiner beruflichen Tätigkeit als römischer Kaiser und Feldherr erfolgreich einsetzte (*Divide et impera!*).[20] Die Modularisierung ist ein probates, ja unverzichtbares Mittel der Software-Entwickler zur Bewältigung von Projekten mit hoher Komplexität.

Aus der Datenkapselung und der Modularisierung ergeben sich gravierende Vorteile für die Softwareentwicklung:

- **Vermeidung von Fehlern, Erleichterung der Fehlersuche**
  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 Null gesetzt wird. Anwender unserer Klasse können einen Nenner einzig über die Methode `setzeNenner()`

[20] Deutsche Übersetzung: Teile und herrsche!

verändern, die aber den Wert Null nicht akzeptiert. Bei einer anderen Klasse kann es wichtig sein, dass für eine Gruppe von Feldern bei jeder Änderung gewissen Konsistenzbedingungen eingehalten werden. Treten in einem Programm trotz Datenkapselung Fehler wegen pathologischer Variablenausprägungen auf, sind diese relativ leicht zu lokalisieren, weil nur wenige Methoden verantwortlich sein können. Per Datenkapselung werden also die **Kosten reduziert**, die durch Programmierfehler entstehen.

- **Produktivität**
  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.
- **Flexibilität, Anpassungsfähigkeit, vereinfachte Wartung**
  Datenkapselung schafft günstige Voraussetzungen für die Wartung bzw. Verbesserung einer Klassendefinition. Solange die Methoden der Schnittstelle unverändert bzw. kompatibel bleiben, kann die interne Architektur einer Klasse ohne Nebenwirkungen auf andere Programmteile beliebig geändert werden.
- **Erfolgreiche Teamarbeit durch abgeschottete Verantwortungsbereiche**
  In großen Projekten können mehrere Programmierer nach der gemeinsamen Entwicklung von Schnittstellen relativ unabhängig an verschiedenen Klassen arbeiten.

#### *4.1.1.2 Vererbung*

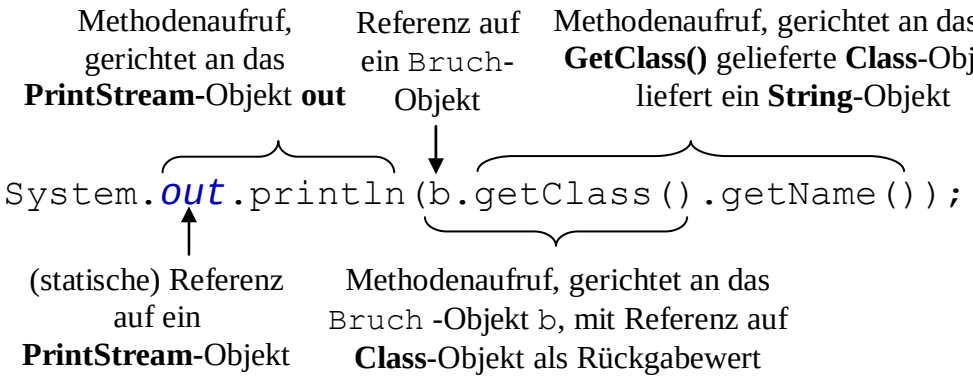
Zu den Vorzügen der „super-modularen“ Klassenkonzeption gesellt sich in der OOP ein Vererbungsverfahren, das beste Voraussetzungen für die Erweiterung von Softwaresystemen bei rationeller Wiederverwendung der bisherigen Code-Basis schafft: Bei der Definition einer neuen Klasse werden alle Eigenschaften (Felder) und Handlungskompetenzen (Methoden) einer *Basisklasse* übernommen. Es ist also leicht, 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 vorhandener Klassen (über statische Methoden oder erzeugte Objekte) bietet die OOP mit der Vererbungstechnik eine weitere Möglichkeit zur (Wieder)verwendung von Software.

Wird bei einer Klassendefinition keine Basisklasse explizit angegeben, erbt die neue Klasse implizit von der Urahnklasse **Object** an der Spitze der streng hierarchisch organisierten Java-Klassensystems. 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 `Bruchrechnung` (vgl. Abschnitt 1.1) wird ein `Bruch`-Objekt nach seiner Klassenzugehörigkeit befragt:

| Quellcode | Ausgabe |
|---|---|
| `class Bruchrechnung {`<br>`  public static void main(String[] args) {`<br>`  Bruch b = new Bruch();`<br>`  System.out.println(b.getClass().getName());`<br>`  }`<br>`}` | `Bruch` |

Die Methode **getClass()** liefert als Rückgabewert ein Objekt der Klasse **Class**, welches nach dem Methodenaufruf **getName()** eine Zeichenfolge mit der gewünschten Information liefert. Diese Zeichenfolge (ein Objekt der Klasse **String**) bildet schließlich den Aktualparameter 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: Die Objekte der Klasse `Bruch` beherrschen dank Vererbung die Methode **getClass()**, obwohl in der `Bruch`-Klassendefinition nichts davon zu sehen ist.

#### *4.1.1.3 Realitätsnahe Modellierung*

Klassen sind nicht nur ideale Bausteine für die rationelle Konstruktion von Softwaresystemen, 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).

### 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 auch umfangreiche Bibliotheken mit Subroutinen 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 fremder Quelle 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 problemadäquate 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.
- Bessere Abbildung des Anwendungsbereichs
- Rationellere (Weiter-)Entwicklung von Software durch die Vererbungstechnik
- 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 der Abschnitte 2 und 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. Darauf basierende Programme sollen Schüler beim Erlernen der Bruchrechnung unterstützen. Eine objektorientierte Analyse der Problemstellung ergab, 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 **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 **int**-Feldern `zaehler` und `nenner`).
- 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 in auf Seite 125).

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 Ref.typ-Init. auf null wird ersetzt

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

  public boolean setzeNenner(int n) {.  .  .}

  public void setzeEtikett(String epar) {
    int rind = epar.length();
    if (rind > 40)
      rind = 40;
    etikett = epar.substring(0, rind);
  }

  public int gibZaehler() {return zaehler;}

  public int gibNenner()  {return nenner;}

  public String gibEtikett()   {return etikett;}

  public void kuerze() {
    // größten gemeinsamen Teiler mit Euklidis Algorithmus bestimmen
    // (performante Variante mit Modulo-Operator)
    if (zaehler != 0) {
      int rest;
      int ggt = Math.abs(zaehler);
      int divisor = Math.abs(nenner);
      do {
        rest = ggt % divisor;
        ggt = divisor;
        divisor = rest;
      } while (rest > 0);
      zaehler /= ggt;
      nenner  /= ggt;
    } else
      nenner = 1;

  }

  public void addiere(Bruch b) {.  .  .}

  public void frage() {.  .  .}

  public void zeige() {
    String luecke = "";
    int el = etikett.length();
    for (int i=1; i<=el; i++)
      luecke = luecke + " ";
    System.out.println(" " + luecke  + "   " + zaehler + "\n" +
                       " " + etikett + " -----\n" +
                       " " + luecke  + "   " + nenner + "\n");
  }
}
```

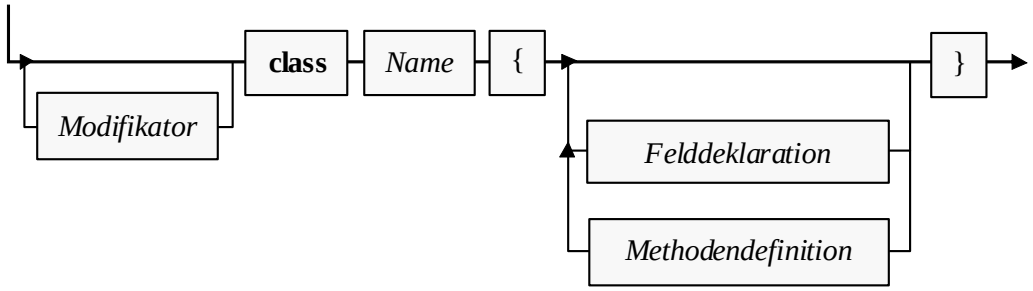
Für die bei diversen Demonstrationen in den folgenden Abschnitten verwendeten Startklassen (mit jeweils spezieller Implementation) werden wir ab jetzt den Namen `Bruchrechnung` verwenden, z.B.:

```
class Bruchrechnung {
  public static void main(String[] args) {
    Bruch b = new Bruch();
    b.setzeZaehler(4);
    b.setzeNenner(16);
    b.kuerze();
    b.setzeEtikett("Der gekuerzte Bruch:");
    b.zeige();
  }
}
```

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. 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 behandeln.

Wir arbeiten weiter mit dem aus Abschnitt 3.1.2.1 bekannten, leicht vereinfachten Syntaxdiagramm zur Klassendefinition:

**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. Dazu muss die Klasse später allerdings noch in ein explizites Paket aufgenommen werden. 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*).[21]

Hinsichtlich der **Dateiverwaltung** ist zu beachten:

- Die `Bruch`-Klassendefinition muss in einer Datei namens **Bruch.java** gespeichert werden, weil die Klasse als **public** definiert ist.
- Auch für den Quellcode der Startklasse `Bruchrechnung`, die nicht als **public** definiert ist, sollte analog eine Datei namens **Bruchrechnung.java** verwendet werden.
- Dateien mit Java-Quellcode benötigen auf jeden Fall die Namenserweiterung **.java**.

[21] Bei einer startfähigen Klasse ist ein komplizierter Name zu vermeiden, wenn dieser vom Benutzer beim Programmstart eingetippt werden muss (mit korrekt eingehaltener Groß-/Kleinschreibung!).

## 4.2 Instanzvariablen

Die Instanzvariablen (bzw. -felder) einer Klasse besitzen viele Gemeinsamkeiten mit den *lokalen* Variablen, die wir im Abschnitt 3 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**)

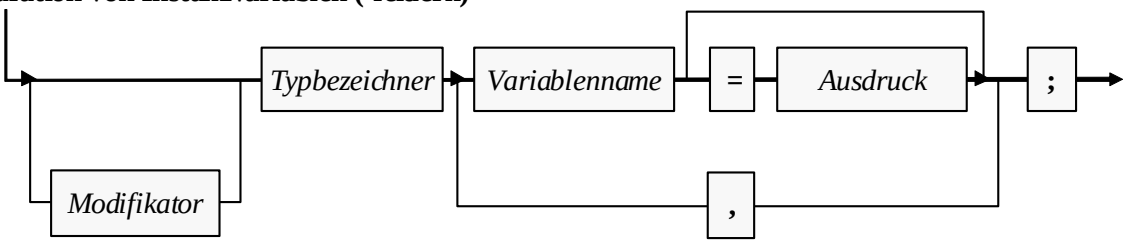
Zu den beiden Feldern `zaehler` und `nenner` vom primitiven Datentyp **int** ist das Feld `etikett` mit dem Referenzdatentyp **String** dazugekommen. Jedes nach dem `Bruch`-Bauplan geschaffene Objekt erhält seine *eigene, komplette* Ausstattung mit diesen Variablen.

### 4.2.1 Deklaration mit Wahl der Schutzstufe

Während lokale Variablen im Anweisungsteil 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.

In der Regel gibt man beim Deklarieren von Instanzvariablen einen Modifikator zur Spezifikation der **Schutzstufe** an, so dass die Syntax im Vergleich zur Deklaration einer lokalen Variablen entsprechend erweitert werden muss.

**Deklaration von Instanzvariablen (-feldern)**




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 Zugriffsrecht, was in der Regel unerwünscht ist.

Auf den ersten Blick scheint die Datenkapselung nur beim Nenner eines Bruches relevant zu sein, doch auch bei den restlichen Instanzvariablen bringt sie (potentiell) Vorteile:

- Zugunsten einer übersichtlichen Bildschirmausgabe soll das Etikett auf 40 Zeichen beschränkt bleiben. Mit Hilfe der Zugriffsmethode `setzteEtikett()` kann dies gewährleistet werden.

- Abgeleitete (erbende) Klassen (siehe unten) werden in die Zugriffsmethoden für `zaehler` und `nenner` neben der Null-Überwachung für den Nenner eventuell noch weitere Wertrestriktionen einbauen (z.B. Beschränkung auf positive Zahlen). Während solche Anpassungen von Zugriffsmethoden problemlos möglich sind, ist beim Ableiten eine Einschränkung von Zugriffsrechten verboten. Für eine als **public** deklarierte Instanzvariable `zaehler` wären also auch in abgeleiteten Klassen keine Wertrestriktionen möglich.
- Oft kann der Datentyp von gekapselten Instanzvariablen geändert werden, ohne dass die Schnittstelle inkompatibel zu vorhandenen Programmen wird, welche die Klasse benutzen. Im Fall der `Bruch`-Felder `zaehler` und `nenner` wird man vielleicht zur Vermeidung von Überlaufproblemen (vgl. Abschnitt 3.6.1) irgendwann den Datentyp **int** durch den größeren Typ **long** ersetzen. Davon würde z.B. die Methode `addiere()` profitieren:

```
public void addiere(Bruch b) {
  zaehler = zaehler*b.nenner + b.zaehler*nenner;
  nenner = nenner*b.nenner;
  kuerze();
}
```

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 eine Änderung des Datentyps nicht in Frage kommt. Um allen Klassen den Direktzugriff auf eine Instanzvariable zu erlauben, wird in deren Deklaration der Modifikator **public** angegeben, z.B.:

```
public int zaehler;
```

Im Zusammenhang mit den Paketen (siehe Abschnitt 7) werden wir uns noch ausführlich mit dem Thema *Zugriffsschutz* beschäftigen.

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*)

### 4.2.2 Gültigkeitsbereich, Existenz und Ablage im Hauptspeicher

Von den lokalen Variablen einer Methode unterscheiden sich die Instanzvariablen einer Klasse vor allem in Bezug auf den Gültigkeitsbereich (synonym: Sichtbarkeitsbereich), die Lebensdauer und die Ablage im Hauptspeicher:

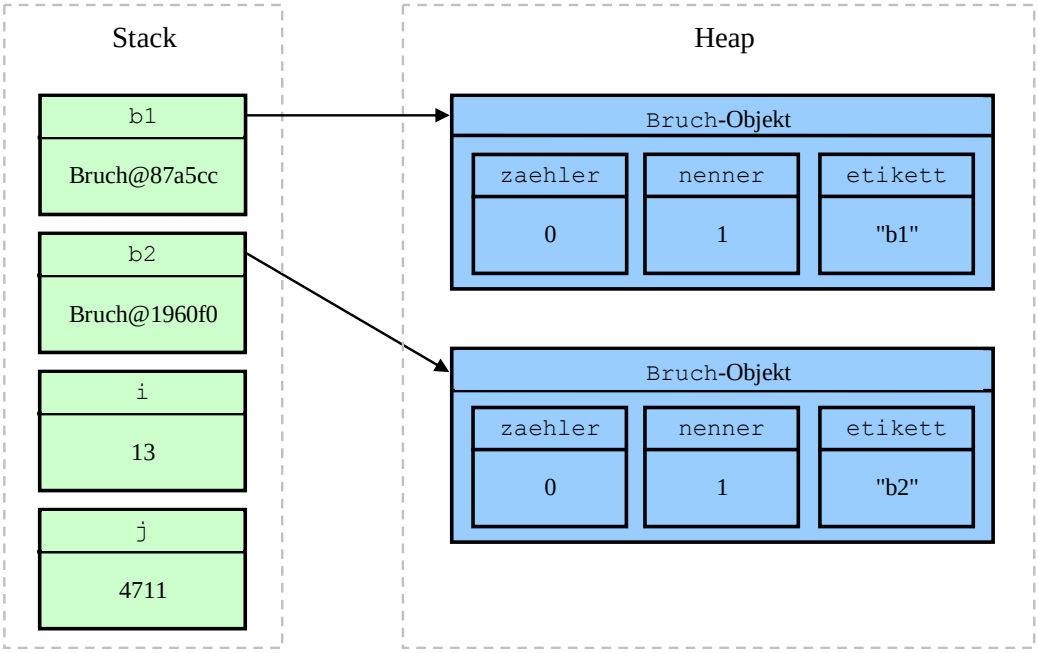
| | lokale Variable | Instanzvariable |
|---|---|---|
| **Gültigkeit, Sichtbarkeit** | Eine lokale Variable ist nur in ihrer eigenen Methode gültig (sichtbar). Nach der Deklarationsanweisung kann sie in den restlichen Anweisungen des lokalsten Blocks angesprochen werden. Eingeschachtelte Blöcke gehören zum Gültigkeitsbereich. In aufgerufenen Methoden ist die lokale Variable nicht sichtbar. | Die Instanzvariablen eines Objekts sind in allen Methoden sichtbar, die eine Referenz zum Objekt kennen. Je nach Schutzstufe ist Methoden fremder Klassen der Zugriff jedoch verwehrt.<br>Instanzvariablen werden durch gleichnamige lokale Variablen überdeckt, können jedoch über das vorgeschaltete Schlüsselwort **this** weiter angesprochen werden (siehe Abschnitt 4.2.4). |

| | lokale Variable | Instanzvariable |
|---|---|---|
| **Lebensdauer** | Sie existiert von der Deklaration bis zum Verlassen des lokalsten Blocks. | 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 Projekt mehr vorhanden ist. |
| **Ablage im Speicher** | Sie wird auf dem so genannten **Stack** (deutsch: *Stapel*) gespeichert. 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:[22]




In der Abbildung sind die Rückgabewerte der (von **Object** geerbten) `Bruch`-Methode **toString()** als Inhalte der lokalen `Bruch`-Referenzvariablen eingetragen. Hinter dem @-Zeichen stehen genau

[22] 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.

genommen die **hashCode()**-Werte (vgl. Abschnitt 6.4.2) der Klasse **Object**, die allerdings wesentlich auf den Speicheradressen basieren.

### 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 Null bedenklich, weshalb eine explizite Initialisierung auf den Wert Eins 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**) zu Folge hätte, wird ein **String**-Objekt mit einer leeren Zeichenfolge erstellt und zur `etikett`-Initialisierung verwendet. Das Erzeugen des **String**-Objekts erfolgt *implizit*, 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 4.4.5.2) 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 durch den **Punktoperator** zu trennen sind. In der folgenden Anweisung aus der `Bruch`-Methode `addiere()` greift das handelnde Objekt lesend auf die Instanzvariablen eines anderen `Bruch`-Objekts zu, das über die Referenzvariable `b` angesprochen wird:

```
zaehler = zaehler*b.nenner + b.zaehler*nenner;
```

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.:

```
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 Zu- bzw. Fehlgriffe jedoch verhindert. Der JDK-Compiler meldet

```
Bruchrechnung.java:8: nenner has private access in Bruch
                b.nenner = 0;
                 ^
1 error
```

und der Eclipse-Compiler äußert sich so

```
Unaufgelöstes Kompilierungsproblem:
  Das Feld Bruch.nenner ist nicht sichtbar (visible)

  at Bruchechnung.main(Bruchrechnung.java:8)
```

### 4.2.5 Finalisierte Felder

Neben der Schutzstufenwahl gibt es weitere Anlässe für den Einsatz von Modifikatoren in Felddeklarationen. Mit dem Modifikator **final** können nicht nur lokale Variablen (siehe Abschnitt 3.3.7) sondern auch (instanz- oder klassenbezogene) Felder als finalisiert deklariert werden. Während normale Felder automatisch mit der typspezifischen Null initialisiert werden (siehe Abschnitt 4.2.3), ist bei finalisierten Feldern eine *explizite* Initialisierung erforderlich, die bei der Deklaration oder in einem Konstruktor bzw. statischen Initialisierer (siehe unten) erfolgen kann. Im weiteren Programmverlauf ist bei finalisierten Feldern keine Wertänderung mehr möglich.

## *4.3 Instanzmethoden*

In einer Bauplan-Klassendefinition werden Objekte entworfen, die eine Anzahl von Verhaltenskompetenzen (Methoden) besitzen, die von anderen Programmbestandteilen (Klassen oder Objekten) per Methodenaufruf genutzt werden können. Objekte sind also Dienstleister, die eine Reihe von Nachrichten interpretieren und mit passendem Verhalten beantworten können.

Wie es im Objekt drinnen aussieht, geht andere Programmierer nichts an (*information hiding*). Seine Instanzvariablen (Eigenschaften) sind bei konsequenter Datenkapselung für die Außenwelt (d.h. für fremde Klassen) unsichtbar. Soll eine Eigenschaft trotz Datenkapselung zugänglich sein, sind entsprechende Methoden zum Lesen bzw. Verändern erforderlich. Darüber hinaus benötigt ein produktiv einsetzbares Objekt natürlich weitere Handlungskompetenzen.

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 gut repräsentieren sollen. Wenn ein anderer Programmierer z.B. ein Objekt aus unserer `Bruch`-Klasse verwendet, kann er zur Bildschirmausgabe auf die Methode `zeige()` zurückgreifen:

```
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");
}
```

Weil diese Methode für *alle* fremden Klassen verfügbar sein soll, wird per Modifikator die Schutzstufe **public** gewählt.

Da es vom Verlauf einer Bildschirmausgabe nichts zu berichten gibt, liefert `zeige()` keinen Rückgabewert. Folglich ist im Kopf der Methodendefinition der Rückgabetyp **void** angegeben.

Weil die `zeige()`-Methode nur eine einzige Arbeitsweise beherrscht, sind keine Parameter vorhanden, wobei die leere Parameterliste aber weder bei der Definition noch beim Aufruf der Methode fehlen darf. Leere Rückgaben und Parameterlisten sind keinesfalls die Regel, so dass wir uns bald näher mit den Kommunikationsregeln beim Methodenaufruf beschäftigen müssen.

In der `zeige()` - Implementierung gelingt mit einfachen Mitteln eine brauchbar formatierte Konsolenausgabe von etikettierten Brüchen, z.B.:

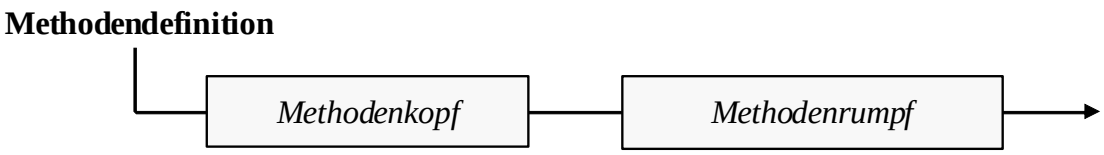
```
                  13
 Erster Bruch:  -----
                  221
```

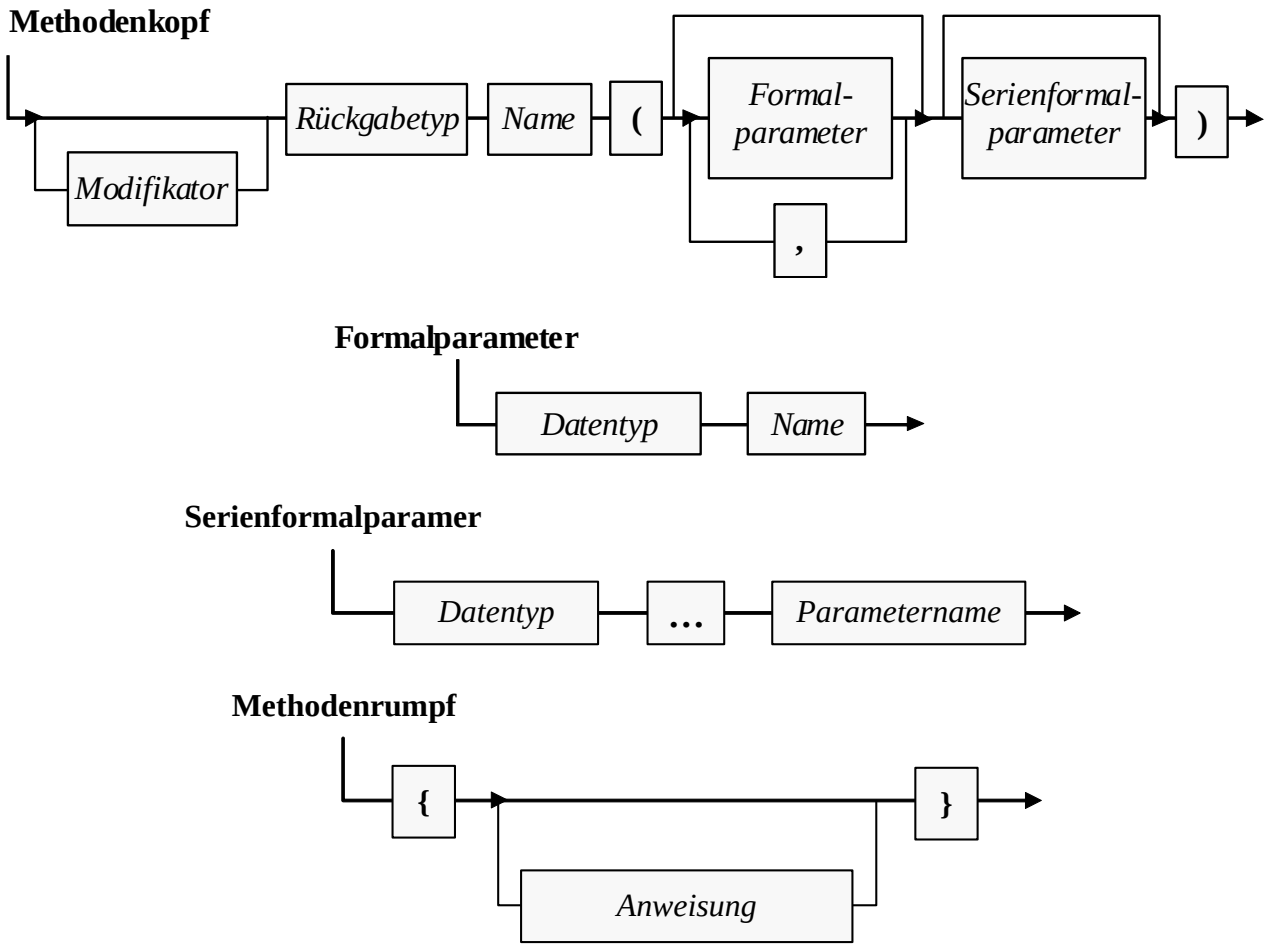
Über und unter dem Etikett steht das **String**-Objekt `luecke`, das als leere Zeichenfolge initialisiert wird. In der **for**-Schleife werden per Plusoperator Leerzeichen angehängt, bis die Länge des Etiketts erreicht ist. Diese Länge wird über den Rückgabewert der **String**-Methode **length()** ermittelt.

Während jedes Objekt einer Klasse seine eigenen Instanzvariablen auf dem Heap besitzt, 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 Variante in Abschnitt 3.1.2.2 durch eine genauere Erklärung der Formalparameterliste:






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.

Vorweg zwei Konventionen zu den **Namen von Java-Methoden**:

- Diese 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 der voreingestellte **Zugriffsschutz** verändert werden. Damit unsere Klasse `Bruch` möglichst universell einsetzbar ist, haben alle Methoden den Zugriffsmodifikator **public** erhalten. Per Voreinstellung (ohne Modifikator) ist eine Methode in allen Klassen verwendbar, die zum selben *Paket* gehören (siehe unten). Über den Modifikator **public** erhalten auch Klassen aus *fremden* Paketen das Nutzungsrecht. Soll eine Methode nur für andere Methoden der eigenen Klasse verfügbar sein, verwendet man den Modifikator **private**.

Wie Sie merken, sind genauere Erläuterungen zu Zugriffsmodifikatoren für Klassen, Instanzvariablen und Methoden erst sinnvoll, nachdem wir uns ausführlich mit Paketen beschäftigt haben.

#### *4.3.1.2 Rückgabewert und return-Anweisung*

Für den Informationstransfer von einer Methode an ihren Aufrufer kann neben Referenzparametern (siehe Abschnitt 4.3.1.3.2) auch ein Rückgabewert genutzt werden. Hier ist man auf einen einzigen Wert (von beliebigem Typ) beschränkt, doch lässt sich die Übergabe sehr elegant in den Programmablauf integrieren. 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.

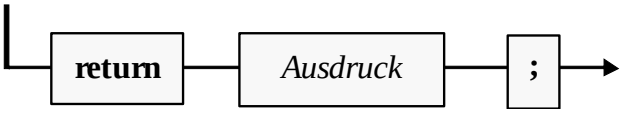
Bei der Definition einer Methode muss festgelegt werden, von welchem Datentyp ihr Rückgabewert ist. Erfolgt *keine* Rückgabe, ist der Ersatztyp **void** anzugeben.

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;
}
```

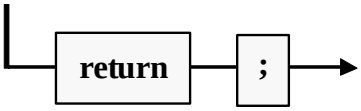
Ist der Rückgabetyp einer Methode von **void** verschieden, dann muss im Rumpf dafür gesorgt werden, dass jeder mögliche Ausführungspfad der Methode mit einer **return**-Anweisung endet, die einen Rückgabewert passenden Typs liefert:

**return-Anweisung für Methoden *mit* Rückgabewert**




Bei Methoden *ohne* Rückgabewert ist die **return**-Anweisung nicht unbedingt erforderlich, kann jedoch (in der Variante *ohne* Ausdruck) dazu verwendet werden, um die Methode vorzeitig zu beenden (z.B. im Rahmen einer bedingten Anweisung):

**return-Anweisung für Methoden *ohne* Rückgabewert**




Soll eine Methode mehr als nur einen Wert von primitivem Datentyp zurückliefern, dann muss eine Klasse als Rückgabetyp benutzt werden (siehe Abschnitt 4.4.5).

#### *4.3.1.3 Formalparameter*

Methodenparameter wurden Ihnen bisher vereinfachend als Informationen über die gewünschte Arbeitsweise einer Methode vorgestellt. Tatsächlich ermöglichen Parameter 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 sind die Formalparameter wie lokale Variablen zu verwenden, die mit den beim Aufruf übergebenen Aktualparameterwerten (siehe Abschnitt 4.3.2) initialisiert wurden.

Methodeninterne Änderungen 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, Klassen). 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 (z.B. **this** bei Objekten) weiter ansprechbar.
- **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.

#### 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 (`zpar`, `npar`) übergebenen Bruch zu seinem eigenen Wert addieren und optional (Parameter `autokurz`) das Resultat gleich kürzen:

```
public boolean addiere(int zpar, int npar, boolean autokurz) {
  if (npar != 0) {
    zaehler = zaehler*npar + zpar*nenner;
    nenner = nenner*npar;
    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 Programmeinheit. 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 {`<br>`  void primParDemo (int ipar) {`<br>`  ipar++;`<br>`  System.out.println(ipar);`<br>`  }`<br>`  public static void main(String[] args) {`<br>`  int imain = 4711;`<br>`  Prog p = new Prog();`<br>`  p.primParDemo(imain);`<br>`  System.out.println(imain);`<br>`  }`<br>`}` | `4712`<br>`4711` |

Die Klasse `Prog` ist startfähig, besitzt also eine statische Methode namens **main()**. Dort wird ein Projekt 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 zu verwenden. Bei den Parametern und beim Rückgabetyp gibt es allerdings keine Unterschiede zwischen den Instanzmethoden 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 rufende Programmeinheit 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 zugreift und dort ggf. Veränderungen vornimmt.

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.

Bei Aufruf der `addiere()`-Variante bleibt das per Referenzparameter ansprechbare Objekt unverändert. Sofern entsprechende Zugriffsrechte vorliegen, was bei Referenzparametern vom eigenen Typ stets der Fall ist, kann eine Methode das Referenzparameter-Objekt aber durchaus auch verändern. Wir erweitern unsere `Bruch`-Klasse um eine Methode `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.

In folgendem Programm wird das `Bruch`-Objekt `b1` beauftragt, die `dupliziere()`-Methode auszuführen, wobei als Parameter eine Referenz auf das Objekt `b2` übergeben wird:

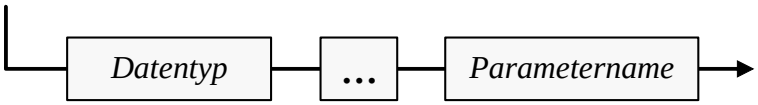
| Quellcode | Ausgabe |
|---|---|
| `class Bruchrechnung {`<br>`  public static void main(String[] args) {`<br>`  Bruch b1 = new Bruch(), b2 = new Bruch();`<br>`  b1.setzeZaehler(1);`<br>`  b1.setzeNenner(2);`<br>`  b1.setzeEtikett("b1 = ");`<br>`  b2.setzeZaehler(5);`<br>`  b2.setzeNenner(6);`<br>`  b2.setzeEtikett("b2 = ");`<br><br>`  b1.zeige();`<br>`  b2.zeige();`<br>`  b1.dupliziere(b2);`<br>`  System.out.println("b2 nach dupliziere():\n");`<br>`  b2.zeige();`<br>`  }`<br>`}` | `  1`<br>`b1 =  -----`<br>`  2`<br><br>`  5`<br>`b2 =  -----`<br>`  6`<br><br>`b2 nach dupliziere():`<br><br>`  1`<br>`b1 =  -----`<br>`  2` |

Die Referenzparametertechnik eröffnet den (berechtigten) Methoden nicht nur unbegrenzte Wirkungsmöglichkeiten, sondern spart auch Zeit und Speicherplatz beim Methodenaufruf.

#### 4.3.1.3.3 Serienparameter

Seit der Version 5.0 (bzw. 1.5) bietet Java auch Parameterlisten variabler Länge, wobei 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);
}
```

Hinter dem Serienparameter steckt ein **Array**, also 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 einer neuen Variante der **for**-

Schleife, die gemeinsam mit den Serienparametern in Java 5.0 eingeführt wurde (siehe Abschnitt 3.7.3.1). Im 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);
```

Methodenintern wird der Serienparameter wie ein Array-Parameter behandelt. Dementsprechend kann man beim Methodenaufruf an Stelle einer Serie von einzelnen Aktualparametern auch einen Array mit diesen Elementen angeben, z.B.:

```
Bruch[] ba = {b2, b3, b4};
b1.addiere(ba);
```

Wer sich darüber wundert, dass die *beiden* `addiere()`-Varianten

```
public void addiere(Bruch b) {. . .}
public void addiere(Bruch ... bar) {. . .}
```

in *einer* Klassendefinition existieren dürfen, sei auf Abschnitt 4.3.4 vertröstet.

Eine weitere Methode mit Serienparameter kennen Sie übrigens schon aus dem Abschnitt 3.2.2 über die formatierte Ausgabe mit der **PrintStream**-Methode **printf()**, die folgenden Definitionskopf besitzt:

**public PrintStream printf(String** *format***, Object ...** *args***)**

Hier wird eine Schema zur Beschreibung einer *vorhandenen Bibliotheksmethode* (im Unterschied zur Beschreibung einer Java-Syntaxregel) im Manuskript erstmals benutzt, die auch in der JDK-Dokumentation in ähnlicher Form Verwendung findet, z.B.:

```
public PrintStream printf(String format,
                          Object... args)
```

Dabei wird die Benutzung einer vorhandenen Methode erläutert durch Angabe von:

- Modifikatoren (z.B. für den Zugriffsschutz)
- Rückgabetyp
- Methodenname
- Parameterliste

Dass die Methode **printf()** eine Referenz auf das handelnde **PrintStream**-Objekt als (meist ignorierten) Rückgabewert liefert, kann uns momentan gleichgültig sein.

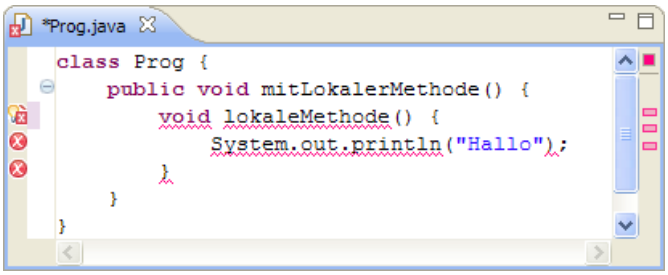
#### *4.3.1.4 Methodenrumpf*

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 von der rufenden Programmeinheit initialisiert werden, so dass diese 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.

Verschachtelte Methodendefinitionen sind verboten, z.B.:




Demgegenüber dürfen in einer Methode **lokale Klassen** definiert werden, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public void mitLokalerKlasse() {`<br>`  class LokaleKlasse {`<br>`  int meilok() {`<br>`  return 4711;`<br>`  }`<br>`  }`<br>`  LokaleKlasse w = new LokaleKlasse();`<br>`  System.out.println(w.meilok());`<br>`  }`<br><br>`  public static void main(String[] args) {`<br>`  Prog p = new Prog();`<br>`  p.mitLokalerKlasse();`<br>`  }`<br>`}` | `4711` |

Innerhalb einer lokalen Klasse sind auch Methodendefinitionen zulässig. Der Gültigkeitsbereich von lokalen Klassen ist wie bei lokalen Variablen geregelt (siehe Abschnitt 3.3.6). Programmiereinsteiger sollten auf die relativ exotische Option lokaler Klassen vorläufig verzichten.

Weil in einer Methode häufig andere Methoden aufgerufen werden, kommt es in der Regel zu mehrstufig verschachtelten Methodenaufrufen, wobei die Höhe des Stacks (Stapelspeichers) zur Verwaltung der Methodenaufrufe entsprechend wächst (siehe Abschnitt 4.3.3).

### 4.3.2 Methodenaufruf und Aktualparameter

Beim Aufruf einer Instanzmethode, z.B.:

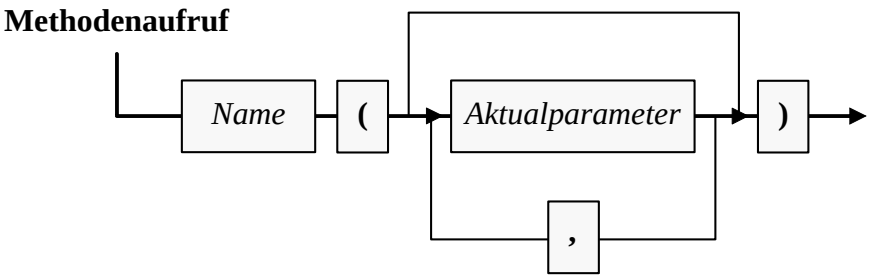
```
b1.zeige();
```

wird nach objektorientierter Denkweise eine *Botschaft* an ein Objekt geschickt:

„b1, zeige dich!“

Als Syntaxregel ist festzuhalten, dass zwischen dem Objektnamen (genauer: dem Namen der Referenzvariablen, die auf das Objekt zeigt) und dem Methodennamen der **Punktoperator** zu stehen hat.

Beim Aufruf einer Methode folgt ihrem Namen die in runde Klammern eingeschlossene Liste mit den **Aktualparametern**, wobei es sich um eine synchron zur Formalparameterliste geordnete Serie von Ausdrücken mit kompatiblen Datentypen handeln muss.




Es muss grundsätzlich eine Parameterliste angegeben werden, ggf. eine leere.

Als Beispiel 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 erlaubt sind Ausdrücke mit einem Typ, der nötigenfalls erweiternd in den Typ des zugehörigen Formalparameters gewandelt werden kann.

Liefert eine Methode einen Wert zurück, stellt ihr Aufruf einen **Ausdruck** dar und kann als Argument in komplexeren Ausdrücken auftreten, z.B.:

```
double a = 3.0, x = 2.0, xha;
xha = Math.exp(a*Math.log(x));
```

Hier wird die allgemeine Exponentialfunktion

$$f(x) = x^a$$

unter Verwendung der Definition

$$x^a = \mathrm{e}^{a \cdot \log(x)}, \quad x > 0$$

mit Hilfe der statischen Methoden **exp()** und **log()** der Klasse **Math** aus dem Paket **java.lang** realisiert.

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();
}
```

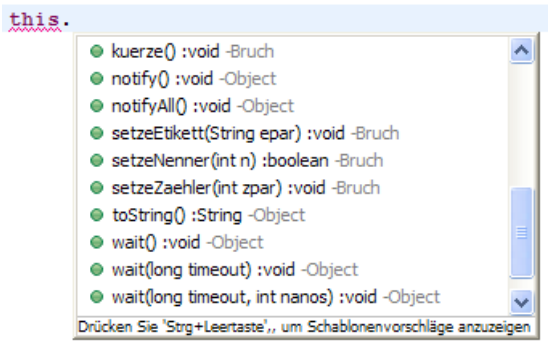
Wer auch solche Methodenaufrufe nach dem Schema

*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 erlaubten Feldern und Methodenaufrufen anzuzeigen:




So kann man lästiges Nachschlagen und Tippfehler vermeiden.

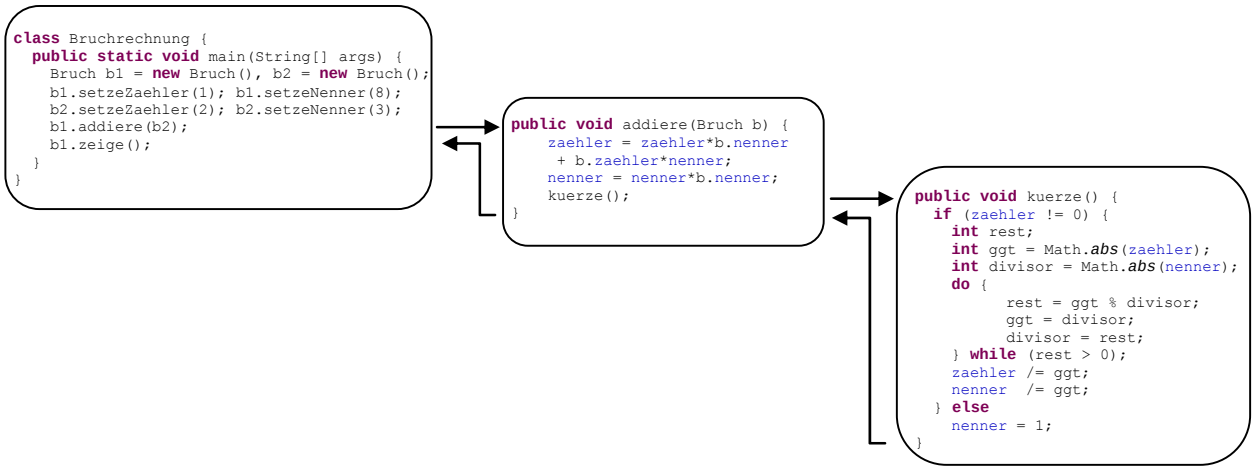
### 4.3.3 Debug-Einsichten zu (verschachtelten) Methodenaufrufen

Verschachtelte Methodenaufrufe stellen keine Besonderheit, sondern den selbstverständlichen Normalfall dar. Nichtsdestotrotz oder gerade deswegen ist es angemessen, das Geschehen etwas genauer zu betrachten. Anhand der folgenden Bruchrechnungsstartklasse

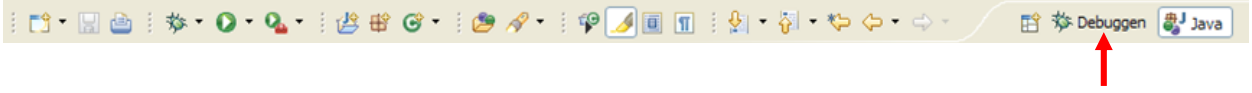
```
class Bruchrechnung {
  public static void main(String[] args) {
    Bruch b1 = new Bruch(), b2 = new Bruch();
    b1.setzeZaehler(1); b1.setzeNenner(8);
    b2.setzeZaehler(2); b2.setzeNenner(3);
    b1.addiere(b2);
    b1.zeige();
  }
}
```

soll mit Hilfe unserer Entwicklungsumgebung Eclipse untersucht werden, was bei folgender Aufrufverschachtelung geschieht:

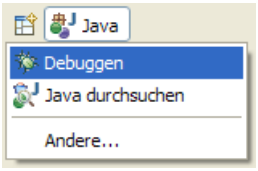
- Die statische Methode **main()** der Klasse `Bruchrechnung` ruft die `Bruch`-Instanzmethode `adddiere()`.
- Die `Bruch`-Instanzmethode `adddiere()` ruft die `Bruch`-Instanzmethode `kuerze()`.




Wir verwenden dabei die zur Fehlersuche konzipierte Debug-Technik von Eclipse und wechseln daher von der voreingestellten 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:

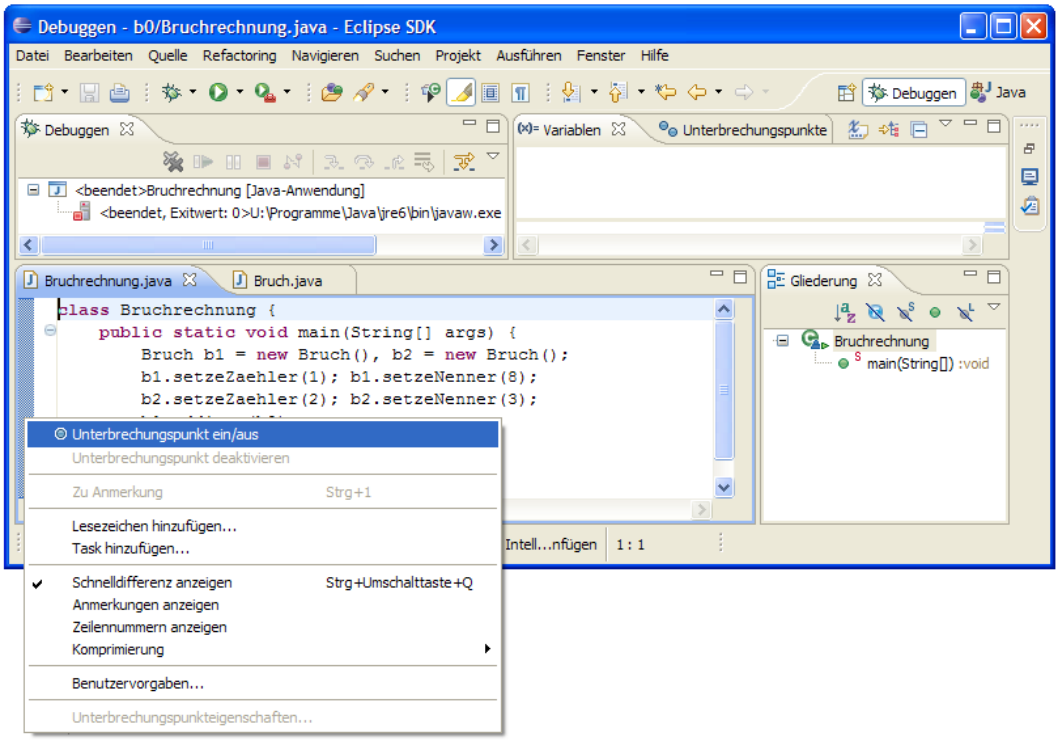



Das Programm s oll an mehreren Stellen durch einen so genannten **Unterbrechungspunkt** (engl. *breakpoint*) angehalten werden, so dass wir jeweils die Lage im Hauptspeicher inspizieren können. Um einen Unterbrechungspunkt festzulegen, …

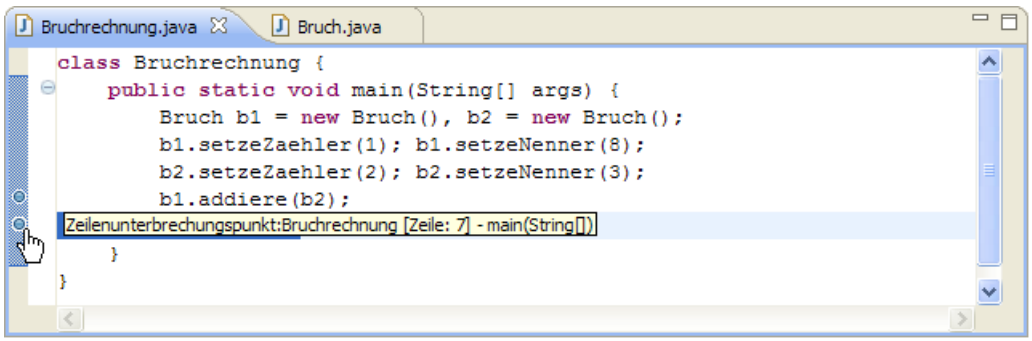
- setzt man in der Infospalte des Editors einen Rechtsklick in Höhe der betroffenen Zeile
- und wählt im Kontextmenü das Item **Unterbrechungspunkt ein/aus**

Zum Entfernen eines Unterbrechungspunkts wählt man das Kontextmenü-Item erneut.

Hier wird die **main()**-Methode vor dem Aufruf der Methode `addiere()` angehalten:




Noch bequemer klappt das Setzen bzw. Entfernen eines Unterbrechungspunkts per Mausdoppelklick in die Infospalte neben der betroffenen Anweisung, z.B.:




Setzen Sie weitere Unterbrechungspunkte …

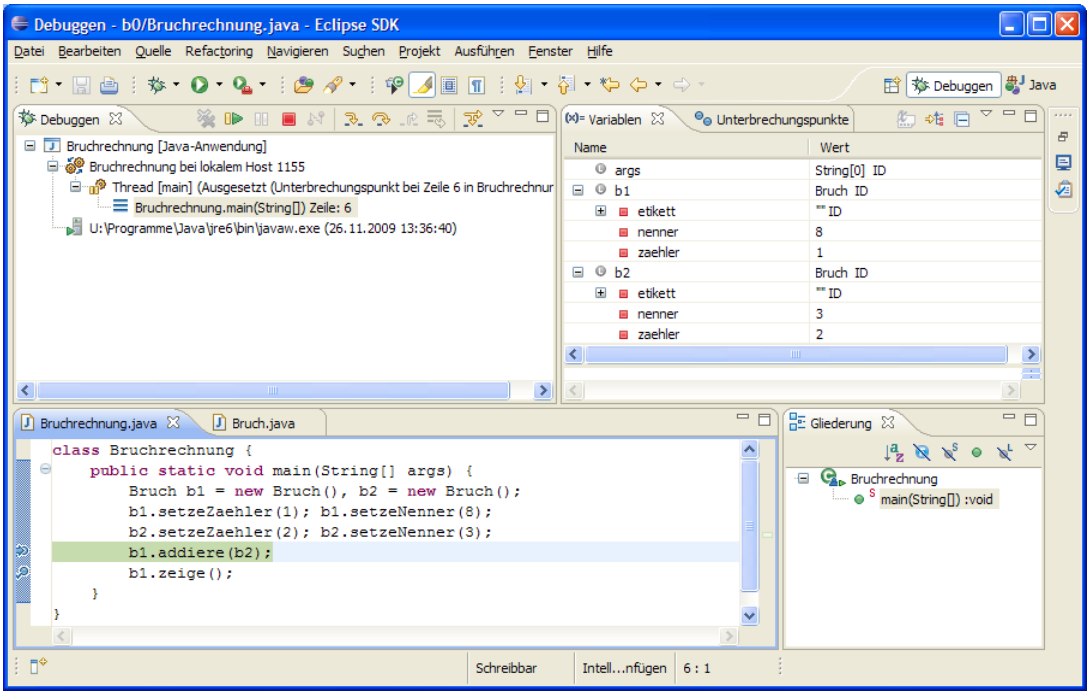
- in der Methode **main()** vor den `zeige()`-Aufruf,
- in der `Bruch`-Methode `addiere()` vor den `kuerze()`-Aufruf,
- in der `Bruch`-Methode `kuerze()` vor die Anweisung „`ggt = divisor;`“ im Block der **do-while** - Schleife.

Starten Sie das Programm im Debug-Modus über den Menübefehl

**Ausführen > Debug ausführen als > Java-Anwendung**

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 befinden. Bei Erreichen des ersten Unterbrechungspunkts (Anweisung „`b1.addiere();`“ in **main()**) ist nur der Stack Frame der Methode **main()** vorhanden:

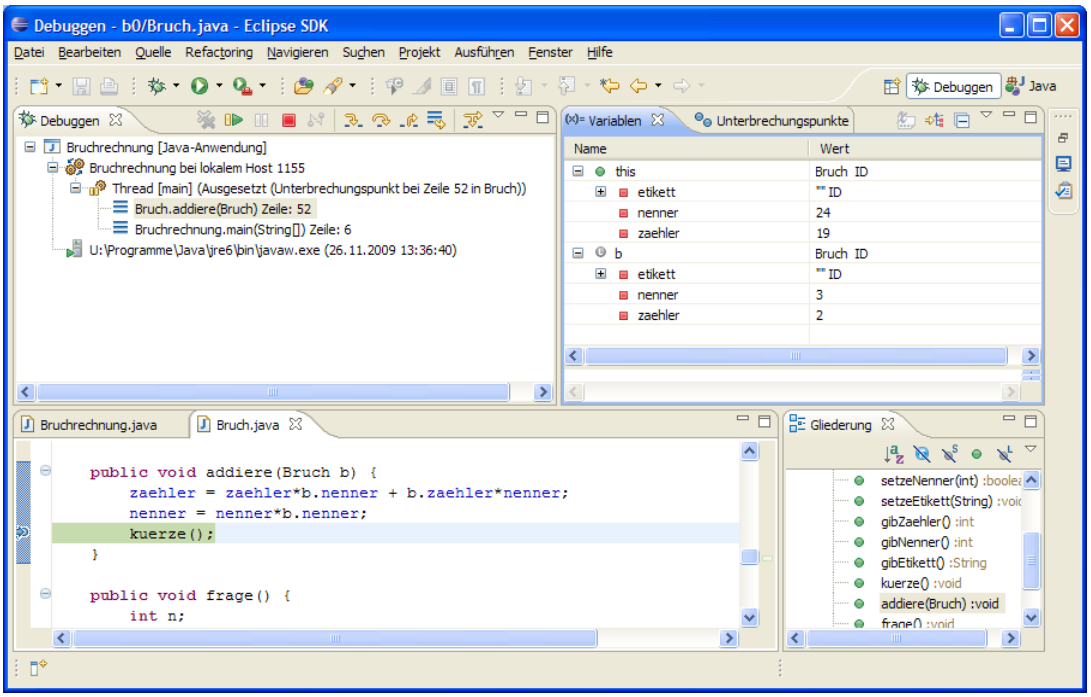



Im **Variablen**-Fenster der **Debuggen**-Perspektive sind die lokalen Variablen der Methode **main()** zu sehen:

- Parameter `args`
- die lokale Referenzvariablen `b1` und `b2`
  Es werden auch die Instanzvariablen der referenzierten `Bruch`-Objekte angezeigt.

Der Ausgabebereich mit dem Konsolenfenster wurde aus Platzgründen per Mausklick auf das Symbol  minimiert und befindet sich als Symbolleiste am rechten Rand des Eclipse-Fensters samt Schalter  zum Wiederherstellen.

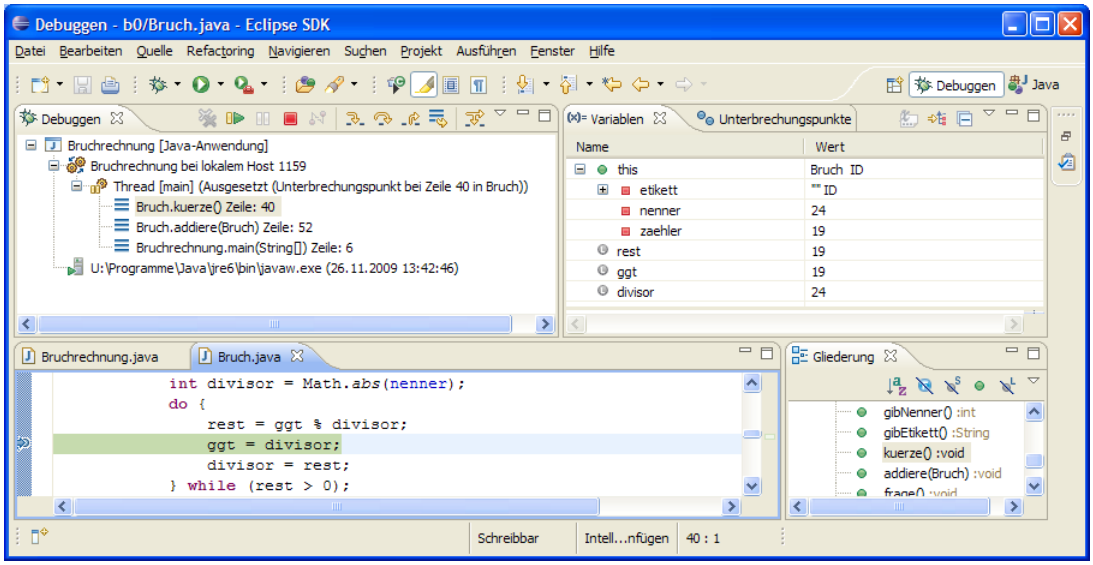
Lassen Sie das Programm mit dem Schalter  oder der Taste **F8** fortsetzen. Beim Erreichen des zweiten Unterbrechungspunkts (Anweisung „`kuerze();`“ in dem Methode `addiere()`) liegen die Stack Frames der Methoden `addiere()` und **main()** übereinander:




Das **Variablen**-Fenster zeigt als lokale Variablen der Methode `addiere()`:

- **this** (Referenz auf das handelnde 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:

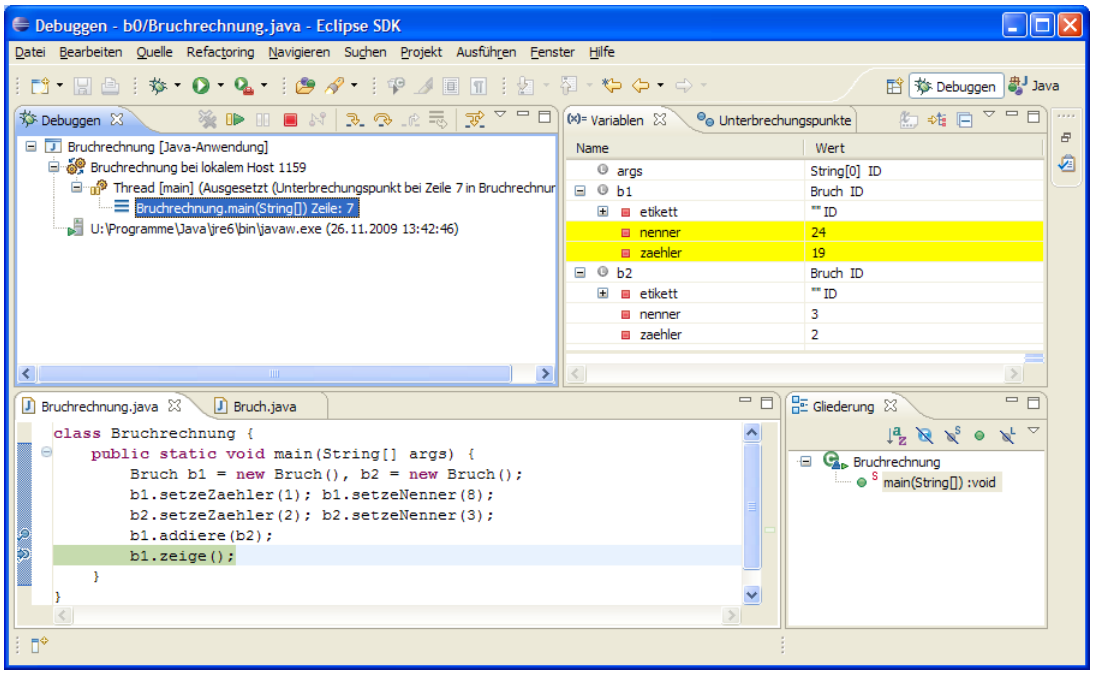



Das **Variablen**-Fenster zeigt als lokale Variablen der Methode `kuerze()`:

- **this** (Referenz auf das handelnde Objekt)
- die lokalen (im Block zur **if**-Anweisung deklarierten) Variablen `rest`, `ggt` und `divisor`.

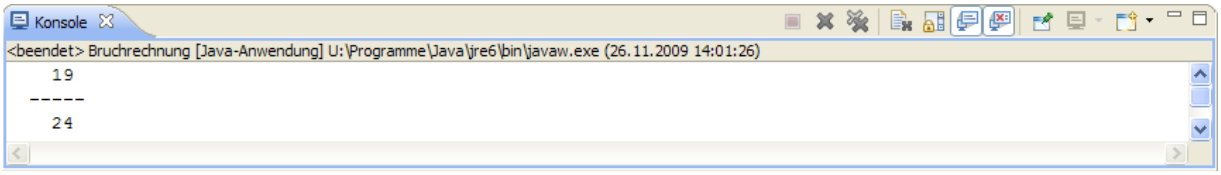
Weil sich der dritte Unterbrechungspunkt in einer **do-while** - Schleife befindet, sind mehrere Fortsetzungsbefehle bis zum Verlassen der Methode `kuerze()` erforderlich.

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.

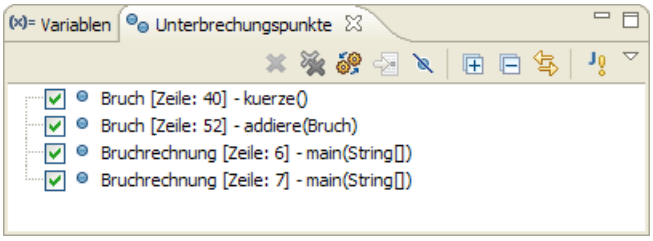
Beenden Sie das Programm durch einen letzten Fortsetzungsklick auf den Schalter , wobei das zuvor aus Platzgründen minimierte Konsolen-Fenster mit der Programmausgabe erscheint:




Im Fenster (in der Sicht) **Unterbrechungspunkte**, das bei aktiver **Debuggen**-Perspektive mit dem Menübefehl

**Fenster > Sicht anzeigen > Unterbrechungspunkte**

verfügbar ist, sind alle Unterbrechungspunkte aufgelistet:




Das Kontextmenü dieses Fensters beitet die Möglichkeit, alle Unterbrechungspunkte zu löschen.

Kehren Sie per Mausklick auf den Schalter Java am rechten Rand der Symbolleiste 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* von Methoden vor.

Eine Überladung ist erlaubt, wenn sich die *Signaturen* der beteiligten Methoden unterscheiden. Zwei Methoden besitzen genau dann *dieselbe* Signatur, wenn die beiden folgenden Bedingungen erfüllt sind:[23]

- Die Namen sind identisch.
- Die Parameterlisten 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 wohl daraus, dass der Rückgabewert einer Methode in Anweisungen oft keine Rolle spielt (ignoriert wird).

Ist bei einem Methodenaufruf die angeforderte Überladung nicht eindeutig zu bestimmen, meldet der Compiler einen Fehler. Um diese Konstellation im Bruchrechnungsbeispiel zu provozieren, sind einige Verrenkungen nötig:

- Die `Bruch`-Instanzvariablen `zaehler` und `nenner` erhalten vorübergehend den Datentyp **long**.
- Es werden zwei neue `addiere()`-Überladungen mit wenig sinnvollen Parameterlisten definiert:

```
public void addiere(long zpar, int npar) {
  if (npar == 0) return;
  zaehler = zaehler*npar + zpar*nenner;
  nenner = nenner*npar;
}
public void addiere(int zpar, long npar) {
  if (npar == 0) return;
  zaehler = zaehler*npar + zpar*nenner;
  nenner = nenner*npar;
}
```

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 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)
```

---

[23] Bei den später zu behandelnden *generischen* Methoden muss die Liste mit den Kriterien für die Identität von Signaturen erweitert werden.

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 berechnen:

  **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 allerdings *generische Methoden* (siehe Abschnitt 6) eine elegantere Lösung für die Unterstützung verschiedener Datentypen, z.B.

```
static <T extends Comparable<T>> T max(T x, T y) {
 return x.compareTo(y) > 0 ? x : y;
}
```

- Für eine Methode sollen unterschiedliche umfangreiche Parameterlisten angeboten werden, sodass zwischen einer bequem aufrufbaren Standardausführung (z.B. mit leerer Parameterliste) und einer individuell gestalteten Ausführungsvariante gewählt werden kann.

## *4.4 Objekte*

Ein zentrales Ziel der OOP ist die Produktion von Software mit hohem Recycling-Potential. Daher wurde im Bruchrechnungsprojekt die recht universell verwendbare `Bruch`-Klasse konzipiert, die schon in Programmen mit stark unterschiedlicher Benutzerschnittstelle (Konsole versus GUI) zum Einsatz kam. In Abschnitt 4.4 geht es darum, wie man Objekte solcher Klassen erzeugen und nutzen kann.

### 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 erklären, beginnen wir mit einer einfacheren Variante *ohne* Initialisierung:

```
Bruch b;
```

Hier wird die Referenzvariable `b` mit dem Datentyp `Bruch` deklariert, die folgende Werte annehmen kann:

- die Adresse eines `Bruch`-Objekts
  In der Variablen wird also kein komplettes `Bruch`-Objekt mit sämtlichen Instanzvariablen abgelegt, sondern ein **Verweis** (eine **Referenz**) auf einen Ort im Heap-Bereich des programmeigenen Speichers, wo sich ein `Bruch`-Objekt befindet.
- **null**
  Dieses Referenzliteral steht für einen leeren Verweis. Eine Referenzvariable mit diesem Wert ist nicht undefiniert, sondern zeigt explizit auf nichts.

Wir nehmen nunmehr offiziell und endgültig zur Kenntnis, dass *Klassen als Datentypen* verwendet werden können und haben damit in Java-Programmen folgende Datentypen zur Verfügung:

- **Primitive Typen** (**boolean**, **char**, **byte**, ..., **double**)
- **Klassentypen**
  Es kommen Klassen aus dem Java-API und selbst definierte Klassen in Frage.

### 4.4.2 Objekte erzeugen

Damit z.B. der folgendermaßen deklarierten Referenzvariablen `b` vom Datentyp `Bruch`

```
Bruch b;
```

ein Verweis auf ein `Bruch`-Objekt als Wert zugewiesen werden kann, muss ein solches Objekt erst erzeugt werden, was per **new**-Operator geschieht, z.B. im folgenden Ausdruck:

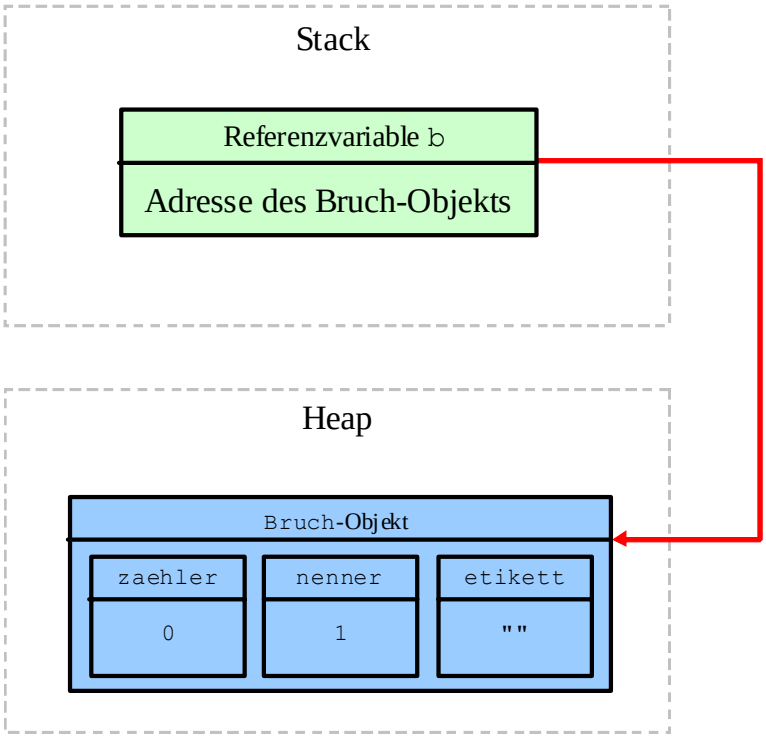
```
new Bruch()
```

Als Operanden erwartet **new** 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 des folgenden Startklasse

```
class Bruchrechnung {
  public static void main(String[] args) {
    Bruch b = new Bruch();
      . . .
  }
}
```

wird die vom **new**-Operator gelieferte Adresse mit dem Zuweisungsoperator in die lokale Referenzvariable `b` geschrieben. Es resultiert die folgende Situation im Speicher des Programms:




Während lokale Variablen bereits beim Aufruf einer Methode (also unabhängig vom konkreten Ablauf) im **Stack**-Bereich des programmeigenen Hauptspeichers angelegt werden, entstehen Objekte (mit ihren Instanzvariablen) erst bei der Auswertung des **new**-Operators. Sie erscheinen auch nicht auf dem Stack, sondern werden im **Heap**-Bereich des programmeigenen Hauptspeichers angelegt.

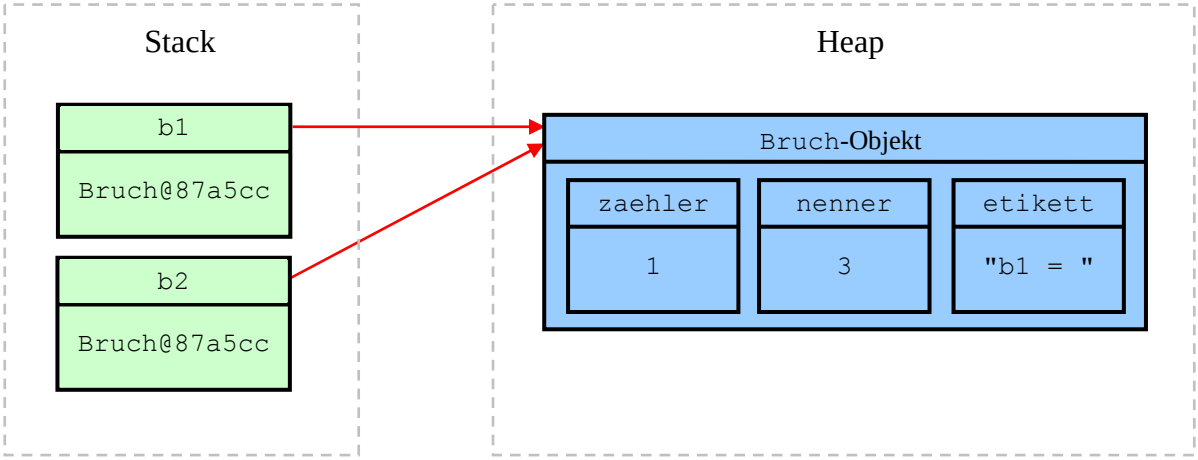
In einem Programm können *mehrere* Referenzvariablen auf *dasselbe* Objekt zeigen, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `class Bruchrechnung {`<br>`  public static void main(String[] args) {`<br>`  Bruch b1 = new Bruch();`<br>`  b1.setzeZaehler(1);`<br>`  b1.setzeNenner(3);`<br>`  b1.setzeEtikett("b1 = ");`<br>`  Bruch b2 = b1;`<br>`  b1.setzeEtikett("b2 = ");`<br>`  b1.zeige();`<br>`  }`<br>`}` | `  1`<br>`b2 = -----`<br>`  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 automatisch aufgerufen werden, um deren Instanzvariablen zu initialisieren. 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, dann kommt ein **Standardkonstruktor** zum Einsatz. Weil dieser 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 Basisklassenkonstruktors (siehe unten) und hat dieselbe Schutzstufe wie die Klasse, so dass beim Standardkonstruktor der Klasse `Bruch` die Schutzstufe **public** resultieren würde.

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:

- Der Konstruktor trägt denselben Namen wie die Klasse.
- Es darf eine Parameterliste definiert werden, was zum Zweck der Initialisierung ja auch unumgänglich ist.
- Der Konstruktor liefert grundsätzlich *keinen* Rückgabewert, und es wird *kein* Typ angegeben, auch nicht der Ersatztyp **void**, mit dem wir bei gewöhnlichen Methoden den Verzicht auf einen Rückgabewert dokumentieren müssen.
- Sobald man einen eigenen Konstruktor definiert, steht der Standardkonstruktor *nicht* mehr zur Verfügung.
- Ist weiterhin ein paramameterfreier Konstruktor erwünscht, so muss dieser *zusätzlich* definiert werden.
- Selbstdefinierte Konstruktoren haben wie andere Klassen-Mitglieder (Felder, Methoden) per Voreinstellung die Schutzstufe **package**, sind also in allen Klassen desselben Pakets nutzbar. Mit der deklarierten Schutzstufe **private** kann man z.B. verhindern, dass ein Konstruktor von fremden Klassen benutzt wird.
- Es sind generell beliebig viele Konstruktoren möglich, die alle denselben Namen und jeweils eine individuelle Parameterliste haben müssen. Das Überladen von Methoden (vgl. Abschnitt 4.3.4) ist also auch bei Konstruktoren erlaubt.

Die folgende Variante unseres Beispielprogrammsenthält 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 zpar, int npar, String epar) {
    zaehler = zpar;
    nenner  = npar;
    etikett = epar;
  }

  public Bruch() {}

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

  . . .
  . . .

}
```

Im folgenden Testprogramm werden beide Konstruktoren eingesetzt:

| Quellcode | Ausgabe |
|---|---|
| `class Bruchrechnung {`<br>`  public static void main(String[] args) {`<br>`  Bruch b1 = new Bruch(1, 2, "b1 = ");`<br>`  Bruch b2 = new Bruch();`<br>`  b1.zeige();`<br>`  b2.zeige();`<br>`  }`<br>`}` | `  1`<br>`b1 =  -----`<br>`  2`<br><br>`  0`<br>`-----`<br>`  1` |

Konstruktoren können nicht direkt aufgerufen, sondern nur als Argument des **new**-Operators verwendet werden. Eine Ausnahme von dieser Regel könnte man in der anschließend demonstrierten Möglichkeit sehen, im Anweisungsblock eines Konstruktors einen anderen Konstruktor derselben Klasse über das Schlüsselwort **this** aufzurufen:

```
public Bruch() {
  this(0, 1, "unbenannt");
}
```

### 4.4.4 Abräumen überflüssiger Objekte durch den Garbage Collector

Wenn keine Referenz mehr auf ein Objekt zeigt, wird es vom **Garbage Collector** (Müllsammler) der virtuellen Maschine automatisch entsorgt, und der belegte Speicher wird frei gegeben.

In unseren bisherigen Bruchrechnungs-Beispielprogrammen verschwindet die letzte (und einzige) Referenz auf ein `Bruch`-Objekt mit dem Beenden der **main()**-Methode, in der das Objekt erstellt wurde. Es ist jedoch durchaus möglich (und normal), dass ein Objekt die erzeugende Methode überlebt (siehe Abschnitt 4.4.5).

Andererseits kann man ein Objekt zu jedem beliebigen Zeitpunkt „aufgeben“, indem man alle Referenzen entfernt. Dazu setzt man die entsprechenden Referenzvariablen entweder auf den Wert **null** oder weist ihnen eine andere Referenz zu, z.B.:

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

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 Objektmassen, 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** (*memory leaks*) durch die vergessene Freigabe des Speichers zu überflüssig gewordenen Objekten.

Der Garbage Collector wird im Normalfall nur dann tätig, wenn die virtuelle Maschine Speicher benötigt und gerade nichts Wichtigeres zu tun hat, 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.

Sollen die Objekte einer Klasse vor dem Entsorgen noch spezielle Aufräumaktionen durchführen, dann muss eine Methode namens **finalize()** nach folgendem Muster definiert werden, die ggf. vom Garbage Collector aufgerufen wird, z.B.:

```
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 4.4.5.2) das aktuell handelnde Objekt angesprochen. Bei der automatischen Konvertierung der Referenz in eine Zeichenfolge wird die vom Laufzeitsystem verwaltete Objektbezeichnung zu Tage fördert.

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.

In der Regel 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.

### 4.4.5 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.

#### *4.4.5.1 Rückgabewerte mit Referenztyp*

Bisher haben die innerhalb einer Methode erzeugten Objekte das Ende der Methode nicht überlebt, waren jedenfalls anschließend nicht mehr nutzbar. Weil keine Referenz außerhalb der Methode existierte, wurden die Objekte dem Garbage Collector überlassen. Soll ein methodenintern erzeugtes Objekt weiter existieren, 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 die erzeugende Methode 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
```

#### 4.4.5.2 *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 void 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 der Deklaration bzw. Definition kaum Unterschiede zwischen einem Instanz- und dem analogen Klassenmitglied.

Abgesehen vom Standardkonstruktor (siehe Abschnitt 4.4.3) gilt auch bei den statischen Mitgliedern 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 Klasen werden ausgeschlossen.
  - **public**
    Alle Klassen dürfen zugreifen.

### 4.5.1 Klassenvariablen

In unserem Bruchrechnungsbeispiel soll ein statisches Feld dazu dienen, die Anzahl der 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 zpar, int npar, String epar) {
    zaehler = zpar;
    nenner  = npar;
    etikett = epar;
    anzahl++;
  }

  public Bruch() {anzahl++;}

  . . .
  . . .

}
```

Die (automatisch auf Null initialisierte) Klassenvariable `anzahl` ist als **private** deklariert, also nur in Methoden der eigenen Klasse sichtbar. Sie wird in den beiden Instanzkonstruktoren sowie in der Methode `klone()` inkrementiert.[24]

Im Java-Editor der Entwicklungsumgebung Eclipse 3.x werden übrigens statische Variablen per Voreinstellung durch *kursive Schrift* gekennzeichnet.

Sofern Methoden fremder Klassen der direkte Zugriff auf eine Klassenvariable gewährt wird, müssen diese dem Variablennamen einen Präfix aus Klassennamen und Punktoperator voranstellen, z.B.:

```
System.out.println("Bisher wurden "+Bruch.anzahl+" Brueche erzeugt");
```

In unserem Beispiel wird das statische Feld `anzahl` aber mit **private**-Modifikator deklariert, so dass der direkte Zugriff klasseneigenen Methoden vorbehalten bleibt.

Während jedes Objekt einer Klasse über einen eigenen Satz mit allen Instanzvariablen verfügt, die beim Erzeugen des Objekts auf dem Heap angelegt werden, existiert eine klassenbezogene Variable nur *einmal*. Sie wird beim Laden der Klasse in der *Method Area* des programmeigenen Speichers angelegt und automatisch genau so initialisiert wie eine Instanzvariable (vgl. Abschnitt 4.2.3).

Wir verwenden übrigens 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 (vgl. Abschnitt 3.3.7), wie die **out**-Deklaration aus der **System**-Klassendefinition zeigt:

```
public final static PrintStream out = nullPrintStream();
```

---

[24] Wer den Abschnitt 4.4.4 komplett gelesen hat, könnte geneigt sein, in einer **finalize()**-Überschreibung die Anzahl der aktuell ansprechbaren `Bruch`-Objekte zu dekrementieren, wobei aber aufgrund der schwer vorhersehbaren Garbage Collector - Arbeitszeiten nur eine obere Schranke für die Anzahl der ansprechbaren `Bruch`-Objekte entstehen würde.

Auch bei häufig benötigten Konstanten bewährt sich die Kombination aus den drei Modifikatoren **public**, **static** und **final**, z.B. beim **double**-Feld **PI** in der API-Klasse **Math** (Paket **java.lang**):

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

Analog zu finalisierten Instanzfeldern (siehe Abschnitt 4.2.5) kann die obligatorische Initialisierung von Read-Only-Klassenvariablen bei der Deklaration oder im statischen Initialisierer (siehe Abschnitt 4.5.4) erfolgen.

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.<br>Sie werden ungültig, wenn das Objekt nicht mehr referenziert ist. | Klassenvariablen werden beim Laden der Klasse angelegt und initialisiert.[25] |

### 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,
  existieren, bis der innerste Block endet.
- **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.

---

[25] 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 2005, S. 330). Eine vom regulären Klassenlader der JRE geladene Klasse wird nicht vor dem Ende des Programms entladen (Ullenboom 2009, Abschnitt 10.4).

- **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. **out** in der Klasse **System** im Paket **java.lang**).

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, Lebensdauer, Gültigkeitsbereich und Initialisierung 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-`<br>`// Methode frage()`<br>`int n;` | `// aus Klasse Bruch`<br>`private int zaehler;` | `// aus Klasse Bruch`<br>`static private int anzahl;` |
| | Referenz | `// aus der Bruch-`<br>`// Methode zeige()`<br>`String luecke = "";` | `// aus Klasse Bruch`<br>`private String etikett="";` | `// aus Klasse System`<br>`public static final`<br>`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 kennen gelernt. So wird im **Math**-Quellcode das Instantiieren 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, je nach Paketzugehörigkeit der Klasse und vorhandenen **import**-Anweisungen am Anfang des Quellcodes.

Oft ist es sinnvoll, klassenbezogene Kompetenzen mit objektbezogenen zu kombinieren, speziell bei Klassen, die neben Instanzvariablen auch Klassenvariablen besitzen. 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 {`<br>`  public static void main(String[] args) {`<br>`  System.out.println(Bruch.hanz() + " Brueche erzeugt");`<br>`  Bruch b1 = new Bruch(1, 2, "Bruch 1");`<br>`  Bruch b2 = new Bruch(5, 6, "Bruch 2");`<br>`  b1.zeige();`<br>`  b2.zeige();`<br>`  System.out.println(Bruch.hanz() + " Brueche erzeugt");`<br>`  }`<br>`}` | `0 Brueche erzeugt`<br>`Bruch 1 -----`<br>`1`<br>`2`<br>`Bruch 2 -----`<br>`5`<br>`6`<br>`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;
}
```

Oft 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."

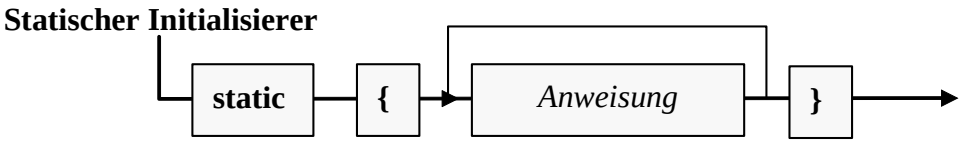
Sofern eine statische Methode eine Referenz zu einem Objekt besitzt, das sie eventuell selbst erzeugt hat, kann sie im Rahmen der eingeräumten Zugriffsrechte (bei Objekten der eigenen Klasse also z.B. uneingeschränkt) auf Instanzvariablen dieses Objekts zugreifen und seine Instanzmethoden aufrufen. In einer Klassenmethode eine Instanzmethode ohne vorangestellte Objektreferenz aufzurufen, wäre reichlich sinnlos. Wer einen Auftrag an ein Objekt schicken möchte, muss den Empfänger natürlich benennen.

In früheren Abschnitten waren mit *Methoden* stets *objektbezogene* Methoden (*Instanzmethoden*) gemeint. Dies soll auch weiterhin so gelten.

### 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 Initialisierungsblöcke, die beim Laden der Klasse ausgeführt werden. Gosling et al. (2005, S. 239) sprechen hier

von *static initializers*, doch liegt auch die Bezeichnung *statische Konstruktoren* ziemlich richtig. Ein syntaktischer Unterschied zu den Instanzkonstruktoren besteht jedenfalls darin, dass beim statischen Initialisierungsblock *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). Enthält eine Klassendefinition *mehrere* statische Initialisierungsblöcke, werden diese 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`;

```
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 Konstruktor noch das Laden der Klasse, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `class Bruchrechnung {`<br>`  public static void main(String[] args) {`<br>`  Bruch b = new Bruch();`<br>`  b.zeige();`<br>`  }`<br>`}` | `Klasse Bruch geladen`<br>`  5`<br>`  -----`<br>`  9` |

Weil sich die Klasse **Random** in Paket **java.util** befindet, das (im Gegensatz zu **java.lang**) *nicht* prinzipiell importiert wird, bestehen folgende Möglichkeiten zur Ansprache der Klasse (vgl. Abschnitt 3.1.6):

- Verwendung des voll vollqualifizierten Klassennamens (mit vorangestelltem Paketnamen), z.B.:

  ```
  java.util.Random zuf = new java.util.Random();
  ```

- Die Klasse oder das gesamte Paket per **import**-Anweisung in die Quellcodedatei importieren, um die Klasse anschließend ohne vorangestellten Paketnamen ansprechen zu können, z.B.:

```
import java.util.Random;
  . .
Random zuf = new Random();
  . .
```

### 4.6 Rekursive Methoden

Innerhalb einer Methode darf man selbstverständlich nach Belieben *andere* Methoden aufrufen. Es ist aber auch zulässig und in vielen Situationen 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. Zu einem rekursiven Algorithmus (per Selbstaufruf einer Methode) existiert stets auch ein iterativer Algorithmus (per Wiederholungsanweisung).

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 Varianten des Euklidischen Lösungsverfahrens kennen gelernt: In Abschnitt 1.1 wurde ein sehr einfacher Algorithmus benutzt, den Sie später in einer Übungsaufgabe (siehe Abschnitt 3.9) 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 ausgelagert wird:

```
private int ggTi(int a, int b) {
  int rest;
  do {
    rest = a % b;
    a = b;
    b = rest;
  } while (rest > 0);
  return a;
}

public void kuerze() {
  if (zaehler != 0) {
    int teiler = ggTi(Math.abs(zaehler), Math.abs(nenner));
    zaehler /= teiler;
    nenner  /= teiler;
  } else
      nenner = 1;
}
```

Die iterative 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 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 von `b`:

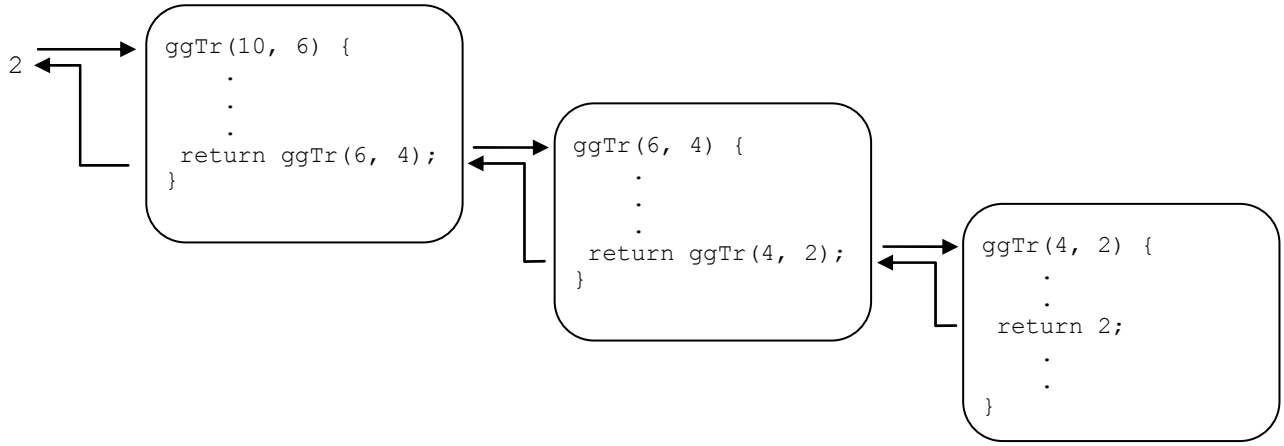
  ```
  return b;
  ```

- Anderenfalls wird das Problem, den ggT von `a` und `b` zu finden, 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 recht elegant im Ausdruck der **return**-Anweisung:

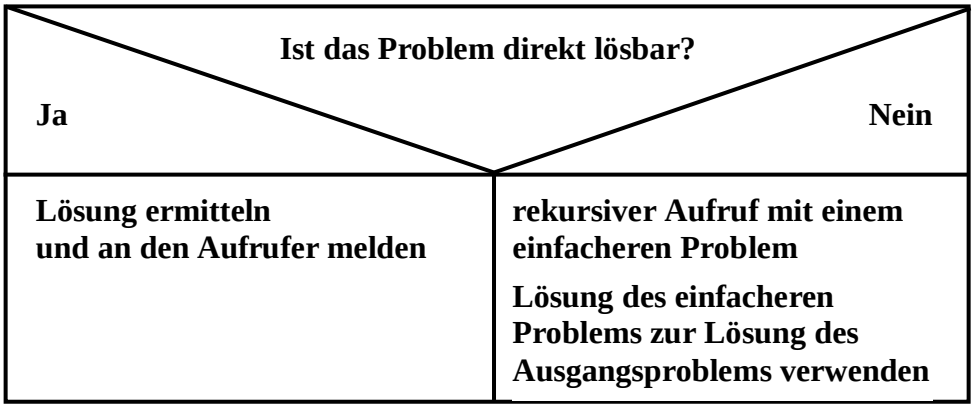
  ```
  return ggTr(b, rest);
  ```

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 ein rekursiver Algorithmus 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
        at Bruch.ggTr(Bruch.java:46)
```

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 Aggregation

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 *Aggregation* von Klassen eine sehr 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 Firma) enthält andere Objekte[26] (z.B. Mitarbeiter, Kunden), die ihrerseits wiederum Objekte enthalten (z.B. ein Gehaltskonto und einen Terminkalender bei den Mitarbeitern) usw.

Wegen der großen Bedeutung der Aggregation soll ihr ein ausführliches Beispiel gewidmet werden, obwohl der aktuelle Abschnitt nur einen neuen Begriff für eine längst vertraute Situation bringt (Felder mit Referenztyp). Wir erweitern das Bruchrechnungsprogramm um eine Klasse namens `Aufgabe`, die Trainingssitzungen unterstützen soll. 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;
  }
```

[26] 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);
    System.out.print("\nWelchen Zaehler hat Dein Ergebnis:     ");
    antwort.setzeZaehler(Simput.gint());
    System.out.println("                                      ------");
    System.out.print("Welchen Nenner hat Dein Ergebnis:      ");
    antwort.setzeNenner(Simput.gint());
  }

  public void pruefe() {
    frage();
    if (korrekt())
      System.out.println("\n Gut!");
    else {
      System.out.println();
      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 Ergebnis.

In folgendem Programm wird die Klasse `Aufgabe` für ein Bruchrechnungstraining eingesetzt:

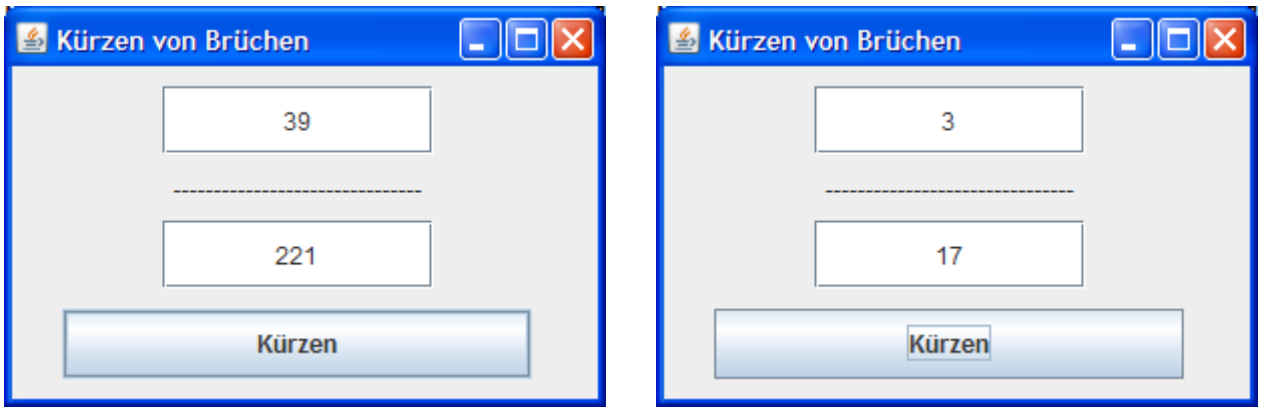
```
class Bruchrechnung {
  public static void main(String[] args) {
    Aufgabe auf = new Aufgabe('+', 1, 2, 2, 5);
    auf.pruefe();
    auf.neueWerte('*', 3, 4, 2, 3);
    auf.pruefe();
  }
}
```

Man kann immerhin schon ahnen, wie die praxistaugliche Endversion des Programms einmal arbeiten wird:

```
Berechne bitte:

   1               2
------     +     -----
   2               5

Welchen Zaehler hat Dein Ergebnis:     3
                                     ------
Welchen Nenner hat Dein Ergebnis:      7

                           9
 Das korrekte Ergebnis: -----
                           10


Berechne bitte:

   3               2
------     *     -----
   4               3

Welchen Zaehler hat Dein Ergebnis:     6
                                     ------
Welchen Nenner hat Dein Ergebnis:      12

 Gut!
```

## 4.8 Bruchrechnungsprogramm mit 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 graphischer Benutzerschnittstelle (engl. *Graphical User Interface*, *GUI*) als Belohnung und Motivationsquelle angemessen. Schließlich gilt es in diesem Kurs auch die Erfahrung zu vermitteln, dass Programmieren Spaß machen kann. Wir erstellen unter Verwendung der Klasse `Bruch` mit Hilfe des Eclipse - Plugins **Visual Editor** (**VE**) ein Bruchkürzungsprogramm mit graphischer Benutzerschnittstelle:




Aufgrund der individuellen Oberflächengestaltung kommt die in Abschnitt 3.8 vorgestellte Klasse **JOptionPane** mit statischen Methoden zur bequemen Realisation von Standarddialogen *nicht* in Frage.

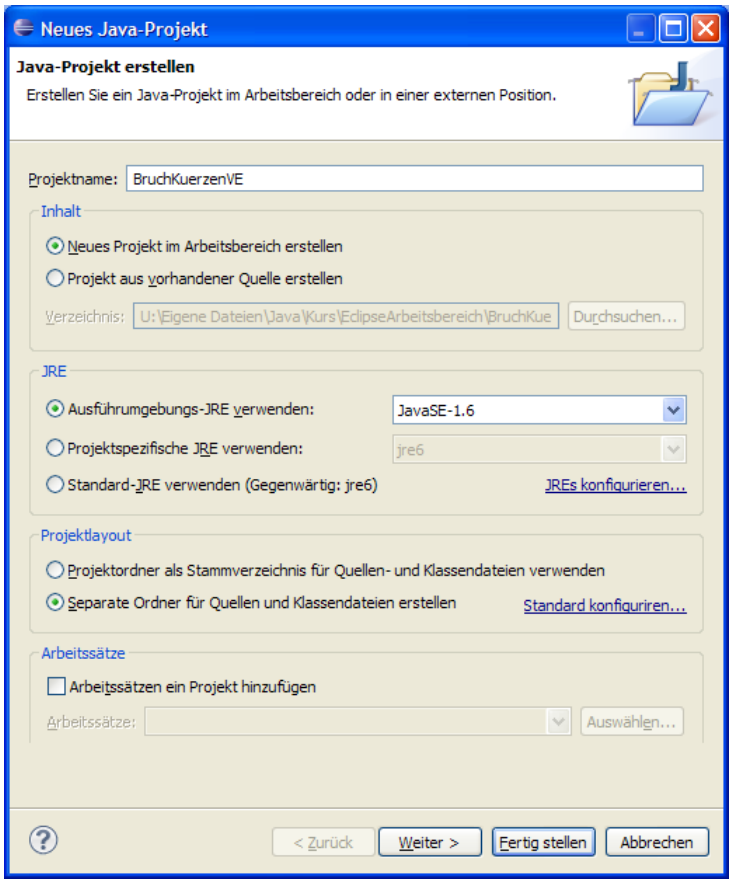
Indem der Visual Editor (oder ein anderer Assistent für Routineaufgaben) wesentliche Teile des Quellcodes erstellt, wird das Programmieren erheblich vereinfacht und beschleunigt. Diese Vorgehensweise bezeichnet man als *Rapid Application Development* (RAD). Grundsätzlich sollten man in der Lage sein, den von Assistenten erstellten Quellcode vollständig verstehen und modifizieren zu können. In diesem Abschnitt werden wir uns aber auf lokale Einblicke beschränken, weil bei der GUI-Programmierung einige noch nicht behandelte Themen beteiligt sind.

### 4.8.1 Projekt mit visueller Klasse anlegen

Legen Sie über den Menübefehl

**Datei > Neu > Java-Projekt**

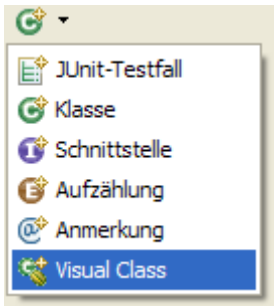
ein neues Java-Projekt an, z.B. mit dem Namen `BruchKuerzenVE`:




Tragen Sie den **Projektnamen** ein, und quittieren Sie mit einem Kick auf den Schalter **Fertig stellen**. Starten Sie anschließend den Assistenten für neue *visuelle* Klassen über den Menübefehl

**Datei > Neu > Visual Class**

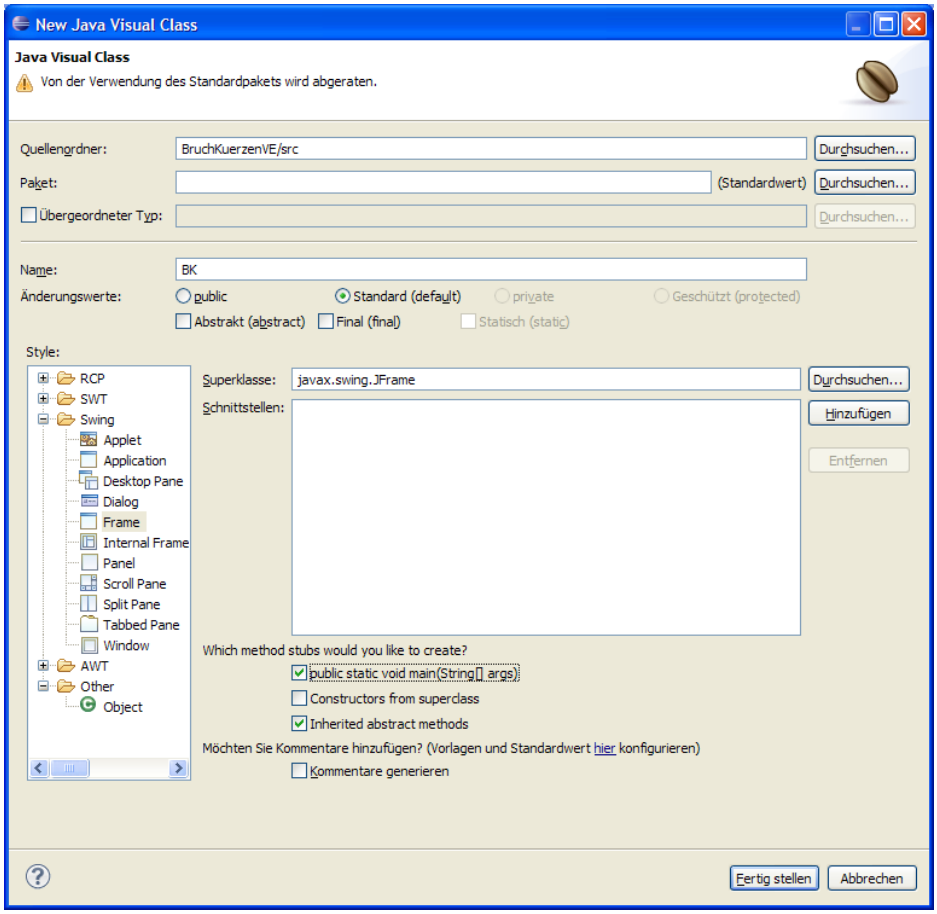
oder das folgende Symbolleistensteuerelement:




Neben dem Namen (im Beispiel: `BK`) und der Schutzstufe (**Standard(default)** ist angemessen, da eine Wiederverwendung in anderen Projekten nicht in Frage kommt) wird für die neue Klasse noch festgelegt:

- **Style** bzw. **Superklasse**
  Hier wählt man zwischen den in Java verfügbaren GUI-Lösungen AWT, Swing und SWT (siehe unten). Wir entscheiden uns für die am weitesten verbreitete Variante Swing. Außerdem wählen wir die Basisklasse **JFrame** im Paket **javax.swing**, so dass unsere neue Klasse die Eigenschaften und Handlungskompetenzen eines selbständigen Anwendungsfensters erbt.
- Rumpf einer **main()**-Methode zum Starten der Anwendung

Insgesamt resultiert das folgende Assistentenfenster:

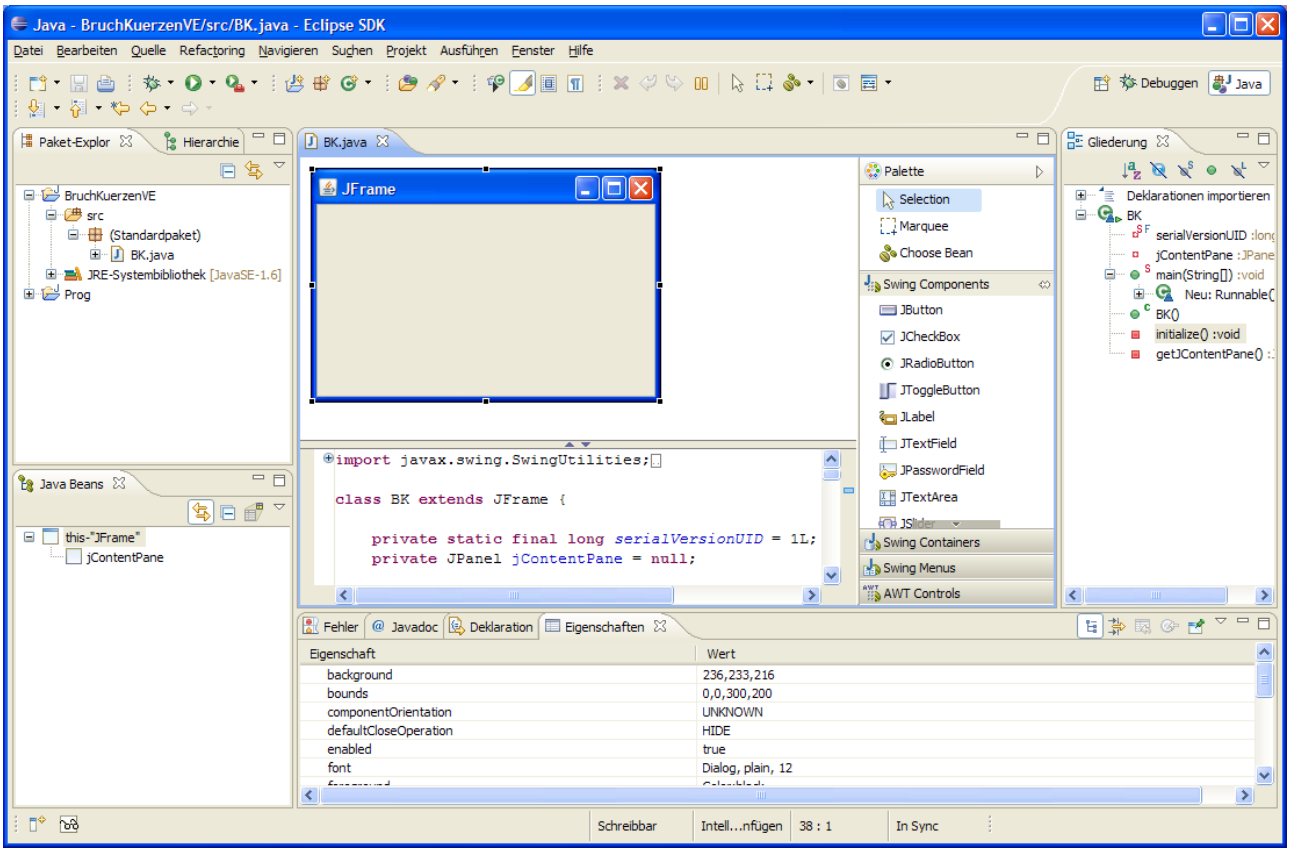



Die Kaffeebohne in der oberen rechten Ecke des Dialogs zeigt an, dass die Objekte der entstehenden Klasse als **Komponenten** (**Java-Beans**) gelten. Komponenten sind besonders gut für die Wiederverwendung in zahlreichen Programmen vorbereitet:

- Sie unterstützen die Verwendung mit Hilfe von visuellen Entwicklungsumgebungen. Durch die systematische Verfügbarkeit von Methoden zum Lesen und Setzen von Eigenschaften und die Einhaltung bestimmter Bennennungsregeln kann eine Entwicklungsumgebung eine Tabelle mit den Eigenschaftsausprägungen zur Entwurfszeit anbieten. Außerdem
- Sie kommunizieren über Ereignisse (siehe unten) als Sender und Empfänger.
- Komponenten für graphische Bedienoberflächen von Programmen (z.B. Befehlsschalter, Textfelder) treten auf dem Bildschirm in Erscheinung (machen Graphikausgaben) und kommunizieren mit dem Benutzer (reagieren auf GUI-Ereignisse wie Mausklicks).

Unsere neu anzulegende Klasse `BK` erbt diese Fähigkeit von ihrer Basisklasse **JFrame**.

Nach dem **Fertigstellen** zeigt der Eclipse-Arbeitsplatz einige Funktionserweiterungen:



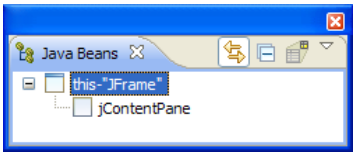

Der Editor für visuelle Klassen erlaubt die Gestaltung der Benutzeroberfläche mit graphischen Werkzeugen und pflegt synchron den äquivalenten Quellcode.

Links unten ist die Sicht **Java Beans** erschienen, die zur Verwaltung der in unsere **JFrame**-Ableitung aufgenommenen Komponenten (z.B. für Textfelder und Schalter) dient.

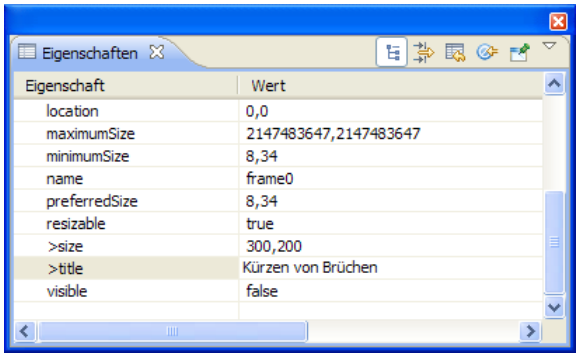
Im unteren Eclipse-Anzeigebereich ist die Registerkarte **Eigenschaften** hinzugekommen, wo die Eigenschaften der **JFrame**-Ableitung `BK` und der darin enthaltenen Komponenten eingesehen und verändert werden können.

### 4.8.2 Eigenschaften von Komponenten ändern

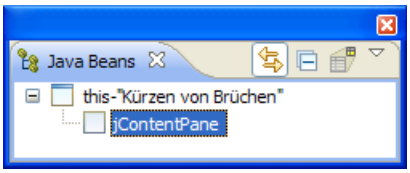
Markieren Sie nötigenfalls in der **Beans**-Sicht den obersten, das Anwendungsfenster repräsentierenden, Eintrag,



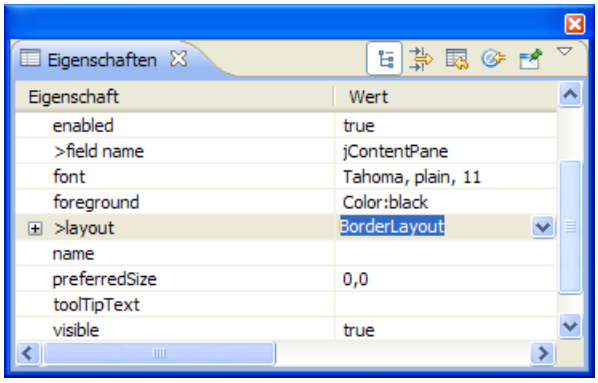

und legen Sie die Titelzeilenbeschriftung per **title**-Eigenschaft fest:

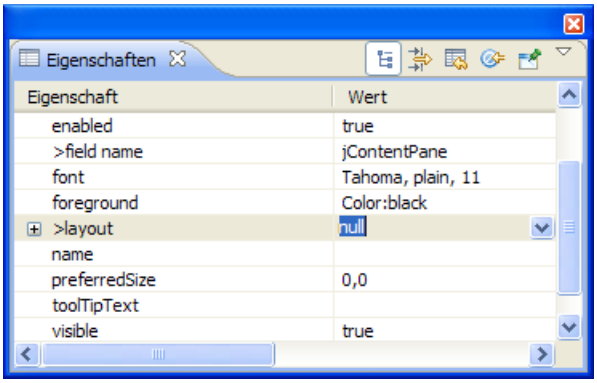


Auf dem Anwendungsfenster platzierte Bedienelemente (siehe unten) landen in einem als *Content Pane* bezeichneten Container. Für die Anordnungslogik der Bedienelemente eines Containers (speziell bei Änderungen der Fenstergröße) wird in der Regel ein so genannter Layout-Manager engagiert. Momentan verzichten wir aus didaktischen Gründen bei der Content Pane des Anwendungsfensters



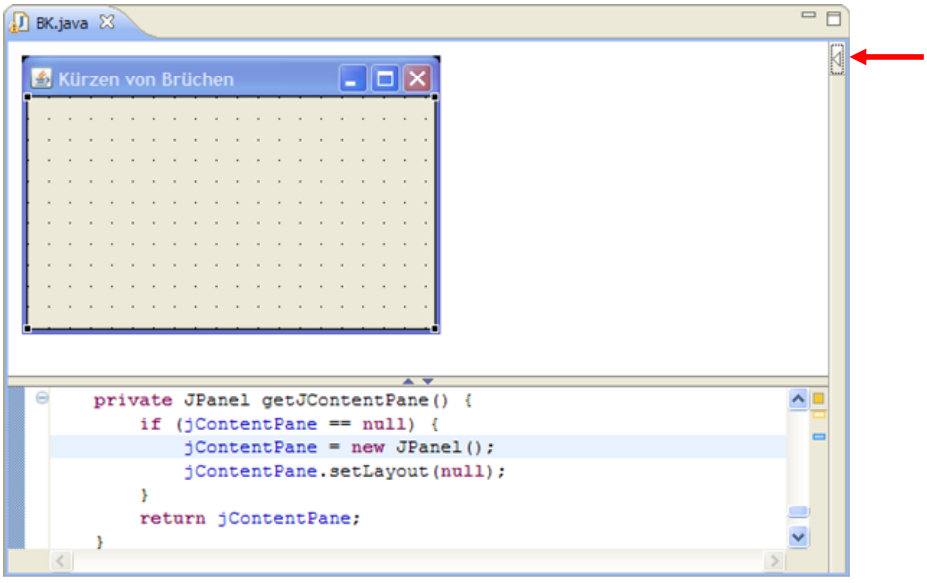

auf den voreingestellten **BorderLayout**-Manager:



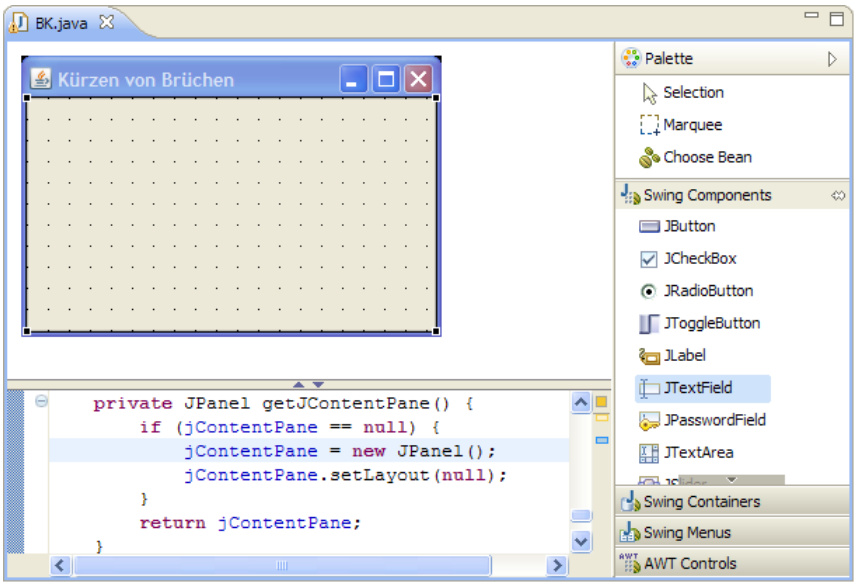




### 4.8.3 Bedienelemente aus der VE-Palette übernehmen und gestalten

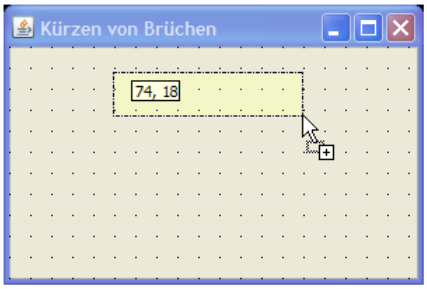
Öffnen Sie nötigenfalls die Palette mit den verfügbaren Bedienelementen per Mausklick auf die Dreieck-Schaltfläche am rechten Rand des VE-Editors:



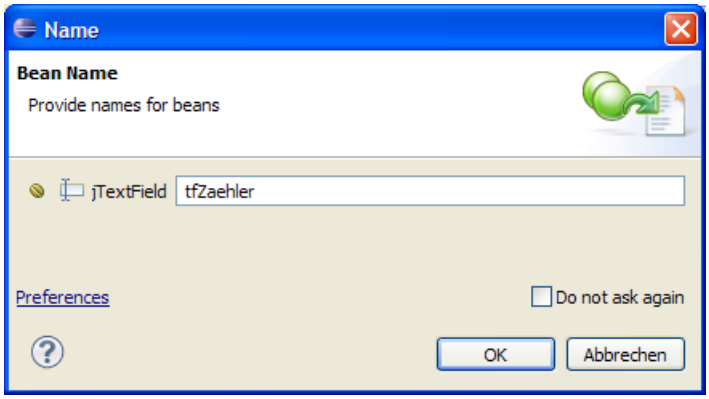

Erweitern Sie im **Paletten-**Fenster die Liste mit den **Swing Components** per Mausklick auf die Beschriftung,




markieren Sie per Mausklick die Komponente **JTextField**, auf setzen Sie per Maus ein Textfeld für den Zähler eines Bruchs auf das Anwendungsfenster:



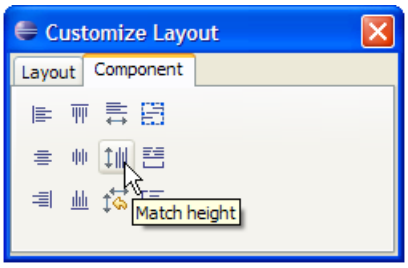

Nachdem Sie die rechte untere Ecke des Textfelds festgelegt haben, erkundigt sich Eclipse nach dem Namen für die `BK`-Referenzinstanzvariable zur **JTextField**-Komponente:




Setzen Sie analog eine **JTextField**-Komponente für den Nenner, eine **JLabel**-Komponente für den Bruchstrich und eine **JButton**-Komponente zur Anforderung der Programmoperation auf das Anwendungsfenster:

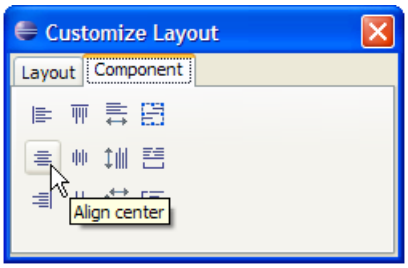
Der VE kann das Design auf vielfältige Weise unterstützen, z.B. beim Textfeld für den Nenner:

- Duplizieren Sie das Textfeld des Zählers via Zwischenablage (z.B. **Strg-C**, **Strg-V**).
- Das neu erstellte Textfeld ist markiert und hat noch keine geeigneten Abmessungen.
- Markieren Sie zusätzlich das Zähler-Textfeld, um dessen Abmessungen übernehmen zu können.
- Wählen Sie aus dem Kontextmenü zur Markierung den Eintrag **Customize Layout**.
- Veranlassen Sie im folgenden Dialog auf der Registerkarte **Component** eine Anpassung von **Breite** und **Höhe**:

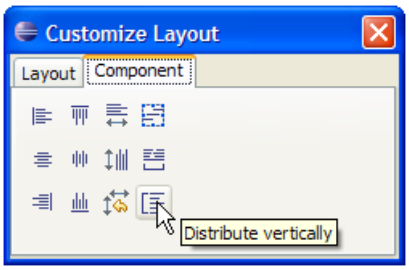



Über den Dialog **Customize Layout** kann man außerdem u.a. erreichen:

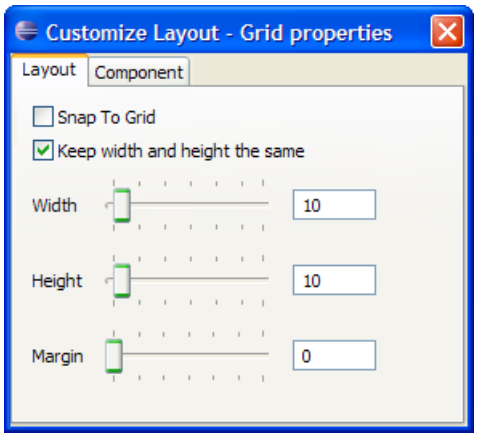
- Das horizontale Zentrieren der markierten Steuerelemente:




- Die gleichmäßige vertikale Verteilung der markierten Steuerelemente:

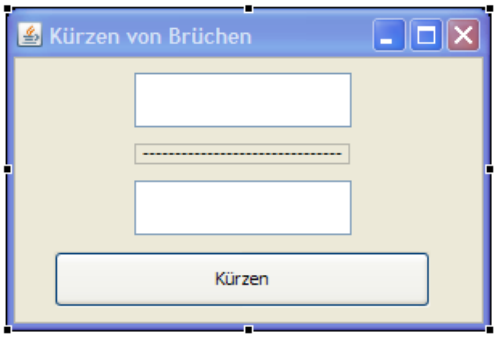



- Abstand und magnetische Aktivität der Gitterlinien:

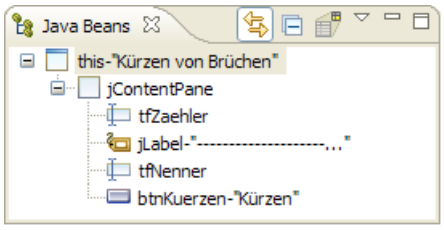



Für die im gestalteten Anwendungsfenster markierte **JLabel**- bzw. **JButton**-Komponente legen wir via **Text**-Eigenschaft noch eine passende Beschriftung fest, die beim Label aus einer geeigneten Anzahl von Bindestrichen bestehen kann. Für die beiden Textfelder und das Label sollte noch das horizontale Zentrieren des Inhalts über den Wert **CENTER** für die Eigenschaft **horizontalAlign-**

**ment** angeordnet werden. Vorerst können wir mit den Bedienelementen und dem Erscheinungsbild des Programms zufrieden sein:




Die Sicht **Java Beans** zeigt die einbezogenen Komponenten mit **Text**-Eigenschaft (falls vorhanden):




### 4.8.4 Der Quellcode-Generator und -Parser

Unsere Gestaltungs- und Konfigurationsarbeiten werden vom Visual Editor in eine recht übersichtliche Klassendefinition umgesetzt:

```
class BK extends JFrame {

    private static final long serialVersionUID = 1L;
    private JPanel jContentPane = null;
    private JTextField tfZaehler = null;
    private JLabel jLabel = null;
    private JTextField tfNenner = null;
    private JButton btnKuerzen = null;
    /**
     * This method initializes tfZaehler
     *
     * @return javax.swing.JTextField
     */
    private JTextField getTfZaehler() {
        if (tfZaehler == null) {
            tfZaehler = new JTextField();
            tfZaehler.setBounds(new Rectangle(75, 10, 135, 34));
        }
        return tfZaehler;
    }
```

Meist werden Sie sich als *Designer* betätigen und das oft mühsame Kodieren dem Visual Editor überlassen. Sie dürfen aber auch direkt den Quelltext ändern, wobei der Visual Editor als Syntaxanalysator (engl. *parser*) aktiv wird und Ihren Code in das Design übernimmt. Dies sollte umso besser gelingen, je stärker Sie sich am Programmierstil des Automaten orientieren.

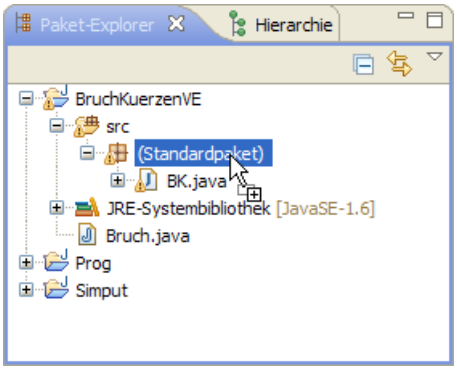
Sollte das ständige Synchronisieren von Design und Code durch den Visual Editor einmal stören (z.B. durch hohen Zeitaufwand), kann es vorübergehend mit dem Pausenschalter der Symbolleiste unterbrochen und später mit dem **Reload**-Schalter wieder aufgenommen werden.

### 4.8.5 Bruch-Klasse einbinden

Zum Kürzen des vom Benutzer der Klasse `BK` über die beiden Textfelder eingegebenen Bruchs soll ein Objekt unserer Klasse `Bruch` verwendet werden. Wir könnten den Bytecode der Klasse einbinden und dabei genauso vorgehen wie bei der Klasse `Simput` (vgl. Abschnitt 3.4.2). Eine weitere Möglichkeit besteht darin, den Quellcode der Klasse `Bruch` in das aktuelle Projekt aufzunehmen. Aus didaktischen Gründen wählen wir die letztgenannte, bislang noch nicht vorgestellte Variante. Ziehen Sie aus einem Fenster des Windows-Explorers die Datei

**…\BspUeb\Klassen und Objekte\Bruch\B7 (mit Aggregation)\Bruch.java**

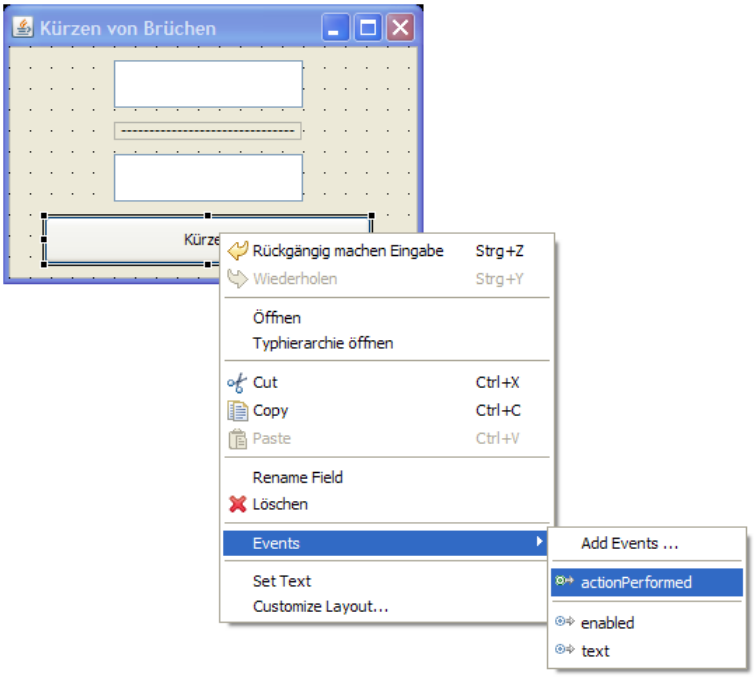
auf das Standardpaket des Projekts im Eclipse-Paket-Explorer:




Es erscheint ein Fehlerindikator, weil in **Bruch.java** die Klasse `Simput` genutzt wird, die über den Klassenpfad des aktuellen Projekts nicht auffindbar ist. Nehmen Sie also die gemäß Abschnitt 3.4.2 angelegte `Simput`-Klassenpfadvariable in das Projekt auf.

### 4.8.6 Ereignisbehandlungsmethoden anlegen

Nun erstellen wir zum Befehlsschalter eine Methode, die durch das Betätigen des Schalters (z.B. per Mausklick) ausgelöst werden soll. Wählen Sie im Visual Editor aus dem Kontextmenü zum Befehlsschalter




den Eintrag **Events > actionPerformed**, so dass der Visual Editor in der Datei **BK.java** einen Rohling zur Behandlungsmethode **actionPerformed()** zum **ActionEvent** des Befehlsschalters anlegt:

```
private JButton getBtnKuerzen() {
   if (btnKuerzen == null) {
      btnKuerzen = new JButton();
      btnKuerzen.setBounds(new Rectangle(25, 121, 234, 35));
      btnKuerzen.setText("Kürzen");
      btnKuerzen.addActionListener(new java.awt.event.ActionListener() {
         public void actionPerformed(java.awt.event.ActionEvent e) {
            System.out.println("actionPerformed()");
         }
      });
   }
   return btnKuerzen;
}
```

Momentan dürfen (und müssen) Sie sich darüber wundern, dass die Definition der Methode **actionPerformed()** als Bestandteil des Aktualparameters in einem Aufruf der Methode **addActionListener()** auftritt. Später wird sich zeigen, dass hier eine anonyme Klasse im Spiel ist.

Mit Hilfe eines Objekts aus unserer Klasse `Bruch` ist die benötigte Funktionalität leicht zu implementieren, z.B.:

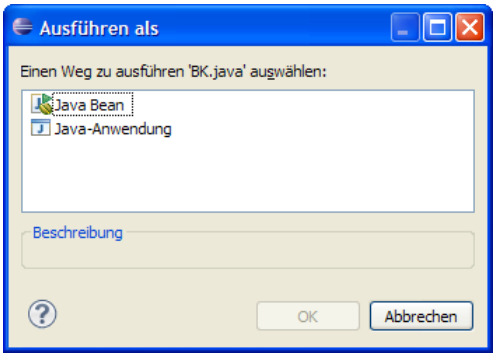
```
public void actionPerformed(java.awt.event.ActionEvent e) {
   int z = Integer.parseInt(tfZaehler.getText());
   int n = Integer.parseInt(tfNenner.getText());
   Bruch b = new Bruch(z, n, "");
   b.kuerze();
   tfZaehler.setText(Integer.toString(b.gibZaehler()));
   tfNenner.setText(Integer.toString(b.gibNenner()));
}
```

Beim Konvertieren zwischen den Datentypen **String** und **int** kommen die statischen Methoden **parseInt()** und **toString()** der Klasse **Integer** zum Einsatz (vgl. Abschnitt 5.3.2).

Der Bequemlichkeit halber wird eine *lokale* `Bruch`-Referenzvariable verwendet, so dass bei jedem Methodenaufruf ein neues Objekt entsteht. Außerdem nehmen wir es vorläufig in Kauf, dass unser Programm bei irregulären Eingaben „abstürzt“. Das Programm ist nun fertig und startklar.

### 4.8.7 Ausführen

Innerhalb der Entwicklungsumgebung Eclipse lässt sich **BK.class** als Java-Bean und (dank **main()**-Methode) als Java-Anwendung ausführen:



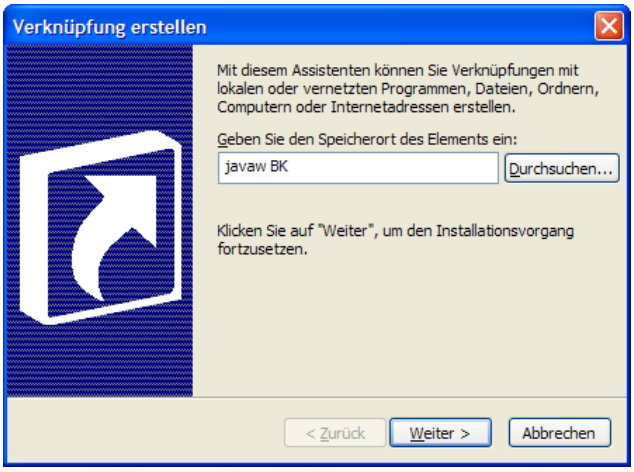

Um das Programm unter Windows unabhängig von Eclipse zu starten, kann man z.B. so vorgehen:

- Konsolenfenster öffnen
- Zu einem Ordner wechseln der alle benötigten **class**-Dateien enthält (Installationsordner) Dazu gehören neben **BK.class** und **Bruch.class** auch die Dateien **BK$1.class** und **BK$2.class** zu anonymen Klassen, die der Compiler im Zusammenhang mit der Ereignisbehandlung erstellt hat.
- Weil das Programm (abgesehen von eventuellen Ausnahmefehlermeldungen) keine Konsolenausgaben macht, starten wir es über **javaw.exe**:


Über eine Verknüpfungsdatei lässt sich ein Start per Doppelklick realisieren:

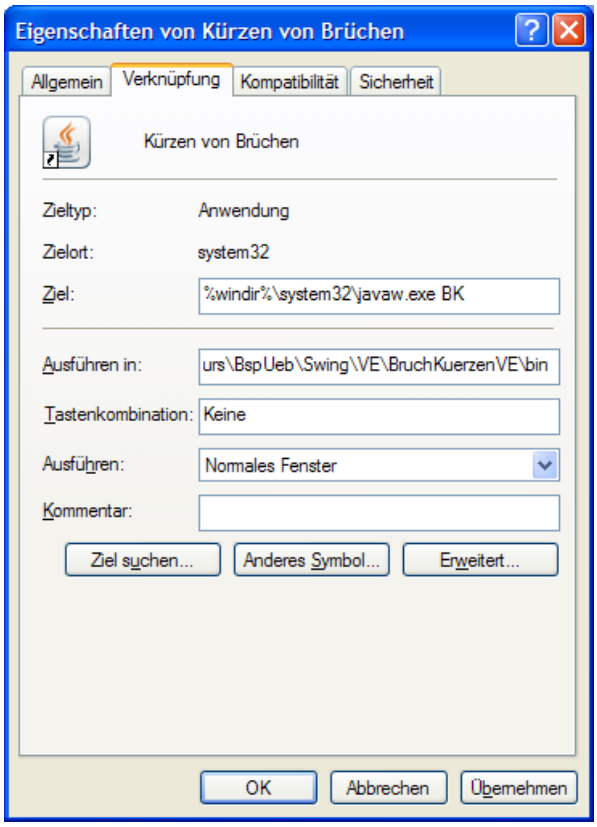
- Verknüpfung anlegen
- Wählen Sie aus über das Kontextmenü zum Zielordner (z.B. Desktop)

  **Neu > Verknüpfung**.
- Tragen Sie den Startbefehl ein, z.B.:

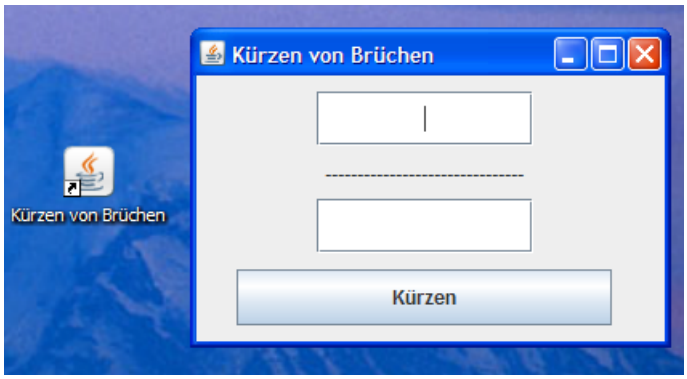



- Wählen Sie einen Namen für die Verknüpfung, z.B.:


- Nach dem **Fertigstellen** müssen Sie über den Eigenschaftsdialog zur neuen Verknüpfung dafür sorgen, dass die Anwendung im Installationsordner ausgeführt wird, z.B.:




Nun gelingt der anwenderfreundliche Start per Doppelklick auf ein Desktop-Symbol:




Das komplette Eclipse-Projekt `BruchKuerzenVE` ist im folgenden Ordner zu finden

**...\BspUeb\Swing\VE\BruchKuerzenVE**

## *4.9 Übungsaufgaben zu Kapitel 4*

1) Welche von den folgenden Aussagen 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 sind richtig bzw. falsch?

1. Methoden müssen generell als **public** deklariert werden, denn sie gehören zur Schnittstelle einer Klasse.
2. Im Rumpf einer Methode sind beliebige Anweisungen erlaubt, aber keine Definitionen. Insbesondere dürfen hier also keine Methoden definiert werden.
3. Ändert man den Rückgabetyp einer Methode, dann ändert sich auch ihre Signatur.
4. Beim Methodenaufruf müssen die Datentypen der Aktualparameter exakt mit den Datentypen der Formalparameter übereinstimmen.
5. Lokale Variablen einer Methode überdecken gleichnamige Felder.

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) Welche programminternen Möglichkeiten hat eine Methode, Informationen an die Außenwelt zu übergeben bzw. Änderungen in der Außenwelt vorzunehmen? Mögliche Wirkungen der Methode auf das Dateisystem oder Netzwerkverbindungen sind bei dieser Frage nicht gemeint.

6) 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();
}
```

7) 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 JRE 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 ein Präfix aus Klassennamen und Punktoperator voranstellen. In Methoden derselben Klasse darf der Klassenname entfallen.

8) 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
- Eine Methode mit dem Namen `toString()` und dem Rückgabetyp **String**, die zu einem `Time`- bzw. `Duration`-Objekt eine gut lesbare Zeichenfolgenrepräsentation liefert
  Tipp: Seit Java 5.0 steht in der Klasse **String** die statische Methode **format()** zur Verfügung, die analog zur **PrintStream**-Methode **printf()** (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:

```
Von 17:34:55 Uhr bis 12:24:12 Uhr vergehen    18:49:17 h:m:s.

   20:23:00 h:m:s nach 17:34:55 Uhr sind es 13:57:55 Uhr.
```

Anmerkungen:

- 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.
- Ist die Datenkapselung perfekt realisiert, kann man bei einer späteren Revision der Klassendefinition z.B. die Instanzvariablen für Stunden, Minuten und Sekunden durch eine einzige Instanzvariable für die Anzahl der Sekunden seit Tagesanbruch bzw. seit dem Beginn des Intervalls ersetzen.

9) Lokalisieren Sie bitte in dieser 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 zpar, int npar, String epar) {
            zaehler = zpar;
            nenner  = npar;
            etikett = epar;
            anzahl++;
      }
      public Bruch() {anzahl++;}

      public void setzeZaehler(int zpar) {zaehler = zpar;}
      public boolean setzeNenner(int n) {
            if (n != 0)  {
                  nenner = n;
                  return true;
            } else
                  return false;
      }
      public void setzeEtikett(String epar) {
            int rind = epar.length();
            if (rind > 40)
                  rind = 40;
            etikett = epar.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

**...\BspUeb\Klassen und Objekte\Begriffe lokalisieren.pdf**

benutzen. Die Idee zu dieser Übungsaufgabe stammt aus Mössenböck (2003).

10) 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. Diese Aufgabe dient dazu, an einem einfachen Beispiel mit rekursiven Methodenaufrufen zu experimentieren. Für die Praxis ist die rekursive Fakultätsberechnung nicht geeignet.

11) 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 folgenden Leistungen erfügt:

- **Länge** ermitteln

  Der Betrag eines Vektors $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ ist definiert durch:

  $$|x| := \sqrt{x_1^2 + x_2^2}$$

  Verwenden Sie die Klassenmethode **Math.sqrt()**, um die Quadratwurzel aus einer **double**-Zahl zu berechnen.

- Vektor auf Länge Eins **normieren**
  Dazu dividiert man beide Komponenten durch die Länge des Vektors, denn mit
  $\tilde{x} := (\tilde{x}_1, \tilde{x}_2)$ sowie $\tilde{x}_1 := \frac{x_1}{\sqrt{x_1^2 + x_2^2}}$ und $\tilde{x}_2 := \frac{x_2}{\sqrt{x_1^2 + x_2^2}}$ gilt:

  $$|\tilde{x}| := \sqrt{\tilde{x}_1^2 + \tilde{x}_2^2} = \sqrt{\left(\frac{x_1}{\sqrt{x_1^2 + x_2^2}}\right)^2 + \left(\frac{x_2}{\sqrt{x_1^2 + x_2^2}}\right)^2} = \sqrt{\frac{x_1^2}{x_1^2 + x_2^2} + \frac{x_2^2}{x_1^2 + x_2^2}} = 1$$

- Vektoren (komponentenweise) **addieren**

  Die Summe der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

  $$\begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

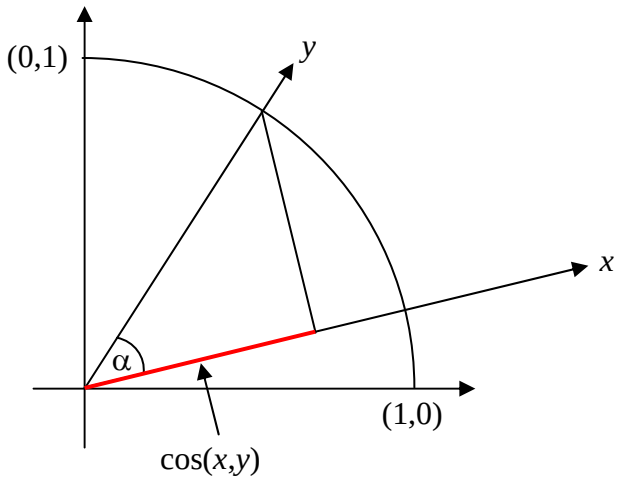
- **Skalarprodukt** zweier Vektoren ermitteln

  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:[27]

  $$\cos(x, y) = \frac{x \cdot y}{|x||y|}$$

  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:

  $$rad = \frac{deg}{360} \cdot 2\pi$$

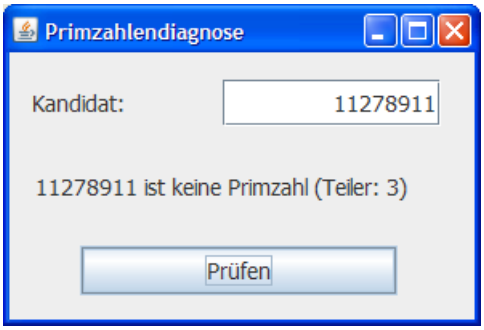
---

[27] Dies folgt aus dem Additionstheorem für den Kosinus.

Erstellen Sie ein Demonstrationsprogramm, das Ihre Vektor-Klasse verwendet und ungefähr den folgenden Programmablauf ermöglicht (Eingabe fett):

```
Vektor 1:                (1,00; 0,00)
Vektor 2:                (1,00; 1,00)

Laenge von Vektor 1:     1,00
Laenge von Vektor 2:     1,41

Winkel:                  45,00 Grad

Um wie viel Grad soll Vektor 2 gedreht werden: 45

Neuer Vektor 2           (0,00; 1,41)
Neuer Vektor 2 normiert  (0,00; 1,00)

Summe der Vektoren       (1,00; 1,00)
```

12) Entwickeln Sie ein Programm zur Primzahlendiagnose mit graphischer Benutzeroberfläche, z.B.:




Gehen Sie dabei analog zu Abschnitt 4.8 vor (individuelles Fensterdesign mit Hilfe des visuellen Editors), obwohl ein oberflächlich ähnliches Programm auch mit der in Abschnitt 3.8 vorgestellten Klasse **JOptionPane** zu realisieren wäre. Erstellen Sie auch eine Verknüpfung zum bequemen Starten Ihrer Anwendung per Doppelklick.

# 5 Elementare Klassen

In diesem Abschnitt wird gewissermaßen die objektorientierte Fortsetzung der elementaren Sprachelemente aus Abschnitt 3 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**)
- Verpackungsklassen zur Integration primitiver Datentypen in das Klassensystem
- Aufzählungstypen (Enumerationen)

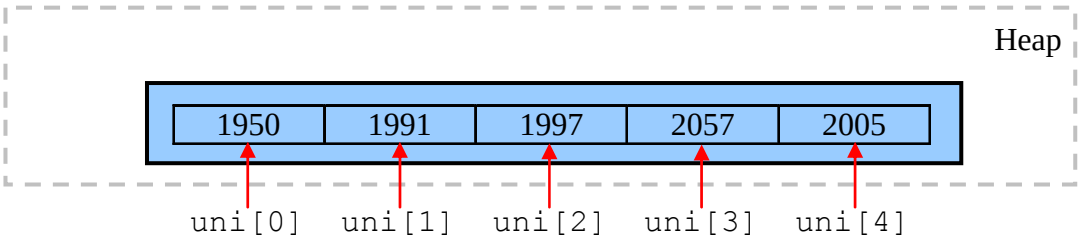
## *5.1 Arrays*

Ein Array ist ein Objekt, das eine feste Anzahl von Elementen desselben Datentyps als Instanzvariablen enthält. Man kann den kompletten Array ansprechen (z.B. als Aktualparameter an eine Methode übergeben) oder auf einzelne Elemente über einen Index zugreifen.

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.

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.

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.

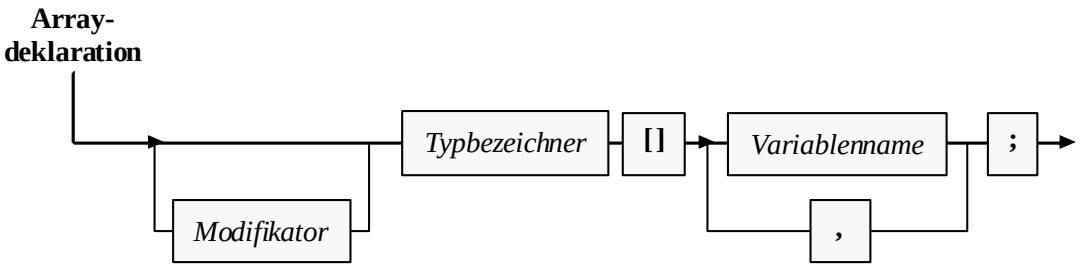
Im Vergleich zur Verwendung einer entsprechenden Anzahl von Einzelvariablen ergibt sich eine erhebliche Vereinfachung der Programmierung:

- 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 (z.B. bei der Ausgabe in eine Datei) per Methodenaufruf mit Array-Aktualparameter veranlassen.
- Viele Algorithmen arbeiten mit Vektoren und Matrizen. Zur Modellierung dieser mathematischen Objekte sind Arrays unverzichtbar.

Wir befassen uns zunächst mit *eindimensionalen* Arrays, behandeln später aber auch den mehrdimensionalen Fall.

### 5.1.1 Array-Referenzvariablen deklarieren

Eine Array-Variable ist vom Referenztyp und wird folgendermaßen deklariert:




Im Vergleich zu der bisher bekannten Variablendeklaration (ohne Initialisierung) ist hinter dem Typbezeichner zusätzlich ein Paar eckiger Klammern anzugeben.[28] In obigem Beispiel kann die Array-Variable `uni` also z.B. folgendermaßen deklariert werden:

```
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 (`max+1`) **int**-Elementen, und seine Adresse landet in der Referenzvariablen `uni`:

```
uni = new int[max+1];
```

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. Existiert ein Array-Objekt erst einmal, kann die Anzahl seiner Elemente allerdings nicht mehr geändert werden. Wer noch dynamischere Datenstrukturen benötigt, kann z.B. die API-Klasse **Vector** benutzen (siehe Abschnitt 5.3.1).

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];
```

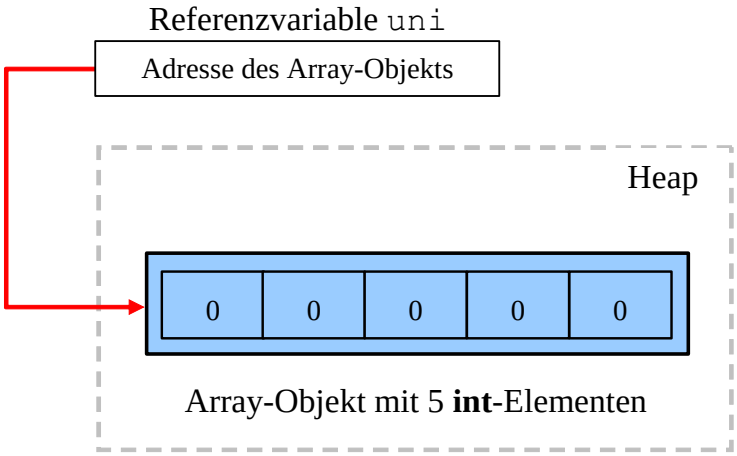
Mit der Verweisvariablen `uni` und dem referenzierten Array-Objekt auf dem Heap haben wir insgesamt folgende Situation:

---

[28] 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.




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.

Aus der Objekt-Natur eines Arrays folgt unmittelbar, dass er vom Garbage Collector entsorgt wird, wenn keine Referenz mehr vorliegt (vgl. Abschnitt 4.4.4). 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 {`<br>`  public static void main(String[] args) {`<br>`  int[] x = new int[3], y;`<br>`  x[0] = 1; x[1] = 2; x[2] = 3;`<br>`  y = x; //y zeigt nun auf das selbe Array-Objekt wie x`<br>`  y[0] = 99;`<br>`  System.out.println(x[0]);`<br>`  }`<br>`}` | `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 = 1; i <= drl; i++)
  uni[zzg.nextInt(5)]++;
```

Den Indexwert liefert die Zufallszahlenmethode **nextInt()** mit Rückgabetyp **int** (siehe unten).

Wie in vielen anderen Programmiersprachen hat auch in Java das erste von *n* Feldelementen 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
        at UniRand.main(UniRand.java:14)
```

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 Instanzvariable **length** feststellen, z.B.:

| Quellcode | Eingabe (fett) und Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  System.out.print("Laenge des Vektors: ");`<br>`  int[] wecktor = new int[Simput.gint()];`<br>`  System.out.println();`<br><br>`  for(int i = 0; i < wecktor.length; i++) {`<br>`  System.out.print("Wert von Element "+i+": ");`<br>`  wecktor[i] = Simput.gint();`<br>`  }`<br>`  System.out.println();`<br><br>`  for(int i = 0; i < wecktor.length; i++)`<br>`  System.out.println(wecktor[i]);`<br>`  }`<br>`}` | `Laenge des Vektors:` **`3`**<br><br>`Wert von Element 0:` **`7`**<br>`Wert von Element 1:` **`13`**<br>`Wert von Element 2:` **`4711`**<br><br>`7`<br>`13`<br>`4711` |

Auch beim Entwurf von Methoden mit Array-Parametern ist es von Vorteil, dass die Länge eines übergebenen Arrays ohne entsprechenden Zusatzparameter in der Methode bekannt ist.

### 5.1.4 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 zur Demonstration verwendeten Anweisungen lassen sich leicht zu einem Programm erweitern, das die Qualität des Java-**Pseudozufallszahlengenerators** 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. Meist 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.

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).

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 Starwert für den Pseudozufall.[29]

Das angekündigte Programm zur Prüfung des Java-Pseudozufallszahlengenerators zieht 10.000 Zufallszahlen und überprüft deren empirische Verteilung:

```
class UniRand {
  public static void main(String[] args) {
    final int DRL = 10000;
    int i;

    int[] uni = new int[5];
    java.util.Random zzg = new java.util.Random();
    for (i = 1; i <= DRL; i++)
      uni[zzg.nextInt(5)]++;
    System.out.println("Absolute Haeufigkeiten:");
    for (i = 0; i < 5; i++)
      System.out.print(uni[i] + " ");

    System.out.println("\n\nRelative Haeufigkeiten:");
    for (i = 0; i < 5; i++)
      System.out.print((double)uni[i]/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 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 $\chi^2$-Signifikanztest mit der Gleichverteilung als Nullhypothese bestätigt durch eine Überschreitungswahrscheinlichkeit von 0,569 (weit oberhalb der 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** |

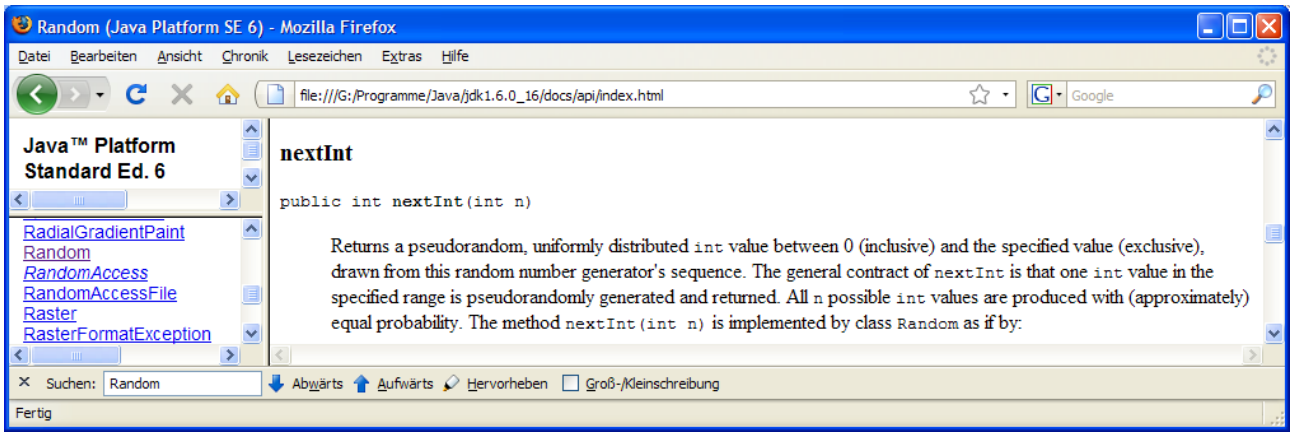
a. Bei 0 Zellen (,0%) werden weniger als 5 Häufigkeiten erwartet. Die kleinste erwartete Zellenhäufigkeit ist 2000,0.

Über die im Beispielprogramm verwendete Klasse **Random** aus dem Paket **java.util** können Sie sich z.B. mit Hilfe der JDK-Dokumentation informieren:

---

[29] 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());
```




Statt explizit 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.:[30]

```
uni[(int) (Math.random()*5)]++;
```

### 5.1.5 Initialisierungslisten

Bei Arrays mit wenigen Elementen ist die Möglichkeit von Interesse, beim Deklarieren der Referenzvariablen eine Initialisierungsliste anzugeben und das Array-Objekt dabei implizit (ohne Verwendung des **new**-Operators) zu erzeugen, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  int[] wecktor = {1, 2, 3};`<br>`  System.out.println(wecktor[2]);`<br>`  }`<br>`}` | 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, z.B.:

```
int[] wecktor;
wecktor = new int[] {1, 2, 3};
```

---

30 Im Hintergrund erzeugt die Methode bei ihrem ersten Aufruf ein **Random**-Objekt über den parameterfreien Konstruktor:

```
new java.util.Random()
```

### 5.1.6 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.

### 5.1.7 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 beliebig verallgemeinern, um Arrays mit höherer Dimensionalität zu erzeugen.

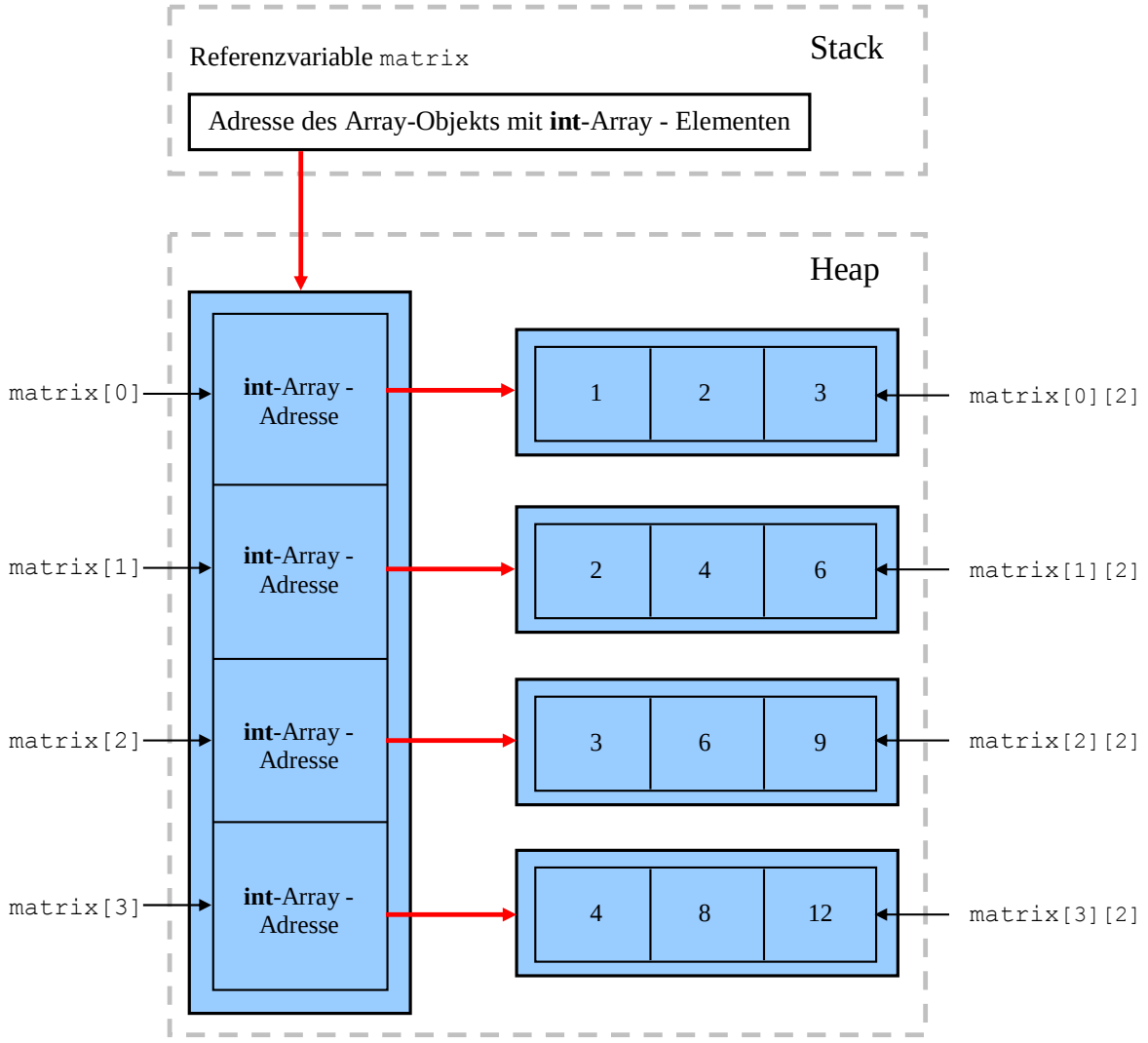
Im Beispiel wird ein Array-Objekt namens `matrix` mit den vier Elementen `matrix[0]` bis `matrix[3]` erzeugt, bei denen es sich jeweils um eine Referenz auf einen Array mit drei **int**-Elementen handelt. Wir haben damit eine zweidimensionale Matrix zur Verfügung, auf deren Zellen man per Doppelindizierung zugreifen kann, wobei sich die Syntax leicht von der mathematischen Schreibweise 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
```

Dabei wird zunächst das äußere Array-Objekt ohne Längenangabe zur zweiten Dimension angelegt:

```
int[][] matrix = new int[5][];
```

Die inneren Array-Objekte (also 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 Zeichenketten

Java bietet für den Umgang mit Zeichenketten zwei Klassen an:

- **String**
  Objekte der Klasse **String** können nach dem Erzeugen nicht mehr geändert werden. Diese Klasse ist für den *lesenden* Zugriff auf Zeichenketten optimiert.
- **StringBuffer**
  Für *variable* Zeichenketten sollte unbedingt die Klasse **StringBuffer** verwendet werden, weil deren Objekte nach dem Erzeugen noch verändert werden können.

### 5.2.1 Die Klasse String für konstante Zeichenketten

#### 5.2.1.1 Implizites und explizites 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

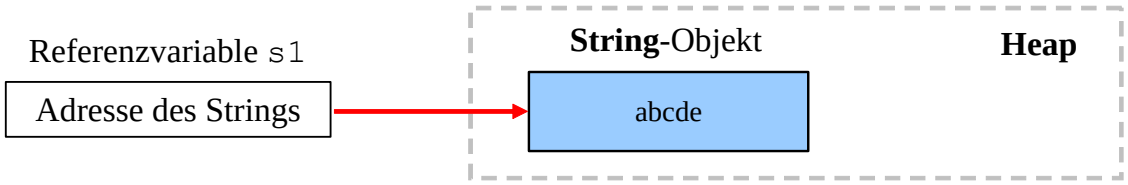
Soviel objektorientierten Hintergrund sieht man der angenehm einfachen Anweisung auf den ersten Blick nicht an. In Java sind jedoch auch Zeichenketten*literale* als **String**-Objekte realisiert, so dass z.B.

```
"abcde"
```

einen Ausdruck darstellt, der als Wert einen Verweis auf ein **String**-Objekt auf dem Heap liefert.

Weil in obiger Deklarations- und Initialisierungsanweisung kein **new**-Operator auftaucht, spricht man auch vom *impliziten* Erzeugen eines **String**-Objekts.

Die Anweisung erzeugt im Hauptspeicher folgende Situation:




Natürlich bietet die Klasse **String** auch Konstruktoren für die explizite Objektkreation per **new** -Operator, z.B.:

```
String s1 = new String("abcde");
```

#### *5.2.1.2 Inhalte und Referenzen*

Erfolgt die Initialisierung einer **String**-Referenzvariablen über einen *konstanten* Ausdruck, kommt der so genannte **interne String-Pool** ins Spiel, den übrigens die Klasse **String** privat verwaltet. Existiert dort bereits ein **String**-Objekt mit demselben Inhalt, wird dessen Adresse verwendet. Anderenfalls wird ein neues Objekt im **String**-Pool angelegt. So wird verhindert, dass für wiederholt im Quellcode auftretende Zeichenkettenliterale jeweils Speicherplatz verschwendend ein neues Objekt entsteht. Diese Vorgehensweise ist sinnvoll, weil sich vorhandene **String**-Objekte garantiert nicht mehr ändern.

Kommt bei der Initialisierung ein *variabler* Ausdruck zum Einsatz, wird auf jeden Fall ein neues Objekt erzeugt, z.B.:

```
String de = "de";
String s3 = "abc"+de;
```

Dies geschieht auch bei expliziter Verwendung des **new**-Operators, z.B.:

```
String s4 = new String("abcde");
```

Beim Vergleich von **String**-Variablen per Identitätsoperator haben obige Ausführungen wichtige Konsequenzen, wie das folgende Programm zeigt:

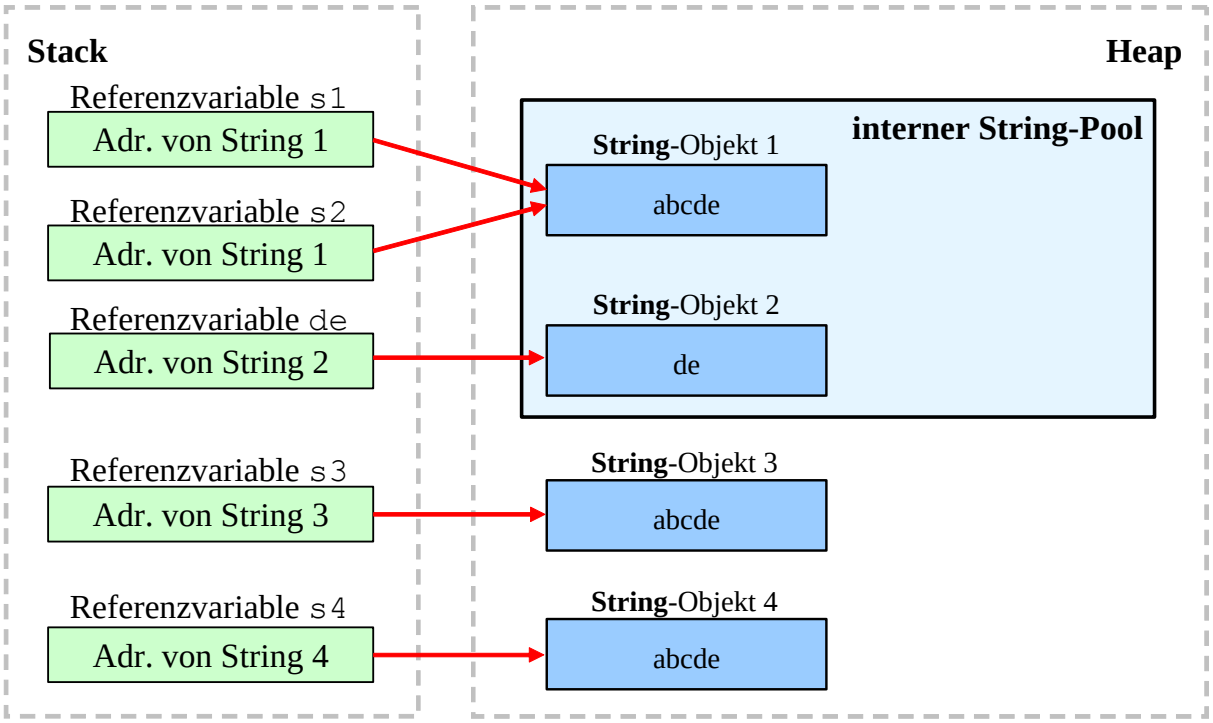
| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  String s1 = "abcde";`<br>`  String s2 = "abc"+"de";`<br>`  String de = "de";`<br>`  String s3 = "abc"+de;`<br>`  String s4 = new String("abcde");`<br>`  System.out.print("(s1 == s2) = "+(s1==s2)+"\n"+`<br>`  "(s1 == s3) = "+(s1==s3)+"\n"+`<br>`  "(s1 == s4) = "+(s1==s4));`<br>`  }`<br>`}` | `(s1 == s2) = true`<br>`(s1 == s3) = false`<br>`(s1 == s4) = false` |

Das merkwürdige[31] 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: Die Variablen stehen genau dann für dieselbe Zeichenfolge, wenn sie denselben Referenzwert haben.

Im Beispielprogramm werden vier **String**-Objekte mit folgenden Referenzen erzeugt:

[31] „Merkwürdig“ bedeutet hier, dass sich eine Aufnahme in das Langzeitgedächtnis lohnt.




In Abschnitt 5.2.1.4.2 lernen Sie die **String**-Methode **equals()** kennen, die auf jeden Fall einen Inhaltsvergleich vornimmt.

### *5.2.1.3 String als WORM - Klasse*

Nachdem ein **String**-Objekt auf dem Heap erzeugt worden ist, kann es nicht mehr geändert werden. 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 Inflexibilität des **String**-Inhalts in Zweifel ziehen und ein Gegenbeispiel der folgenden Art vorbringen:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  String testr = "abc";`<br>`  System.out.println("testr = " + testr);`<br>`  testr = testr + "def";`<br>`  System.out.println("testr = " + testr);`<br>`  }`<br>`}` | `testr = abc`<br>`testr = abcdef` |

In der Zeile

```
testr = testr + "def";
```

wird aber das per `testr` ansprechbare **String**-Objekt (mit dem Text „abc“) nicht geändert, sondern durch ein neues **String**-Objekt (mit dem Text „abcdef“) ersetzt. Das alte Objekt ist noch vorhanden, aber nicht mehr referenziert. Sobald das Laufzeitsystem Langeweile hat oder Speicher benötigt, wird das alte Objekt vom Garbage Collector eliminiert.

### *5.2.1.4 Methoden für String-Objekte*

Von den ca. 50 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-SDK.

#### 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 Java die „+“ - Operatoren jeweils sinnvoll interpretiert (Verketten von Strings bzw. Addieren von Zahlen):

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  System.out.println("4 + 3 = " + (4 + 3));`<br>`  }`<br>`}` | `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 Vergleichen von Strings

Für den Test auf identischen **Inhalt** kann man die **String**-Methode **equals()**

**public boolean equals(String** *vergl***)**

verwenden, um den oben 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 {`<br>`  public static void main(String[] args) {`<br>`  String s1 = "abc";`<br>`  String s2 = new String("abc");`<br>`  System.out.println(s1==s2);`<br>`  System.out.println(s1.equals(s2));`<br>`  }`<br>`}` | `false`<br>`true` |

Allerdings sollte man den internen **String**-Pool nicht als Speicherspartechnik[32] mit Risikopotential bei unbedachten Vergleichen betrachten, sondern auch als eine Option zur Leistungssteigerung bei Programmen, die sehr viele **String**-Vergleiche vornehmen müssen. Vor einer solchen Pool-Verwendung kommt ein Befüllen über konstante Ausdrücke wohl kaum in Frage. In diesem Zusammenhang ist die **String**-Instanzmethode **intern()** von Nutzen, mit deren Hilfe man den internen **String**-Pool bevölkern kann. Sie besitzt keinen Parameter und liefert als so genannte *kanonische Repräsentation* eine Referenz vom Typ **String**:

- Ist im internen Pool ein inhaltsgleicher **String** vorhanden (im Sinne der **equals()**-Methode), wird dessen Adresse als Rückgabe geliefert.
- Anderenfalls wird der Parameterstring in den internen **String**-Pool aufgenommen und seine Adresse als Rückgabe geliefert.

Bei Referenzvariablen zu internalisierten Strings folgt aus der Gleichheit der Adressen bereits die Inhaltsgleichheit. Folglich kann man beim Identitätstest statt des relativ aufwändigen Inhaltsvergleichs den erheblich flotteren Referenzvergleich durchführen. Dazu müssen die **String**-Objekte aber zunächst interniert werden, was einigen Zeitaufwand erfordert. Im folgenden Programm werden `ANZ` Zufallszeichenfolgen der Länge `LEN` jeweils `N` mal mit einem zufällig gewählten Partner

[32] Weil die im **String**-Pool vorhandenen Referenzen eine Entsorgung der zugehörigen Objekte durch den Garbage Collector verhindern, kommt der gewünschte Speichereinspareffekt *nicht* unbedingt zu Stande.

verglichen. Dies geschieht zunächst per **equals()**-Methode und dann nach dem zwischenzeitlichen Internieren per Adressvergleich.

```
import java.util.*;

class StringIntern {
   public static void main(String[] args) {
      final int ANZ = 50000, LEN = 20, N = 50;
      StringBuffer sb = new StringBuffer();
      Random ran = new Random();
      String[] sar  = new String[ANZ];
      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;
      // N * ANZ Inhaltsvergleiche
      for (int n = 1; n <= N; n++)
         for (int i = 0; i < ANZ; i++)
            if (sar[i].equals(sar[ran.nextInt(ANZ)]))
               hits++;
      System.out.println((N * ANZ)+" Inhaltsvergleiche ("+hits+
         " hits) benoetigen "+(System.currentTimeMillis()-start)+
         " Millisekunden");

      start = System.currentTimeMillis();
      hits = 0;
      // Internieren
      for (int j = 1; j < ANZ; j++)
         sar[j] = sar[j].intern();
      // N * ANZ Adressvergleiche
      for (int n = 1; n <= N; n++)
         for (int i = 0; i < ANZ; i++)
            if (sar[i] == sar[ran.nextInt(ANZ)])
               hits++;
      System.out.println((N * ANZ)+" Adressvergleiche ("+hits+
         " hits) benoetigenn (inkl. Internieren) "+
         (System.currentTimeMillis()-start)+" Millisekunden");
   }
}
```

Es hängt von den Aufgabenparametern `ANZ`, `LEN` und `N` ab, welche Vergleichsmethode überlegen ist:[33]

| | Laufzeit in Millisekunden | |
|---|---|---|
| | **equals()**-Vergleiche | Internieren plus Adress-Vergl. |
| `ANZ` = 50000, `LEN` = 20, `N` = 5 | 78 | 172 |
| `ANZ` = 50000, `LEN` = 20, `N` = 50 | 781 | 281 |

Erwartungsgemäß ist das Internieren umso rentabeler, je mehr Vergleiche anschließend mit den Zeichenfolgen angestellt werden. Bei **String**-Vergleichen sind sicher noch weitere Verbesserungen möglich, z.B. durch Ausnutzen der lexikographischen Ordnung.

Zum Testen auf **lexikographische Priorität** (z.B. beim Sortieren) kann die **String**-Methode

**public int compareTo(String** *vergl***)**

dienen, z.B.:

[33] Die Ergebnisse stammen von einem PC mit Intel - CPU (Pentium 4, 3-GHz).

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  String a = "Müller, Anja", b = "Müller, Kurt",`<br>`  c = "Müller, Anja";`<br>`  System.out.println("< : " + a.compareTo(b));`<br>`  System.out.println("= : " + a.compareTo(c));`<br>`  System.out.println("> : " + b.compareTo(a));`<br>`  }`<br>`}` | `< : -10`<br>`= : 0`<br>`> : 10` |

Die Methode **compareTo()** liefert folgende **int**-Rückgabewerte:

| | | **compareTo()**-Rückgabe |
|---|---|---|
| Das **String**-Objekt, dessen **compareTo()**-Methode aufgerufen wird, ist im Vergleich zum Argument: | kleiner | negative Zahl |
| | gleich | 0 |
| | größer | positive Zahl |

#### 5.2.1.4.3 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 {`<br>`  public static void main(String[] args) {`<br>`  char[] cvek = {'a', 'b', 'c'};`<br>`  String str = "abc";`<br>`  System.out.println(cvek.length);`<br>`  System.out.println(str.length());`<br>`  }`<br>`}` | `3`<br>`3` |

#### 5.2.1.4.4 Zeichen(folgen) extrahieren, suchen oder ersetzen

Im Abschnitt 3.7.2.3 zur Fallunterscheidung per **switch**-Anweisung haben wir erstmals mit dem Array von **String**-Elementen gearbeitet, über den Java-Programme Zugang zu den beim Start übergebenen Programmargumenten haben. Im folgenden Beispielprogramm wird das erste Element dieses Arrays untersucht (`args[0]`):

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  if (args.length > 0) {`<br>`  System.out.println(args[0].substring(1, 3));`<br>`  System.out.println(args[0].indexOf("t"));`<br>`  System.out.println(args[0].indexOf("x"));`<br>`  System.out.println(args[0].startsWith("r"));`<br>`  System.out.println(args[0].charAt(0));`<br>`  }`<br>`  }`<br>`}` | `ot`<br>`2`<br>`-1`<br>`true`<br>`r` |

**substring()**

Mit der Methode

**public String substring(int** *start***, int** *ende***)**

lassen sich alle Zeichen zwischen *start* (inklusive) und *ende* (exklusive) extrahieren.

**indexOf()**

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

**startsWith()**

Mit der Methode

**public boolean startsWith**(**String** *start*)

lässt sich feststellen, ob ein **String** mit einer bestimmten Zeichenfolge beginnt.

**charAt()**

Weil ein **String** *kein* Array ist, kann auf die einzelnen Zeichen *nicht* per Indexoperator ([]) zugegriffen werden. Mit der **String**-Methode

**public char charAt()**

steht aber ein Ersatz zur Verfügung, wobei die Nummerierung der Zeichen wiederum bei 0 beginnt.

**toCharArray()**

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 {`<br>`  public static void main(String[] args) {`<br>`  String s = "abc";`<br>`  char[] c = s.toCharArray();`<br>`  for (int i = 0; i < c.length; i++)`<br>`  System.out.println(c[i]);`<br>`  }`<br>`}` | `a`<br>`b`<br>`c` |

**replace()**

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.

#### 5.2.1.4.5 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 {`<br>`  public static void main(String[] args) {`<br>`  String a = "Otto", b = "otto";`<br>`  System.out.println(a.toUpperCase().equals(b.toUpperCase()));`<br>`  }`<br>`}` | `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.

### 5.2.2 Die Klasse StringBuffer für veränderliche Zeichenketten

Für häufig zu ändernde Zeichenketten sollte man statt der Klasse **String** unbedingt die Klasse **StringBuffer** verwenden, weil hier beim Ändern einer Zeichenkette die (zeitaufwändige) Erzeugung eines neuen Objektes entfällt.

Ein **StringBuffer** kann nicht *implizit* erzeugt werden, jedoch stehen bequeme Konstruktoren zur Verfügung, z.B.:

- **public StringBuffer**(**String** *str*)
  Beispiel: `StringBuffer sb = new StringBuffer("abc");`
- **public StringBuffer**()
  Beispiel: `StringBuffer sb = new StringBuffer();`

In folgendem Programm wird eine Zeichenkette 20000-mal verlängert, zunächst mit Hilfe der Klasse **String**, dann mit Hilfe der Klasse **StringBuffer**:

```
class Prog {
  public static void main(String[] args) {
    final int DRL = 20000;
    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\":     "+diff);
```

```
        s = "*";
        StringBuffer t = new StringBuffer(s);
        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 StringBuffer-Verlaengerung: "+diff);
    }
}
```

Die Laufzeiten (gemessen in Millisekunden auf einem PC mit 3-GHz-CPU Intel Pentium 4) unterscheiden sich erheblich:

```
Zeit fuer die String-"Verlaengerung":     765
Zeit fuer die StringBuffer-Verlaengerung: 5
```

Ein **StringBuffer**-Objekt beherrscht u.a. die folgenden **public**-Methoden:

| StringBuffer-Methode | Erläuterung |
|---|---|
| **int length()** | Diese Methode liefert die Anzahl der Zeichen. |
| **append()** | Der **StringBuffer** wird um die Stringrepräsentation des Argumentes verlängert, z.B.:<br>`sb.append("*");`<br>Es sind **append()**-Überladungen für zahlreiche Datentypen vorhanden. |
| **insert()** | Die Stringrepräsentation des Arguments, das von nahezu beliebigem Typ sein kann, wird an einer bestimmten Stelle eingefügt, z.B.:<br>`sb.insert(4, 3.14);` |
| **delete()** | Die Zeichen von einer Startposition (einschließlich) bis zu einer Endposition (ausschließlich) werden gelöscht, in folgendem Beispiel also gerade zwei Zeichen, falls der **StringBuffer** mindestens drei Zeichen enthält:<br>`sb.delete(1,3);` |
| **replace()** | Ein Bereich des **StringBuffer**-Objekts wird durch den Argument-**String** ersetzt, z.B.:<br>`sb.replace(1,3, "xy");` |
| **String toString()** | Es wird ein **String**-Objekt mit dem Inhalt des **StringBuffer**-Objekts erzeugt. Dies ist z.B. erforderlich, um zwei **StringBuffer**-Objekte mit Hilfe der **String**-Methode **equals()** vergleichen zu können:<br>`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).

In der Regel verfügen die Wrapper-Klassen über zwei Konstruktoren mit jeweils einem Parameter, der vom zugehörigen primitiven Typ oder vom Typ **String** sein darf, z.B. bei **Integer**:

- **public Integer**(**int** *value*)
  Beispiel: `Integer iw = new Integer(4711);`
- **public Integer**(**String** *str*)
  Beispiel: `Integer iw = new Integer(args[0]);`

Der beim Erzeugen festgelegte Wert eines Wrapper-Objekts kann nicht mehr geändert werden kann. Daher besitzen die Wrapper-Klassen keinen parameterfreien Konstruktor.

### 5.3.1 Autoboxing

Seit der Java-Version 5 kann der Compiler das Einpacken automatisch erledigen, z.B.:

```
Integer iw = 4711;
```

Damit vereinfacht sich die Nutzung von Methoden, die **Object**-Parameter erwarten. Im folgenden Beispielprogramm wird ein Objekt der Klasse **Vector** aus dem Paket **java.util** als bequemer und flexibler Container verwendet:

- Ein **Vector** kann beliebige Objekte als Elemente aufnehmen.
- Die Größe des Vektors wird automatisch an den Bedarf angepasst.
- **Vector** ist eine *generische* Klasse (siehe Abschnitt 6) und wird in der Regel mit Elementen *eines* bestimmten Datentyps benutzt. Dieser ist beim Instantiieren anzugeben, wenn der Compiler die Typhomogenität überwachen soll. Wir verwenden in diesem Abschnitt den so genannten Rohtyp der Klasse **Vector**, der sich gut zur Autoboxing-Demonstration eignet, aber ansonsten eher zu vermeiden ist.

Um Werte primitiver Typen in einen **Vector**-Container einzufügen, müssen sie in Wrapper-Objekte verpackt werden, was aber dank Autoboxing keine Mühe macht:

```
class Autoboxing {
  public static void main(String[] args) {
    java.util.Vector v = new java.util.Vector();
    v.addElement("Otto");
    v.addElement(13);
    v.addElement(23.77);
    v.addElement('x');

    System.out.println("Der Vector enthält:");
    for(Object o : v)
      System.out.println(" " + o + "\t Typ: " + o.getClass());
  }
}
```

Wie die Programmausgabe zeigt, sind tatsächlich diverse Wrapper-Klassen im Spiel:

```
Der Vector enthält:
 Otto      Typ: class java.lang.String
 13        Typ: class java.lang.Integer
 23.77     Typ: class java.lang.Double
 x         Typ: class java.lang.Character
```

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 nun auch zuweisungskompatibel zur Klasse **Object**, wobei zum Auspacken aber eine explizite Typumwandlung erforderlich ist, z.B.:

```
Object o = 4711;
int i = (Integer) o;
```

### 5.3.2 Konvertierungs-Methoden

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***)**

  liefert einen **double**-Wert zurück, falls die Konvertierung der Zeichenfolge gelingt.
- Die **Double**-Klassenmethode

  **public static Double valueOf(String** *str***)**

  liefert einen verpackten **double**-Wert zurück, falls die Konvertierung der Zeichenfolge gelingt.

Bei einer großen Anzahl von Konvertierungen ist die Methode **valueOf()** wegen der aufwändigen Objekt-Kreationen *nicht* empfehlenswert.

Das folgende Beispielprogramm berechnet die Summe der numerisch interpretierbaren Kommandozeilenparameter:

```
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 Abschnitt über Ausnahmebehandlung ausführlich besprochen wird, versucht das Programm für jeden Kommandozeilenparameter 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 eine Zeichenkette 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);
```

### 5.3.3 Konstanten mit Grenzwerten

In den numerischen Wrapper-Klassen sind öffentliche, finalisierte und statische Instanzvariablen für diverse Grenzwerte definiert, z.B. in der Klasse **Double**:

| Konstante | Inhalt |
|---|---|
| **MAX_VALUE** | Größter positiver (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 {`<br>`  public static void main(String[] args) {`<br>`  System.out.println("Max. double-Zahl:\n"+`<br>`  Double.MAX_VALUE);`<br>`  }`<br>`}` | `Max. double-Zahl:`<br>`1.7976931348623157E308` |

### 5.3.4 Character-Methoden zur Zeichen-Klassifikation

Die Wrapper-Klasse **Character** 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**. |

| Methode | Erläuterung |
|---|---|
| **boolean isWhitespace(char ch)** | Die Methode liefert den Wert **true** zurück, wenn ein Trennzeichen übergeben wurde, sonst **false**. Zu den Trennzeichen gehören u.a.:<br>• Leerzeichen (\u0020)<br>• Tabulatorzeichen (\u0009)<br>• Wagenrücklauf (\u000D)<br>• 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 einer Adressdatenbank auch den Charakter der erfassten Personen notieren 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 == "Phlekmatisch") ...
  ```
- 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) ...
  ```

Bei beiden Vorschlägen ist nicht garantiert, dass eine Variable zur Temperamentsverwaltung ausschließlich die vier vorgesehenen Werte aufnehmen kann (mangelnde Typsicherheit).

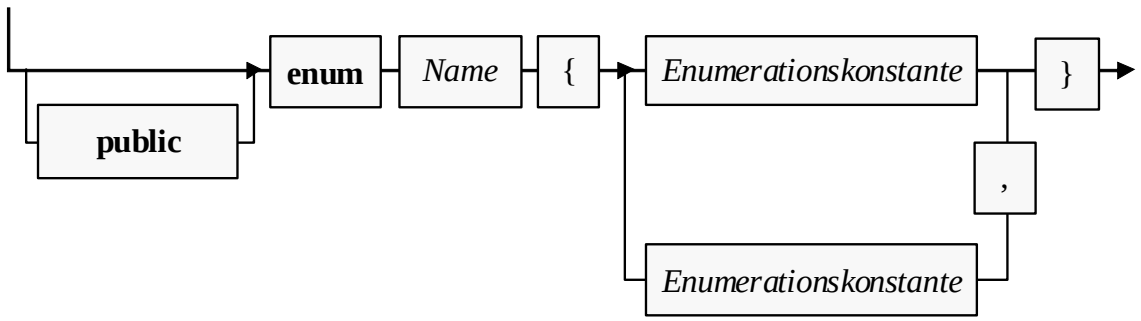
Java bietet seit der Version 5 mit den **Enumerationen** (**Aufzählungstypen)** eine Lösung, die folgende Vorteile bietet:

- Perfekte Typsicherheit
- Gut lesbarer Quellcode

### 5.4.1 Einfache Enumerationen

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 Enumerationskonstanten:

**Einfache Enumerationsdefinition**



Weil Syntaxdiagramme zwar sehr präzise, aber nicht unbedingt mit einem Blick verständlich sind, betrachten wir ergänzend gleich ein Beispiel:

```
public enum Temperament {MELANCHOLISCH, CHOLERISCH, PHLEGMATISCH, SANGUIN}
```

Man sollte die Namen der Enumerationskonstanten komplett groß schreiben, wie es sich bei Konstanten (d.h. finalisierten Variablen) generell eingebürgert hat.

Objekte der folgenden Klasse `Person` (mit ungeschützten Eigenschaften!) 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:

```
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");
  }
}
```

Ausdrücke mit Aufzählungstyp sind auch in **switch**-Anweisungen erlaubt, wobei aber ausnahmsweise der Typname nicht nur überflüssig, sondern sogar *verboten* ist, 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 haben wir erfolgreich mit der relativ einfachen Vorstellung gearbeitet, dass eine Enumeration ein Ganzzahltyp mit einer kleinen Menge von benannten Werten sei. Tatsächlich sind Enumerationen in Java aber als *Klassen* realisiert (mit der Basisklasse **Enum** aus dem Paket **java.lang**), wobei im Vergleich zu gewöhnlichen Klassen einige Besonderheiten zu beachten sind:

- Die Enumerationskonstanten zeigen als statische und finalisierte Referenzvariablen auf Objekte der Enumerationsklasse, die beim Laden der Klasse automatisch entstehen.
- Es ist nicht möglich, weitere Objekte einer Enumerationsklasse (per **new**-Operator oder auf andere Weise) zu erzeugen.
- Man kann eine Enumeration nicht beerben.
  In Abschnitt 8.1 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**.

In vielen Programmiersprachen stellen die Werte eines Enumerationstyps lediglich symbolische Namen für die natürlichen Zahlen (0, 1, 2, …) in der Definitionsreihenfolge dar. Auch die Enumerationsobjekte der Programmiersprache Java kennen ihre Position in der definierenden Liste und liefern diese als Rückgabewert der Instanzmethode **ordinal()**, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `class PersonTest {`<br>`  public static void main(String[] args) {`<br>`  Person otto = new Person("Otto", "Hummer",`<br>`  35, Temperament.SANGUIN);`<br>`  System.out.println(otto.temp.ordinal());`<br>`  }`<br>`}` | `3` |

Bei jeder Enumerationsklasse kann man mit der statischen Methode **values()** einen Array mit ihren Objekten anfordern, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `class PersonTest {`<br>`  public static void main(String[] args) {`<br>`  for (Temperament t : Temperament.values())`<br>`  System.out.println(t.name());`<br>`  }`<br>`}` | `MELANCHOLISCH`<br>`CHOLERISCH`<br>`PHLEGMATISCH`<br>`SANGUIN` |

### 5.4.2 Erweiterte Enumerationen

Es ist möglich, eine Enumerationsklasse mit Instanzvariablen, Methoden und (privaten) Konstruktoren auszustatten. Objekte der folgenden `TemperamentEx` - Enumeration geben über die Methoden `stable()` bzw. `extra()` Auskunft darüber, ob die zugehörige Persönlichkeit emotional stabil bzw. extravertiert ist:[34]

```
public enum TemperamentEx {MELANCHOLISCH(false, false),
                           CHOLERISCH(false, true),
                           PHLEGMATISCH(true, false),
                           SANGUIN(true, true);
  private boolean stable, extra;
  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 kann nur innerhalb der Enumerationsklasse genutzt werden. Dazu werden Aktualparameterlisten an die Enumerationskonstanten angehängt.

[34] 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** (ca. 275 - 195 v. Chr.). Dieser Algorithmus reduziert sukzessive eine Menge von Primzahlkandidaten, die initial alle natürlichen Zahlen bis zu einer Obergrenze *K* enthält, also {1, 2, 3, ..., *K*}:

- Im ersten Schritt werden alle echten Vielfachen der Basiszahl 2 (also 4, 6, ...) aus der Kandidatenmenge gestrichen, während die Zahl 2 in der Liste verbleibt.
- Dann geschieht iterativ folgendes:
    - Als neue Basis *b* wird die kleinste Zahl gewählt, welche die beiden folgenden Bedingungen erfüllt:
      *b* ist größer als die vorherige Basiszahl.
      *b* ist im bisherigen Verlauf nicht gestrichen worden.
    - Die echten Vielfachen der neuen Basis (also 2·*b*, 3·*b*, ...) werden aus der Kandidatenmenge gestrichen, während die Zahl *b* in der Liste verbleibt.
- Das Streichverfahren kann enden, wenn für eine neue Basis *b* gilt:

$$b > \sqrt{K}$$

Ist eine natürliche Zahl $n \leq K$ ein Vielfaches von $b$, dann gibt es eine natürliche Zahl $n_b$ mit

$$n = n_b \cdot b \text{ und } n_b < \sqrt{K}$$

Wir nehmen es ganz genau und unterscheiden zwei Fälle:

- $n_b$ war zuvor als Basis dran:
  Dann wurde $n$ bereits als Vielfaches von $n_b$ gestrichen.
- $n_b$ wurde zuvor als Vielfaches einer Basis $\tilde{b}$ gestrichen ( $n_b = u\tilde{b}$ )
  Dann wurde $n$ bereits als Vielfaches von $\tilde{b}$ gestrichen: $n = n_b\, b = u\tilde{b}\, b = ub\tilde{b}$

Die Kandidatenmenge enthält also nur noch Primzahlen.

Sollen z.B. alle Primzahlen kleiner oder gleich 18 bestimmt werden, so startet man mit folgender Kandidatenmenge:

| 1 | 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:

| 1 | 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:

| 1 | 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:

1, 2, 3, 5, 7, 11, 13 und 17

4) Definieren Sie eine Klasse für eine zweidimensionale Matrizen mit Elementen vom Typ **double** zur Aufnahme von Beobachtungswerten. 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 die Berechnung von Mittelwerten und Varianzen bei *einer* einzigen Passage durch die Zeilen der Matrix, während die Originalformel eine vorgeschaltete Passage zur Berechnung der Mittelwerte benötigt.

**Abschnitt 5.2 (Klassen für Zeichenketten)**

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 einer Array per Index-Syntax 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 Kommandozeilenparameter 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 Buchstaben-Nummern addieren und die Summe als Startwert für den Pseudozufallszahlengenerator verwenden.

Beenden Sie Ihr Programm mit einer Fehlermeldung, wenn weniger als zwei Kommandozeilenparameter übergeben wurden.

Tipp: Um ein Programm spontan zu beenden, kann die Methode **exit()** der Klasse **System** verwendet werden.

4) Die Klassen **String** und **StringBuffer** besitzen beide eine Methode namens **equals()**, doch bestehen gravierende Verhaltensunterschiede:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  StringBuffer sb1 = new StringBuffer("abc");`<br>`  StringBuffer sb2 = new StringBuffer("abc");`<br>`  System.out.println("sb1 = sb2 = "+sb1);`<br>`  System.out.println("StringBuffer-Vergl.: "+`<br>`  sb1.equals(sb2));`<br><br>`  String s1 = sb1.toString();`<br>`  String s2 = sb1.toString();`<br>`  System.out.println("\ns1 = s2 = "+s1);`<br>`  System.out.println("String-Vergl.:  "+`<br>`  s1.equals(s2));`<br>`  }`<br>`}` | `sb1 = sb2 = abc`<br>`StringBuffer-Vergl.: false`<br><br>`s1 = s2 = abc`<br>`String-Vergl.:  true` |

Ermitteln Sie mit Hilfe der JDK-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, " ", 40);
    StringUtil.wrapln(s, 40);
    StringUtil.wrapln(s);
  }
}
```

Der zweite Methodenaufruf sollte folgende Ausgabe erzeugen:

```
Dieser Satz passt nicht in eine Schmal-
Zeile, die nur wenige Spalten umfasst.
```

Das Zerlegen eines Strings in einzelne *Tokens* 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 {`<br>`  public static void main(String[] args) {`<br>`  String s = "Dies ist der Satz, der zerlegt werden soll.";`<br>`  java.util.StringTokenizer stok =`<br>`  new java.util.StringTokenizer(s, " ", false);`<br>`  while (stok.hasMoreTokens())`<br>`  System.out.println(stok.nextToken());`<br>`  }`<br>`}` | `Dies`<br>`ist`<br>`der`<br>`Satz,`<br>`der`<br>`zerlegt`<br>`werden`<br>`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 in Java 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 **Vector** 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.

4) Erweitern Sie die in einer Übungsaufgabe zu Abschnitt 5.2 erstellte Klasse `StringUtil` um eine statische Methode namens `countLetters()`, die für einen **String**–Parameter die enthaltenen Buchstaben mit der Häufigkeiten des Auftretens protokolliert. In folgendem Programm wird die Verwendung der Methode demonstriert:

| Quellcode | Ausgabe |
|---|---|
| `class StringUtilTest {`<br>`  public static void main(String[] args) {`<br>`  StringUtil.countLetters("Otto spielt Lotto." , true);`<br>`  }`<br>`}` | `L: 1`<br>`O: 1`<br>`e: 1`<br>`i: 1`<br>`l: 1`<br>`o: 3`<br>`p: 1`<br>`s: 1`<br>`t: 5` |

**Abschnitt 5.4 (Aufzählungstypen)**

1) Erstellen und erproben Sie einen Datentyp `Wochentage`, der folgende Bedingungen erfüllt:

- Typsicherheit
  Einer Variablen vom Typ `Wochentage` können nur sieben verschiedene Werte zugewiesen werden, die den Wochentagen Sonntag, Montag, etc. (in dieser Reihenfolge) entsprechen.
- Ordnungsinformation
  Für zwei Werte des Typs `Wochentage` kann leicht die zeitliche Anordnung festgestellt werden.
- Leicht lesbarer Quellcode
- Verwendbarkeit in **switch**-Anweisungen

# 6 Typgenerizität und Kollektionen

Während sich die Fortentwicklung der Java-Plattform meist auf die Standardbibliothek konzentrierte, ist mit der Version 5 auch die *Programmiersprache* Java wesentlich erweitert worden, indem generische Typen und Methoden Einzug gehalten haben. Wir werden in diesem Abschnitt erste Erfahrungen mit dieser Technik zur weiteren Verbesserung der Wiederverwendbarkeit von Quellcode sammeln. Wegen der starken Verschränkung mit anderen noch unbehandelten Begriffen bzw. Prinzipien (vor allem Vererbung und Interfaces) folgen später noch wesentliche Ergänzungen zur Generizität. Weitere Details zu generischen Typen und Methoden finden sich z.B. im Tutorium von Bracha (2004).

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 Abschnitt 6.4 vorgestellt wird. Für die Objekte dieser Klassen wird im Manuskript alternativ zur offiziellen Bezeichnung *Kollektion* aus sprachlichen Gründen oft auch die Bezeichnung *Container* verwendet. Dass später auch von Containern zur Verwaltung von GUI-Bedienelementen die Rede sein wird, sollte keine Verwirrung stiften.

## *6.1 Die generische Kollektionsklasse Vector*

In Abschnitt 5.3.1 haben wir die Klasse **Vector** aus dem Paket **java.util** als Container für Objekte beliebigen Typs verwendet:

```
java.util.Vector v = new java.util.Vector();
v.addElement("Otto");
v.addElement(13);
v.addElement(23.77);
v.addElement('x');
```

Im Unterschied zu einem gewöhnlichen Java-Array (siehe Abschnitt 5.1) bietet die Klasse **Vector**:

- eine automatische Größenanpassung
- Typflexibilität (durch Verwendung des Elementtyps **Object**)

Während das obige Beispiel die Größen- *und* die Typflexibilität nutzt, ist oft ein Container mit automatischer Größenanpassung (ein dynamischer Array) für Objekte eines bestimmten, *festen* 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 sicher stellen, 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 **Vector**-Container verwaltet werden.

```
class RawVector {
   public static void main(String[] args) {
      java.util.Vector v = new java.util.Vector();
      v.addElement("Otto");
      v.addElement("Rempremerding");
      v.addElement('.');
      int i = 0;
      for (Object s: v)
         System.out.printf("Laenge von Zeile %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()
```

Weil der dritte **addElement()**-Aufruf ein **Character**-Objekt (Autoboxing!) in den Container befördert, endet das Programm mit einem Ausnahmefehler:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Character
cannot be cast to java.lang.String
  at GenVector.main(GenVector.java:9)
```

Es ist nicht allzu schwer, eine spezielle Container-Klasse zur Verwaltung von **String**-Objekten zu definieren, welche die beiden 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 Typparametern wird die gesamte Handlungskompetenz der Klasse typunabhängig formuliert. Bei jeder Instantiierung wird der Typ jedoch konkretisiert, so dass Typsicherheit und syntaktische Eleganz resultieren.

Wie ein Blick in die API-Dokumentation zeigt, ist die Klasse **Vector** selbstverständlich generisch realisiert und verwendet einen Typparameter:

**java.util**

## Class Vector<E>

```
java.lang.Object
  └java.util.AbstractCollection<E>
      └java.util.AbstractList<E>
          └java.util.Vector<E>
```

Wir haben bisher naiv den aus Kompatibilitätsgründen noch unterstützten **Rohtyp** der generischen Klasse **Vector** (mit Elementtyp **Object**) verwendet (siehe Abschnitt 6.2.5). Er ist wenig geeignet, wenn ein sortenreiner Container (alle Elemente vom selben Typ) benötigt wird. Die beiden Nachteile dieser Konstellation (Typunsicherheit, lästige Typanpassungen) wurden oben beschrieben.

Wird im Beispiel der Container mit Angabe des gewünschten Elementtyps instantiiert,

```
java.util.Vector<String> v = new java.util.Vector<String>();
```

dann verhindert der Compiler die Aufnahme des ungeeigneten **Character**-Objekts. Der Eclipse-Compiler unterschlängelt die Defektstelle

```
v.addElement('.');
```

und der JDK-Compiler meldet:

```
>javac GenVector.java
GenVector.java:13: addElement(java.lang.String)
in java.util.Vector<java.lang.String> cannot be applied to (char)
                v.addElement('.');
                  ^
1 error
```

Die Elemente des auf **String**-Objekte spezialisierten **Vector**-Containers beherrschen ohne Typanpassung die Methoden ihrer Klasse, z.B.:

```
for (String s: v)
  System.out.printf("Laenge von Zeile %d: %d\n", ++i, s.length());
```

Generische Klassen ermöglichen robustere Programme (dank Typüberwachung durch den Compiler), die zudem leichter lesbar sind.

## 6.2 Generische Klassen

Aus der Entwicklerperspektive besteht der wesentliche Vorteil einer generischen Klasse darin, dass mit *einer* Definition beliebig viele konkrete Klassen für spezielle Datentypen geschaffen werden. Dieses Konstruktionsprinzip ist speziell bei den Kollektionsklassen sehr verbreitet (siehe Abschnitt 6.4), aber keinesfalls auf Container mit ihrer weitgehend inhaltstypunabhängigen Verwaltungslogik beschränkt.

### 6.2.1 Definition

Bei der generischen Klassendefinition verwendet man **Typformalparameter**, die im Kopf der Definition hinter dem Klassennamen zwischen spitzen Klammern und durch Kommata getrennt angegeben werden. Wir erstellen als einfaches Beispiel eine generische Container-Klasse namens `SimpleList<E>`, die hinsichtlich Einsatzzweck und Konstruktion den Listenverwaltungsklassen aus dem Java Collection Framework ähnelt (z.B. **Vector<E>**, siehe oben und Abschnitt 6.4.1), aber nicht annähernd denselben Funktionsumfang bietet. Für den Datentyp seiner Elemente benutzt der generische Container `SimpleList<E>` den Typformparameter `E`:

```
public class SimpleList<E> {
  private Object[] elements;
  private int initSize = 10;
  private int size;

  public SimpleList(int len) {
    if (len > 0) {
      elements = new Object[len];
      initSize = len;
    } else
      elements = new Object[initSize];
  }

  public SimpleList() {
       elements = new Object[initSize];
  }

  public void add(E element) {
    if (size ==  elements.length) {
      Object[] temp = new Object[elements.length + initSize];
      for (int i = 0; i  < elements.length; i++)
        temp[i] = elements[i];
      elements = temp;
    }
    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 wird der Typformalparameter wie ein konkreter Referenzdatentyp verwendet.

Es wird empfohlen, für Typformalparameter einzelne Großbuchstaben zu verwenden, z.B.:

**T** „Type“
**E** „Element“
**K** „Key“
**V** „Value“

Es überrascht, dass die generische Container-Klasse `SimpleList<E>` bei dem zur internen Datenverwaltung genutzten Array den festen Urahntyp **Object** nutzt. In Abschnitt 6.2.6 werden Sie erfahren, warum der Typparameter **E** an dieser Stelle leider *nicht* verwendbar ist.

Beim Erstellen des intern zur Datenspeicherung verwendeten Arrays wird als Länge die per Konstruktor-Parameter festgelegte initialen Listenlänge verwendet. In der Methode `add()` wird bei Bedarf 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 {`<br>`  public static void main(String[] args) {`<br>`  SimpleList<String> sls = new SimpleList<String>(3);`<br>`  sls.add("Otto");`<br>`  sls.add("Rempremmerding");`<br>`  System.out.println("Laenge: "+sls.size()+`<br>`  " Kapazitaet: "+sls.capacity());`<br><br>`  sls.add("Hans");`<br>`  sls.add("Brgl");`<br>`  System.out.println("Laenge: "+sls.size()+`<br>`  " Kapazitaet: "+sls.capacity());`<br><br>`  for (int i=0; i < sls.size();i++)`<br>`  System.out.println(sls.get(i));`<br>`}` | `Laenge: 2 Kapazitaet: 3`<br>`Laenge: 4 Kapazitaet: 6`<br>`Otto`<br>`Rempremmerding`<br>`Hans`<br>`Brgl` |

Als Beispiel für eine generische Klasse mit *zwei* Typformalparametern betrachten wird die API-Klasse **HashMap**, die eine Tabelle mit Schüssel-Wert - Paaren verwaltet

**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

Beim Verwenden eines generischen Typs durch Wahl konkreter Datentypen an Stelle der Typformalparameter entsteht ein so genannter **parametrisierter Typ**, z.B. `SimpleList<String>`.

### 6.2.2 Gebundene Typformalparameter

Häufig muss eine generische Klassendefinition bei den Klassen, welche einen Typparameter konkretisieren dürfen, gewisse Handlungskompetenzen voraussetzen. Soll z.B. ein generischer Container seine Elemente *sortieren*, kann für jede konkrete Elementklasse gefordert werden, dass sie das das Interface **Comparable<E>** implementieren muss. Wir benötigen hier den wichtigen Begriff *Interface*, mit dem wir uns in Kapitel 10 ausführlich beschäftigen werden. Allerdings stellt der Vor-

griff kein didaktisches Problem dar, weil die Forderung an eine zulässige Konkretisierungsklasse leicht mit vertrauten Begriffen formuliert werden kann: Diese muss eine Methode namens **compareTo()** besitzen (hier beschrieben unter Verwendung des Typparameters **E**):

**public int compareTo(E** *vergl***)**

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 genug sein, was die Schnittstelle (das Interface) **Comparable<E>** von einem Typ verlangt. Wie das Beispiel **Comparable<E>** zeigt, sind auch bei Schnittstellen typgenerische Varianten von großer Bedeutung.

Wir erstellen nun eine Variante der simplen Container-Klasse aus Abschnitt 6.2.1, die eingefügte Elemente automatisch einsortiert und daher ihren Typformalparameter auf den Datentyp **Comparable<E>** einschränkt:

```
public class SimpleSortedList<E extends Comparable<E>> {
   private Comparable[] elements;
   private int initSize = 10;
   private int size;

   public SimpleSortedList(int len) {
      if (len > 0) {
         elements = new Comparable[len];
         initSize = len;
      } else
         elements = new Comparable[initSize];
   }

   public SimpleSortedList() {
         elements = new Comparable[initSize];
   }

   public void add(E element) {
      if (size == elements.length) {
         Comparable[] temp = new Comparable[elements.length + initSize];
         for (int i = 0; i < elements.length; i++)
            temp[i] = elements[i];
         elements = temp;
      }
      boolean inserted = false;
      for (int i = 0; i < size; i++) {
         if (element.compareTo((E)elements[i]) <= 0) {
            for (int j = size; j > i; j--)
               elements[j] = elements[j-1];
            elements[i] = element;
            inserted = true;
            break;
         }
      }
      if (!inserted)
         elements[size] = element;
      size++;
   }

   public E get(int index) {
      if (index >= 0 && index < size)
         return (E) elements[index];
      else
         return null;
   }

   public int size() {return size;}

   public int capacity() {return elements.length;}
}
```

Bei der Formulierung von Einschränkungen (synonym: *Bindungen*) für Typparameter wird das Schlüsselwort **extends** verwendet, das Sie bald im Zusammenhang mit Vererbungsbeziehungen zwischen Klassen kennen lernen werden. Wie das Beispiel zeigt, ist das Schlüsselwort auch bei einer Typparameterrestriktion unter Verwendung einer *Schnittstelle* zu verwenden.

Beim intern zur Datenspeicherung verwendeten Array wird die Schnittstelle **Comparable** als Datentyp verwendet, worüber Sie sich jetzt noch wundern dürfen. Im Kapitel 10 über Schnittstellen wird diese Verwendungsart als wichtige Option behandelt.

Wie schon in Abschnitt 6.2.1 ist zu bedauern, dass der Typparameter **E** leider *nicht* als Elementtyp für den internen Array verwendbar ist. Den Grund erfahren Sie in Abschnitt 6.2.6.

Wie das folgende Testprogramm zeit, hält ein Objekt der Klasse `SimpleSortedList<E extends Comparable<E>>` seine Elemente stets in sortiertem Zustand:

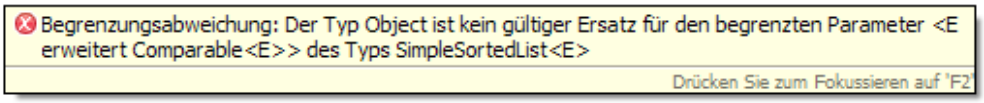
| Quellcode | Ausgabe |
|---|---|
| `class SimpleSortedListTest {`<br>`  public static void main(String[] args) {`<br>`  SimpleSortedList<Integer> si =`<br>`  new SimpleSortedList<Integer>(3);`<br>`  si.add(4); si.add(11); si.add(1); si.add(2);`<br>`  System.out.println("Laenge: "+si.size()+`<br>`  " Kapazitaet: "+si.capacity());`<br>`  for (int i=0; i < si.size();i++)`<br>`  System.out.println(si.get(i));`<br>`  }`<br>`}` | `Laenge: 4 Kapazitaet: 6`<br>`1`<br>`2`<br>`4`<br>`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<Object>(3);
```




Man kann für einen Typparameter auch *mehrere* Bindungen (Restriktionen) definieren, die mit dem &-Zeichen verknüpft werden. Im folgenden Beispiel

```
class MultiRest<E extends SuperKlasse & Comparable<E>> {...}
```

steht **E** für einen Datentyp, der …

- von `SuperKlasse` abstammt und
- die Schnittstelle **Comparable<E>** unterstützt.
  Die später noch ausführlich zu behandelnden Schnittstellen dienen dazu, Verhaltenskompetenzen von Objekten über eine Liste von Methodendefinitionsköpfen festzulegen. Um eine Schnittstelle zu erfüllen, muss eine Klasse alle vorgeschriebenen Methoden implementieren.

### 6.2.3 Zuweisungskompatibilität

Bei der ersten Beschäftigung mit generischen Klassen könnte man z.B. den parametrisierten Datentyp `SimpleList<String>` für eine Spezialisierung des parametrisierten Typs `SimpleList<Object>` halten, weil schließlich die Klasse **String** eine Spezialisierung der Urahnklasse

**Object** ist. Wie bald im Kapitel über Vererbung zu sehen sein wird, können Objekte einer abgeleiteten Klasse über Referenzvariablen der Basisklasse angesprochen werden. Der Compiler verbietet jedoch zu Recht, ein Objekt der Klasse `SimpleList<String>` über eine Referenzvariable vom Typ `SimpleList<Object>` anzusprechen, z.B.:

```
public static void main(String[] args) {
    SimpleList<String> slString = new SimpleList<String>(5);
    SimpleList<Object> slObject = slString;
}
```

Typabweichung: Konvertierung von SimpleList<String> auf SimpleList<Object> nicht möglich
Drücken Sie 'F2', um weitere Informationen anzuzeigen.

Die oben formulierte Abstammungsvermutung ist also *falsch*.

Es ist möglich (aber *nicht* ratsam), ein Objekt der Klasse `SimpleList<String>` über eine Referenzvariable vom so genannten Rohtyp `SimpleList` (siehe Abschnitt 6.2.5) anzusprechen, z.B.:

```
public static void main(String[] args) {
   SimpleList<String> slString = new SimpleList<String>(5);
   SimpleList slObject = slString;
   slObject.add(13);
   System.out.println(slString.get(0).length());
}
```

Ein Aufruf dieser **main()**-Methode führt zu einer **ClassCastException**, weil das eingeschmuggelte **Integer**-Objekt keine **length()**-Methode beherrscht:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer
cannot be cast to java.lang.String at SimpleList.main(SimpleList.java:37)
```

Die Compiler bemerken das Risiko und warnen, z.B.:

```
SimpleList slObject = sls;
```

SimpleList ist ein unformatierter Typ. Verweise auf den generischen Typ SimpleList<E> sollten mit Parameterangabe erfolgen

3 Schnellreparaturen verfügbar:

- Add type arguments to 'SimpleList'
- Generische Typargumente ableiten...
- @SuppressWarnings 'unchecked' zu 'main()' hinzufügen

Drücken Sie zum Fokussieren auf 'F2'

### 6.2.4 Wildcard-Datentypen

#### *6.2.4.1 Ungebunden*

In Abschnitt 6.2.3 wurde erläutert, dass z.B. der parametrisierte Datentyp `SimpleList<String>` *keine* Spezialisierung des parametrisierten Typs `SimpleList<Object>` ist. Um bei einer Variablen oder Parameterdeklaration unter Verwendung einer generischen Klasse für einen Typformalparameter beliebige Konkretisierungen zu erlauben, wurde in Java 5 im Zusammenhang mit den generischen Typen eine Wildcard-Syntax eingeführt. Die folgendermaßen definierte Referenzvariable `slObject`

```
SimpleList<?> slObject;
```

kann auf `SimpleList<E>` - Konkretisierungen mit beliebigen Elementtyp zeigen, z.B.:

```
SimpleList<String> slString = new SimpleList<String>(5);
slString.add("Emma");
slObject = slString;
```

Hinsichtlich der Zuweisungskompatibilität kann man also den Wildcard-Typ `SimpleList<?>` wie eine Superklasse zu allen `SimpleList<E>` - Konkretisierungen verwenden.

#### *6.2.4.2 Gebunden*

Oft soll die Beliebigkeit bei der Wildcard-Konkretisierung analog zu den Typformalparametern durch ein Bindung eingeschränkt werden (engl.: *bounded wildcards*). Die folgendermaßen definierte Referenzvariable `slNumber`

```
SimpleList<? extends Number> slNumber;
```

kann auf `SimpleList<E>` - Konkretisierungen mit der Klasse **Number** oder einer beliebigen Spezialisierung als Elementtyp zeigen, z.B.:

```
SimpleList<Integer> slInteger = new SimpleList<Integer>(5);
slNumber = slInteger;
```

Neben der eben vorgestellten **extends**-Bindung (auch als *upper bound* bezeichnet) bietet Java noch die **super**-Bindung (auch als *lower bound* bezeichnet), wobei zur Wildcard-Konkretisierung eine bestimmte Klasse und ihre sämtlichen (auf verschiedene Ebenen angesiedelten) Basisklassen bis hinauf zur Urahnklasse **Object** zugelassen sind. Zur Erläuterung der **super**-Bindung soll die in Abschnitt 6.2.2 vorgestellte generische Klasse `SimpleSortedList<E>` aufgegriffen und verbessert werden:

```
public class SimpleSuperSortedList<E extends Comparable<E>> {
   . . .
}
```

Hier wird für den Elementtyp `E` das Interface **Comparable<E>** gefordert, d.h. eine Methode mit dem folgenden Kopf:

**public int compareTo(E** *vergl***)**

Wird z.B. aus einer Klasse namens `Figur`, welche die geforderte Methode besitzt, eine Klasse namens `Kreis` abgeleitet, dann sind alle `Kreis`-Objekte miteinander über die geerbte **compareTo()**-Methode vergleichbar. In der `Kreis`-Klasse ist eine Methode

```
public int compareTo(Figur fig)
```

verfügbar, aber *keine* Methode

```
public int compareTo(Kreis kr)
```

Daher wird `Kreis` nicht als `SimpleSortedList<E>` - Elementtyp akzeptiert. Um dies zu ermöglichen, ist nur eine kleine Änderung in der `SimpleSortedList<E>` - Definition erforderlich:

```
public class SuperSimpleSortedList<E extends Comparable<? super E>> {
   . . .
}
```

#### *6.2.4.3 Kein Einfügen von Container-Elementen via Wildcard-Referenz*

Bei folgender Vorgeschichte

```
SimpleList<? extends Number> slNumber;
SimpleList<Integer> slInteger = new SimpleList<Integer>(5);
slInteger.add(4711);
slNumber = slInteger;
```

lässt sich die `SimpleList<E>` - Methode `get()` per Wildcard-Referenz verwenden, sofern an der Aufrufstelle der Rückgabetyp **Number** akzeptabel ist, z.B.:

```
Number num = slNumber.get(0);
```

Demgegenüber gelingt das Einfügen neuer Elemente über die Methode `add()`, die den Typparameter der Klasse `SimpleList<E>` in ihrer Parameterliste verwendet, *nicht* per Wildcard-Referenz, z.B.:

```
slNumber.add(13);
```


Im Beispiel steht `slNumber` für eine `SimpleList<E>` - Konkretisierung mit der Klasse **Number** oder einer beliebigen Spezialisierung (z.B. **Byte**) als Elementtyp. Folglich kann der Compiler es z.B. nicht erlauben, ein **Integer**-Objekt (Autoboxing!) in den per `slNumber` ansprechbaren Container einzufügen.

### 6.2.5 Rohtyp und Typlöschung

Der Java-Compiler erzeugt für jede generische Klasse unabhängig von der Anzahl der im Quellcode vorhandenen Konkretisierungen ausschließlich den so genannten **Rohtyp**. Hier sind Typformalparameter durch den breitesten zulässigen Datentyp ersetzt. Bei unrestringierten Parametern ist diese *obere Schranke* (engl.: *upper bound*) der Urahntyp **Object**, bei restringierten Parametern ist sie entsprechend kleiner (z.B. **Interface**-Typ **Comparable** beim Beispiel in Abschnitt 6.2.2). Man spricht hier von **Typlöschung** (engl.: *type erasure*). Im Bytecode existieren also keine parametrisierten Typen, sondern nur Rohtypen. Dementsprechend resultiert bei der Übersetzung einer generischen Klasse genau *eine* **class**-Datei. Den Rohtyp zu `SimpleList<E>` kann man sich ungefähr so vorstellen:

```
public class SimpleList {
   private Object[] elements;
   private int initSize = 10;
   private int size;

   public SimpleList(int len) {
      if (len > 0) {
         elements = new Object[len];
         initSize = len;
      } else
         elements = new Object[initSize];
   }

   public SimpleList() {
         elements = new Object[initSize];
   }

   public void add(Object element) {
      if (size ==  elements.length) {
         Object[] temp = new Object[elements.length + initSize];
         for (int i = 0; i < elements.length; i++)
            temp[i] = elements[i];
         elements = temp;
      }
      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;}
}
```

Während die Entwickler typsichere und bequeme Konkretisierungen einer generischen Klasse verwenden, bleibt für die JRE alles beim Alten. Die damit fälligen expliziten Typanpassungen fügt der Compiler automatisch in den Bytecode ein.

Auf ihre Klassenzugehörigkeit befragt, nennen Objekte eines parametrisierten Typs stets den zugehörigen Rohtyp, z.B.:

| Quellcodefragment | Ausgabe |
|---|---|
| `public static void main(String[] args) {`<br>`  SimpleList<String> sls = new SimpleList<String>(3);`<br>`  System.out.println(sls.getClass());`<br>`}` | `class SimpleList` |

Die Typlöschung ist auch bei Verwendung des (im Kurs noch nicht vorgestellten) **instanceof**-Operators zu berücksichtigen, der die Zugehörigkeit eines Objekts zu einer bestimmten Klasse prüft, z.B.:

| Quellcodefragment | Ausgabe |
|---|---|
| `System.out.println(sls instanceof SimpleList);` | `true` |

Der Operator akzeptiert keine parametrisierten Typen, so dass z.B. die folgende Anweisung *nicht* übersetzt werden kann:

```
System.out.println(sls instanceof SimpleList<String>);
```

instanceof-Abgleich gegen Typ SimpleList<String> mit Parameterangabe nicht möglich. Verwenden Sie stattdessen seine unformatierte Form SimpleList, da die generischen Typinformationen während der Ausführung gelöscht werden

Drücken Sie 'F2', um weitere Informationen anzuzeigen.

Wie Sie aus Abschnitt 6.1 wissen, kann der Rohtyp verwendet werden durch Verzicht auf die Konkretisierung der Typparameter, wobei die Vorteile generischer Klassen wegfallen. Diese Praxis ist nur dann akzeptabel, wenn Kompatibilität mit älterer Software erforderlich ist.

Als Motive für die Typlöschung bei der Generizitätslösung in Java werden neben der Kompatibilität mit Software-Altlasten auch Einsparungen bei Rechenzeit und Speicherplatz genannt. Allerdings sind mit dieser Designentscheidung einige Einschränkungen verbunden, die nun diskutiert werden.

### 6.2.6 Schwächen der Typgenerizität in Java

Als Konkretisierung für einen Typformalparameter kommt nur ein *Referenztyp* in Frage, was dank Wrapper-Klassen und Auto(un)boxing keine Einschränkung darstellt. Allerdings ist bei Verwendung eines generischen Containers zur Verwaltung von zahlreichen Werten eines primitiven Typs durch die hohe Anzahl von Auto(Un)boxing-Operationen mit Leistungseinbußen zu rechnen.

Wie Sie aus Abschnitt 6.2.5 wissen, ist eine generische Klasse unabhängig von der Anzahl der im Quellcode vorhandenen Konkretisierungen (parametrisierten Typen) im Bytecode nur durch ihren Rohtyp vertreten. Damit verwenden alle parametrisierten Typen dieselben statischen Variablen und Methoden der Klasse. Folglich darf bei der Deklaration von statischen Feldern oder der Definition von statischen Methoden kein Typparameter verwendet werden.

Weil zu Laufzeit alle Typformalparameter durch ihre obere Schranke (z.B. **Object**) ersetzt sind, kann der Typ eines zu erzeugenden Objekts nicht über Typformalparameter festgelegt werden, was zu Lücken in der Typsicherheit führt. In der `SimpleList`-Definition (siehe Abschnitt 6.2.1) wird zur internen Verwaltung der Listenelemente ein **Object**-Array verwendet:

```
private Object[] elements;
private int initSize = 10;
  . . .
public SimpleList(int len) {
  if (len > 0) {
    elements = new Object[len];
    initSize = len;
  } else
    elements = new Object[initSize];
}
```

Während in der `SimpleList`-Definition die *Deklaration*

```
private E[] elements;
```

erlaubt wäre, weigert sich der Compiler, die folgende Zeile zu übersetzen:

```
elements = new E[len];
```

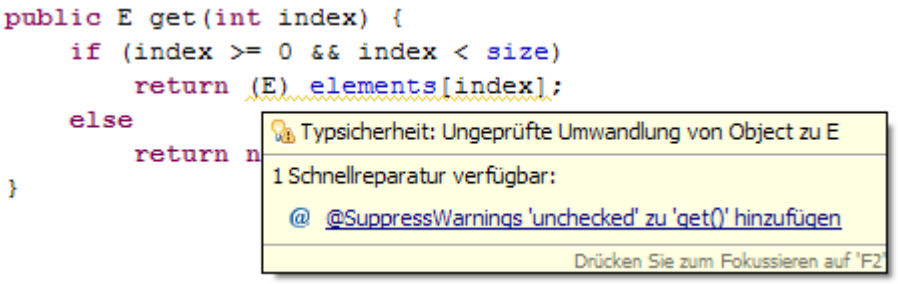
Der Elementtyp des Arrays kann *nicht* über den Typparameter bestimmt werden! In der `SimpleList`-Methode `get()`, die ihren Rückgabetyp per Typparameter definiert, ist daher eine explizite Typumwandlung erforderlich:

```
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 JDK-Compiler macht mit einer **unchecked**-Warnung darauf aufmerksam, dass er keine Kontrollmöglichkeit hat:

```
>javac SimpleList.java
Note: SimpleList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Auch der Eclipse-Compiler bemerkt die Konstellation und kann ebenfalls nur empfehlen, bei der betroffenen Methode die **unchecked**-Warnung abzuschalten:




Die bei `SimpleList` benutzte Lösung wird übrigens auch bei der API-Klasse **Vector** verwendet, z.B.:[35]

[35] Das Schlüsselwort **synchronized** wird später im Zusammenhang mit Threads erläutert. Es verhindert eine Änderung des Containers durch einen anderen Thread (Ausführungsfaden) des Programms während der Ausführung von **lastElement()**, weil diese Konstellation zu einem falschen Rückgabewert der Methode führen könnte.

```
protected Object[] elementData;
  . . .
this.elementData = new Object[initialCapacity];
  . . .
public synchronized E lastElement() {
  if (elementCount == 0) {
      throw new NoSuchElementException();
  }
  return (E)elementData[elementCount - 1];
}
```

Weil der Compiler die Gültigkeit der Typkonvertierung in der **return**-Anweisung nicht garantieren kann, erscheint obige Warnung natürlich auch beim Übersetzen der API-Klasse **Vector**. Hier zeigt sich eine Schwäche der in Java gewählten Realisierung generischer Klassen, die aber nur bei fehlerhaft konzipierten generischen Klassen zum Problem wird. Bei der *Verwendung* generischer Klasen überwacht der Java-Compiler die Typsicherheit. Beim *Klassendesign* ist der Programmierer für die Typsicherheit verantwortlich.

## 6.3 Generische Methoden

Wenn überladene Methoden analoge Operationen mit verschiedenen Datentypen ausführen, stellt *eine* generische Methode oft die bessere Lösung dar. Im folgenden Beispiel liefert eine statische und generische Methode das Maximum zweier Argumente, wobei der gemeinsame Datentyp der Argumente die Schnittstelle **Comparable<T>** erfüllen, also eine Methode **compareTo()** besitzen muss:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  static <T extends Comparable<T>> T max(T x, T y) {`<br>`   return x.compareTo(y) > 0 ? x : y;`<br>`  }`<br><br>`  public static void main(String[] args) {`<br>`   System.out.println("int-max:\t"+max(12, 4711));`<br>`   System.out.println("double-max:\t"+max(2.16, 47.11));`<br>`  }`<br>`}` | `int-max:  4711`<br>`double-max:  47.11` |

In der Definition einer generischen Methode befindet sich vor dem Rückgabetyp zwischen spitzen Klammern mindestens ein **Typformalparameter**. Mehrere Typparameter werden durch Kommata getrennt.

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 6.2.1 beschriebenen Klasse `SimpleList<E>`:

```
public void add(E element) {
  . . .
}
```

Beim Aufruft einer generischen Methode ermittelt der Compiler die Datentypen der per Referenzparameter übergebenen Objekte und stellt ggf. sicher, dass Typrestriktionen eingehalten werden. Werte mit primitivem Datentyp werden per Autoboxing in ein Objekt der zugehörigen Verpackungsklasse gesteckt.

Bei seiner Bytecode-Produktion erstellt der Compiler *eine* Methode und ersetzt dabei die Typparameter jeweils durch den breitesten erlaubten Typ (z.B. **Comparable**). Eine mehrfach konkretisierte

Methode landet also nur einfach im Bytecode. Die gelöschten Typkonkretisierungen (vgl. Abschnitt 6.2.5) 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 {`<br>`  static <T extends Comparable<T>> T max(T x, T y) {`<br>`  return x.compareTo(y) > 0 ? x : y;`<br>`  }`<br><br>`  static int max(int x, int y) {`<br>`  System.out.print("trad. ");`<br>`  return x > y ? x : y;`<br>`  }`<br><br>`  public static void main(String[] args) {`<br>`  System.out.println("int-max:\t"+max(12, 4711));`<br>`  System.out.println("double-max:\t"+max(2.16, 47.11));`<br>`  }`<br>`}` | `trad. int-max: 4711`<br>`double-max: 47.11` |

Der Compiler ermittelt zu einem konkreten Aufruf die am besten passende kompatible Methode und beschwert sich bei Zweifelsfällen.

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. Bei Arrays mit primitivem Elementtyp findet *kein* Autoboxing statt. Hier hilft nur die Erstellung von Überladungen für jeden relevanten primitiven Typ, was analog z.B. bei den **sort()**-Methoden der API-Klasse **java.util.Arrays** geschehen ist.

## *6.4 Java Collection Framework*

Die in diesem Abschnitt vorgestellten Interfaces (Listen von Methodenköpfen zur Funktionalitätsdefinition) und Klassen zur Verwaltung von Listen, Mengen oder (Schlüssel-Wert) - Tabellen mit Objekten stammen aus dem **Java Collection Framework**, dessen Weiterentwicklung seit der Einführung in Java 2 insbesondere von der seit Java 5 verfügbaren Generizität profitiert hat. In Abschnitt 6.1 haben Sie einen Eindruck davon erhalten, welchen Fortschritt die generische Klasse **Vector<E>** gegenüber der auf unsicheren **Object**-Referenzen und expliziter Typumwandlung basierenden Vorgängerlösung bei der häufig benötigten Verwaltung von Elementen *desselben* Typs darstellt.

An dieser Stelle im Kursverlauf die Kollektionen zu präsentieren, hat vor allem didaktische Gründe. Man erlebt dabei zahlreiche generische Typen mit hohem praktischem Nutzwert, 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. Ein Nachteil der relativ frühen Kollektionsbehandlung sind zahlreiche Vorgriffe auf das Kapitel 10 über (generische) Interfaces.

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**, ansonsten **false**. Manche Kollektionen verweigern z.B. die Aufnahme von Dubletten.
- **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**, ansonsten **false**. Per Wildcard-Typdeklaration (siehe Abschnitt 6.2.4) wird für die Aufnahmekandidaten der Elementtyp der im **addAll()**-Aufruf angesprochenen Kollektion oder eine Spezialisierung vorgeschrieben.
- **public void clear()**
  Entfernt alle Elemente aus der Kollektion
- **public boolean contains(Object** *object***)**
  Diese Methode informiert darüber, ob das fragliche Element in der Kollektion vorhanden ist.
- **public Iterator<E> iterator()**
  Diese Methode liefert ein Iterator-Objekt, das ein sequentielles Aufsuchen der Kollektionselemente unterstützt (siehe Abschnitt 6.4.1.4). Sie gehört zum Interface **Iterable<E>**, das vom Interface **Collection<E>** erweitert wird.
- **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.
- **public int size()**
  Liefert die Anzahl der Elemente in der Kollektion

### 6.4.1 Listen

Zur Realisation von Listen stehen neben der Klase **Vector<E>**, mit der Sie schon in Abschnitt 6.1 Bekanntschaft gemacht haben, noch die Klassen **ArrayList<E>** und **LinkedList<E>** zur Verfügung.

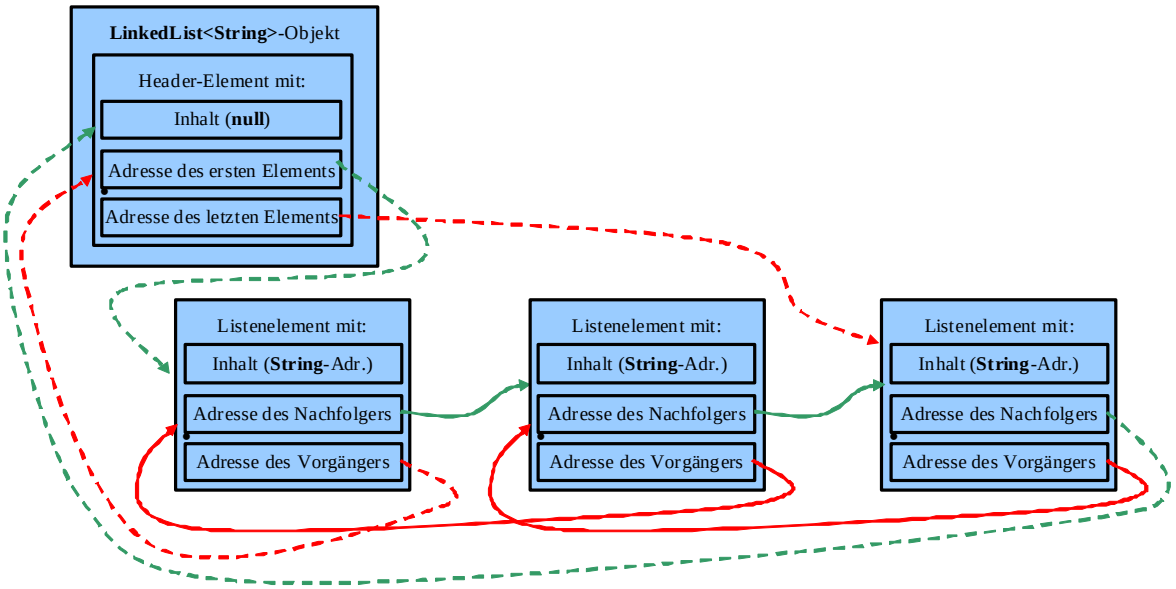
#### *6.4.1.1 Listenarchitekturen*

**Vector<E>** und **ArrayList<E>** arbeiten intern mit einem Array und bieten daher einen schnellen wahlfreien Zugriff auf vorhandene Elemente. 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 jedoch die neuen bzw. früheren rechten Nachbarn zeitaufwändig nach rechts bzw. links verschoben werden.

Ein wesentlicher Unterschied zwischen den beiden Klassen besteht darin, dass **Vector<E>** Threadsicher 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 Multi-Thread-Anwendung einsetzen, müssen Sie 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 Klasse **Collections** 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>());
```

Die Klasse **LinkedList<E>** arbeitet intern mit einer **doppelt verketteten Liste** bestehend aus selbständigen Objekten, die jeweils ihren Nachfolger und ihren Vorgänger kennen, z.B.:




Vorteile der Listenarchitektur:

- Beim Einfügen und Löschen von Elementen müssen keine anderen Elemente verschoben, sondern nur einige Adressen geändert werden.
- Die Länge der Liste ist zu keiner Zeit festgelegt.

Um ein Listenelement mit bestimmtem Indexwert aufzufinden, muss die Liste 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 Elementadressen im Speicher hintereinander liegen und direkt ermittelt werden können.

Insgesamt sind verkettete Listen besonders geeignet für Algorithmen, die …

- häufig Elemente einfügen oder entfernen und sich dabei nicht auf das Listenende beschränken,
- betroffene Elemente überwiegend sequentiell aufsuchen.

Zum sequentiellen Aufsuchen der Listenelemente muss bei der Klasse **LinkedList<E>** aus Performanzgründen an Stelle eines Index-Parameters unbedingt ein Iterator-Objekt verwendet werden (siehe Abschnitt 6.4.1.4).

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 oben Hinweis auf die **Collections**-Methode **synchronizedList()**).

### *6.4.1.2 Das generische Interface List<E>*

Die Klassen **Vector<E>**, **ArrayList<E>** und **LinkedList<E>** implementieren das von **Collection<E>** (siehe oben) abstammende generische Interface **List<E>** und bieten folglich u.a. die folgenden Instanzmethoden:

- **public boolean add(E** *element***)**
  Es wird ein neues Element am Ende der Liste angehängt.
- **public void add(int** *index***, E** *element***)**
  Es wird ein neues Element an der gewünschten Indexposition eingefügt.

- **public E get(int** *index***)**
  Das Element mit dem gewünschten Index wird geliefert (wahlfreier Zugriff).
- **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.

Bei der Klasse **LinkedList<E>** machen es die Methoden mit Index-Parameter (z.B. **get()**, **remove()**) erforderlich, sich vom Startpunkt (Index ≤ halbe Länge) oder Endpunkt (Index > halbe Länge) ausgehend bis zur gesuchten Position vorzuarbeiten, wobei diese aufwändige Prozedur bei jedem Methodenaufruf neu startet. Genügt ein sequentieller Zugriff, sollte bei einer verketteten Liste unbedingt ein **Iterator**-Objekt verwendet werden, um die Elemente nacheinander zu besuchen (siehe Abschnitt 6.4.1.4). Wird ein wahlfreier Zugriff tatsächlich benötigt, ist eine Array-basierte Klasse zu bevorzugen.

Man kann je nach Einsatzschwerpunkt und benötigter Thread-Sicherheit zwischen den drei Klassen wählen und sogar unproblematisch wechseln, sofern man sich auf die gemeinsamen, durch das Interface **List<E>** vorgeschriebenen Methoden beschränkt.

#### *6.4.1.3 Leistungsunterschiede und Einsatzempfehlungen*

Im folgenden Testprogramm

```
import java.util.*;
class Listen {
  static final int ANZ = 20000;

  static void testList(List<String> lis_) {
    List<String> liste = lis_;
    StringBuffer sb = new StringBuffer();
    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 Fuellen:\t" +
        (System.currentTimeMillis()-start));

    // Abrufen
    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
    start = System.currentTimeMillis();
    for (int i = 0; i < ANZ; i++)
      liste.add(0, "neu");
    System.out.println(" Zeit zum Einfuegen:\t" +
        (System.currentTimeMillis()-start));
```

```
        // Löschen
        start = System.currentTimeMillis();
        for (int i = 0; i < 2*ANZ; i++)
          liste.remove(0);
        System.out.println(" Zeit zum Loeschen:\t" +
             (System.currentTimeMillis()-start) + "\n");
    }

    public static void main(String[] args) {
      testList(new Vector<String>());
      testList(new ArrayList<String>());
      testList(new LinkedList<String>());
    }
}
```

mit den Aufgaben

- eine Liste mit 20000 Zeichenketten füllen
- aus der Liste 20000 Elemente mit zufällig bestimmter Indexposition abrufen
- 20000 neue Elemente am Anfang der Liste einfügen
- 40000 Elemente einzeln am Listenanfang löschen

zeigen die drei Klassen **Vector<String>**, **ArrayList<String>** und **LinkedList<String>** folgende Leistungen:[36]

```
Kollektionsklasse:      class java.util.Vector
 Zeit zum Fuellen:      47
 Zeit zum Abrufen:      0
 Zeit zum Einfuegen:    984
 Zeit zum Loeschen:     1391

Kollektionsklasse:      class java.util.ArrayList
 Zeit zum Fuellen:      46
 Zeit zum Abrufen:      0
 Zeit zum Einfuegen:    1047
 Zeit zum Loeschen:     1422

Kollektionsklasse:      class java.util.LinkedList
 Zeit zum Fuellen:      31
 Zeit zum Abrufen:      2672
 Zeit zum Einfuegen:    0
 Zeit zum Loeschen:     0
```

Wir beobachten:

- Das Befüllen verläuft bei allen Klassen recht flott, wobei die Thread-sichere Klasse **Vector<String>** etwas 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 Listenarchitektur 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 (ein Interface) 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()**). Diese Programmiertechnik fördert die Flexibilität und ist so emp-

[36] Die Zeiten stammen von einem PC unter Windows XP mit Intel Pentium 4 - CPU (3 GHz).

fehlenswert, dass wir einen Vorgriff auf das Interface-Kapitel in Kauf nehmen (siehe Abschnitt 10.3). Im Beispiel hätten wir dieselbe Eleganz auch mit einem Vorgriff auf das Kapitel 8 über Vererbung erreichen können, weil die Klassen **Vector<E>**, **ArrayList<E>** und **LinkedList<E>** von der gemeinsamen Basisklasse **AbstractList<E>** abstammen, welche ebenfalls das Interface **List<E>** implementiert und somit die benötigten Methoden beherrscht.

#### *6.4.1.4 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 Listenelement, 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 Listenelement 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 **NoSuchElementException** -Ausnahme (zur Ausnahmebehandlung siehe Kapitel 9).

- **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.

Das folgende Programm demonstriert die Erstellung und 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, sind aber bei den Klassen zur Verwaltung von Mengen verwendbar (vgl. Abschnitt 6.4.2).

Dank der **for**-Schleife für Kollektionen (vgl. Abschnitt 3.7.3.2) ist der Iterator-Einsatz oft ohne nennenswerten Aufwand zu realisieren.

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 weiteres 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.
- **public void set(E *element*)**
  Das zuletzt von **next()** oder **previous()** gelieferte Element wird durch das Parameterobjekt ersetzt.

Über die Methode **listIterator()** der Klasse **LinkedList<E>** erhält man ein Objekt aus einer Klasse, welche das Interface **ListIterator<E>** implementiert.

Zu einer Liste dürfen *mehrere lesende* Iteratoren existieren und operieren. Bei einem schreibenden Iterator ist hingegen Exklusivität Pflicht. Stellt ein Iterator bei einem Methodenaufruf (z.B. **next()**) fest, dass die Liste „hinter seinem Rücken“ verändert worden ist (z.B. über einen anderen Iterator), dann schlägt er Alarm, z.B.:

```
Exception in thread "main" java.util.ConcurrentModificationException
  at java.util.LinkedList$ListItr.checkForComodification(LinkedList.java:761)
  at java.util.LinkedList$ListItr.next(LinkedList.java:696)
  at Prog.main(Prog.java:10)
```

#### *6.4.1.5 Stapel*

Wer einen leistungsfähigen Stapelspeicher ohne nennenswerten eigenen Aufwand einsetzen möchte, kann die API-Klasse **Stack<E>** verwenden. Sie erweitert die Klasse **Vector<V>**, wie die JDK-Dokumentation zeigt:

**java.util**
## Class Stack<E>

```
java.lang.Object
  └java.util.AbstractCollection<E>
      └java.util.AbstractList<E>
          └java.util.Vector<E>
              └java.util.Stack<E>
```

Im Vergleich zur Klasse **Vector<E>** kommen u.a. die folgenden Methoden hinzu:

- **public E push(E *element*)**
  Es wird ein neues Element auf den Stapel gelegt.
- **public E pop()**
  Diese Methode hebt das oberste Element vom Stapel ab und liefert seine Adresse an den Aufrufer.
- **public E peek()**
  Diese Methode liefert ebenfalls die Adresse des obersten Elements als Rückgabewert, belässt es aber auf dem Stapel.

### 6.4.2 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 die generischen Klassen **HashSet<E>** und **TreeSet<E>**. Sie 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.
- **public boolean contains(Object** *object***)**
  Diese Methode informiert darüber, ob das fragliche Element in der Kollektion vorhanden ist und arbeitete bei den Klassen **HashSet<E>** und **TreeSet<E>** erheblich flotter als bei den Klassen zur Listenverwaltung (siehe Abschnitt 6.4.1).
- **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.

Einen Indexzugriff auf ihre Elemente bieten die Kollektionsklassen zur Mengenverwaltung nicht, Iteratoren (vgl. Abschnitt 6.4.1.4) sind jedoch verfügbar.

Aus Performanzgründen sind die Klassen **HashSet<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 20000 **String**-Objekten füllen
- für 20000 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:[37]

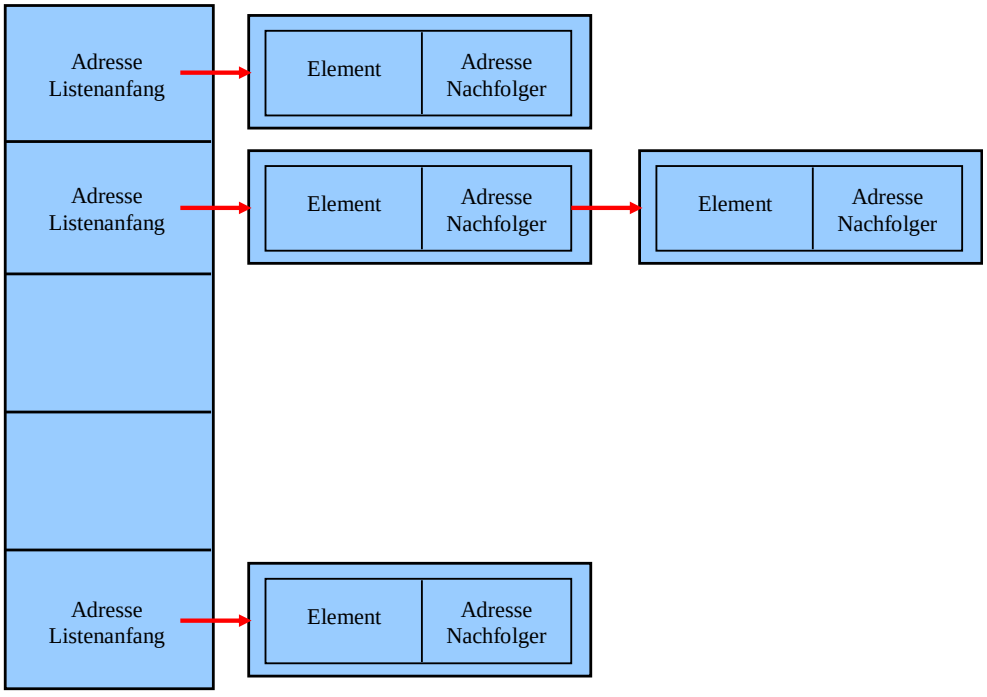
[37] Die Zeiten stammen von einem PC unter Windows XP mit Intel Pentium 4 - CPU (3 GHz).

```
Kollektionsklasse:                 class java.util.ArrayList
 Zeit zum Fuellen:                 47
 Zeit fuer die Existenzpruefungen: 8735

Kollektionsklasse:                 class java.util.LinkedList
 Zeit zum Fuellen:                 31
 Zeit fuer die Existenzpruefungen: 11297

Kollektionsklasse:                 class java.util.HashSet
 Zeit zum Fuellen:                 47
 Zeit fuer die Existenzpruefungen: 47

Kollektionsklasse:                 class java.util.TreeSet
 Zeit zum Fuellen:                 46
 Zeit fuer die Existenzpruefungen: 63
```

Die Klassen **HashSet<E>** und **TreeSet<E>** sind also nützlich, wenn Mengenzugehörigkeitsprüfungen in großer Zahl anfallen. Außerdem bieten sie bequeme Lösungen für Aufgaben aus dem Bereich der Mengenlehre (z.B. Durchschnitt, Vereinigung oder Differenz von zwei Mengen bilden).

#### *6.4.2.1 Hashtabellen*

Benötigt ein Algorithmus zahlreiche Existenzprüfungen, 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 Implementation der bereits in der Urahnklasse **Object** definierten **hashCode()**-Instanzmethode

**public int hashCode()**

über den Array-Index der zu verwendenden Liste.

Wesentliche Anforderungen an die **hashCode()**-Methode:

- 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.
- Es ist nicht erforderlich, aber aus Performanzgründen wünschenswert, dass sich die **hashCode()**-Rückgabewerte für zwei im Sinne der **equals()**-Methode verschiedene Objekte unterscheiden. Bei der **hashCode()**-Implementation der Klasse **Object** ist dies gewährleistet, weil die Speicheradressen der Objekte wesentlich verwendet werden.
- Die **hashCode()**-Rückgabewerte sollten möglichst gleichmäßig über den erlaubten Wertebereich verteilt sein.
- Überschreibt eine Klasse die **Object**-Methode **equals()**, ist in der Regel auch eine **hashCode()**-Überschreibung erforderlich, um die zweite Bedingung sicher zu stellen.

Aus dem Hashcode eines Objekts wird der Array-Index per Modulo-Operation ermittelt.[38]

Um für ein Objekt 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 die meisten Listen möglichst kurz bleiben, sollte ihre Zahl ungefähr das 1,5 - fache der Elementzahl im **HashSet<E>** - Objekt 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).

#### *6.4.2.2 Balancierte Binärbäume*

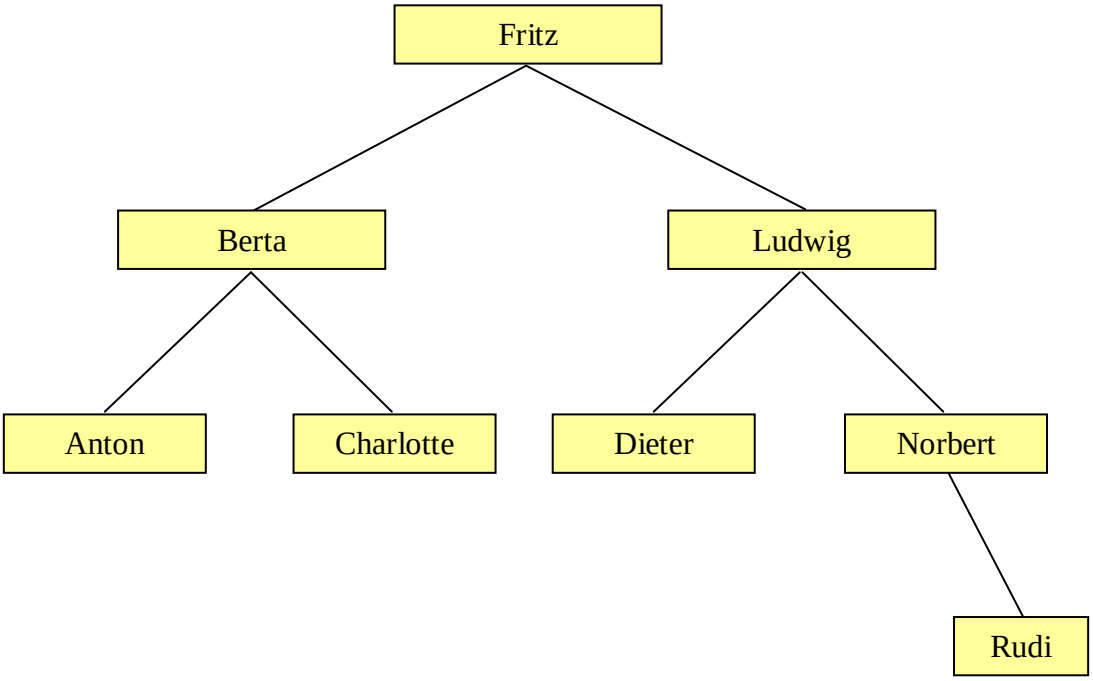
Existiert über den Elementen einer Menge eine **vollständige Ordnung** (z.B. Zeichenketten mit der lexikografischen Ordnung), kann man über einen Binärbaum die Elemente im sortierten Zustand halten, ohne den Aufwand (bei den zentralen Methoden **add()**, **contains()** und **remove()**) im Vergleich zur Hashtabelle wesentlich steigern zu müssen. Das macht sich die API-Klasse **TreeSet<E>** zu Nutze, die über das Interface **Set<E>** hinaus auch das Interface **SortedSet<E>** für vollständig geordnete Mengen erfüllt.

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:

---

[38] Im API-Quellcode wird aus Performanzgründen die Modulo-Operation äquivalent über die bitweise UND-Operation (siehe Abschnitt 3.5.6) realisiert:

```
static int indexFor(int h, int length) {
 return h & (length-1);
}
```




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. Die bei der Klasse **TreeSet<E>** verwendete **Rot-Schwarz** -Architektur stellt sicher, dass der längste Pfad höchstens doppelt so lang ist wie der kürzeste.

Die benötigte Ordnung wird in der Regel über die **compareTo()**-Methode des Elementtyps realisiert. Wird eine alternative Ordnung benötigt, über gibt man dem **TreeSet<E>** - Konstruktor ein Objekt, das die Schnittstelle **Comparator<E>** erfüllt und folglich für den Typ **E** geeignete Vergleichsmethoden bietet (**compare()** und **equals()**).

### 6.4.3 Abbildungen

Hier geht es um Klassen zur Verwaltung einer Menge von Schlüssel-Wert - Paaren, die das Interface **Map<K,V>** erfüllen und daher u.a. die folgenden Instanzmethoden beherrschen:

- **public V put(K** *key*, **V** *value***)**
  Wenn der Schlüssel *key* noch nicht existiert, wird ein neues Schlüssel-Wert - Paar angelegt. Anderenfalls wird der alte Wert überschrieben.
- **public V get(Object** *key***)**
  Liefert den zu einem Schlüssel gehörigen Wert
- **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 boolean containsKey(Object** *key***)**
  Die Methode liefert **true** zurück, wenn der angegebene Schlüssel in der Tabelle vorhanden ist, sonst **false**.
- **public Set<K> keySet()**
  Diese Methode liefert ein Objekt, das die Schnittstelle **Set<K>** - erfüllt (vgl. Abschnitt 6.4.2) und als *Sicht* (engl.: *View*) mit der Menge aller Schlüssel aus der angesprochenen

Abbildung operiert. Man löscht also z.B. mit der **Set<K>** - Methode **clear()** sämtliche Schlüssel, d.h. sämtliche Elemente der Abbildung.

Über einen Mengenverwaltungscontainer (z.B. aus der Klasse **HashMap<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. Über die im aktuellen Abschnitt zu beschreibenden 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. Aus dem 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 hier tatsächlich eine Abbildung im mathematischen Sinn (von einer Teilmenge der Buchstaben in die Menge der natürlichen Zahlen) geleistet wird.

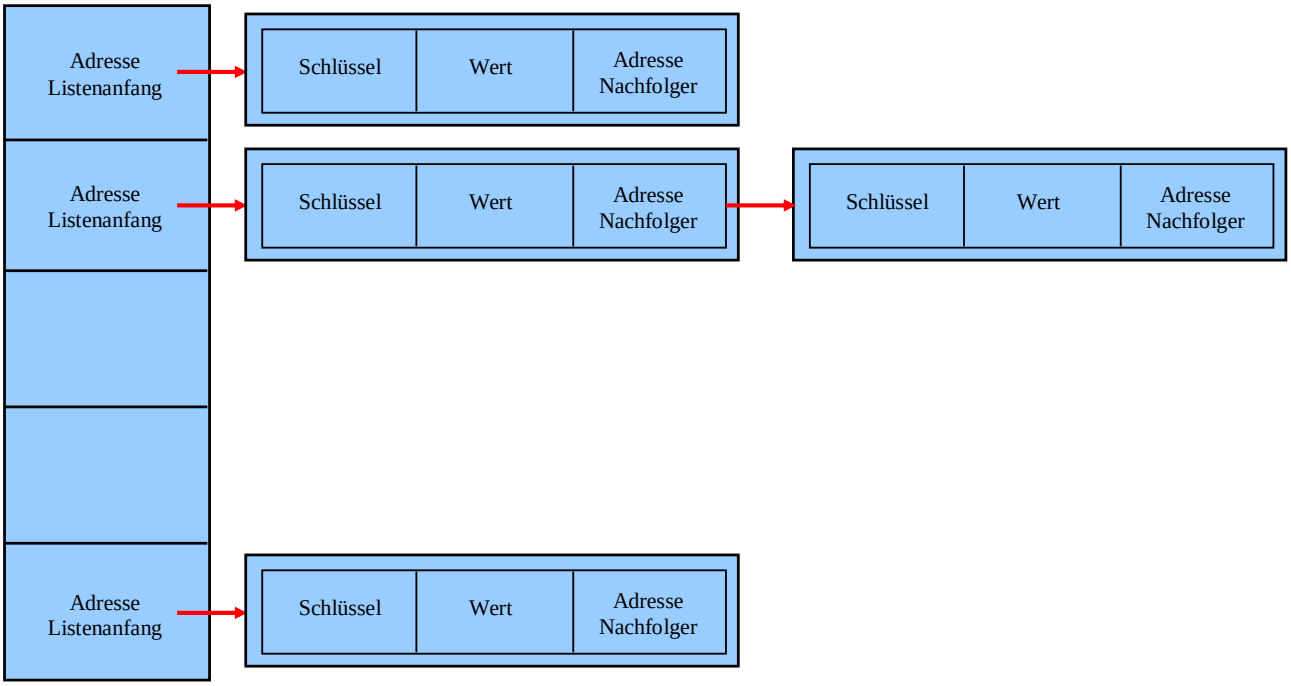
Die Paare aus einem Schlüssel vom Typ **Character** und einem Wert vom Typ `Mint` (eine selbst entworfene **int**-Hüllenklasse, vgl. Übungsaufgabe in Abschnitt 5.5) liefert die folgende Methode `countLetters()` als Elemente eines **HashMap<Character,**`Mint`**>** - Objekts:

```
public static Map<Character, Mint> countLetters(String text) {
  HashMap<Character,Mint> fred = new HashMap<Character,Mint>();
  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 6.4.2.1 beschriebene Klasse **HashSet<E>** arbeitet auch die Klasse **HashMap<K,V>** mit einer Hashtabelle, d.h. einem Array aus einfach verketteten Listen. Ein Blick in den API-Quellcode zeigt sogar, dass die Klasse **HashSet<E>** intern ein **HashMap<E,V>** - Objekt 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>();
}
```

Ein **HashMap<K,V>** - Objekt kann so skizziert werden:




Wie zur Mengenverwaltungsklasse **HashSet<E>** gibt es auch zur Abbildungsverwaltungsklasse **HashMap<K,V>** für vollständig geordnete Schlüsseltypen eine Alternative mit einem balancierten Binärbaum als Designprinzip. Ersetzt man im obigen Beispiel das **HashMap<Character,**`Mint`**>** - Objekt durch ein **TreeMap<Character,**`Mint`**>** - Objekt,

```
TreeMap<Character,Mint> fred = new TreeMap<Character,Mint>();
```

dann sind die Elemente 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 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>());
```

## *6.5 Übungsaufgaben zu Kapitel 6*

1) Erstellen Sie zu der in Abschnitt 6.3 vorgestellten generischen Methode `max()` eine Überladung, die das maximale Element zu einer beliebig langen Serie von Argumenten zurück gibt. Beim Typ des Serienparameters soll nur vorausgesetzt werden, dass er das Interface **Comparable<T>** erfüllt.

2) 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

Haeufigkeiten Merkmal 0:
    Wert       N
    1,00       2
    2,00       2
    3,00       1

Haeufigkeiten Merkmal 1:
    Wert       N
    1,00       2
    2,00       3

Haeufigkeiten Merkmal 2:
    Wert       N
    4,00       2
    5,00       2
    6,00       1
```

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 6.4.2) 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
```

# 7 Pakete

Das Java-API und auch jede größere Einzelanwendung enthält sehr viele Klassen, und schon zur Vermeidung von Namenskonflikten haben die Java-Designer so genannte *Pakete* (engl.: *packages*) eingeführt, um die Menge der Klassen zu strukturieren. Im Java-6-API gibt es ca. 200 Pakete mit zusammengehörigen Klassen. Neben Klassen können sie auch *Schnittstellen* (*Interfaces*) enthalten, mit denen wir uns noch beschäftigen werden.

Pakete erfüllen in Java viele wichtige Aufgaben:

- **Große Projekte strukturieren**
  Wenn sehr viele Klassen vorhanden sind, kann man mit Paketen Ordnung schaffen. Alle **class**-Dateien eines Pakets befinden sich in einem Dateiverzeichnis, dessen Name mit dem Paketnamen übereinstimmt.

  Es ist auch ein *hierarchischer* Aufbau über *Unterpakete* möglich, wobei die Paketstruktur auf einen Dateiverzeichnisbaum abgebildet wird. Im Namen eines konkreten (Unter-)Pakets folgen dann die Verzeichnisnamen aus dem zugehörigen Pfad durch Punkte getrennt aufeinander, z.B.:

  **java.util.zip**

  Vor allem bei der Weitergabe von Programmen ist es nützlich, eine komplette Paketstruktur in eine **Java-Archivdatei** (mit Extension **.jar**) zu verpacken (siehe Abschnitt 7.4).
- **Namenskonflikte vermeiden**
  Jedes Paket bildet einen eigenen Namensraum. Identische Bezeichner stellen also kein Problem dar, solange sie sich in verschiedenen Paketen befinden.
- **Zugriffskontrolle steuern**
  Per Voreinstellung ist eine Klasse nur innerhalb des eigenen Paketes sichtbar. Damit sie auch von Klassen aus fremden Paketen genutzt werden kann, muss in der Klassendefinition der Zugriffsmodifikator **public** gesetzt werden. In Abschnitt 7.3 wird die Rolle der Pakete bei der Zugriffsverwaltung genauer erläutert.

Bei der Paketierung handelt es sich aber nicht um eine *Option* für große Projekte, sondern um ein universelles Prinzip: Jede Java-Klasse gehört zu einem Paket. Wird eine Klasse keinem Paket explizit zugeordnet, gehört sie zusammen mit allen anderen ebenfalls nicht zugeordneten Klassen zum (namenlosen) **Standardpaket** (engl. *default package*). Diese Situation war bei all unseren bisherigen Anwendungen gegeben und aufgrund der geringen Komplexität dieser Projekte auch angemessen.

Im Java-Quellcode müssen fremde Klassen i. A. über ein durch Punkt getrenntes Paar aus Paketnamen und Klassennamen angesprochen werden, wie Sie es schon bei etlichen Beispielen kennen gelernt haben. Bei manchen Klassen ist aber *kein* Paketname erforderlich:

- Klassen aus **demselben** Paket
  Bei unseren bisherigen Beispielprogrammen befanden sich alle Klassen im Standardpaket, so dass kein Paketname erforderlich war. Im speziellen Fall des Standardpakets existiert auch gar kein Name.
- Klassen aus **importierten** Paketen
  Importiert man ein Paket in eine Quellcodedatei (siehe unten), können seine Klassen ohne Paketnamen angesprochen werden. Das Paket **java.lang** mit besonders fundamentalen Klassen (z.B. **System**, **String**, **Math**) wird bei jeder Anwendung *automatisch* importiert.

In Abschnitt 7.2.2 folgen noch weitere Details zur Ansprache von Klassen(mitgliedern).

## 7.1 Pakete erstellen

Vor der *Verwendung* von Paketen behandeln wir deren *Produktion*, weil dabei die technischen Details gut veranschaulicht werden können.

### 7.1.1 package-Anweisung und Paketordner

Wir erstellen zunächst ein einfaches Paket namens `demopack` mit den Klassen `A`, `B` und `C`. An den Anfang jeder einzubeziehenden Quellcodedatei setzt man die **package**-Anweisung mit dem Paketnamen, 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);
  }
}
```

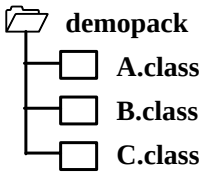
Vor der **package**-Anweisung dürfen höchstens Kommentar- oder Leerzeilen stehen.

Sind in einer Quellcodedatei *mehrere* Klassendefinitionen vorhanden, was in Java eher unüblich ist, so werden *alle* Klassen dem Paket zugeordnet.

Die Klassen eines Pakets können von Klassen aus fremden Paketen nur dann verwendet werden, wenn sie als **public** definiert sind. Pro Quellcodedatei ist nur eine **public**-Klasse erlaubt. Zusätzlich müssen auch Methoden und Felder explizit per **public**-Modifikator für fremde Pakete freigegeben werden. Steht z.B. kein **public**-Konstruktor zur Verfügung, können fremde Pakete eine Klasse zwar „sehen“, aber keine Objekte dieses Typs erzeugen. Mit den Zugriffsrechten für Klassen, Methoden und Felder werden wir uns in Abschnitt 7.3 ausführlich beschäftigen.

Die beim Übersetzen der zu einem Paket gehörigen Quellcodedateien entstehenden **class**-Dateien gehören in ein gemeinsames Dateiverzeichnis, dessen Name mit dem Paketnamen identisch ist. 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.
- Die drei Bytecodedateien **A.class**, **B.class** und **C.class** befinden sich in einem Ordner namens **demopack**:



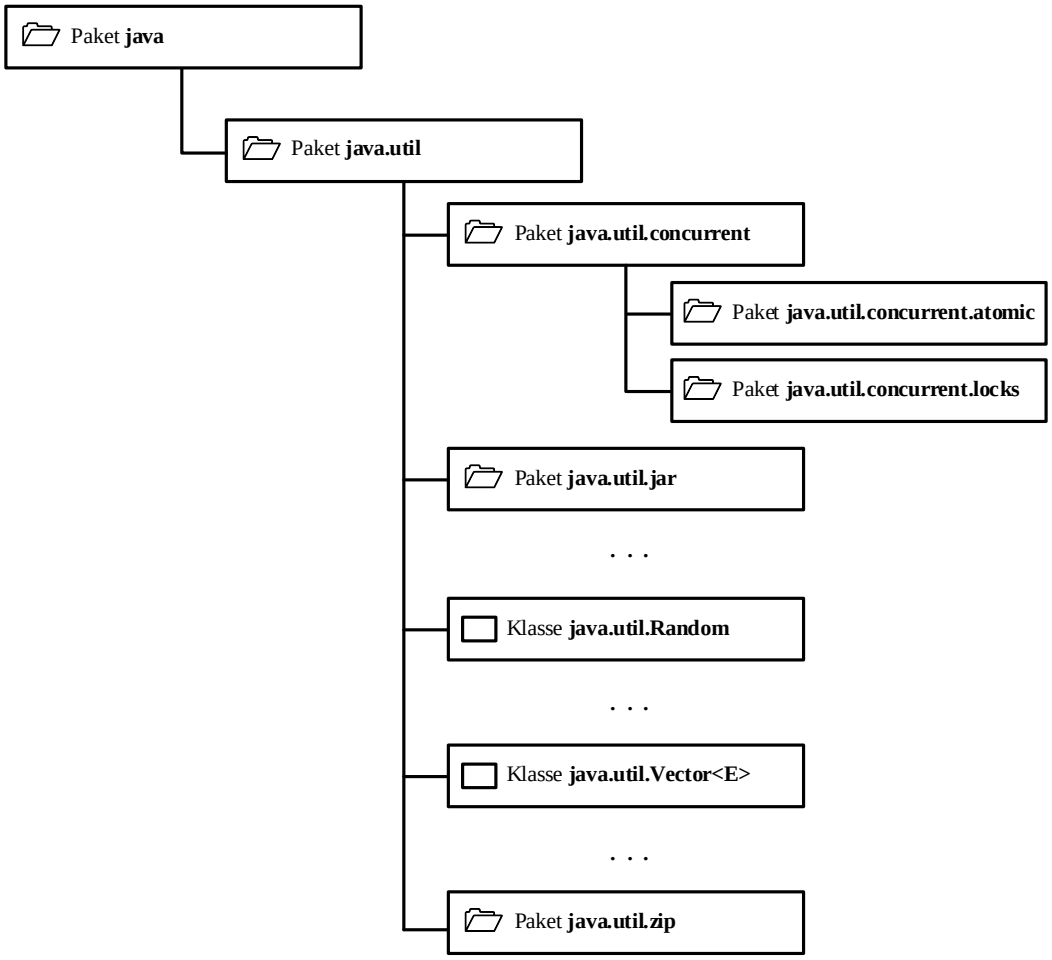

In Abhängigkeit von der verwendeten Java-Entwicklungsumgebung geschieht das Erstellen des Paketordners und das Einsortieren der Bytecode-Dateien eventuell automatisch (siehe Abschnitt 7.1.3 für Eclipse 3.x).

Für die Weitergabe von Programmen ist es oft sinnvoll, den Paketordner mit dem JDK-Werkzeug **jar.exe** in eine Java-Archivdatei zu verpacken (siehe Abschnitt 7.4).

Ohne **package**-Definition am Beginn der Quellcodedatei gehören die resultierenden Klassen zusammen mit allen anderen ebenfalls nicht zugeordneten Klassen zum (unbenannten) **Standardpaket** (engl. *default package*).

### 7.1.2 Unterpakete

Ein Paket kann hierarchisch in **Unterpakete** aufgeteilt werden, was bei den API-Paketen in der Regel geschehen ist, z.B.:




Auf jeder Stufe der Pakethierarchie können sowohl Klassen als auch Unterpakete enthalten sein. So enthält z.B. das Paket **java.util** u.a.

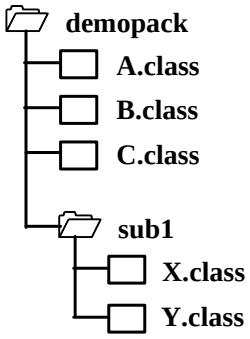
- die Klassen **Random**, **Vector<E>**, ...
- die Unterpakete **concurrent**, **jar**, **zip**, ...

Soll eine Klasse einem Unterpaket zugeordnet werden, muss in der **package**-Anweisung 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 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);
    }
}
```

Die **class**-Dateien müssen in einem zur Pakethierarchie analog aufgebauten Dateiverzeichnisbaum abgelegt werden, der in unserem Beispiel folgendermaßen auszusehen hat:

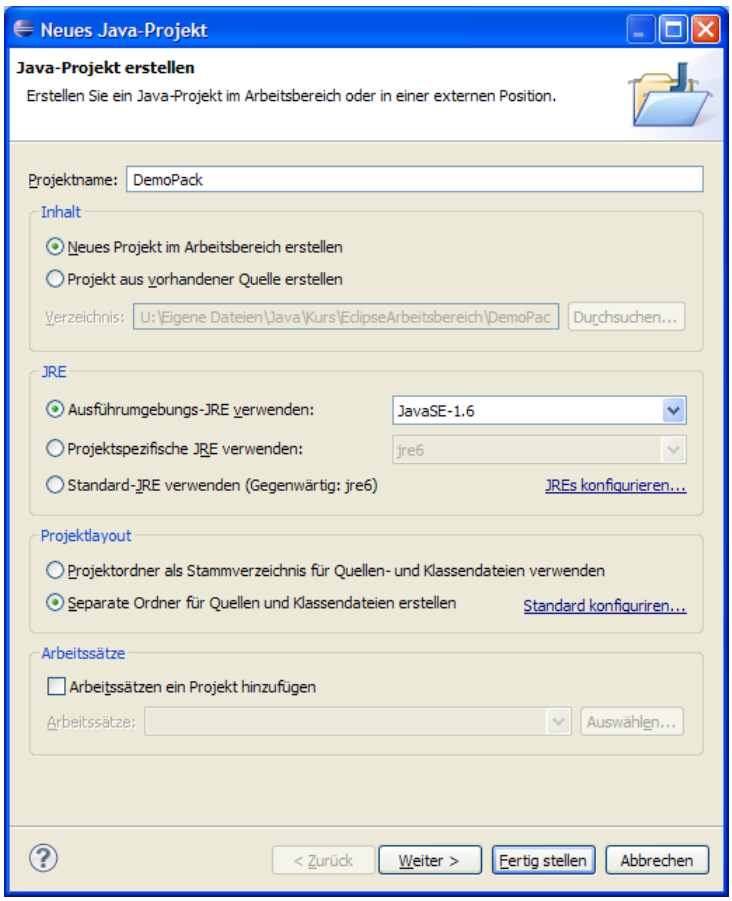



### 7.1.3 Paketunterstützung in Eclipse 3.x

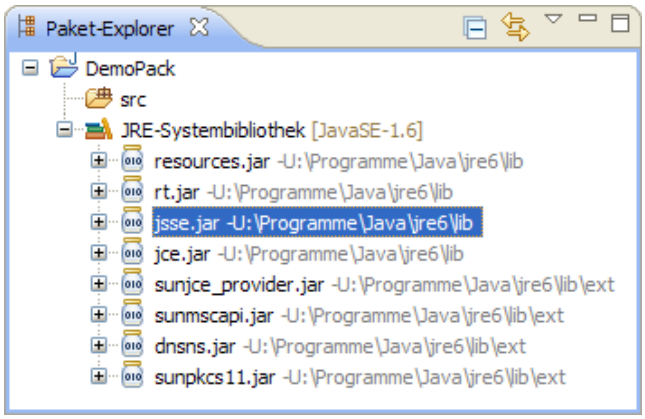
Wir starten in Eclipse über

**Datei > Neu > Java-Projekt**

ein neues Java-Projekt mit dem Namen `DemoPack`:



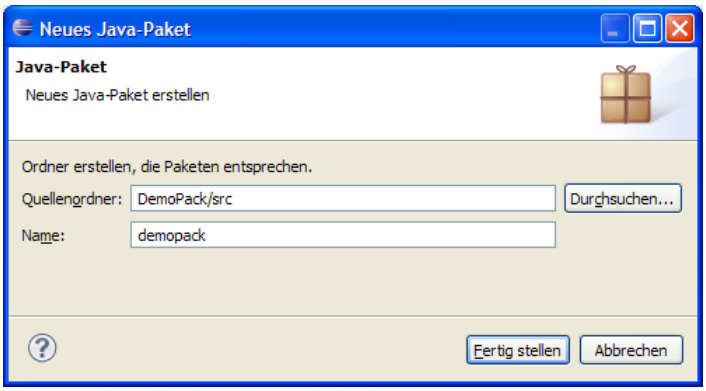

Zunächst zeigt der Paket-Explorer zum neuen Projekt nur die API-Klassenbibliothek an:



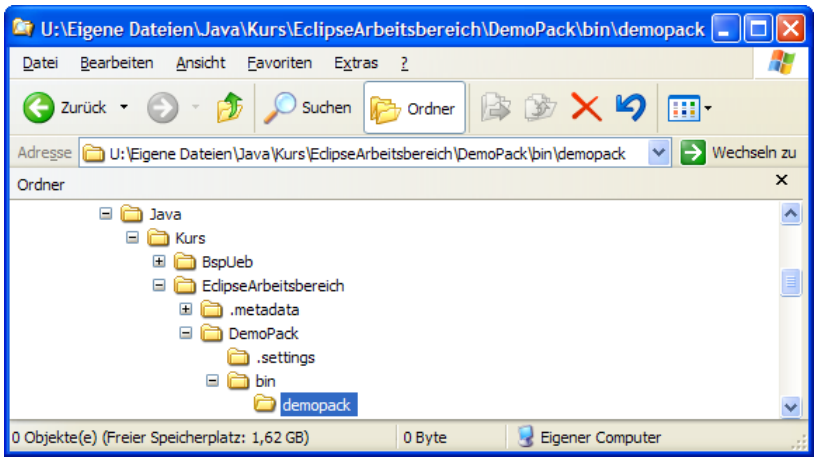

Aus dem Kontextmenü zum Projekteintrag im Paket-Explorer wählen wir die Option

**Neu > Paket**

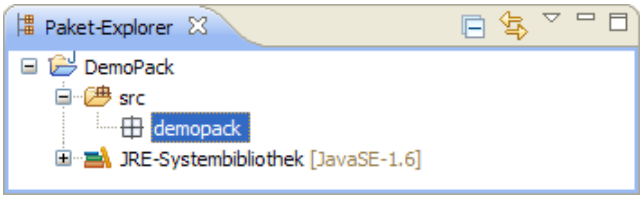
und geben in folgender Dialogbox den gewünschten Namen für das neue Paket an:



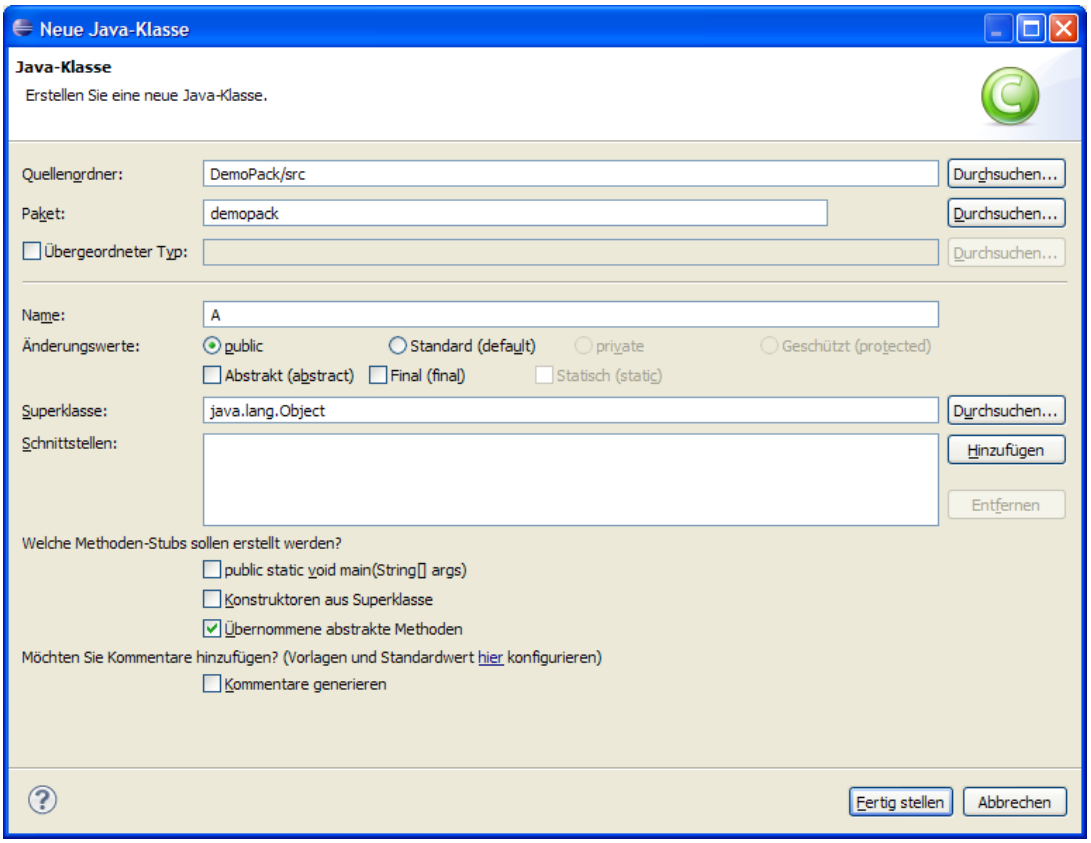

Anschließend erzeugt Eclipse den Ordner **demopack** im **bin**-Unterordner des Projekts



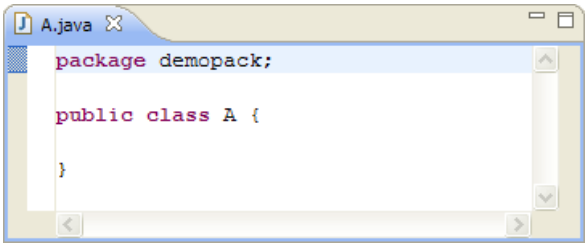

und zeigt ihn im Paket-Explorer an:



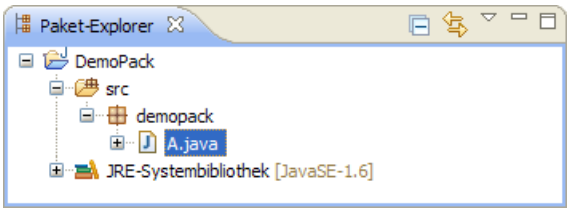

Nun legen wir im Paket `demopack` die Klasse `A` an, z.B. über den Eintrag **Neu > Klasse** im Kontextmenü zum Paket:




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 7.1.1) und legen analog auch die Klassen `B` und `C` im Paket `demopack` an:

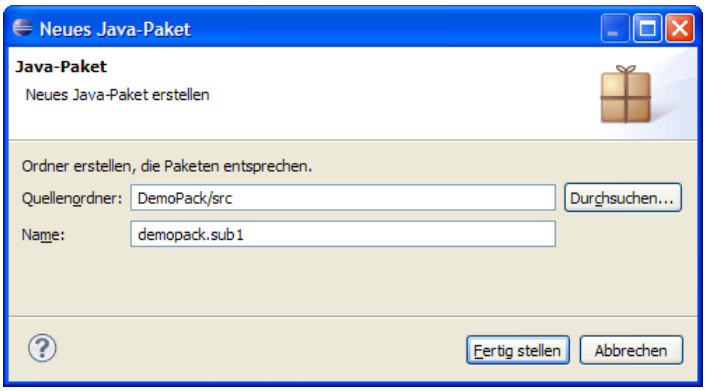
```
package demopack;

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);
  }
}
```

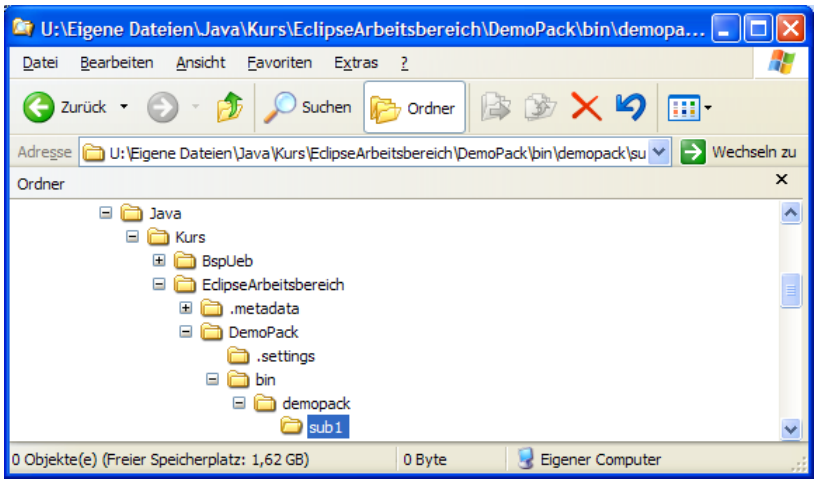
Um das Unterpaket `sub1` zu erzeugen, wählen wir im Paket-Explorer das Item

**Neu > Paket**

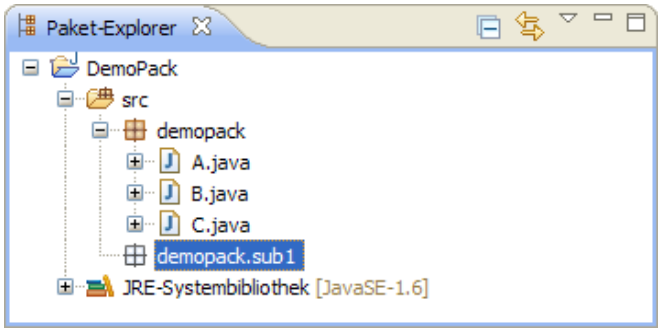
aus dem Kontextmenü zum Paket `demopack` und geben in folgender Dialogbox den gewünschten Namen für das Unterpaket an:



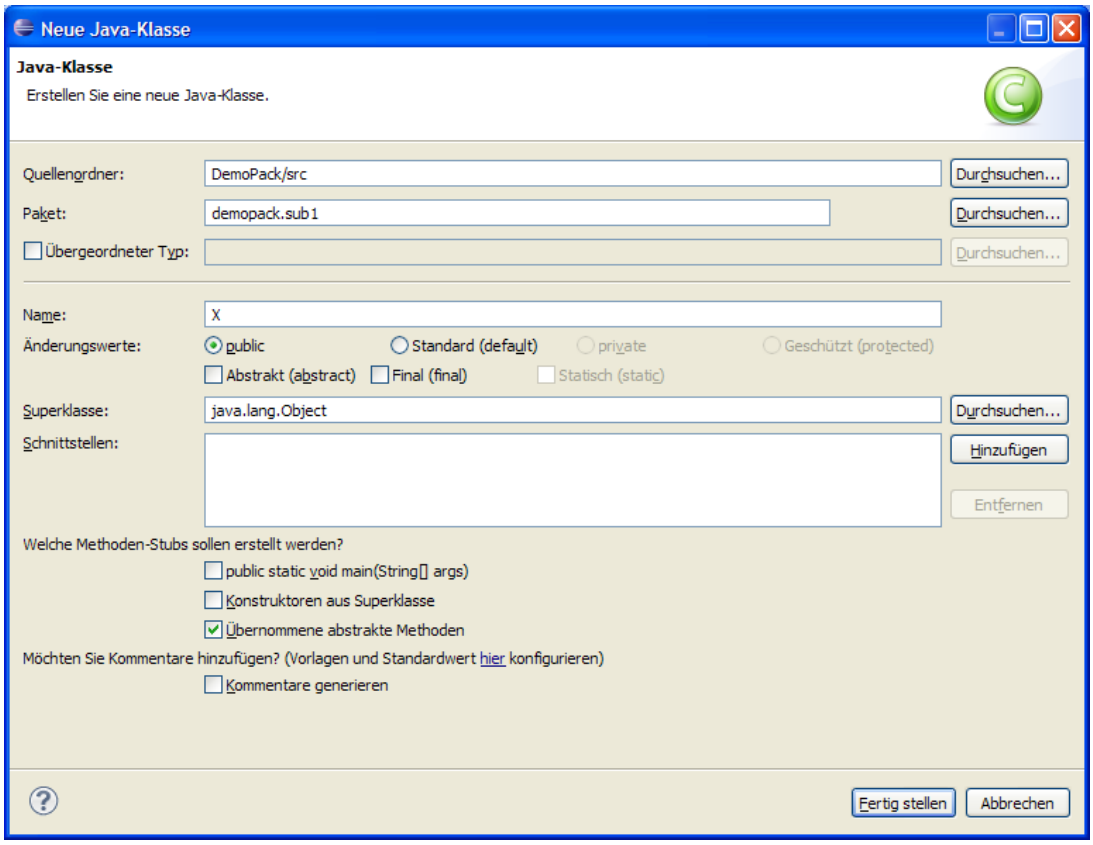

Anschließend erzeugt Eclipse den Ordner **demopack\sub1** im **bin**-Unterordner des Projekts



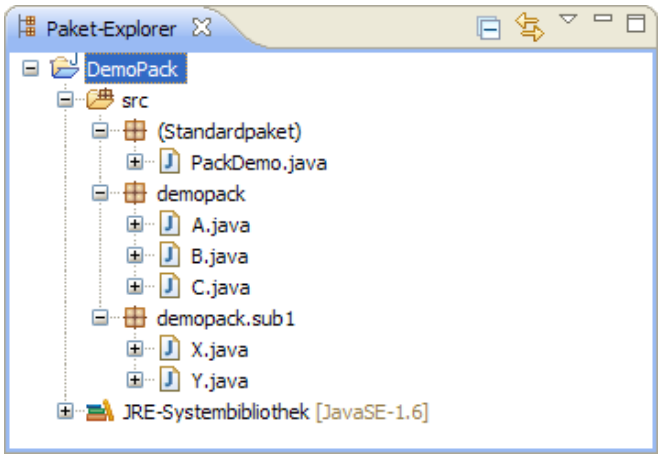

und zeigt ihn im Paket-Explorer an:




Nun legen wir im Unterpaket `demopack.sub1` die Klasse `X` an, deren Quellcode schon in Abschnitt 7.1.2 zu sehen war,




und danach die analog aufgebaute Klasse `Y`. Schließlich erstellen wir noch eine Startklasse namens `PackDemo` (bitte die Reihenfolge der Namensbestandteile beachten) zu Testzwecken (Quellcode folgt in Abschnitt 7.2.2), so dass im Paket-Explorer folgendes Bild entsteht:

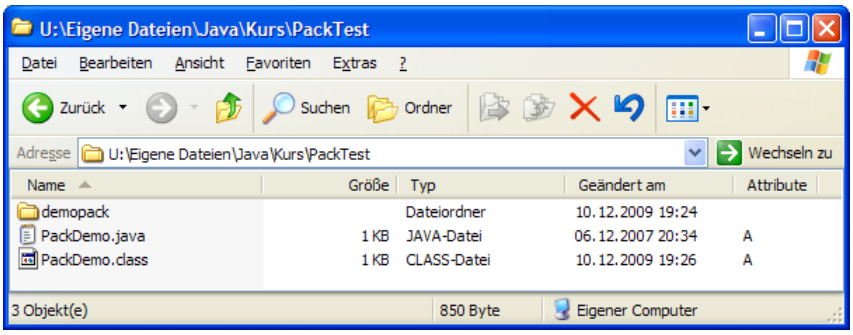



## *7.2 Pakete verwenden*

### 7.2.1 Verfügbarkeit der class-Dateien

Damit ein Paket genutzt werden kann, muss es sich an einem Ort befinden, der vom Compiler bzw. Interpreter bei Bedarf nach Klassen durchsucht wird. Die API-Pakete werden auf jeden Fall gefunden. Sonstige Pakete können auf unterschiedliche Weise in den Suchraum aufgenommen werden (vgl. Abschnitt 2.2.4). Wir ignorieren vorläufig Pakete in Java-Archiven (siehe Abschnitt 7.4) und beschränken uns passend zum Entwicklungsstadium des `DemoPack`-Beispiels auf Pakete in Verzeichnissen:

- Paket im **aktuellen Verzeichnis** ablegen
  Befindet sich das oberste Verzeichnis im Paketnamen (in unserem Beispiel: **demopack**) im selben Ordner wie die zu übersetzende oder auszuführende Klasse (im Beispiel: **PackDemo.java**), dann werden die Paketklassen vom JDK-Compiler bzw. von der JRE gefunden:




  Dies ist natürlich keine sinnvolle Option, wenn ein Paket in mehreren Projekten eingesetzt werden soll.

- Paket in der **CLASSPATH**-Umgebungsvariablen berücksichtigen
  Per CLASSPATH können nicht nur einzelne **class**-Dateien aus dem unbenannten Standardpaket verfügbar gemacht werden, sondern auch komplette Pakete. Wenn sich das oberste Verzeichnis im Paketnamen (in unserem Beispiel: **demopack**) im Ordner

  **U:\Eigene Dateien\Java\lib**

  befindet, kann die CLASSPATH-Definition etwa lauten:

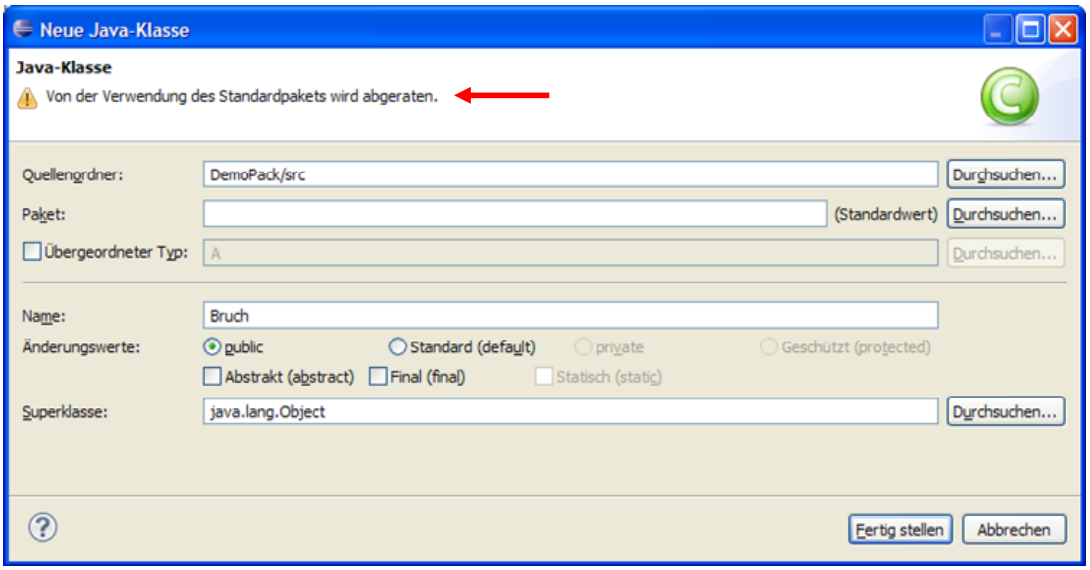
```
>set classpath=".;U:\Eigene Dateien\Java\lib"
```

  Eclipse berücksichtigt diese Umgebungsvariable allerdings *nicht*.

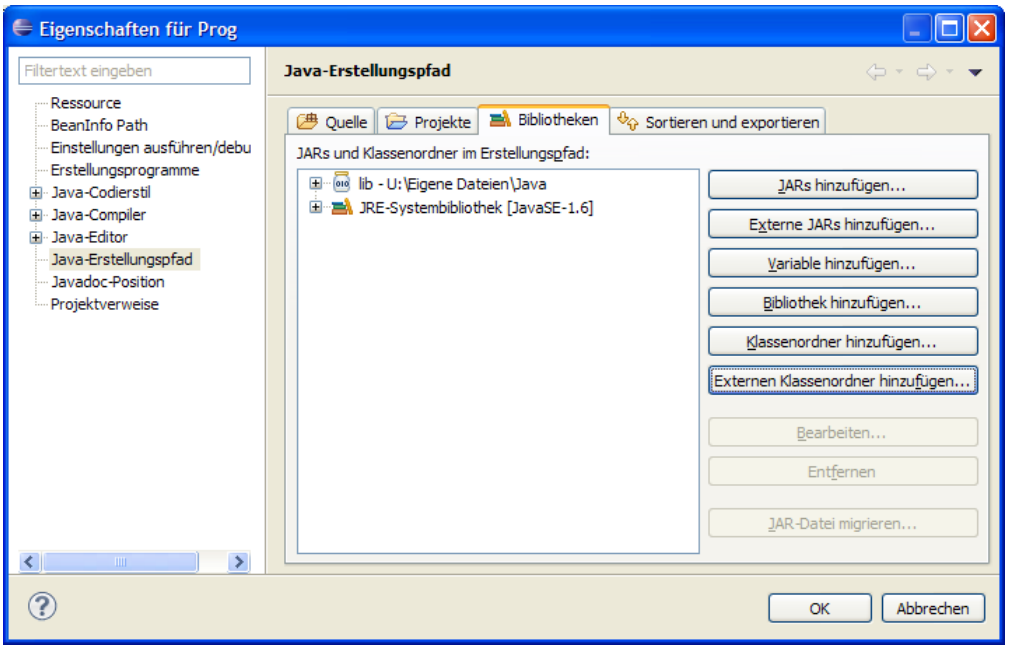
- **classpath**-Option beim Aufruf des JDK-Compilers bzw. Interpreters angeben, z.B.:

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

Die Klassen im unbenannten **Standardpaket** (engl. *default package*) sind in Klassen anderer Pakete generell (auch bei Zugriffsstufe **public**) **nicht** verfügbar. Diese gravierende Einschränkung haben wir 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.:



In Eclipse kann man einen *nicht* zum Projekt gehörigen Paketordner verfügbar machen, indem man per Projekt-Eigenschaftsdialog das übergeordnete Verzeichnis als **externen Klassenordner** in den **Java-Erstellungspfad** aufnimmt, z.B.:




Für einen in vielen Projekten benötigten externen Klassenordner (oder ein Archiv) sollte man auf Arbeitsbereichsebene eine Klassenpfadvariable definieren (vgl. Abschnitt 3.4.2).

Wie Sie bereits wissen, sind nur die **public**-Klassen eines Pakets für paketfremde Klassen zu sehen.

### 7.2.2 Namensregeln

Für den Zugriff auf die Klassen eines fremden, von **java.lang** verschiedenen Pakets bietet Java folgende Möglichkeiten:

- **Verwendung des vollqualifizierten Klassennamens**
  Dem Klassennamen ist der durch Punkt abgetrennte Paketnamen 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();
```

- **Import einer benötigten Klasse oder eines kompletten Pakets**
  Um die lästige Angabe von Paketnamen zu vermeiden, kann man einzelne Klassen und/oder komplette Pakete in eine Quellcodedatei *importieren*. Anschließend sind alle importierten Klassen direkt (ohne Paket-Präfix) ansprechbar. Die zuständigen **import**-Anweisungen sind an den Anfang einer Quellcodedatei zu setzen, ggf. aber *hinter* eine **package**-Deklaration. 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));
  }
}
```

  Um *alle* Klassen aus dem Paket **java.util** zu importieren, schreibt man:

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

Beachten Sie bitte, dass *Unterpakete* durch den Joker-Stern *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, falls keine Namenskollisionen (durch identische Klassennamen in verschiedenen Paketen) auftreten. Für das wichtige API-Paket **java.lang** übernimmt der Compiler den Import, so dass seine Klassen stets direkt angesprochen werden können.

- **Import von statischen Methoden und Feldern einer Klasse**
  Seit Java 5 besteht die Möglichkeit, statische Methoden und Variablen fremder Klassen zu importieren, so dass sie wie klasseneigene Elemente genutzt werden können. 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 besteht ist folgende Alternative verfügbar:

```
import static java.lang.Math.*;
class Prog {
 public static void main(String[] args) {
   System.out.println("Sin(Pi/2) = " + sin(PI/2));
 }
}
```

  Hier wird nicht nur der Paket- sondern auch noch der Klassenname eingespart.

In folgendem Programm wird das Beispielpaket `demopack` samt Unterpaket `sub1` importiert:

| Quellcode | Ausgabe |
|---|---|
| `import demopack.*;`<br>`import demopack.sub1.*;`<br><br>`class PackDemo {`<br>`  public static void main(String[] args) {`<br>`  A a1 = new A(), a2 = new A();`<br>`  a1.prinr();`<br>`  a2.prinr();`<br><br>`  B b = new B();`<br>`  b.prinr();`<br><br>`  C c = new C();`<br>`  c.prinr();`<br><br>`  X x = new X();`<br>`  x.prinr();`<br><br>`  Y y = new Y();`<br>`  y.prinr();`<br>`  }`<br>`}` | `Klasse A, Objekt Nr. 1`<br>`Klasse A, Objekt Nr. 2`<br>`Klasse B, Objekt Nr. 1`<br>`Klasse C, Objekt Nr. 1`<br>`Klasse X, Objekt Nr. 1`<br>`Klasse Y, Objekt Nr. 1` |

## *7.3 Zugriffsschutz (Sichtbarkeit)*

Nach der Beschäftigung mit Paketen kann endlich präzise erläutert werden, wie in Java die Zugriffsrechte für Klassen, Felder und Methoden festgelegt werden.

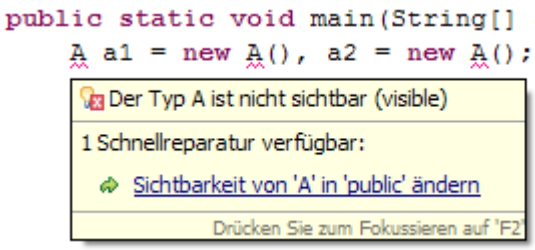
### 7.3.1 Zugriffsschutz für Klassen

Bisher haben wir uns nahezu ausschließlich mit *äußeren Klassen* (*Top-Level - Klassen*) beschäftigt, die *nicht* innerhalb des Quellcodes anderer Klassen definiert werden. Von den inneren Klassen wurden bislang die lokalen Klassen (siehe Abschnitt 4.3.1.4) und die anonymen Klassen (siehe Abschnitt 4.8.6) kurz erwähnt. Mitgliedsklassen, die Sie noch nicht kennen gelernt haben (siehe Abschnitt 11.6.6.1), sind beim Zugriffsschutz wie andere Mitglieder (Felder und Methoden) zu behandeln (siehe Abschnitt 7.3.2). Für die sonstigen inneren Klassen sind Zugriffsmodifikatoren irrelevant und verboten.

Die Top-Level - Klassen eines Paketes sind für Klassen aus fremden Paketen nur dann sichtbar, wenn bei der Definition der Zugriffsmodifikator **public** angegeben wird, z.B.:

```
package demopack;

public class A {
   . . .
   . . .
}
```

Wird im `demopack`-Paket die Klasse `A` ohne **public**-Zugriffsmodifikator definiert, scheitert das Übersetzen des in Abschnitt 7.2.2 vorgestellten Programms `PackDemo`. Der Eclipse-Compiler hilft durch Markieren der Fehlerstellen und mit einem sinnvollen Vorschlag zur **Schnellreparatur**:




Auch der JDK-Compiler liefert eine hilfreiche Problembeschreibung:

```
> javac PackDemo.java
PackDemo.java:6: demopack.A is not public in demopack;
cannot be accessed from outside package
                A a = new A(), aa = new A();
                ^
```

Pro Quellcodedatei darf nur *eine* Klasse als **public** deklariert werden. Eventuell vorhandene zusätzliche Klassen sind also nur paketintern zu verwenden. Diese Regel stellt allerdings keine Einschränkung dar, weil man in der Regel ohnehin für jede Klasse eine eigene Quellcodedatei verwendet (mit dem Namen der Klasse plus angehängter Erweiterung **.java**).

Bei aufmerksamer Lektüre der (z.B. im Internet) zahlreich vorhandenen Java-Beschreibungen stellt man fest, dass bei **ausführbaren Klassen** neben der statischen Methode **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 durchaus plausibel und systematisch, wird jedoch vom Java-Compiler bzw. –Interpreter *nicht* gefordert und stellt daher eine vermeidbare Mühe dar. Bei der Wahl einer Regel für dieses Manuskript habe ich mich am Verhalten der Java-Urheber orientiert: Gosling et at. (2005) lassen bei ausführbaren Klassen den Modifikator **public** systematisch weg.

### 7.3.2 Zugriffsschutz für Klassenmitglieder (Felder, Methoden, Mitgliedklassen)

Bei der Deklaration bzw. Definition von Feldern, Methoden und Mitgliedklassen[39] (objekt- oder klassenbezogen) können die Modifikatoren **private**, **protected** und **public** angegeben werden, um die Zugriffsrechte festzulegen. In der folgenden Tabelle wird die Wirkung 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 hier relevanten Zugriffsmodifikator **protected** werden wir uns bald beschäftigen.

Der vom Compiler bereitgestellte Standardkonstruktor (vgl. Abschnitt 4.4.3) hat den Zugriffsschutz der Klasse.

Wird im `demopack`-Beispiel die Klasse `A` mit **public**-Zugriffsmodifikator versehen, ihre `prinr()`-Methode jedoch nicht, dann scheitert das Übersetzen des Programms `PackDemo` durch den JDK-Compiler mit folgender Meldung:

```
>javac PackDemo.java
PackDemo.java:7: prinr() is not public in demopack.A;
cannot be accessed from outside package
                a.prinr();
                  ^
```

Für die *voreingestellte* Schutzstufe (nur Klassen aus dem eigenen Paket dürfen zugreifen) wird gelegentlich die Bezeichnung *package* verwendet.

## *7.4 Java-Archivdateien*

Wenn zu einem Programm zahlreiche **class**-Dateien (eventuell aufgeteilt in Pakete) und zusätzliche Hilfsdateien (z.B. mit Multimedia-Inhalten) gehören, dann bietet die Zusammenfassung zu einer Java-Archivdatei (Namenserweiterung **.jar**) Vorteile.

### 7.4.1 Eigenschaften von Archivdateien

Java-Archivdateien haben aus Sicht der Anwender 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 (siehe unten), bei entsprechender Konfiguration des Betriebssystems auch per Maus(doppel)klick.
- **Verkürzte Zugriffs- bzw. Übertragungszeiten**
  Eine einzelne Archivdatei reduziert im Vergleich zu zahlreichen einzelnen Dateien die Wartezeit beim Laden von einer Festplatte oder über ein Netzwerk.

---

[39] Mitgliedsklassen haben wir bisher noch nicht verwendet (siehe Abschnitt 11.6.6.1).

- **Kompression**
  Java-Archivdateien können komprimiert werden, was für Applets wegen des beschleunigten Internet-Transports sinnvoll ist, bei lokal installierten Anwendungen jedoch wegen der erforderlichen Dekomprimierung eher nachteilig. Die Archivdateien mit den Java-API-Klassen (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.

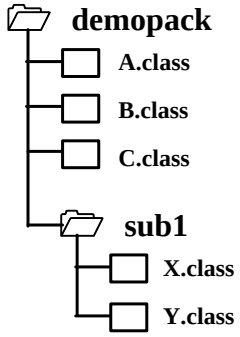
Eine Archivdatei kann beliebig viele Pakete und eigenständige Klassendateien enthalten. Damit diese vom Compiler und der JRE gefunden werden, muss die Archivdatei analog zu einem Verzeichnis mit demselben Inhalt in den **Suchpfad** für **class**-Dateien und Pakete aufgenommen werden (vgl. Abschnitte 3.4.2 und 7.2.1), z.B. über die Umgebungsvariable CLASSPATH:

```
>set classpath=.;U:\Eigene Dateien\Java\lib\Simput.jar
```

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 7.4.2 und 7.4.4) oder einer entsprechend ausgestatteten Entwicklungsumgebung überlassen (siehe Eclipse 3.x - Lösung in Abschnitt 7.4.5).

### 7.4.2 Archivdateien mit dem JDK-Werkzeug jar erstellen

Zum Erstellen und Verändern von Java-Archivdateien dient das JDK-Werkzeug **jar.exe**. Wir verwenden es dazu, eine Archivdatei mit dem `demopack`-Paket (siehe Abschnitt 7.1) samt Unterpakt `sub1`




zu erzeugen. Zur Vermeidung von Missverständnissen soll betont werden, dass generell in eine Archivdatei beliebig viele Pakete und auch sonstige Dateien aufgenommen werden können.

Wir öffnen ein Konsolenfenster und wechseln zum Verzeichnis, das den **demopack**-Ordner enthält. Dann lassen wir mit folgendem **jar**-Aufruf das Archiv **demarc.jar** mit der gesamten Paket-Hierarchie erstellen:[40]

```
>jar cf0 demarc.jar demopack
```

Darin bedeuten:

- 1. Parameter: Optionen
  Die Optionen bestehen aus jeweils einem Buchstaben und müssen unmittelbar hintereinander geschrieben werden:

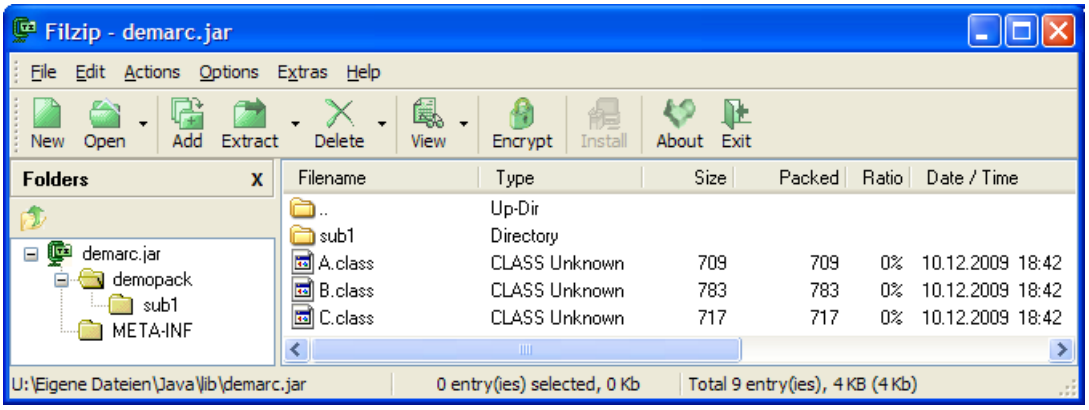
[40] Sollte der Aufruf nicht klappen, befindet sich vermutlich das JDK-Unterverzeichnis **bin** (z.B. **C:\Programme\Java\jdk1.6.0_16**) nicht im Suchpfad für ausführbare Programme. In diesem Fall muss das Programm mit kompletter Pfadangabe gestartet werden.

  - **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 weiterer Kommandozeilenparameter zu folgen hat.
  - **0**
    Mit der Ziffer **0** wird die ZIP-Kompression abgeschaltet.
- 2. Parameter: Archivdatei
  Der Archivdateiname muss einschließlich Extension (üblicherweise **.jar**) angegeben werden.
- 3. Parameter: zu archivierende Dateien und Ordner
  Bei einem Ordner wird rekursiv der gesamte Verzeichnisast einbezogen. Dabei werden nicht nur Bytecode-, sondern z.B. auch Multimediadateien berücksichtigt.

  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.

Weitere Informationen über das Archivierungswerkzeug finden Sie z.B. in der JDK-Dokumentation über den Link Java Archive (JAR) Files.

Aus obigem **jar**-Aufruf resultiert die folgende Java-Archivdatei (hier angezeigt vom kostenlosen Programm **Filzip**[41]):




Die Quellcodedateien sind für die Verwendung des Archivs (als Programm bzw. Klassenbibliothek) nicht erforderlich und können daher (z.B. aus urheberrechtlichen Gründen) entfernt werden.

### 7.4.3 Archivdateien verwenden

Um ein Archiv mit seinen Paketklassen nutzen zu können, muss es in den Klassensuchpfad des Compilers bzw. Interpreters aufgenommen werden. Befindet z.B. die Archivdatei **demarc.jar** im Ordner **U:\Eigene Dateien\Java\lib** und die Quellcodedatei der Klasse `PackDemo`, die 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 Werkzeug-Aufrufen kann eine Archivdatei in die CLASSPATH-Umgebungsvariable des Betriebssystems aufgenommen werden. Für die Nutzung von Ar-

[41] Verfügbar über die Webseite: http://www.filzip.com/de/

chivdateien unter Eclipse ist das Setzen von Klassenpfadvariablen sehr zu empfehlen (siehe Abschnitt 3.4.2).

### 7.4.4 Ausführbare JAR-Dateien

Um eine als Applikation ausführbare JAR-Datei zu erhalten, 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 noch keine Beachtung geschenkt haben, als **Hauptklasse** eingetragen werden. Das Manifest befindet sich in der Datei **MANIFEST.MF**, die das **jar**-Werkzeug im Archiv-Ordner **META-INF** anlegt, wobei im **jar**-Aufruf eine Datei mit Manifestinformationen übergeben werden kann. Um z.B. die Hauptklasse `PackDemo` auszuzeichnen, legt man eine Textdatei an, die folgende Zeile und eine anschließende Leerzeile (!) enthält:

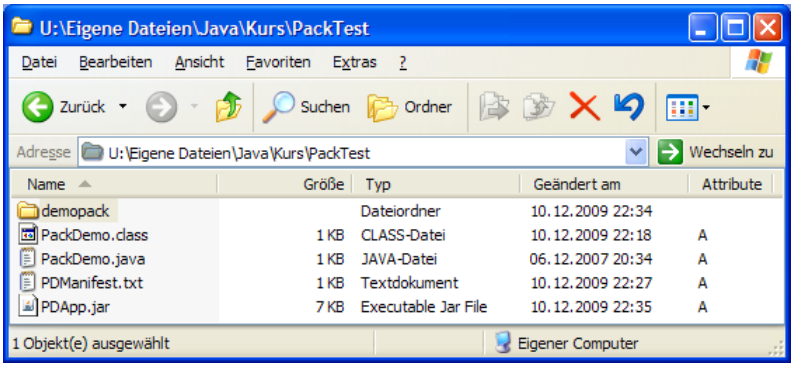
```
Main-Class: PackDemo
```

Im **jar**-Aufruf zum Erstellen des Archivs wird über die Option **m** eine Datei mit Manifestinformationen angekündigt, z.B. mit dem Namen `PDManifest.txt`:

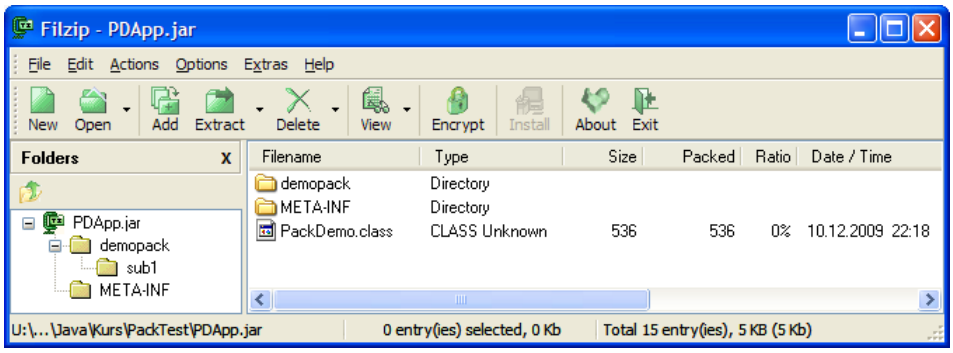
```
>jar cmf0 PDManifest.txt PDApp.jar PackDemo.class demopack
```

Es ist zu beachten, dass die Namen der Manifest- und der Archivdatei in derselben Reihenfolge wie die zugehörigen Optionen auftauchen müssen.

Obiger **jar**-Aufruf erwartet die Datei **PDManifest.txt** mit den Manifestinformationen, die Datei **PackDemo.class** mit der Startklasse und das Paketverzeichnis **demopack** im aktuellen Ordner, z.B.:




Es entsteht die Datei **PDApp.jar** mit folgendem Inhalt:




Unter Verwendung dieser Archivdatei kann das Programm `PackDemo` so gestartet werden:

```
>java -jar PDApp.jar
```

Dabei muss auf dem Zielrechner natürlich eine JRE mit geeigneter Version installiert sein.

Wenn mit dem Betriebssystem die Behandlung von **jar**-Dateien passend vereinbart wird, klappt der Start sogar per Maus(doppel)klick auf die Archivdatei. Unter Windows sind dazu folgende Registry-Einträge geeignet:

```
Windows Registry Editor Version 5.00

[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:\\Programme\\Java\\jre6\\bin\\java.exe\" -jar \"%1\" %*"
```

An Stelle des für Konsolenprogramme erforderlichen Starters **java.exe** wird bei der JRE-Installation allerdings in der Regel 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:\\Programme\\Java\\jre6\\bin\\javaw.exe\" -jar \"%1\" %*"
```

Weil **javaw.exe** kein Konsolenfenster anzeigt, bleibt der Doppelklick auf **PDApp.jar** ohne sichtbare Folgen.

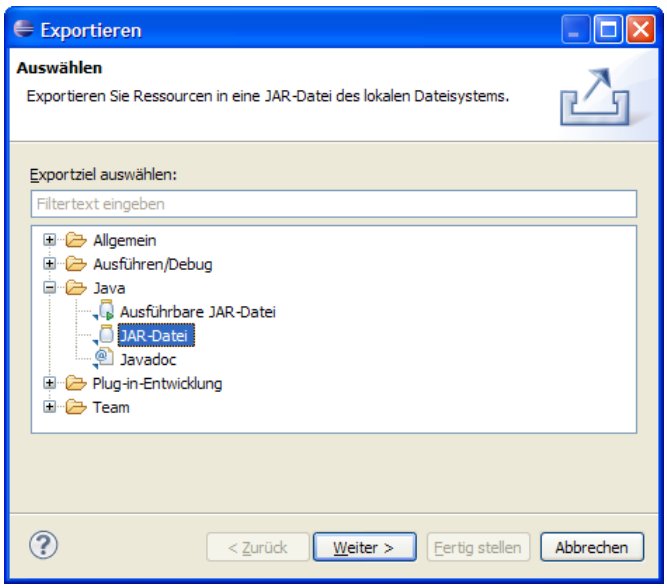
Wird ein Java-Programm per **jar**-Datei gestartet, dann legt allein das Manifest den Klassensuchpfad fest. Weder die Umgebungsvariable CLASSPATH, noch die Kommandozeilenoption –classpath sind wirksam. Die Klassen im Java-API werden aber auf jeden Fall gefunden.

### 7.4.5 Archivdateien in Eclipse erstellen

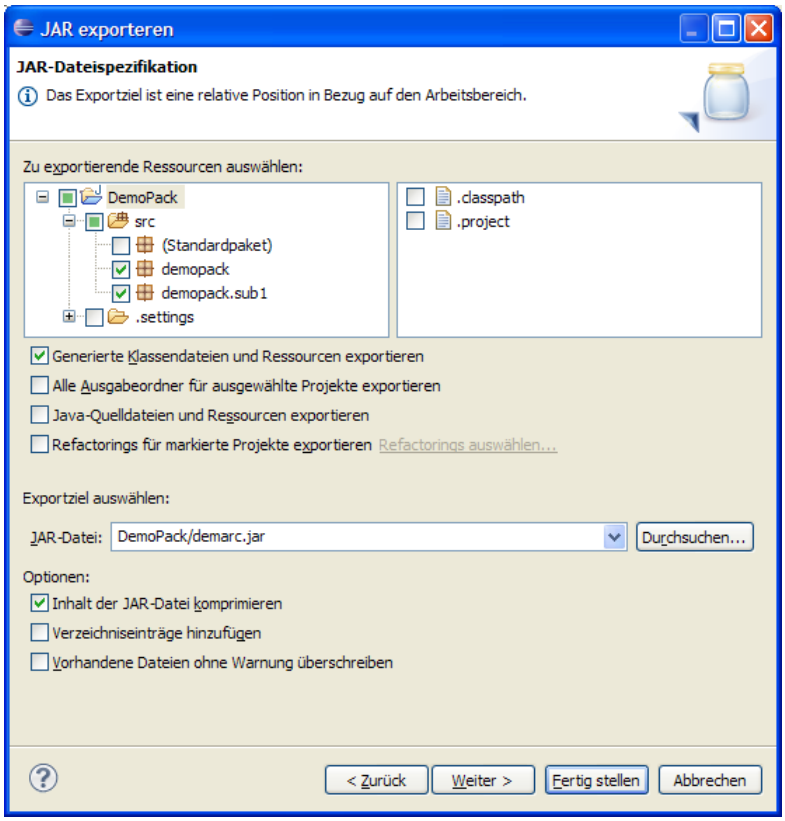
In Eclipse 3.x ist über

**Datei > Exportieren > Archivdatei**

ein bequemer Assistent zum Erstellen von JAR-Dateien verfügbar:

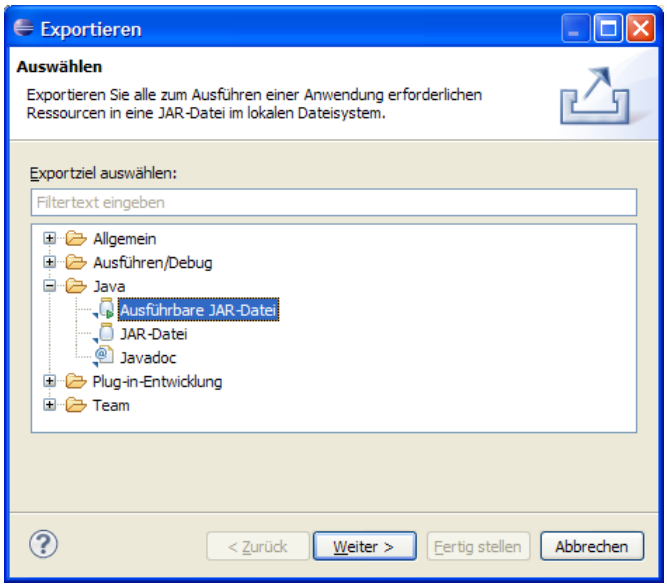



Mit dieser Wahl von Exportumfang und -ziel:

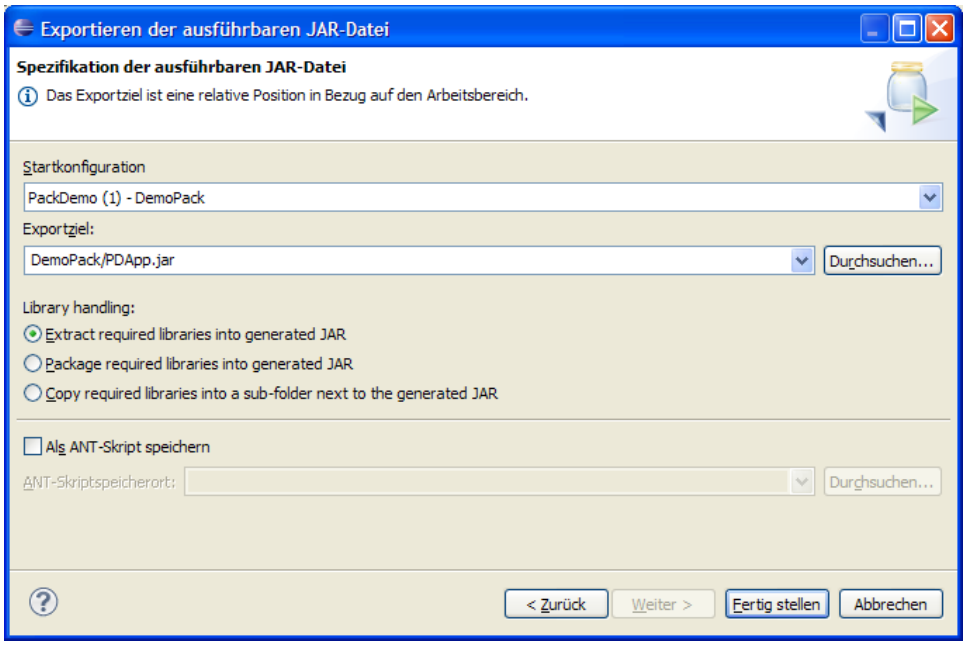



liefert der Assistent nach dem **Fertigstellen** dasselbe Ergebnis wie das in Abschnitt 7.4.5 vorgestellte **jar**-Kommando.

Wählt man im ersten Exportassistentendialog eine **ausführbare JAR-Datei**




kann man im nächsten Schritt eine vorhandene (nötigenfalls vorher angelegte, siehe Abschnitt 3.7.2.4) **Startkonfiguration** wählen, das **Exportziel** nennen




und die Anwendung **fertig stellen** lassen.

## *7.5 Das API der Java Standard Edition*

Zur Java-Plattform gehören zahlreiche Pakete, die Klassen für wichtige Aufgaben der Programmentwicklung (z.B. Zeichenkettenverarbeitung, Netzverbindungen, Datenbankzugriff) 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 beschränken, bietet die Firma Sun Microsystems noch weitere Java-APIs an, z.B.:

- Java Enterprise Edition (JEE)
- Java Micro Edition (JME)

Nähere Informationen finden Sie z.B. auf der Webseite http://java.sun.com/.

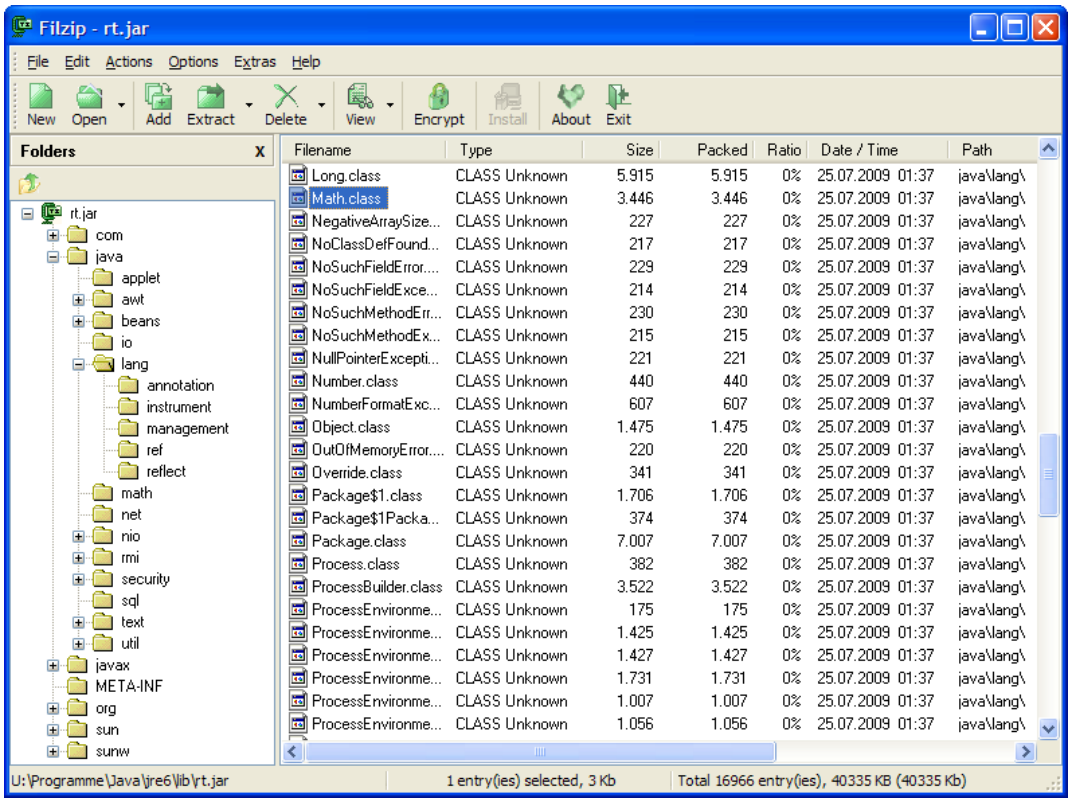
In der JDK-Dokumentation zur Standard Edition sind deren Pakete umfassend dokumentiert:

- Klicken Sie nach dem Start auf den Link Java Platform API Specification.
- Im linken oberen Rahmen kann ein Paket (oder die Option **All Classes**) gewählt werden, und im linken unteren Rahmen erscheinen alle Klassen des ausgewählten Pakets. Neben den in normalem Schriftschnitt notierten Klassen erscheinen in kursiver Schrift die Schnittstellen (Interfaces), mit denen wir uns später beschäftigen werden. Nach dem Anklicken einer Klasse oder Schnittstelle wird diese im Hauptrahmen ausführlich erläutert (z.B. mit einer Beschreibung der öffentlichen Methoden).

Sie haben vermutlich 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. **C:\Programme\Java\jdk6\lib**). Den größten Brocken bildet die Datei **rt.jar**. Hier 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:[42]

[42] Zur Anzeige der Datei **rt.jar** wird hier das kostenlose Programm **Filzip** verwendet (http://www.filzip.com/de/).




Es folgen kurze Beschreibungen wichtiger Pakete im API der Java Standard Edition:

- **java.applet**
  Das Paket enthält Klassen zum Programmieren von Applets. Dies sind kleine Programm, die per Web-Browser von einem Web-Server herunter geladen und dann im Rahmen des Browsers ausgeführt werden.
- **java.awt**
  Das AWT-Paket (**A**bstract **W**indowing **T**oolkit) enthält Klassen zum Programmieren von graphischen Benutzerschnittstellen. Heute werden allerdings meist die Komponenten aus dem aktuelleren Swing-Paket bevorzugt (siehe unten).
- **java.awt.event**
  Das AWT-Event-Paket enthält Klassen zur Ereignisverwaltung (siehe unten), die für Komponenten aus den Paketen **java.awt** und **javax.swing** relevant sind.
- **java.beans**
  Das Paket enthält Klassen zum Programmieren von Java-Komponenten (**beans** genannt).
- **java.io**
  Das Paket enthält Ein- und Ausgabeklassen.
- **java.lang**
  Das Paket enthält fundamentale Klassen, die in jedem Programm automatisch importiert werden und daher 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 Kollektionsklassen (z.B. **Vector<E>**) wichtige Hilfsmittel wie den Pseudozufallszahlengenerator **Random**, den wir schon mehrfach benutzt haben.
- **javax.swing**
  Im **swing**-Paket 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. Wir werden uns noch ausführlich mit den AWT- und Swing-Klassen beschäftigen.
- **javax.swing.event**
  Das Swing-Event-Paket enthält Klassen zur Ereignisverwaltung (siehe unten) für die Komponenten aus dem **javax.swing**.

## *7.6 Übungsaufgaben zu Kapitel 7*

1) Legen Sie das seit Abschnitt 7.1 als Beispiel verwendete Paket `demopack` an. Es sind folgende Klassen zu erstellen:

- im Paket `demopack`: `A`, `B`, `C`
- im Unterpaket `sub1`: `X`, `Y`

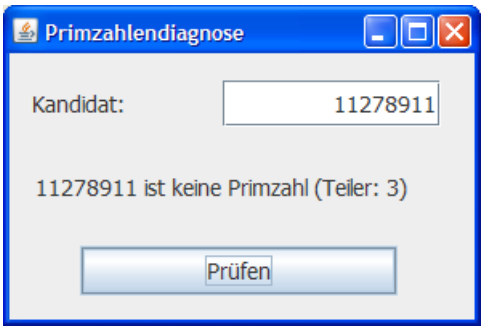
Erstellen Sie aus dem Paket eine Archivdatei, und verwenden Sie diese beim Übersetzen und Ausführen der im Abschnitt 7.2 wiedergegebenen Klasse `PackDemo`.

2) Im folgenden Programm

```
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?

3) Erstellen Sie ein ausführbares Java-Archiv mit dem in einer Übungsaufgabe zu Kapitel 4 (siehe Abschnitt 4.9) erstellten Primzahlendiagnoseprogramm mit graphischer Bedienoberfläche:




Benutzen Sie zunächst das JDK-Werkzeug **jar.exe**. Einzupacken sind die Dateien:

- **PrimDiag.class**
- **PrimDiag$1.class**
- **PrimDiag$2.class**

Außerdem wird noch eine Manifest-Datei benötigt, welche die auszuführende Klasse angibt (siehe Abschnitt 7.4.4).

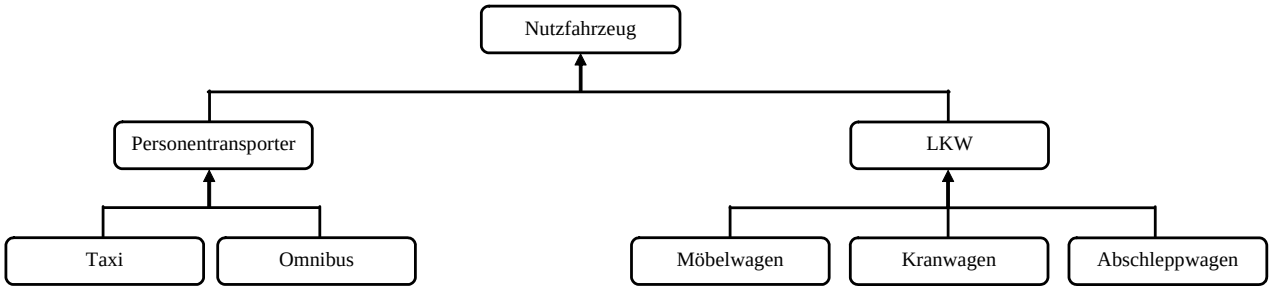
Testen Sie anschließend auch die automatisierte Erstellung mit Eclipse (vgl. Abschnitt 7.4.5).

# 8 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

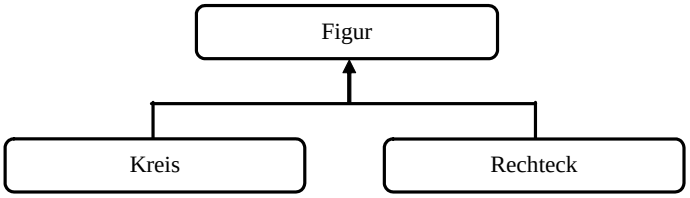
Beim Modellieren eines Gegenstandbereiches durch Klassen, die durch Eigenschaften (Instanz- und Klassenvariablen) sowie Handlungskompetenzen (Instanz- und Klassenmethoden) gekennzeichnet sind, müssen auch die Spezialisierungs- bzw. Generalisierungsbeziehungen zwischen real existierenden Klassen abgebildet werden. Eine Firma für Transportaufgaben aller Art mag ihre Nutzfahrzeuge folgendermaßen klassifizieren:




Einige Eigenschaften sind für alle Nutzfahrzeuge relevant (z.B. Anschaffungspreis, momentane Position, maximale Geschwindigkeit), andere betreffen nur spezielle Klassen (z.B. Anzahl der Fahrgäste, maximale Anhängelast, Hebekraft des Krans). Ebenso sind einige Handlungsmöglichkeiten bei allen Nutzfahrzeuge vorhanden (z.B. eigene Position melden), während andere speziellen Fahrzeugen vorbehalten sind (z.B. Fahrgäste aufnehmen, Klaviere 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:

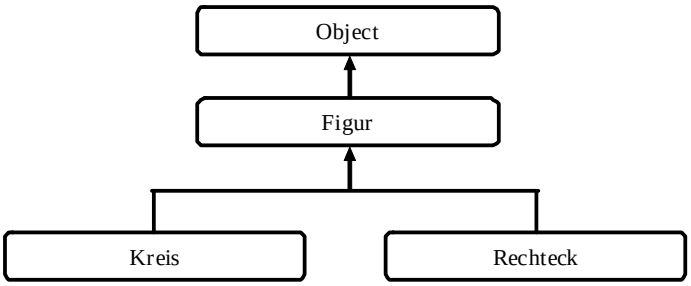



### Vererbung

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** 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

In Java stammen *alle* Klassen (sowohl die im API mitgelieferten als auch die vom Programmierer definierten) von der Klasse **Object** aus dem Paket **java.lang** ab. Wird (wie bei unseren bisherigen Beispielen) 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 hat eine Klasse stets nur *eine* Basisklasse, während andere Programmiersprachen (z.B. C++) auch eine *Mehrfachvererbung* erlauben. Somit ist in Java der kritische Fall ausgeschlossen, dass eine abgeleitete Klasse *mehrere* Methoden mit *identischer* Signatur erbt, woraus leicht Missverständnisse zwischen Programmierer und Programmiersprache über die tatsächlich auszuführende Methode resultieren können. Einen Ersatz bieten die Schnittstellen (siehe unten).

**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 und dabei doch robuste und wartungsfreundliche Softwaresysteme entstehen. Spätere Verbesserungen bei einer Basisklasse (z.B. bei einem Methodenimplementation) kommen allen (direkt oder indirekt) abgeleiteten Klassen zu Gute. Die noch weit verbreitete Praxis, vorhanden Code per *Copy & Paste* in neuen Projekten 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 Programmierergemeinde eingesetzt wird.

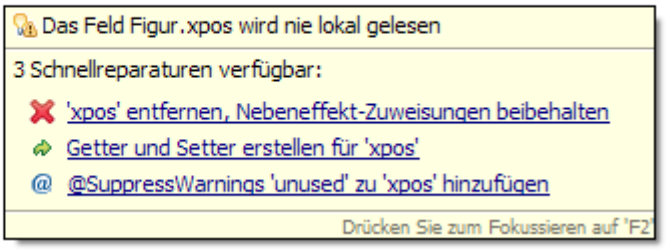
## *8.1 Definition einer abgeleiteten Klasse*

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:

```
public class Figur {
  private double xpos = 100.0, ypos = 100.0;

  public Figur(double x, double y) {
    xpos = x;
    ypos = y;
    System.out.println("Figur-Konstruktor");
  }

  public Figur() {}
}
```

Eclipse sieht keinen Nutzen in den privaten Instanzvariablen `xpos` und `ypos`, weil die Entwicklungsumgebung abgeleitete Klassen nicht berücksichtigen kann:

```
public class Figur {
    private double xpos = 100.0, ypos = 100.0;
```




Wir planen aber, die kritisierten Instanzvariablen in diversen abgeleiteten Klassen zu erben und dort auch zu verwenden, womit die Deklaration in der Basisklasse gerechtfertigt ist. Außerdem wird die Klasse `Figur` noch Methoden erhalten, welche die Koordinaten verwenden.

Mit Hilfe des Schlüsselworts **extends** wird die Klasse `Kreis` als Spezialisierung der Klasse `Figur` definiert. Sie erbt die beiden Positionsvariablen und ergänzt eine zusätzliche Instanzvariable für den Radius:

```
public class Kreis extends Figur {
  private double radius = 50.0;

  public Kreis(double x, double y, double rad) {
    super(x, y);
    radius = rad;
    System.out.println("Kreis-Konstruktor");
  }

  public Kreis() {}
}
```

Es wird ein initialisierender `Kreis`-Konstruktor definiert, der über das Schlüsselwort **super** den parametrisierten Konstruktor der Basisklasse aufruft, anstatt dessen Arbeit selbst zu erledigen. Das Schlüsselwort hat übrigens den oben eingeführten Begriff *Superklasse* motiviert. In Abschnitt 8.4 werden wir uns mit einigen Regeln für **super**-Konstruktoren beschäftigen.

Weil eine abgeleitete Klasse generell keine Konstruktoren erbt, wird in der `Kreis`-Klasse ein parameterfreier Konstruktor neu definiert, obwohl die Basisklasse einen solchen Konstruktor besitzt (natürlich mit anderem Namen!).

Das folgende Programm erzeugt ein Objekt aus der Basisklasse und ein Objekt aus der abgeleiteten Klasse:

| Quellcode | Ausgabe |
|---|---|
| `class FigurenDemo {`<br>`  public static void main(String[] args) {`<br>`  Figur fig = new Figur(50.0, 50.0);`<br>`  System.out.println();`<br>`  Kreis krs = new Kreis(10.0, 10.0, 5.0);`<br>`  }`<br>`}` | `Figur-Konstruktor`<br><br>`Figur-Konstruktor`<br>`Kreis-Konstruktor` |

## 8.2 Finalisierte Klassen

Gelegentlich gibt es Gründe dafür, eine Klasse mit dem Modifikator **final** als *endgültig* zu deklarieren, so dass sie zwar verwendet (instantiiert), aber nicht mehr beerbt werden kann. Für das Finalisieren der Klasse **String** im API-Paket **java.lang**

```
public final class String
```

sprach vermutlich ihre spezielle Behandlung durch den Compiler, z.B. beim impliziten Instantiieren (siehe Abschnitt 5.2.1.1) und beim Überladen des Plusoperators (siehe Abschnitt 5.2.1.4.1).

Finalisierte Klassen eignen sich auch zum Aufbewahren von Konstanten, z.B.:

```
public final class Busch {
   public final static int MAX = 333;
   public final static int MORITZ = 500;
   private Busch(){}
}
```

Um das unsinnige Erzeugen von Objekten zu einer öffentlichen Konstantenklasse zu verhindern, sollte man explizit einen parameterfreien Konstruktor mit dem Modifikator **private** definieren, um den Standardkonstruktor zu ersetzen, welcher dieselbe Schutzstufe hat wie die Klasse.

## *8.3 Der Zugriffsmodifikator protected*

Die folgende Variante des Figurenbeispiels soll den Effekt des Zugriffsmodifikators **protected** demonstrieren. Zunächst wird dafür gesorgt, dass die Klasse `Figur` zu einem anderen Paket gehört als die abgeleitete Klasse `Kreis` und die Startklasse `FigurenDemo`, weil *innerhalb* eines Paketes die abgeleiteten Klassen dieselben großzügigen Zugriffsrechte haben wie beliebige andere Klassen (vgl. Abschnitt 7.3.2).

```
package fipack;

public class Figur {
   protected double xpos=100.0, ypos=100.0;

   public Figur(double x, double y) {
      xpos = x;
      ypos = y;
   }

   public Figur() {}
}
```

Weil die Basisklasse `Figur` die Instanzvariablen `xpos` und `ypos` als **protected** deklariert, können Methoden abgeleiteter Klassen unabhängig von ihrer Paketzugehörigkeit direkt darauf zugreifen. Dies geschieht in der neuen `Kreis`-Methode `abstand()`, die für einen beliebigen Punkt (mit X- und Y-Koordinate) den Abstand zum Kreismittelpunkt berechnet:

```
import fipack.Figur;

public class Kreis extends Figur {
   protected double radius = 50.0;

   public Kreis(double x, double y, double rad) {
      super(x, y);
      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 hier *geerbte Instanzvariablen* von `Kreis`-Objekten verändert werden. Auf das `xpos`-Feld eines `Figur`-Objekts könnte eine Methode der `Kreis`-Klasse jedoch 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:

```
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);
  }
}
```

## 8.4 super-Konstruktoren und Initialisierungsmaßnahmen

Abgeleitete Klassen erben *nicht* die Basisklassenkonstruktoren, können diese aber in eigenen Konstruktoren über das Schlüsselwort **super** aufrufen. Dieser Aufruf muss am Anfang eines Konstruktors stehen. Dadurch ist es z.B. möglich, geerbte Instanzvariablen zu initialisieren, die in der Basisklasse als **private** deklariert wurden. Diese Konstellation war in der ursprünglichen Figurenhierarchie gegeben (siehe Abschnitt 8.1).

Wird in einem Konstruktor einer abgeleiteten Klasse kein Basisklassenkonstruktor aufgerufen, dann ruft der Compiler implizit den *parameterfreien* Konstruktor der Basisklasse auf. Fehlt ein solcher, weil der Programmierer einen eigenen, parametrisierten Konstruktor erstellt und nicht durch einen expliziten parameterfreien Konstruktor ergänzt hat, dann protestiert der JDK-Compiler z.B. so:

```
Kreis.java:8: cannot find symbol
symbol  : constructor Figur()
location: class Figur
        public Kreis() {}
                       ^
```

Es gibt zwei offensichtliche Möglichkeiten, das Problem zu lösen:

- Im Konstruktor der abgeleiteten Klasse über das Schlüsselwort **super** einen parametrisierten Basisklassenkonstruktor aufrufen.
- In der Basisklasse einen parameterfreien Konstruktor definieren.

Der parameterfreie Basisklassenkonstruktor wird auch vom Standardkonstruktor 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. 2005, 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 der Urahn **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, 100.0)`:

erzeugt wird:

- Das `Kreis`-Objekt wird mit allen Instanzvariablen (`xpos`, `ypos`, `radius`) auf dem Heap angelegt, und die Instanzvariablen werden mit Nullen initialisiert.
- Aktionen für die Klasse **Object**:
  - Die Instanzvariablen der Klasse **Object** erhalten den Initialisierungswert laut Deklaration.
    Derzeit sind mir zwar keine **Object**-Instanzvariablen bekannt, doch ist die Existenz von gekapselten Exemplaren durchaus möglich.
  - Der Rumpf des parameterfreien **Object**-Konstruktors wird ausgeführt.
    Dies ist der Standard-Konstruktor. Weitere Konstruktoren besitzt die Klasse **Object** auch nicht.
- 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, 100,0)` wird ausgeführt, wobei `radius` den Wert 100,0 erhält.

## *8.5 Überschreiben und Überdecken*

### 8.5.1 Methoden überschreiben

Eine Basisklassenmethode darf in einer Unterklasse durch eine Methode mit gleichem Namen und gleicher Parameterliste (also mit gleicher Signatur, vgl. Abschnitt 4.3.4) überschrieben werden, welche für die speziellere Unterklassensituation ein angepasstes Verhalten realisiert. Es liegt übrigens *keine* Überschreibung vor, wenn in der Unterklasse 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:

```
public class Figur {
  protected double xpos = 100.0, ypos = 100.0;

  public Figur(double x, double y) {
    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:[43]

```
public class Kreis extends Figur {
  protected double radius = 50.0;

  public Kreis(double x, double y, double rad) {
    super(x, y);
    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));
  }

  public void wo() {
    super.wo();
    System.out.println("Unten Rechts: (" + (xpos+2*radius) +
                       ", " + (ypos+2*radius) + ")");
  }
}
```

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 |
|---|---|
| `class Test {`<br>`  public static void main(String[] ars) {`<br>`  Figur f = new Figur(10.0, 20.0);`<br>`  f.wo();`<br>`  Kreis k = new Kreis(50.0, 100.0, 25.0);`<br>`  k.wo();`<br>`  }`<br>`}` | `Oben Links: (10.0, 20.0)`<br><br>`Oben Links: (50.0, 100.0)`<br>`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 die Zeichenfolgendarstellung zu einem Objekt zu ermitteln:

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  Prog tst1 = new Prog();`<br>`  Prog tst2 = new Prog();`<br>`  System.out.println(tst1);`<br>`  System.out.println(tst2);`<br>`  }`<br>`}` | `Prog@16f0472`<br>`Prog@18d107f` |

[43] Falls Sie sich über die Berechnungsvorschrift für die Y-Koordinate der rechten unteren `Kreis`-Ecke 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 Benutzeroberflächen schon jetzt daran gewöhnen.

In der JDK-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() {}

  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 {`<br>`  public static void main(String[] args) {`<br>`  Mint zahl = new Mint(4711);`<br>`  System.out.println(zahl);`<br>`  }`<br>`}` | `4711` |

Auch *statische* Methoden können überschrieben werden, wobei die Basisklassenvariante durch Voranstellen des Klassennamens angesprochen werden kann.

### 8.5.2 Finalisierte Methoden

Gelegentlich möchte man das Überschreiben einer Methode *verhindern*, damit sich der Nutzer einer Unterklasse darauf verlassen kann, dass dort die geerbte Methode *nicht* überschrieben worden ist. 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. Damit kann dem Nutzer der Klasse `Bcl` die ursprüngliche Funktionalität von `passwd()` garantiert werden.

Um das Überscheiben einer Methode zu verhindern, leitet man ihre Definition mit dem Modifikator **final** ein. Unsere `Figur`-Klasse 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 Methode noch einen Performanzvorteil: Während bei nicht-finalisierten Methoden *das Laufzeitsystem* feststellen muss, welche Variante in Abhängigkeit von der Klassenzugehörigkeit des angesprochenen Objekts tatsächlich ausgeführt werden soll (vgl. Abschnitt 8.7 über Polymorphie), steht eine **final**-Methode schon beim Übersetzen fest.

### 8.5.3 Felder überdecken

Wird in der abgeleiteten Klasse `Spezial` für eine Instanzvariable 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:

- Vom `General` geerbte Methoden verwenden weiterhin die `General`-Variable, während die zusätzlichen Methoden der `Spezial`-Klasse auf die `Spezial`-Variable zugreifen.
- Analog zu einer überschriebenen Methode kann die überdeckte Variable mit Hilfe des Schlüsselworts **super** weiterhin benutzt werden.

Im folgenden Beispielprogramm führt ein `Spezial`-Objekt eine `General`- und eine `Spezial`-Methode aus, um die beschriebenen Zugriffsvarianten 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
```

Auch *statische* Felder können überdeckt werden, wobei allerdings die Basisklassenvariante nicht über das Schlüsselwort **super**, sondern durch Voranstellen des Klassennamens angesprochen werden sollte.

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.

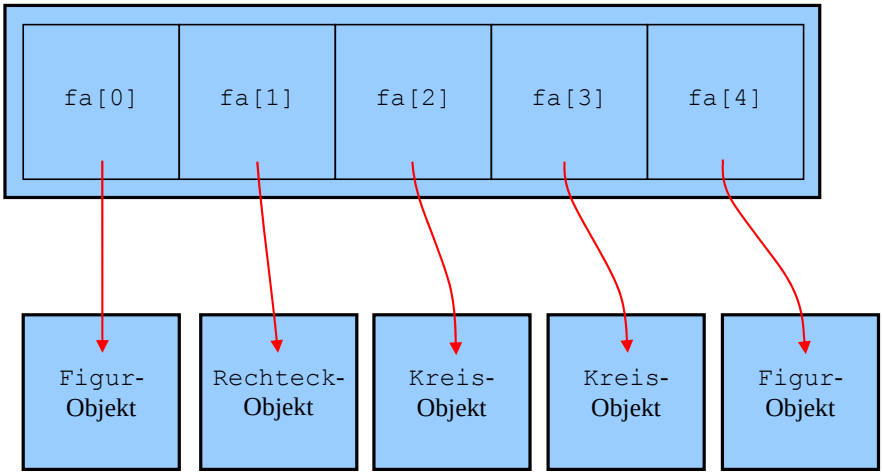
## *8.6 Verwaltung von Objekten über Basisklassenreferenzen*

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 8.1 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:

Array `fa` mit Elementtyp `Figur`




Das folgende Programm verwaltet Referenzen auf Figuren und Kreise in einem Array vom Typ `Figur`:

| Quellcode | Ausgabe |
|---|---|
| `class Test {`<br>`  public static void main(String[] ars) {`<br>`  Figur[] fa = new Figur[4];`<br>`  fa[0] = new Figur(10.0, 20.0);`<br>`  fa[1] = new Kreis(50.0, 50.0, 25.0);`<br>`  fa[2] = new Figur(0.0, 30.0);`<br>`  fa[3] = new Kreis(100.0, 100.0, 10.0);`<br>`  for (int i = 0; i < fa.length; i++)`<br>`  if (fa[i] instanceof Kreis)`<br>`  System.out.println("Figur "+i+": Radius = "+`<br>`  ((Kreis)fa[i]).getRadius());`<br>`  else`<br>`  System.out.println("Figur "+i+": kein Kreis");`<br>`  }`<br>`}` | `Figur 0: kein Kreis`<br>`Figur 1: kein Kreis`<br>`Figur 2: Radius = 25.0`<br>`Figur 3: Radius = 10.0`<br>`Figur 4: kein Kreis` |

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]).getRadius())
```

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]).getRadius());
```

Um den Zugriff auf Unterklassenerweiterungen demonstrieren zu können, hat die Klasse `Figur` im Vergleich zur Version in Abschnitt 8.5.1 die zusätzliche Methode `getRadius()` erhalten:

```
public int getRadius() {
  return radius;
}
```

## 8.7 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:

```
class Test {
  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?"+
     "\nWaehlen Sie durch Abschicken von \"f\" oder \"k\": ");
    if (Character.toUpperCase(Simput.gchar()) == 'F')
      fa[2] = new Figur();
    else
      fa[2] = new Kreis();
    fa[2].wo();
    if (fa[2] instanceof Kreis)
      System.out.println("Radius: "+((Kreis)fa[2]).getRadius());
  }
}
```

Hier werden Referenzen auf `Figur`- und `Kreis`-Objekte in einem Array vom gemeinsamen Basistyp `Figur` verwaltet (vgl. Abschnitt 8.6). 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 der Compiler die Klassenzugehörigkeit nicht kennen muss, darf im Beispielprogramm die Klasse des Array-Elementes `fa[2]` vom Benutzer festgelegt werden.

Dank Polymorphie kann eine Basisklasse auch mit solchen Unterklassen kooperieren, die beim Übersetzen der Basisklasse noch gar nicht existierten. Somit verbessert auch diese OOP-Technik die Chancen für produktives Software-Recycling.

## 8.8 Abstrakte Methoden und Klassen

Um die eben beschriebene gemeinsame Verwaltung von Objekten aus diversen Unterklassen über Referenzvariablen vom Basisklassentyp nutzen und dabei artgerechte 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, dem der Modifikator **abstract** vorangestellt wird.
- Den Methodenrumpf ersetzt man durch ein Semikolon.

Im Figurenbeispiel ergänzen wir eine Methode namens `inhalt()` zum Ermitteln des Flächeninhalts in der Klasse `Kreis`

```
public double inhalt() {
  return 2 * Math.PI * radius*radius;
}
```

sowie in der von Ihnen im Rahmen einer Übungsaufgabe zu erstellenden Klasse `Rechteck`:

```
public double inhalt() {
  return breite * hoehe;
}
```

Weil die Methode zum Ermitteln des Flächeninhalts in der Basisklasse `Figur` nicht sinnvoll realisierbar ist, wird sie hier abstrakt definiert:

```
public abstract class Figur {
  . . .
  public abstract double inhalt();
  . . .
}
```

Enthält eine Klasse mindestens eine abstrakte Methode, dann handelt es sich um eine **abstrakte Klasse**, und bei der Klassendefinition muss der Modifikator **abstract** angegeben 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:

```
class Test {
  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 (%-15s): %10.2f\n",
                        i, fa[i].getClass(), fa[i].inhalt());
      ges += fa[i].inhalt();
    }
    System.out.printf("\nGesamtflaeche:  %10.2f",ges);
  }
}
```

liefert die Ausgabe:

```
Fläche Figur 0 (class Kreis    ):    3926,99
Fläche Figur 1 (class Rechteck ):   20000,00

Gesamtflaeche:    23926,99
```

Die Methode `inhalt()` eignet sich dazu, am Ende von Abschnitt 8 den Nutzen der Polymorphie noch einmal 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.

## 8.9 Übungsaufgaben zu Kapitel 8

1) Warum kann der folgende Quellcode (mit zwei Klassen im Standardpaket) nicht übersetzt werden?

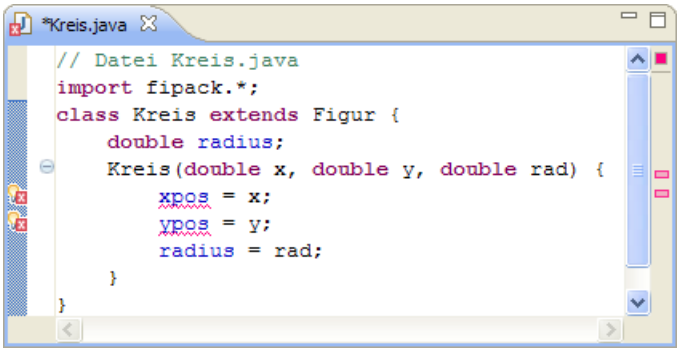
```
// Datei General.java
class General {
  int ig;
  General(int igp) {
    ig = igp;
  }
  void hallo() {
    System.out.println("hallo-Methode der Klasse General");
  }
}
// Datei Spezial.java
class Spezial extends General {
  int is = 3;
  void hallo() {
    System.out.println("hallo-Methode der 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 8.8. 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 `inhalt()`-Methode, welche die abstrakte `Figur`-Version implementiert.

In der `Kreis`-Klasse ist seit Abschnitt 8.3 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:

- Die Klasse `Figur` ist mittlerweile abstrakt, also nicht instanziierbar.
- Abgeleitete Klassen sollten keine formunabhängige Abstandsmessung erben können.

Erstellen Sie ein Testprogramm, das polymorphe Objektverwaltung und entsprechende Methodenaufrufe demonstriert.

5) Von verbesserten Methodenimplementation in der Basisklasse 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 übersetzten.

# 9 Ausnahmebehandlung

Durch Programmierfehler (z.B. versuchter Feldzugriff mit ungültigem Indexwert) oder durch besondere Umstände (z.B. irreguläre Eingabedaten, Speichermangel, unterbrochene Netzverbindungen) kann die reguläre Ausführung einer Methode scheitern. Solche Probleme sollten entdeckt, behoben oder zusammen mit hilfreichen Informationen an den Aufrufer der Methode und eventuell schlussendlich an den Benutzer gemeldet werden, statt zu einem Absturz und sogar zu einem Datenverlust zu führen.

Java bietet ein modernes Verfahren zur Meldung und Behandlung von Problemen: An der Unfallstelle wird ein Ausnahmeobjekt aus der Klasse **Exception** aus dem Paket **java.lang** oder aus einer problemspezifischen Unterklasse erzeugt und der unmittelbar verantwortlichen Methode „zugeworfen“. Diese wird über das Problem informiert und mit relevanten Daten für die Behandlung versorgt.

Die Initiative beim Auslösen einer Ausnahme kann ausgehen …

- vom **Laufzeitsystem**
  Wir dürfen annehmen, dass die JRE praktisch immer stabil bleibt. Entdeckt sie einen Fehler, der nicht zu schwerwiegend ist und vom Benutzerprogramm prinzipiell behoben werden kann, wirft sie ein Ausnahmeobjekt, bei einem versuchten Feldzugriff mit ungültigem Indexwert z.B. ein Objekt aus der Klasse **IndexOutOfBoundsException**.
- vom **Programm**, wozu auch die verwendeten Bibliotheksklassen gehören
  In jeder Methode kann mit der **throw**-Anweisung (siehe Abschnitt 9.6) eine Ausnahme erzeugt werden.

Die unmittelbar von einer Ausnahme betroffene Methode steht oft am Ende einer Sequenz verschachtelter Methodenaufrufe, und entlang der Aufrufersequenz haben die beteiligten Methoden jeweils folgende Reaktionsmöglichkeiten:

- Ausnahmeobjekt abfangen und das Problem behandeln
  Dabei ist der im Ausnahmeobjekt enthaltene Unfallbericht von Nutzen. Scheidet die Fortführung des ursprünglichen Handlungsplans auch nach der Ausnahmebehandlung aus, sollte erneut ein Ausnahmeobjekt geworfen werden, entweder das ursprüngliche oder ein informativeres.
- Ausnahmeobjekt ignorieren und dem Vorgänger in der Aufrufersequenz überlassen

Wir werden uns anhand verschiedener Versionen eines Beispielprogramms damit beschäftigen,

- was bei unbehandelten Ausnahmen geschieht,
- wie man Ausnahmen abfängt,
- wie man selbst Ausnahmen wirft,
- wie man eigene Ausnahmeklassen definiert.

Man kann von keinem Programm erwarten, dass es unter allen widrigen Umständen normal funktioniert. Doch müssen Datenverluste verhindert werden, und der Benutzer sollte nach Möglichkeit eine nützliche Information zum aufgetretenen Problem erhalten. Bei vielen Methodenaufrufen ist es realistisch und erforderlich, auf ein Scheitern vorbereitet zu sein. Dies folgt schon aus **Murphy’s Law** (zitiert nach Wikipedia):

> „Whatever can go wrong, will go wrong.“

### 9.1 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.:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "vier"
   at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
   at java.lang.Integer.parseInt(Integer.java:449)
   at java.lang.Integer.parseInt(Integer.java:499)
   at Fakul.getFakulArg(Fakul.java:3)
   at Fakul.main(Fakul.java:14)
```

Das folgende Programm soll die Fakultät zu einer Zahl ausrechnen, die beim Start als Programmargument übergeben wird. Dabei beschränkt sich die **main()**-Methode auf die eigentliche Fakultätsberechnung und überlässt die Konvertierung und Validierung des übergebenen Strings der Methode `getFakulArg()`. Diese wiederum stützt sich bei der Konvertierung auf die statische Methode **parseInt()** der Klasse **Integer**:

```
class Fakul {
   static int getFakulArg(String instr) {
      int arg = Integer.parseInt(instr);
      if (arg >= 0  &&  arg <= 170)
         return arg;
      else
         return -1;
   }

   public static void main(String[] args) {
      int argument = -1;

      if (args.length > 0)
         argument = getFakulArg(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 ist durchaus bemüht, einige kritische Fälle zu vermeiden. Die Methode **main()** überprüft, ob `args[0]` tatsächlich vorhanden ist, bevor dieses **String**-Objekt beim Aufruf der Methode `getFakulArg()` als Parameter verwendet wird. Damit wird verhindert, dass es zu einer **ArrayIndexOutOfBoundsException** kommt, wenn der Benutzer das Programm ohne Kommandozeilenparameter 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 Betriebsystem und steht unter Windows in der Umgebungsvariablen ERRORLEVEL zur Verfügung, z.B.:

```
>java Fakul
Kein Argument angegeben

>echo %ERRORLEVEL%
1
```

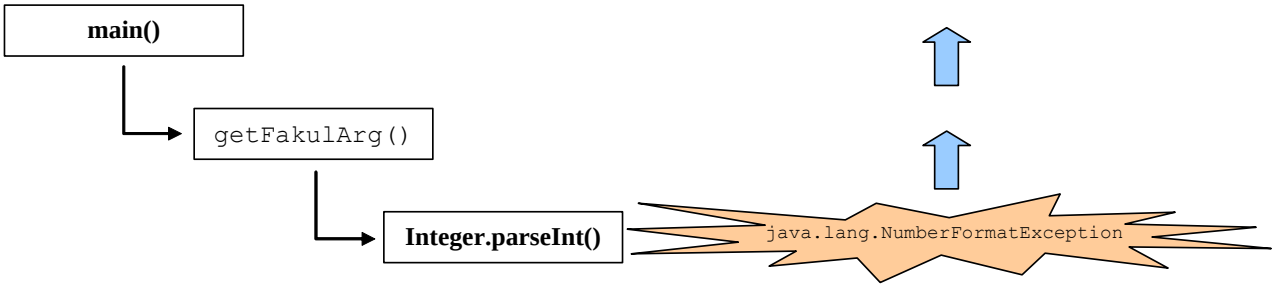
Nach einem störungsfrei verlaufenen Programmeinsatz enthält ERRORLEVEL den Exitcode 0.

Die Reaktion auf ein fehlendes Programmargument kann als akzeptabel gelten:

- An Stelle der für Benutzer irritierenden und wenig hilfreichen Ausnahmemeldung durch das Laufzeitsystem (siehe oben) erscheint eine verwertbare Information.
- Falls das Programm von einem anderen Programm (z.B. einer Kommando-Prozedur) gestartet worden ist, steht dem Aufrufer ein Exitcode zur Verfügung.

Die Methode `getFakulArg()` 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. **main()** erkennt die spezielle Bedeutung dieses Rückgabewerts, so dass z.B. unsinnige Fakultätsberechnungen für negative Argumente vermieden werden. Diese traditionelle Fehlerbehandlung per Rückgabewert ist nicht grundsätzlich als überholt und ineffizient zu bezeichnen, aber in vielen Situationen doch der gleich vorzustellenden Kommunikation über Ausnahmeobjekte unterlegen (siehe Abschnitt 9.3 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 Zeichnfolge füttert (z.B. „vier“). Die zunächst betroffene Methode[44] **Integer.parseInt()** wirft daraufhin eine **NumberFormatException**. Diese wird vom Laufzeitsystem entlang der Aufrufreihenfolge an `getFakulArg()` 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.

## *9.2 Ausnahmen abfangen*

### 9.2.1 Die try-catch-finally - Anweisung

In Java wird die Behandlung von Ausnahmen über die **try-catch-finally** - Anweisung unterstützt:

[44] Aufrufverschachtelungen *innerhalb* der Java-Klassenbibliothek ignorieren wir an dieser Stelle. In Abschnitt 9.2.3wird die Angelegenheit mit Hilfe des API-Quellcodes genauer untersucht.

**try {**
*Überwachter Block mit Anweisungen für den normalen Programmablauf*
**}**
**catch (***Ausnahmeklasse Parametername***) {**
*Anweisungen für die Behandlung von Ausnahmen der ersten Ausnahmeklasse*
**}**
// Optional können weitere Ausnahmen abgefangen werden:
**catch (***Ausnahmeklasse Parametername***) {**
*Anweisungen für die Behandlung von Ausnahmen der zweiten Ausnahmeklasse*
**}**

. . .

// Optionaler Block mit Abschlussarbeiten.
// Bei vorhandenem **finally**-Block ist kein **catch**-Block erforderlich.
**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 erstellen (siehe Abschnitt 9.8).

#### *9.2.1.1 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 analog zu einer Methode in seinem Kopfbereich über eine einelementige Parameterliste, wobei als Formalparametertyp eine mehr oder weniger breite (allgemeine) Ausnahmeklasse in Frage kommt. Das Laufzeitsystem sucht nach einem **catch**-Block mit einer zum behandelnden Objekt passenden Ausnahmeklasse und führt ggf. den zugehörigen Anweisungsblock aus. 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 `getFakulArg()` 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 getFakulArg(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 JDK-Dokumentation zeigt, sind von **parseInt()** keine weiteren Ausnahmeklassen 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 = getFakulArg(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 Kommandozeilenparameter erscheint nun eine informative Fehlermeldung des Programms an Stelle eines „Absturzprotokolls“ der JRE, z.B.:

```
Fehler beim Konvertieren von: vier
```

Nach der Ausführung eines **catch**-Blocks wird das betroffene Programm hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**–Block ausgeführt wird.

#### *9.2.1.2 finally*

In einen **finally**-Block gehören solche Anweisungen, die auf jeden Fall ausgeführt werden sollen (z.B. zur Freigabe von Ressourcen wie Datei - und Netzverbindungen):

- nach der ungestörten Ausführung des **try**-Blocks
  Auch einvorzeitiges 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

Dies ist der ideale Ort für Anweisungen, die unter möglichst allen Umständen ausgeführt werden sollen, z.B. zur Freigabe von Ressourcen wie Datei - und Netzverbindungen. Wir verwenden (dem Kapitel 12 vorgreifend) zur **finally**-Demonstration eine statische Methode namens `mean()`, die mit Hilfe eines **DataInputStream**-Objekts aus einer Binärdatei 100 dort erwartete **double**-Zahlen liest und den Mittelwert daraus berechnet:

```
import java.io.*;
class Mean {
  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 ignored) {};
    }
  }

  public static void main(String args[]) {
    mean("eingabe.dat");
  }
}
```

Ein Programm muss generell geöffnete Dateien möglichst früh per **close()**-Aufruf schließen, um den Benutzer und andere Programme möglichst wenig zu behindern. 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, die vor geraumer Zeit mit einem Programm bearbeitet wurde, das noch aktiv ist und die Datei ohne Grund weiterhin blockiert. Genau diese Beschränkung würde im Beispielprogramm auftauchen, das eine Datei zum Lesen öffnet. Bei zum Schreiben geöffneten Dateien kommen noch weitere Motive für das möglichst frühzeitige Schließen hinzu (siehe Abschnitt 12.1.5).

API-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 9.5). Im Beispiel sind der **FileInputStream**-Konstruktor und die **DataInputStream**-Methode **readDouble()** betroffen. Es könnte 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 möglicht frühe Schließen der Datei auch im Ausnahmefall sichergestellt ist, findet der erforderliche **close()**-

Aufrufe 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.

Ist bereits das Öffnen der Datei im **FileInputStream**-Konstruktor misslungen, existiert 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 diese Ausnahmeklasse explizit zu berücksichtigen ist (siehe Abschnitt 9.5), muss der **close()**-Aufruf in einer **try-catch** - Anweisung stattfinden, und es resultiert eine **try**-Verschachtelung.

### 9.2.2 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 eine optimierte Einsatzplanung *nicht* 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 = 10 % Integer.parseInt(instr);
    }
    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 den Aufruf in eine **try**-Anweisung mit **catch**-Block für die **ArithmeticException** ein, die das Laufzeitsystem z.B. bei einer versuchten Division 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 = 10 % 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)

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:22)
```

In einer komplexen Methode ist es oft sinnvoll, **try**-Anweisungen zu schachteln, wobei sowohl innerhalb eines **try**- als auch innerhalb eines **catch**-Blocks wiederum eine komplette **try**-Anweisung stehen darf. Daraus ergeben sich weitere Ablaufvarianten für eine flexible Ausnahmebehandlung.

## 9.2.3 Diagnostische Ausgaben

Viele Informationen eines Ausnahmeobjekts eignen sich direkt für diagnostische Ausgaben. 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** (*stack trace*) von der **main()**-Methode bis zur Unfallstelle. Ergänzt man im **catch**-Block des Beispielprogramms von Abschnitt 9.2.1.1 einen Aufruf der Methode **printStackTrace()**,

```
catch (NumberFormatException e) {
   e.printStackTrace(System.out);
   return -2;
}
```

dann führt z.B. ein Programmstart mit dem Kommandozeilenargument „vier“ zu folgender Ausgabe:

```
java.lang.NumberFormatException: For input string: "vier"
   at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
   at java.lang.Integer.parseInt(Integer.java:449)
   at java.lang.Integer.parseInt(Integer.java:499)
   at Fakul.getFakulArg(Fakul.java:5)
   at Fakul.main(Fakul.java:25)
Fehler beim Konvertieren von: vier
```

Vielleicht wunden 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 `getFakulArg()` aufgerufene **parseInt()**-Überladung mit einem Parameter vom Typ **String**

```
public static int parseInt(String s) throws NumberFormatException {
  return parseInt(s,10);
}
```

die eigentliche Arbeit der Überladung mit einem zusätzlichen Parameter für die Basis des Zahlensystems überlässt, die auf das Problem stößt und die **NumberFormatException** wirft:

```
public static int parseInt(String s, int radix)
    throws NumberFormatException {
  . . .
}
```

### 9.3 Ausnahmeobjekte im Vergleich mit der traditionellen Fehlerbehandlung

Die konventionelle Fehlerbehandlung verwendet meist die Rückgabewerte von Methoden zur Berichterstattung über Probleme bei der Ausführung von Aufträgen. Ein Rückgabewert kann …

- ausschließlich zur Fehlermeldung dienen
  Meist wird dann ein ganzzahliger **Returncode** 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 über spezielle Wert Problemfälle signalisieren
  Diese Variante wird in Abschnitt 9.2.1.1 vorgeführt.

Sollen z.B. drei Methoden, deren Rückgabewerte ausschließlich zur Fehlermeldung dienen, nacheinander aufgerufen werden, dann wird die vom Algorithmus diktierte simple Sequenz:

```
m1();
m2();
m3();
```

nach Ergänzen der Fehlerbehandlungen zu einer länglichen und recht unübersichtlichen Konstruktion (nach Mössenböck 2005, S. 254):

```
returncode = m1();
if (returncode == 0) {
  returncode = m2();
  if (returncode == 0) {
    returncode = m3();
    if (returncode == 0) {
      .  .  .
    }
    else {
      // Ausnahmebehandlung für diverse m3()-Fehlercodes}
  }
  else {
    // Ausnahmebehandlung für diverse m2()-Fehlercodes}
}
else {
  // Ausnahmebehandlung für diverse m1()-Fehlercodes
}
```

Mit Hilfe der Ausnahmetechnik bleibt beim Kernalgorithmus die Übersichtlichkeit erhalten:

```
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
}
```

Ein gut gesetzter Rückgabewert nutzt 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 1.1.3) 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 Klasse **PrintWriter** (siehe Abschnitt 12.4.1.4) 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 Returncodes oder Fehlerstatusvariablen 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 9.5). 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 beachtet werden.
- Automatische Weitermeldung bis zur bestgerüsteten Methode
  Oft ist der unmittelbare Aufrufer 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 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 Statusvariable nicht in jedem Fall der moderneren, aber auch aufwändigeren Kommunikation per Ausnahmeobjekt unterlegen. Die Verwendung der traditionellen Technik im Beispielprogramm von Abschnitt 9.2 kann z.B. als akzeptabel gelten. Im weiteren Verlauf von Abschnitt 9 wird aber auch eine alternative Variante der Methode `getFakulArg()` 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 Statusvariable kommunizieren.

Eine weitere Möglichkeit besteht darin, dem Aufrufer eine zusätzliche Methode zur Prüfung der Erfolgsaussicht anzubieten. In der Klasse **Scanner** (vgl. Abschnitt 12.5), die sich dazu eignet, aus einer Textdatei Werte primitiver Datentypen zu lesen, 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**-Literal zu interpretieren.
- **public boolean hasNextDouble()**
  Es wird überprüft, ob das eben beschriebene Unterfangen gelingen kann.

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.:

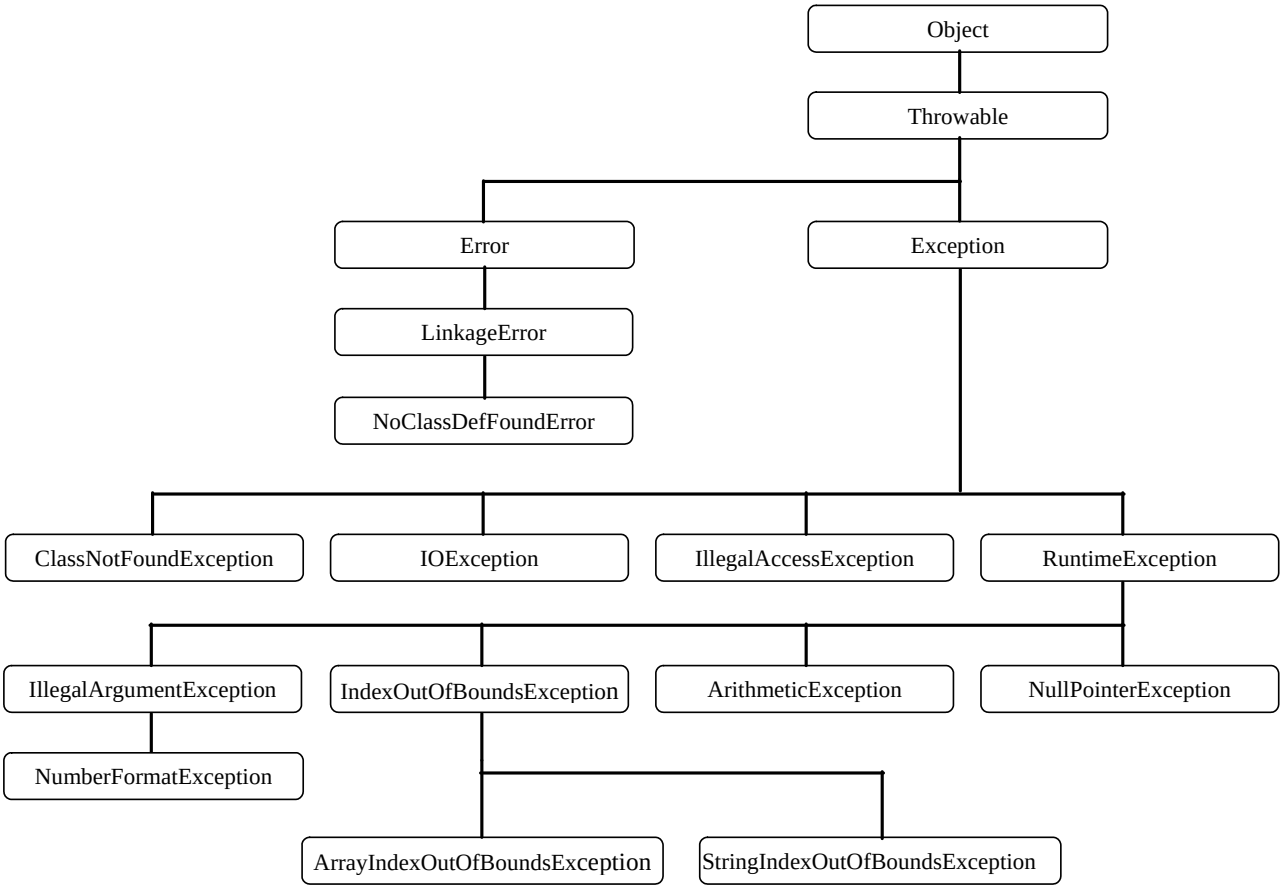
```
while (input.hasNextDouble()) {
 sum += input.nextDouble();
 n++;
}
```

Bei Fehlern mit geringer Wahrscheinlichkeit haben jedoch häufige, meist überflüssige Kontrollen eine Performanzeinbuße zur Folge. Hier sollte man es besser auf eine Ausnahme ankommen lassen.

Eine Überwachung über 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.

## *9.4 Ausnahmeklassen in Java*

Java kennt zahlreiche Ausnahmeklassen, die mit ihren Vererbungsbeziehungen eine Klassenhierarchie bilden, aus der die folgende Abbildung einen kleinen Ausschnitt zeigt:




In einem **catch**-Block 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 der Compiler freundlicherweise überwacht.

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

JRE z.B. eine für den Programmablauf benötigte Klasse nicht finden, wird ein **NoClassDefFoundError** gemeldet:

```
>java PackDemo
Exception in thread "main" java.lang.NoClassDefFoundError: demopack/A
        at PackDemo.main(packdemo.java:7)
```

Wenn man über die statische **Class<T>** - Methode **forName()**

**public static Class<?> forName(String** *name***)**

das zu einer per Zeichenfolge spezifizierten Klasse gehörige Beschreibungsobjekt anfordert, und keine Klasse mit diesem Namen zu finden ist, wirft die Methode **forName()** ein Ausnahmeobjekt aus der Klasse **ClassNotFoundException**. Diesmal wird das Programm nicht zwangsweise beendet, sondern zu einer Ausnahmebehandlung aufgefordert.

## *9.5 Obligatorische und freiwillige Vorbereitung auf eine Ausnahme*

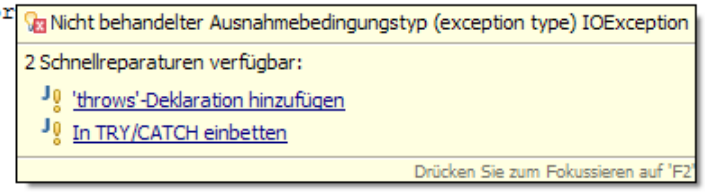
Bei Ausnahmeobjekten aus der Klasse **RuntimeException** und aus daraus abgeleiteten Klassen (siehe Klassenhierarchie in Abschnitt 9.4) ist es dem Programmierer *freigestellt*, ob er sich auf eine Behandlung vorbereiten möchte (**unchecked exceptions**). Alle übrigen Ausnahmeobjekte *müssen* hingegen behandelt werden (**checked exceptions**, z.B. aus der Klasse **IOException**). Beim Einsatz einer Methode, die Probleme über obligatorische Ausnahmen meldet, muss der Aufrufer …

- entweder eine **try**-Anweisung mit geeignetem **catch**-Block verwenden (vgl. Abschnitt 9.2)
- oder im eigenen Definitionskopf das Durchreichen der Ausnahmen ankündigen (vgl. Abschnitt 9.6).

Bei einer Ausnahmeklasse *ohne* Behandlungszwang ist eine solche Vorbereitung nicht erforderlich. Allerdings muss *jede* geworfene Ausnahme (unabhängig von der Klassenzugehörigkeit) beachtet werden, um die Beendigung des Programms durch das Laufzeitsystem zu verhindern.

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 **read()** potentiell eine **IOException** auslöst (siehe JDK-Dokumentation), protestiert der Eclipse-Compiler:

```
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(e);
      }
      System.out.println(key);
   }
}
```

Noch ist der Compiler nicht in der Lage, eine tatsächliche Ausnahmebehandlung einzufordern und akzeptiert auch die folgende, im konkreten Beispiel (einen Tastendruck in Empfang nehmen) durchaus angemessene Variante:

```
try {
   key = System.in.read();
} catch(Exception e) {}
```

Gleich lernen Sie eine Möglichkeit kennen, auf die Behandlung einer obligatorischen Ausnahme zu verzichten und dem Vorgänger in der Aufrufersequenz das Problem zu überlassen.

## 9.6 Ausnahmen auslösen (throw) und deklarieren (throws)

Unsere eigenen Methoden 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. In der folgenden Variante unseres Beispielprogramms zur Fakultätsberechnung wird in der Methode `getFakulArg()` 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 getFakulArg(String instr) throws IllegalArgumentException {
 int arg;
 try {
  arg = Integer.parseInt(instr);
  if (arg < 0  || arg > 170)
  throw new IllegalArgumentException (
       "Unzulaessiges 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)
- 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.

Die Benutzer einer Methode sollten wissen, welche Ausnahmen sie (direkt oder indirekt) auslöst und dann *nicht* lokal behandelt, so dass der Aufrufer ggf. vom Laufzeitsystem zur Ausnahmebe-

handlung aufgefordert wird. Mit der **throws**-Klausel im Methodenkopf kann man den Nutzer einer Methode darüber informieren, z.B.:

```
static int getFakulArg(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 9.4) ist es dem Programmierer freigestellt, ob er die in einer Methode (direkt oder indirekt) ausgelösten, aber nicht behandelten Ausnahmen deklarieren möchte. Alle übrigen Ausnahmen (z.B. **IOException**) müssen entweder behandelt oder deklariert werden. In obigem Beispiel erfolgt die Deklaration freiwillig.

Die aktuelle `getFakulArg()`-Variante informiert den Aufrufer mit einem Ausnahmeobjekt aus der Klasse **IllegalArgumentException**. Um auf dieses (im Methodenkopf deklarierte) Ausnahmeobjekt reagieren zu können, muss der Aufrufer eine **try**-Anweisung verwenden, z.B.:

```
try {
   argument = getFakulArg(args[0]);
} catch (IllegalArgumentException iae) {
   System.out.println(iae.getMessage());
   System.exit(1);
}
```

Dass eine Methode die selbst geworfenen 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 = getFakulArg(args[0]);
} catch (IllegalArgumentException iae) {
   System.out.println(iae.getMessage());
   System.exit(1);
}
```

In Abschnitt 9.5 haben Sie erfahren, dass man beim Aufruf einer Methode, die potentiell *obligatorische* Ausnahmen 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** die Verantwortung auf den Vorgänger in der Aufrufhierarchie abzuschieben. Im Beispielprogramm aus Abschnitt 9.5 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 checked exceptions *weiterleiten*, die von aufgerufenen Methoden stammen.

## 9.7 Ausnahmen definieren

Mit Hilfe von Ausnahmeobjekten kann eine Methode beim Auftreten von Fehlern die aufrufende Methode ausführlich und präzise über Ursachen und Begleitumstände informieren. Dabei muss man sich keinesfalls auf die im Java-API vorhandenen Ausnahmeklassen beschränken, sondern kann auch eigene Ausnahmen definieren, z.B.:

```
public class BadFakulArgException extends Exception {
  protected int error, value;
  protected String instr;
  public BadFakulArgException(String desc, String instr_,
                              int error_, int value_) {
    super(desc);
    instr = instr_;
    error = error_;
    value = value_;
  }
  public String getInstr() {return instr;}
  public int getError() {return error;}
  public int getValue() {return value;}
}
```

Der `BadFakulArgException()`-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 `BadFakulArgException` kann unsere Methode `getFakulArg()` beim Auftreten von irregulären Argumenten neben einer Fehlermeldung noch weitere Informationen auf bequem zugängliche Weise an aufrufende Methoden übergeben:

- in `instr` die zu konvertierende Zeichenfolge
- in `error` einen numerischen Indikator für die Fehlerart
  - 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** haben wir uns für eine *checked exception* entschieden, die im `getFakulArg()`-Methodenkopf deklariert werden *muss*:

```
static int getFakulArg(String instr) throws BadFakulArgException {
  int arg;
  try {
    arg = Integer.parseInt(instr);
    if (arg < 0  || arg > 170)
      throw new BadFakulArgException(
        "Unzulaessiges Argument (erlaubt: 0 bis 170)", instr, 3, arg);
    else
      return arg;
  }
  catch (NumberFormatException e) {
    throw new BadFakulArgException("Fehler beim Konvertieren", instr, 2, -1);
  }
}
```

Ebenso sind `getFakulArg()`-Aufrufer gezwungen, entweder die `BadFakulArgException` in einem **catch**-Block zu behandeln oder ihrerseits im Methodenkopf die potentielle Ausnahme zu deklarieren:

```
public static void main(String[] args) {
  int argument = -1;

  try {
    if (args.length == 0)
      throw new BadFakulArgException("Kein Argument angegeben", "", 1, -1);
    argument = getFakulArg(args[0]);
    double fakul = 1.0;
    for (int i = 1; i <= argument; i++)
      fakul = fakul * i;
    System.out.println("Fakultaet: " + fakul);
  }
  catch (BadFakulArgException 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.

## 9.8 Übungsaufgaben zu Kapitel 9

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 9.2.1).

3) Erstellen Sie eine Variante des Fakultätsprogramms aus Abschnitt 9, die vom Benutzer via Konsole oder **JOptionPane**-Standarddialog (vgl. Abschnitt 3.8) ein Argument entgegen nimmt und bei Eingabefehlern Gelegenheit zur Nachbesserung bietet.

4) Erstellen Sie ausnahmsweise ein Programm, das eine **NullPointerException** auslöst, indem es auf ein nicht existentes Objekt zugreift.

5) 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 weiter gerechnet wird, und am Ende eines längeren Rechenwegs das Ergebnis **Double.NaN** steht. In folgendem Beispiel wird eine Methode

namens `dualog()` zur Berechnung des dualen Logarithmus[45] 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.

| Quellcode | Ausgabe |
|---|---|
| `public class DuaLog {`<br>`  final static double LOG2 = Math.log(2);`<br>`  public static double dualog(double arg) {`<br>`  return Math.log(arg) / LOG2;`<br>`  }`<br><br>`  public static void main(String[] args) {`<br>`  double a = dualog(8);`<br>`  double b = dualog(-1);`<br>`  System.out.println(a*b);`<br>`  }`<br>`}` | `NaN` |

Erstellen Sie eine Variante, die bei ungeeigneten Argumenten eine **IllegalArgumentException** wirft.

[45] 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).

# 10 Interfaces

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 JDK-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, d.h.:

- **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 12 über die Ein- und Ausgabe kennen lernen.
- **Comparable<Double>**
  Analog zu generischen Klassen (vgl. Abschnitt 6.2) unterstützt Java seit der Version 5 auch Interfaces mit Typformalparametern. Weil die Klasse **java.lang.Double** das Interface **Comparable<Double>** im Paket **java.lang** implementiert, ist für ihre Objekte ein Größenvergleich definiert. Folglich können die Objekte in einem **Double**-Array mit der statischen Methode **java.util.Arrays.sort()** bequem sortiert werden, z.B.:

  ```
  Double[] da = new Double[13];
   . . .
  java.util.Arrays.sort(da);
  ```

  Um das Interface **Comparable<Double>** zu implementieren, muss die Klasse **Double** eine Methode mit folgendem Definitionskopf besitzen:

  **public int compareTo(Double** *d***)**

  Wir kennen die Methode **compareTo()** schon (z.B. aus dem Abschnitt über die Klasse **String**) und wissen, dass eine vernünftige Realisation keinen beliebigen **int**-Wert abliefert darf, sondern das Vergleichsergebnis so mitteilen muss:
  - negativ, wenn das angesprochene Objekt kleiner als das Parameterobjekt ist
  - 0, wenn beide gleich sind
  - positiv, wenn das angesprochene Objekt größer als das Parameterobjekt ist

Ein Interface (dt.: eine *Schnittstelle*) dient in der Regel dazu, Verhaltenskompetenzen von Objekten über eine Liste von Methodenköpfen zu definieren. Man kann dieses in Java erstmals verwendete OOP-Sprachelement in erster Näherung als Klasse mit ausschließlich abstrakten Methoden begreifen. Ein Interface ist also ein **Datentyp**. Es lassen sich zwar keine *Objekte* von diesem Datentyp erzeugen, aber *Referenzvariablen* sind erlaubt und als Abstrakationsmittel 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 (mit später bzw. dynamischer Bindung, siehe Abschnitt 8.7).

Implementiert eine Klasse ein Interface, dann …

- muss sie die im Interface enthaltenen Methoden implementieren, wenn keine abstrakte Klasse entstehen soll (vgl. Abschnitt 8.8),
- werden Variablen vom Typ dieser Klasse vom Compiler überall dort akzeptiert, wo der Interface-Datentyp vorgeschrieben ist.

Im Programmieralltag kommen wir auf unterschiedliche Weise mit Schnittstellen in Kontakt, z.B.:

- Verwendung fremder Klassen mit Interface-konformen Verhaltenskompetenzen
  Bei der Verwendung von Objekten aus fremden Klassen in eigenen Methodendefinitionen nutzen wir Verhaltenskompetenzen, die durch Schnittstellen-Verpflichtungen dieser Klassen garantiert sind. Es ist nicht unbedingt erforderlich, die fremden Klassen namentlich zu kennen. Bei einer eigenen Methodendefinition kann es z.B. sinnvoll sein, Parameterdatentypen über Schnittstellen zu definieren. Damit wird Typsicherheit ohne überflüssige Einengung erreicht.
- 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.
- Definition von eigenen Schnittstellen
  Beim Entwurf eines Softwaresystems, das als Halbfertigprodukt (oder Programmgerippe) 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 sicher zu stellen. In diesem Fall spricht man von einem **Framework** (z.B. Java Collection Framework, Hibernate Persistenz Framework). Auch bei einem **Entwurfsmuster** (engl.: **design pattern**), das für eine konkrete Aufgabe bewährte Lösungsverfahren vorschreibt, spielen Schnittstellen oft eine wichtige Rolle.

## *10.1 Interfaces definieren*

Wir behandelt zuerst das im Programmieralltag vergleichweise 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**:[46]

```
public interface Comparable<T> {
  public int compareTo(T o);
}
```

In einem Schnittstellenkopf ist neben dem Schlüsselwort **interface** und dem Typnamen noch der Modifikator **public** mit der üblichen Bedeutung erlaubt.

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. In der Einleitung zum aktuellen Kapitel 10 haben wir z.B. das Interface **Comparable<Double>** betrachtet. Aus dem Abschnitt 6.4.3 über Kollektionen mit Schlüssel-Wert - Elementen kennen Sie das Interface **Map<K,V>** mit zwei Typformalparametern (für *Key* und *Value*).

Im Schnittstellenrumpf werden in der Regel abstrakte Methoden aufgeführt, deren Rumpf durch ein Semikolon ersetzt ist. Dabei werden die Typformalparameter wie gewöhnliche Typbezeichner ver-

[46] 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.

wendet. Mit einer Schnittstelle wird also 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 **Comperable<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 bei einer implementierenden Klasse nur die Einhaltung syntaktischer Regeln sicher stellen, so dass z.B. auch die folgende **compareTo()** - Realisation akzeptiert wird:

```
public int compareTo(Double a) {return 1;}
```

Einige Regeln für Schnittstellen-Definitionen:

- Modifikatoren für Schnittstellen
  - **public**
    Wird **public** nicht angegeben, ist die Schnittstelle nur innerhalb ihres Pakets verwendbar.
  - **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 (durch Komma getrennte) Typformalparameter folgen lassen (siehe Abschnitt 6).
- Vererbung bei Schnittstellen
  Die von Klassen bekannte Vererbung (siehe Abschnitt 8.1) über das Schlüsselwort **extends** wird auch bei Interfaces unterstützt, z.B.:
  ```
  public interface SortedSet<E> extends Set<E> {
      . . .
  }
  ```
  Dabei gilt:
  - 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 {
    . . .
    }
    ```
  - Es gibt keine mit der Urahnklasse **Object** vergleichbare Urahnschnittstelle.

  Bei einer Schnittstelle mit (direkten und indirekten) Basisschnittstellen muss eine implementierende Klasse die Methoden aller beteiligten Schnittstellen realisieren.
- Methodendeklarationen
  In einer Schnittstelle sind alle Methoden grundsätzlich **public** und **abstract**. Die beiden Schlüsselwörter können also weggelassen werden. In der *Java Language Specification* findet sich sogar die Mahnung (Gosling 2005, S. 267):

  > *It is permitted, but strongly discouraged as a matter of style, to redundantly specify the public modifier for interface methods.*

Im Quellcode der wichtigen Schnittstelle **Comparable<T>** findet sich allerdings zur einzigen Methode **compareTo()** diese Definition:

```
public int compareTo(T o);
```

Ein dem Schlüsselwort **public** widersprechender Zugriffsmodifikator ist verboten. Für spezielle Zwecke sind auch Schnittstellen *ohne* Methoden erlaubt (siehe unten).
Statische Interface-Methoden sind verboten.

- 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 ROT = 1, GRUEN = 2, BLAU = 3;
   double PIHALBE = 1.5707963267948966;
}
```

  Implementierende Klassen können auf die Konstanten ohne Angabe des Schnittstellennamens zugreifen. Hier besteht also ein kleiner Vorteil im Vergleich zu Konstanten, die in einer Klasse als **static**, **public** und **final** deklariert werden. Interface-Konstanten können auch von nicht implementierenden Klassen verwendet werden, wobei der Interface-Name samt Punktoperator voranzustellen ist.

- Dateiverwaltung
  Hinsichtlich der Dateiverwaltung gelten dieselben Regeln wie bei Klassen. In der Regel befindet sich ein Interface also in einer eigenen Datei mit der Namenserweiterung **.java**.

Die `Demo`-Schnittstelle in folgendem Beispiel enthält eine **int**-Konstante namens `ONE` und verlangt das Implementieren einer Methode namens `say1()`:

```
interface Demo {
  int ONE = 1;
  int say1();
}
```

Es sind auch Schnittstellen erlaubt, die weder abstrakte Methoden noch Konstanten 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 12.6) höchst 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.

Abschließend soll noch von einer Kuriosität bei manchen Schnittstellendefinitionen im Java Collection Framework 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 Widerspruch zu obigen Erläuterungen über Schnittstellen als *Verpflichtungserklärungen* befindet:

| Method Summary | |
|---|---|
| boolean | **add**(E e)<br>Ensures that this collection contains the specified element (optional operation). |
| void | **clear**()<br>Removes all of the elements from this collection (optional operation). |
| boolean | **contains**(Object o)<br>Returns true if this collection contains the specified element. |
| ... | ... |

Mit diesem Zusatz 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** - Klassendefinition) empfohlen:

```
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 (nämlich die Sichten bzw. Views, siehe Beschreibung der Methode **keySet()** in Abschnitt 6.4.3) einerseits z.B. die Schnittstelle **Collection<E>** erfüllen sollen, aber andererseits keine Strukturveränderungen (z.B. durch Aufnahme neuer Elemente) vornehmen dürfen.

## *10.2 Interfaces implementieren*

Soll für die Objekte einer Klasse angezeigt werden, dass sie auch 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 begrenzte Ähnlichkeit mit namensgleichen früheren Beispielklassen besitzt und die Datenkapselung sträflich vernachlässigt. Sie implementiert das Interface **Comparable<Figur>**, damit `Figur`-Arrays bequem sortiert werden können:

```
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 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 10.1 wiedergegebenen **Comparable<T>** - Definition ist also im letzten 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 linken oberen Ecke verglichen:

- Liegt die angesprochene Figur links vom Vergleichspartner, dann wird –1 zurück gemeldet.
- Haben beide Figuren in der linken oberen Ecke 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 Wege.

Wenn eine implementierende Klasse eine Schnittstellenmethode weglässt (oder abstrakt implementiert), dann resultiert eine abstrakte Klasse, die auch als solche deklariert werden muss (vgl. Abschnitt 8.8).

Weil die Methoden einer Schnittstelle grundsätzlich als **public** definiert sind, und beim Implementieren (wie übrigens auch beim Überschreiben) eine Einschränkungen der Schutzstufe verboten ist, muss beim Definieren von implementierenden Methoden der Zugriffsmodifikator **public** explizit angegeben werden.

Während eine Klasse nur *eine direkte Basisklasse* besitzt, kann sie *beliebig viele Schnittstellen* implementieren, so dass ihre Objekte entsprechend viele Datentypen erfüllen, z.B.:

```
public class HashSet<E>
  extends AbstractSet<E>
  implements Set<E>, Cloneable, java.io.Serializable {
  . . .
}
```

Wie wir inzwischen wissen, wird der Klasse aber nichts geschenkt (mal abgesehen von den Konstanten mancher Schnittstellen), sondern sie muss die Methoden aller Schnittstellen implementieren. Es ist *kein* Problem, wenn zwei implementierte Schnittstellen über Methoden mit identischem Definitionskopf verfügen, weil keine konkurrierenden Realisationen geerbt werden, sondern von der implementierenden Klasse *eine* Realisation neu erstellt werden muss.

Implementiert eine Klasse eine Schnittstelle mit (direkten und indirekten) Basisschnittstellen, dann muss sie die Methoden aller beteiligten Schnittstellen realisieren. Weil z.B. die Klasse **TreeSet<E>** aus dem Java Collection Framework (siehe Abschnitt 6.4.2.2) neben den Schnittstellen **Cloneable** und **Serializable** auch die Schnittstelle **Navigatable<E>** implementiert,

```
public class TreeSet<E> extends AbstractSet<E>
  implements NavigableSet<E>, Cloneable, java.io.Serializable {
 . . .
}
```

sammelt sich einiges an Lasten an, denn **Navigatable<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),
- hat der **AbstractSet<E>** - Programmierer keine zusätzlichen Interface-Verträge abgeschlossen und unvollständig realisiert.

Auch Schnittstellen ändern nichts daran, dass für Java-*Klassen* eine Mehrfach*vererbung* (vgl. Abschnitt 8) ausgeschlossen ist. Allerdings erlauben Schnittstellen in vielen Fällen eine Ersatzlösung, denn:

- Eine Klasse darf beliebig viele Schnittstellen implementieren.
- Bei Schnittstellen ist Mehrfachvererbung erlaubt.

Im Zusammenhang mit dem Thema *Vererbung* ist noch von Bedeutung, dass eine abgeleitete Klasse die in Basisklassen implementierten Schnittstellen erbt. Wird z.B. die Klasse `Kreis` von der oben vorgestellten Klasse `Figur` abgeleitet,

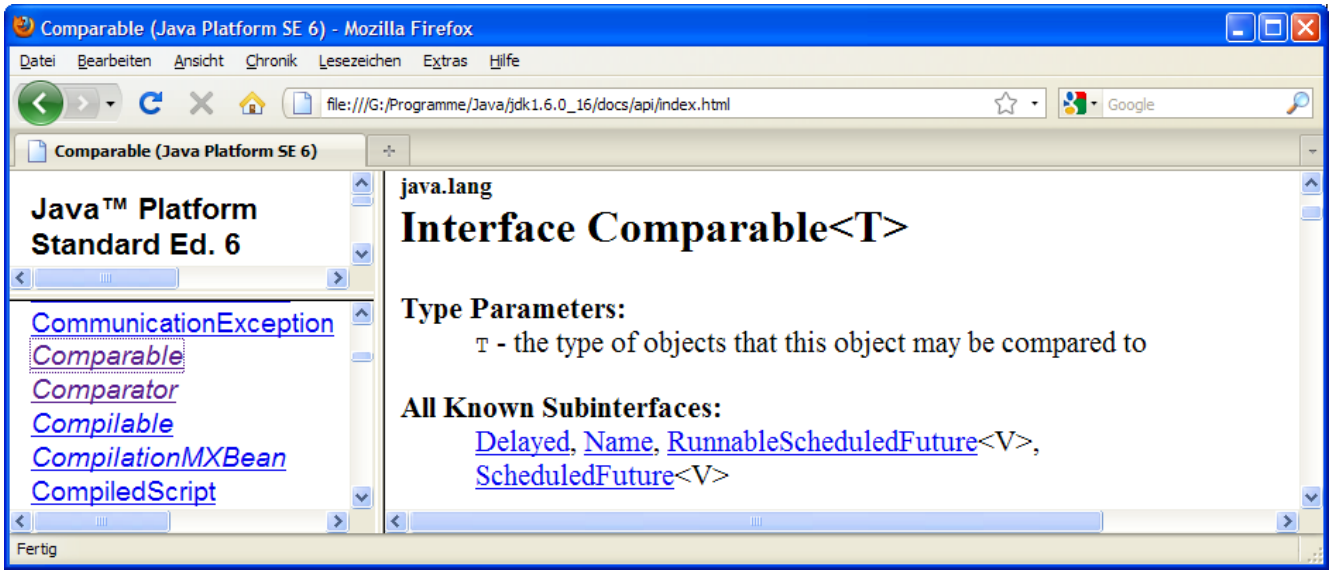
```
public class Kreis extends Figur {
  public int radius;
  public Kreis(String name_, int xpos_, int ypos_, int rad_) {
    super(name_, xpos_, ypos_);
    radius = rad_;
  }
}
```

so 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 Test {`<br>`  public static void main(String[] args) {`<br>`  Kreis[] ka = new Kreis[3];`<br>`  ka[0] = new Kreis("C", 250, 50, 10);`<br>`  ka[1] = new Kreis("B", 150, 50, 20);`<br>`  ka[2] = new Kreis("A", 50, 50, 30);`<br>`  for (Kreis ko : ka)`<br>`  System.out.print(ko.name+" ");`<br>`  java.util.Arrays.sort(ka);`<br>`  System.out.println();`<br>`  for (Kreis ko : ka)`<br>`  System.out.print(ko.name+" ");`<br>`  }`<br>`}` | `C B A`<br>`A B C` |

Die Schnittstelle **Comparable**<`Kreis`> befindet sich übrigens weder im Erbe der `Kreis`-Klasse noch darf sie hier zusätzlich implementiert werden, wozu auch kein Anlass besteht.

In der Java-API – Dokumentation sind die Schnittstellen in der Paketübersicht (unten links) an den kursiv gesetzten Namen zu erkennen, z.B.:




### *10.3 Interfaces als Referenzdatentypen verwenden*

Mit der Definition einer Schnittstelle wird ein neuer Referenzdatentyp vereinbart, der anschließend in Variablendeklarationen und Parameterlisten einsetzbar ist. Eine Referenzvariable des neuen Typs kann auf Objekte jeder Klasse zeigen, welche die Schnittstelle implementiert, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `interface Quatsch {`<br>`  void sagWas();`<br>`}`<br><br>`class Ritter implements Quatsch {`<br>`  void ritterlichesVerhalten() { ... }`<br>`  public void sagWas() {`<br>`  System.out.println("Bin ein Ritter.");`<br>`  }`<br>`}` | `Bin ein Ritter.`<br>`Bin ein Wolf.` |

| Quellcode | Ausgabe |
|---|---|
| `class Wolf implements Quatsch {`<br>`  public void jagdausflug() { ... }`<br>`  public void sagWas() {`<br>`  System.out.println("Bin ein Wolf.");`<br>`  }`<br>`}`<br><br>`class Intereferenz {`<br>`  public static void main(String[] args) {`<br>`  Quatsch[] demintiar = {new Ritter(),new Wolf()};`<br>`  for (Quatsch di : demintiar)`<br>`  di.sagWas();`<br>`  }`<br>`}` | |

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()`).

## *10.4 Annotationen*

Seit der JSE-Version 5 kann man an Pakete, Typen (Klassen, Schnittstellen, Annotationen), Methoden, Konstruktoren, Parameter und lokale Variablen Annotationen anheften, um zusätzliche **Metainformationen** bereit zu stellen, die …

- vor dem Übersetzen,
- beim Übersetzen
- oder zur Laufzeit

berücksichtigt werden können.[47] 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 schlichten An- bzw. Abwesenheit der Annotation, jedoch kann eine Annotation auch Detailinformationen enthalten.

Neben den im Java-API enthalten Annotation (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 10.4.1), und der Compiler erzeugt tatsächlich aus jeder Annotationsdefinition, die nicht auf den Quellcode beschränkt bleiben soll (siehe Abschnitt 10.4.4), ein Interface.

Eine angeheftete Annotation kann das Laufzeitverhalten eines Programms indirekt beeinflussen über ihre Signalwirkung auf Methoden, welche sich über die Existenz bzw. Ausgestaltung der Annotation informieren und ihr Verhalten daran orientieren (siehe Abschnitt 10.4.3). Wir lernen hier eine weitere Technik zur Kommunikation zwischen Programmbestandteilen kennen. In komplexen objektorientierten Softwaresystemen (Frameworks) spielt generell die als *Reflexion* (engl.: *reflection*) bezeichnete Ermittlung von Informationen über Typen zur Laufzeit eine zunehmende Rolle. Dabei leisten Annotationen einen wichtigen Beitrag. Man spricht in diesem Zusammenhang auch von *Meta-Programmierung*.

Viele Annotationen beeinflussen das Verhalten des Compilers, der z.B. durch die Annotation **Deprecated** zur Ausgabe einer Warnung veranlasst wird. Neben dem Compiler und reflektierenden

[47] Wer die Programmiersprache C# kennt, fühlt sich zu Recht an die dortigen *Attribute* erinnert.

Programmbestandteilen kommen auch Fremdprogramme (Werkzeuge) als Adressaten für 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*. Neben Quellcode lassen sich auch andere Daten aufgrund von Annotationen automatisch erstellen (z.B. in XML-Dateien). Im JDK findet sich mit dem *Annotation Processing Tool* (*APT*) ein Konsolenprogramm, das die Erstellung von Quellcode und anderen Daten aufgrund von Annotationen unterstützt.

Unsere Entwicklungsumgebung Eclipse 3 bezeichnet Annotationen als *Anmerkungen*, und man kann daher 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 Typen eine eigene Quellcode- und Bytecodedatei.

### 10.4.1 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 Bestandteile (siehe Abschnitt 10.4.4). 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   maintainer() default "engineer at star soft";
  String[] contributors() default {};
}
```

Als Rückgabetypen sind bei Annotationen erlaubt:

- Primitive Typen
- Die Klassen **String** und **Class**
- Aufzählungstypen
- Annotationstypen
- Eindimensionale Arrays mit einem Basistyp aus der vorgenannten Liste

Verboten sind bei den Annotationselementen:

- Rückgabetyp **void**
- Parameter
- **throws** - Klauseln

Nach dem Schlüsselwort **default** kann zu einem Annotationselement ein Voreinstellungswert angegeben werden, was zwei Vorteile bietet:

- Bei der Annotationsverwendung (siehe Abschnitt 10.4.2) spart man Aufwand, wenn der Voreinstellungswert zugewiesen werden soll, weil man in diesem Fall das Element weglassen kann.
- Bei der Erweiterung einer Annotation um ein Element mit Voreinstellungswert bleiben frühere Verwendungen der Annotation kompatibel.

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 10.4.2) 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 Datentyp) Träger von Annotationen werden (im Beispiel: **Target** und **Retention**) werden, wobei man von *Meta-Annotationen* spricht. Die drei im Beispiel auftauchenden API-Annotationen werden in Abschnitt 10.4.4 näher beschrieben.

Alle Annotationstypen stammen implizit vom Interface **Annotation** im Paket **java.lang.annotation** ab. Eine explizite Ableitung von Annotationstypen ist *nicht* möglich.

### 10.4.2 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 Annotationsinstanz besteht erlaubt aus einem Namen samt Präfix „@“ und einer Elementenliste mit (Name = Wert) - Paaren. Hier wird einer Methode die Annotation `VersionInfo` zugewiesen, deren Definition in Abschnitt 10.4.1 zu sehen war:

```
@VersionInfo(version = "7.1.4", build = 3124, maintainer = "Saft",
    contributors = {"Häcker", "Kwikki"})
public static void meth() {
  // Not yet implemented
}
```

Für die Elementenliste einer Annotationsinstanz gelten folgende Regeln:

- Sie wird analog zu einem Konstruktoraufruf in runde Klammern eingeschlossen
- 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"}
contributors = "Häcker"
```

### 10.4.3 Auswertung per Reflexion

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 10.4.4) entsprechend gesetzt werden:

```
@Retention(RetentionPolicy.RUNTIME)
```

Diese Zeile eignet sich auch für die in Abschnitt 10.4.1 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, maintainer = "Saft",
       contributors = {"Häcker", "Kwikki"})
  public static void meth() {
    // Not yet implemented
  }

  public static void main(String args[]) {
    for (Method meth : AnnoReflection.class.getMethods()) {
      System.out.println("\npublic method "+meth.getName()+"()");
      VersionInfo vi = meth.getAnnotation(VersionInfo.class);
      if (vi != null) {
        System.out.println(" "+vi.version()+" ("+vi.build()+") "+vi.date());
        System.out.print(" "+vi.maintainer()+" ");
        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` angehängte Schlüsselwort **class** wird ein Objekt der Klasse **Class** angesprochen, das diverse Kenntnisse über die Klasse `AnnoReflection` besitzt:

```
AnnoReflection.class
```

- 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
 Saft Häcker Kwikki
```

### 10.4.4 API-Annotationen

Nun werden wichtige Annotationen aus dem Java-API beschrieben, die teilweise im bisherigen Verlauf von Abschnitt 10.4.4 bereits zum Einsatz kamen. Im Paket **java.lang** finden sich u.a. die folgenden Annotationen:

- **Deprecated**
  Diese Annotation wird an veraltete (überholte, abgewertete) Programmbestandteile (z.B. Methoden oder Klassen) geheftet, um Programmierer von ihrer weiteren Verwendung abzuhalten. Eventuell hat sich die Verwendung des Programmelements als problematisch herausgestellt, oder es ist eine bessere Lösung entwickelt worden. Im Kapitel 15 über Multithreading 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 3.1.4) 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 3 sind abgewertete Programmelemente an einer durchgestrichenen Bezeichnung zu erkennen.
- **Override**
  Mit dieser Marker-Annotation kann man seine Absicht bekunden, bei einer Methodendefinition eine Basisklassenvariante zu überschreiben. Misslingt dieser Plan z.B. aufgrund eines Tippfehlers, warnt der Compiler.

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 auf 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, nicht aber zur Laufzeit.
  - **RUNTIME**
    Die Annotation ist auch noch zur Laufzeit verfügbar.

  Um für eine Annotation die in Abschnitt 10.4.3 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.

## *10.5 Übungsaufgaben zu Kapitel 10*

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. Eine Schnittstelle kann auch optionale Methoden enthalten.

2) Man hat in Java auf die Mehrfachvererbung verzichtet, damit keine Klasse *mehrere* Methoden mit *identischen* Definitionsköpfen erben kann, woraus eine Unklarheit über die beim Aufruf tatsächlich auszuführende Methode resultieren würde. Kann es zu analogen Problemen kommen, wenn eine Klasse mehrere Schnittstellen implementiert, welche Methoden mit identischen Definitionsköpfen enthalten?

3) 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.

4) Nennen Sie mindestens fünf generische Interfaces, die im bisherigen Kursverlauf schon eine wesentliche Rolle gespielt haben.

5) Welche Methoden benötigt eine Klasse, um die generische Schnittstelle **Comparator<E>** zu implementieren?

6) Definieren Sie eine generische Methode mit einem Parameter, dessen Typ von einer bestimmten Klasse abstammen und zwei Interfaces implementieren muss.

# 11 GUI-Programmierung mit Swing

Eine Anwendung mit graphischer Bedienoberfläche (engl.: ***G**raphical **U**ser **I**nterface*)[48] 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*[49]. Wie Sie aus dem Abschnitt 4.8 (mit einem Vorausblick auf die Entwicklung graphischer Bedienoberflächen) wissen, bezeichnet man in Java die Komponenten auch als *Beans*. Weil die Steuerelemente intuitiv und in verschiedenen Programmen weitgehend konsistent zu bedienen sind, erleichtern Sie den Umgang mit moderner Software erheblich.

Im Vergleich zu Konsolenprogrammen geht es bei GUI-Anwendungen nicht nur anschaulicher und intuitiver, sondern vor allem auch ereignisreicher und mit mehr Mitspracherechten für den Anwender zu. 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 graphischer 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, Touch Screen etc. praktisch permanent in der Lage ist, unterschiedliche Wünsche zu artikulieren. Ein GUI-Programm präsentiert mehr oder weniger viele Bedienelemente und wartet die meiste Zeit darauf, dass eine der zugehörigen Ereignisbehandlungmethoden durch ein (meistens) vom **Benutzer** ausgelöstes **Ereignis** aufgerufen wird.

Im Vergleich zu einem Konsolenprogramm ist bei einem GUI-Programm die dominante Richtung im Kontrollfluss zwischen Anwendung und Laufzeitsystem invertiert. Die Ereignisbehandlungmethoden 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*".

Bei vereinfachter Betrachtungsweise kann man sagen:

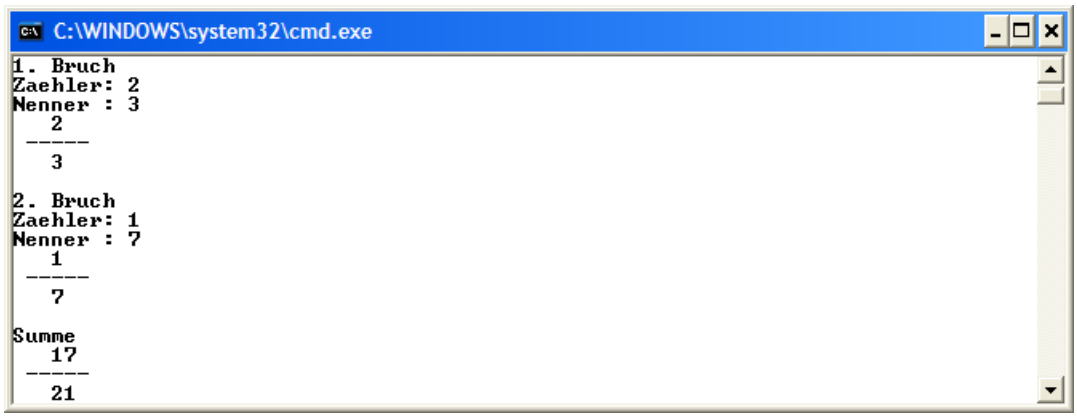
- Eine Konsolenanwendung diktiert den Ablauf und erlaubt dem Benutzer gelegentlich eine Eingabe.
- Eine GUI-Anwendung stellt eine Sammlung von Ereignisbehandlungmethoden dar, wobei die zugehörigen Ereignisse vom Benutzer ausgelöst werden, indem er eines der zahlreichen, für ihn verfügbaren Bedienelemente benutzt.

Betrachten wir zur Illustration eine Konsolen- und eine GUI-Anwendung zum Addieren von Brüchen. Bei der Konsolenanwendung (vgl. Abschnitt 1.1.4)

---

[48] Das dritte Wort in der Bezeichnung *Graphical User Interface* unterscheidet sich in seiner Bedeutung stark von der Kapitelüberschrift *Interface*.

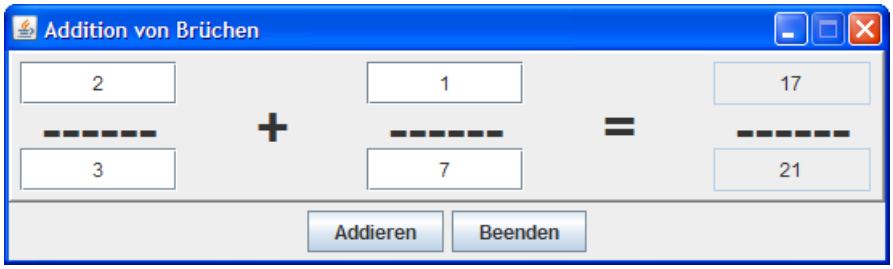
[49] Diese Wortkombination aus *window* und *gadgets* steht für ein *praktisches Fenstergerät*.




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.

Im Unterschied zu diesem **programmgesteuerten Ablauf** wird bei der GUI-Variante




das Geschehen vom Benutzer diktiert, der die sechs Bedienelemente (vier Eingabefelder und zwei Schaltflächen) in beliebiger Reihenfolge verwenden kann, wobei das Programm mit seinen Ereignisbehandlungsmethoden reagiert (**benutzergesteuerter Ablauf**).

Im aktuellen Abschnitt ragen zwei Themen heraus:

- Gestaltung graphischer Bedienoberflächen
- Ereignisbehandlung

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.

In diesem Abschnitt soll ein grundlegendes Verständnis von Aufbau und Funktionsweise einer GUI-Anwendung vermittelt werden. Dabei verzichten wir auf GUI-Design - Assistenten, die einen relativ komplexen Quellcode produzieren, der den Blick auf das Wesentliche erschwert. Im späteren Programmieralltag sollten Sie zur Steigerung der Produktivität eine Entwicklungsumgebung mit graphischem Fensterdesigner verwenden (z.B. Eclipse mit dem Plugin *Visual Editor*, NetBeans mit dem GUI-Designer *Mantisse*). Erste Erfahrungen mit dem Visual Editor haben Sie schon in Abschnitt 4.8 gemacht, der einen frühen Eindruck vom Alltag der Software-Entwicklung mit Java vermitteln sollte. M.E. ist es erst dann sinnvoll, den zu einem Fenster gehörenden Java-Code automatisch erstellen zu lassen, wenn man die Assistentenprodukte verstehen kann. Zudem sind auch beim grafischen Fensterdesign gewisse Kenntnisse der technischen Basis erforderlich. Wer z.B. mit

dem Visual Editor ein Swing-GUI erstellt, muss für jeden Container einen Layout-Manager wählen, wobei die Informationen aus Abschnitt 11.5 recht nützlich sein dürften.

## *11.1 GUI-Lösungen in Java*

In der Java-Standardbibliothek sind leistungsfähige Klassen zur **plattformunabhängigen GUI-Programmierung** mit Hilfe vorgefertigter Steuerelemente enthalten, wobei die Verteilung auf verschiedene Pakete teilweise historisch bedingt ist. Die ursprüngliche, als ***A**bstract **W**indowing **T**oolkit* (AWT) bezeichnete GUI-Technologie wurde in Java 1.2 durch das *Swing Toolkit* erweitert und teilweise ersetzt.

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.

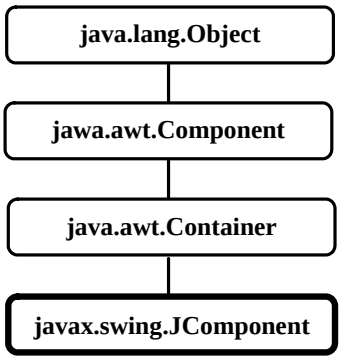
Wir beschränken uns in diesem Kurs auf die Lösungen der Standardbibliothek, die anschließend näher skizziert werden:

- **Abstract Windowing Toolkit** (AWT, enthalten im Paket **java.awt)**
  Das bereits in Java 1.0 vorhandene AWT ist zwar teilweise überholt, stellt aber immer noch wichtige Basisklassen für die aktuelle GUI-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. Offenbar hat diese Lösung ursprünglich viele Ressourcen in Anspruch genommen. Mittlerweile beweist das SWT, dass eine „schwergewichtige“ Lösung durchaus flüssig arbeiten kann.
  Aus der notwendigen Beschränkung auf den damals recht kleinen gemeinsamen Nenner der zu unterstützenden Plattformen resultierte ein beschränkter AWT-Funktionsumfang.
  In diesem Manuskript 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 entfällt, stehen **mehr Komponenten** zur Verfügung.
  - Java-Anwendungen können auf allen Betriebsystemen 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 11.8.1 zu den verfügbaren Alternativen), während die AWT-Komponenten auf das GUI-Design des Betriebssystems festgelegt sind.
  - Weitere Vorteile gegenüber dem AWT sind: QuickInfo-Fenster (Tool Tipps), Steuerung per Tastatur, Unterstützung für die Anpassung an verschiedene Sprachen und Konventionen (Lokalisierung).

## 11.2 Swing im Überblick

### 11.2.1 Leichtgewichtige Komponenten

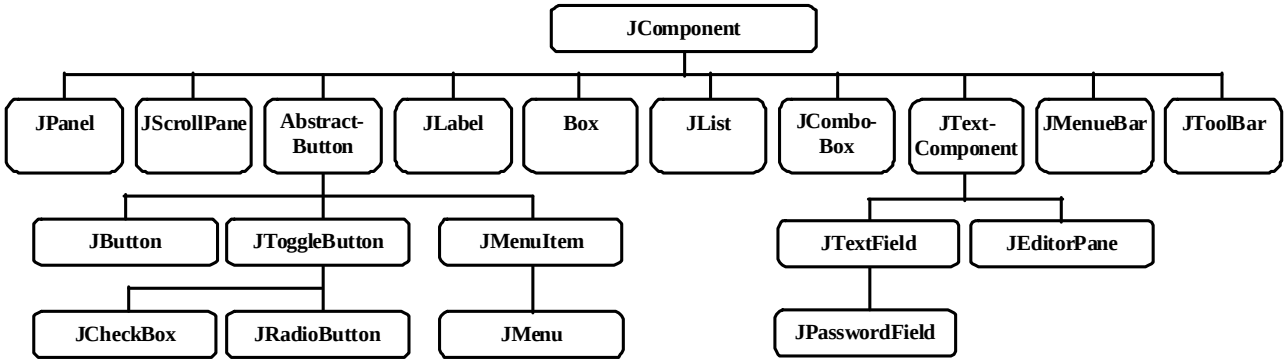
Die leichtgewichtigen (fensterinternen) Swing-Komponenten stammen meist von der Klasse **javax.swing.JComponent** ab, die wiederum zahlreiche Handlungskompetenzen und Eigenschaften über folgende Ahnenreihe erwirbt:




Komponenten sind also auch Objekte, allerdings mit einigen zusätzlichen Kompetenzen:

- Sie unterstützen die Verwendung mit Hilfe von visuellen Entwicklungsumgebungen. Durch die systematische Verfügbarkeit von Methoden zum Lesen und Setzen von Eigenschaften und die Einhaltung bestimmter Bennennungsregeln kann eine Entwicklungsumgebung eine Tabelle mit den Eigenschaftsausprägungen zur Entwurfszeit anbieten. Außerdem
- Sie bieten Ereignisse an, über die sich andere Objekte informieren lassen können (siehe Abschnitt 11.6).
- Visuelle Komponenten[50] treten auf dem Bildschirm in Erscheinung (machen Graphikausgaben) und kommunizieren mit dem Benutzer (reagieren auf GUI-Ereignisse wie Mausklicks).

In der folgenden Abbildung sehen Sie die Abstammungsverhältnisse für die im Abschnitt 11 behandelten **JComponent**-Abkömmlinge:

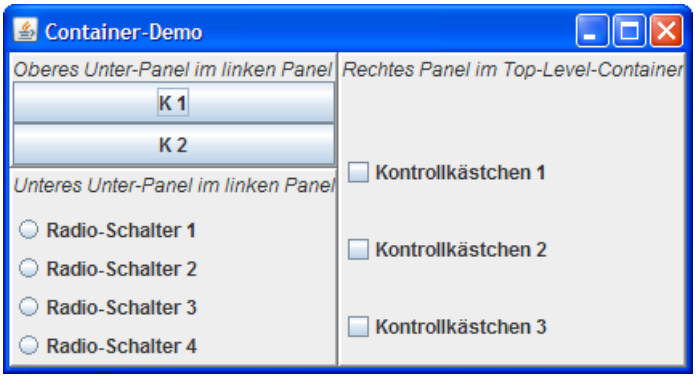



Die Klasse **JTextComponent** und ihre Erweiterungen befinden sich im Paket **javax.swing.text**, alle anderen Klassen befinden sich im Paket **javax.swing**. Wie an den Namen unschwer zu erkennen ist, stehen die meisten Klassen für Bedienelemente, die aus GUI-Systemen wohlbekannt sind (Befehlsschalter, Label etc.).

In der Sammlung befindet sich mit **JPanel** aber auch ein **Container**. Diese Komponente dient zur Aufnahme und damit zur Gruppierung von anderen Swing-Komponenten. 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. Im folgenden Beispiel befinden sich innerhalb eines **JFrame** - Top-Level-Containers (sie-

[50] Nicht-visuellen Komponenten werden wir z.B. bei der Datenbank-Programmierung begegnen.

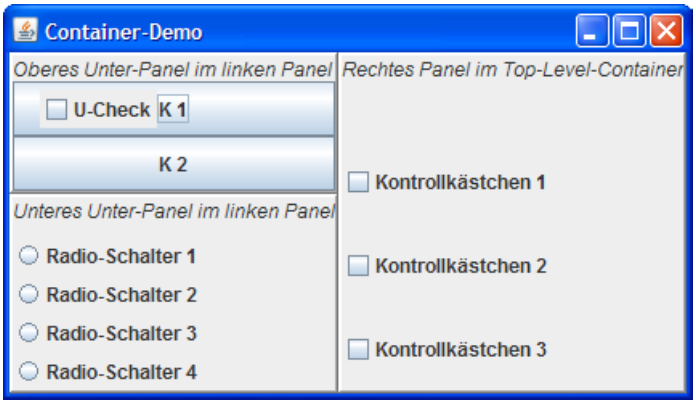
he unten) insgesamt vier **JPanel**-Container, um Komponenten aus den Klassen **JLabel**, **JButton**, **JCheckBox** und **JRadioButton** anzuordnen:




Die Frage, welcher von den beiden Begriffen *Component* und *Container* in Javas GUI-Technologie dem anderen vorgeordnet ist, kann aufgrund des Klassenstammbaums nicht perfekt 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. Tatsächlich erben alle Klassen im **JComponent**-Baum von **java.awt.Container** die Methode **add()** zum „Einfüllen“ von Komponenten. Die Swing-Komponenten sind also allesamt Container. Man kann z.B. ein Kontrollkästchen in einen Befehlsschalter einbauen:




Allerdings fördern derartige Konstruktionen nicht unbedingt die Bedienbarkeit eines Programms.

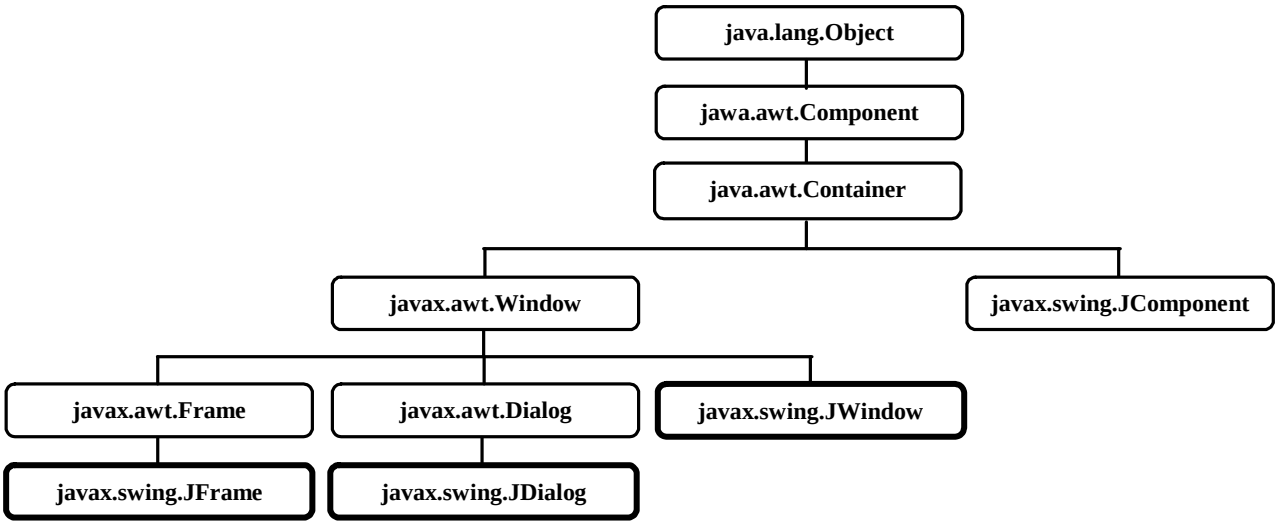
Aus Zeitgründen können leider viele attraktive Swing-Komponenten im Kurs nicht behandelt werden, z.B.:

- **JTree**, **JTable**
- **JSplitPane**, **JTabbedPane**
- **JPopupMenu**

Gute Darstellungen findet man z.B. bei Krüger & Stark (2007), Ullenboom (2009) und im Java Tutorial (Sun Microsystems 2009).

### 11.2.2 Top-Level - Container

Jedes Programm mit Swing-GUI benötigt mindestens einen Top-Level - Container, der als Fenster zu der vom Betriebssystem verwalteten Bedienoberfläche dient 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:




In Abschnitt 11 werden alle Beispielprogramme als Top-Level - Container eine Komponente vom Typ **JFrame** benutzen, die ein **Rahmenfenster** mit folgender Ausstattung realisiert (siehe obige Container-Beispielprogramme):

- **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)

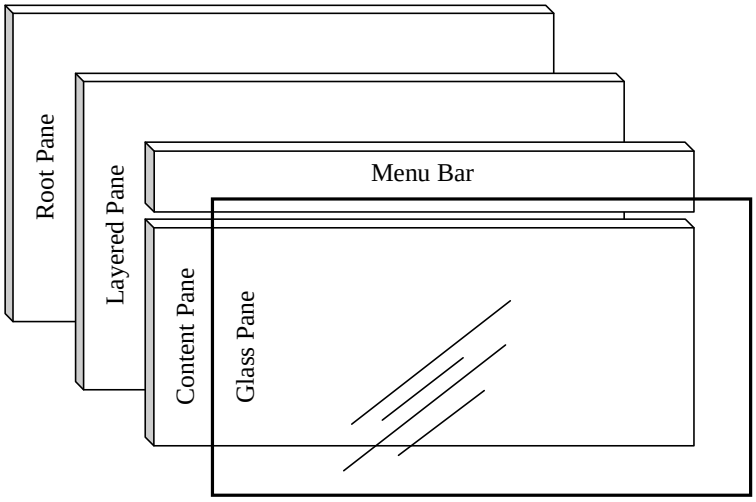
Mit der Klasse **JDialog** werden *Dialogfenster* realisiert, denen im Vergleich zum Rahmenfenster die Titelzeilenbedienelemente zum Maximieren und zum Minimieren fehlen. Sie können zwar auch als selbständige Fenster einer Anwendung arbeiten, werden aber meist in Rahmenfenster-Anwendungen für bestimmte 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.

Bei den später zu behandelnden Java-Applets, die im Rahmen eines Web-Browser - Fensters ausgeführt werden, kann die Klasse **JApplet** einen Swing-basierten Top-Level - Container realisieren.

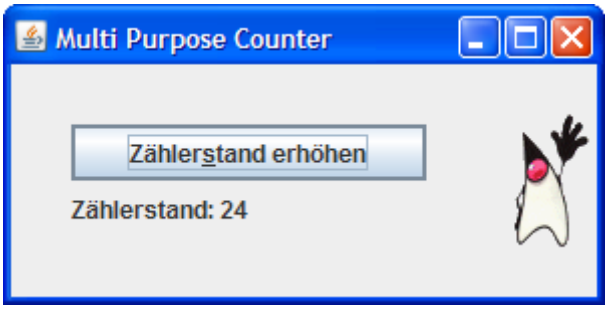
Die Top-Level - Container benutzen zur Verwaltung ihrer Komponenten einen leichtgewichtigen Container aus der Klasse **JRootPane**, der wiederum Container aus den Klassen **JLayeredPane**, **JPanel** und **JMenuBar** als Mitgliedsobjekte enthält. Bei aufgabenorientierter Betrachtung kann man die strukturellen und funktionalen Beziehungen zwischen den Objekten durch ein Scheibenmodell skizzieren (vgl. Java-Tutorial, Sun Microsystems 2009):




Wir werden meist mit der **Content Pane** (Inhaltsschicht) arbeiten und dort die Bedienelemente platzieren (mit Ausnahme von Menüzeilen). Für einige Verabredungen (z.B. Auswahl einer voreingestellten Schaltfläche, welche auf die **Enter**-Taste wie auf einen Mausklick reagieren soll) ist die **Root Pane** zuständig (ein Objekt der Klasse **JRootPane**). Die **Layered Pane** (ein Objekt der Klasse **JLayeredPane**) verwaltet die Inhaltesschicht und die optionale Menüzeile (ein Objekt der Klasse **JMenuBar**). Außerdem lässt sich mit ihrer Hilfe die Schichtung (Z-Anordnung) der Komponenten beeinflussen. Die **Glass Pane** ermöglicht das Übermalen von mehreren Komponenten sowie das Abfangen von GUI-Ereignissen. Hier landen z.B. Tool-Tip - Texte (vgl. Abschnitt 11.4.5.1). Wie ein Blick in den API-Quellcode zeigt, sind Content Pane und Glass Pane Objekte aus der Klasse **JPanel**.

## *11.3 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:




Den Quellcode werden wir im weiteren Verlauf von Abschnitt 11 vollständig besprechen:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class MPC extends JFrame {
   private final static String labelPrefix = "Zählerstand: ";
   private final static String titel = "Multi Purpose Counter";
   private JLabel lblText;
   private JPanel panLeft;
   private JButton cbCount;
   private int numClicks = 0;
   private final byte maxIcon = 3;
   private ImageIcon[] icons;
   private JLabel lblIcon;
   private byte iconInd = 0;
```

```
MPC() {
   super(titel);

   Container cont = getContentPane();

   cbCount = new JButton("Zählerstand erhöhen");
   cbCount.setMnemonic(KeyEvent.VK_S);
   getRootPane().setDefaultButton(cbCount);
   cbCount.setToolTipText("Befehlsschalter");

   lblText = new JLabel(labelPrefix + "0    ");
   lblText.setToolTipText("Label mit Text");

   panLeft = new JPanel();

   panLeft.setBorder(BorderFactory.createEmptyBorder(30, 30, 30, 30));

   panLeft.setLayout(new GridLayout(0, 1));

   panLeft.add(cbCount);
   panLeft.add(lblText);

   cont.add(panLeft, BorderLayout.CENTER);

   icons = new ImageIcon[maxIcon];
   icons[0] = new ImageIcon("duke.gif");
   icons[1] = new ImageIcon("fight.gif");
   icons[2] = new ImageIcon("snooze.gif");

   lblIcon = new JLabel(icons[0]);
   lblIcon.setToolTipText("Label mit Icon");

   cont.add(lblIcon, BorderLayout.EAST);

   cbCount.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         numClicks++;
         lblText.setText(labelPrefix + numClicks);
         if (iconInd < maxIcon-1)
            iconInd++;
         else
            iconInd = 0;
         lblIcon.setIcon(icons[iconInd]);
      }
   });

   addWindowListener(new WC());

   setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);

   setSize(300, 150);

   this.setVisible(true);
}

public static void main(String[] args) {
   new MPC();
}
```

```
    class WC extends WindowAdapter {
      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);
      }
    }
}
```

Zu Beginn werden alle Klassen aus den Paketen **javax.swing**, **java.awt** und **java.awt.event** importiert, um sie bequem ansprechen zu können.

Kernstück des Programms ist die von **JFrame** abgeleitete Klasse `MPC`. In der **main()**-Methode wird ein Objekt dieser Klasse erzeugt:

```
public static void main(String[] args) {
    new MPC();
}
```

Weil das Rahmenfenster mit seinen Steuerelementen ein vergleichsweise komplexes Objekt darstellt, hat der `MPC`-Konstruktor einige Arbeit:

- Das Fenster erhält seinen Titel durch expliziten Aufruf des Basisklassen-Konstruktors:
  ```
  super(titel);
  ```
- Um bequem auf die Inhaltsschicht (engl.: *content pane*) des Rahmenfensters zugreifen zu können, wird mit der **JFrame**-Methode **getContentPane()** eine lokale Referenzvariable auf diesen Container angelegt:
  ```
  Container cont = getContentPane();
  ```
  In der folgenden Anweisung wird die Inhaltsschicht per **add()**-Methodenaufruf gebeten, die Komponente `panLeft` aufzunehmen:
  ```
  cont.add(panLeft, BorderLayout.CENTER);
  ```
  Dass der folgende, direkt an die **JFrame**-Komponente gerichtete Methodenaufruf
  ```
  add(panLeft, BorderLayout.CENTER);
  ```
  denselben Zweck erreicht, liegt an den seit Java 5 vorhandenen Bequemlichkeitsüberschreibungen in der Klasse **JFrame**.[51] Wir werden im Kurs der Klarheit halber die letztlich zuständige Inhaltsschicht meist explizit ansprechen.
- Wie der Konstruktor die Komponenten des Rahmenfensters erzeugt, konfiguriert und positioniert sowie die Ereignisbehandlung vorbereitet, wird gleich im Detail erklärt.
- 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:
  ```
  setSize(300, 150);
  setVisible(true);
  ```

---

[51] Wer neugierig darauf ist, wie in der **JFrame**-Klassendefinition die **add()**-Aufrufe weitergeleitet werden, sollte die Datei **JFrame.java** öffnen und z.B. einen Blick auf die Methode **addImpl()** werfen:

```
protected void addImpl(Component comp, Object constraints, int index) {
    if(isRootPaneCheckingEnabled()) {
        getContentPane().add(comp, constraints, index);
    } else {
        super.addImpl(comp, constraints, index);
    }
}
```

> 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.

Wie in Abschnitt 0 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 sehr gering ist, verzichten wir (in Übereinstimmung mit den meisten Lehrbüchern) vorläufig auf eine absolut wasserdichte, aber etwas aufwändigere Lösung. Nach dem **Visible(true)**-Aufruf dürfen im Fensterkonstruktor aber keine GUI-Manipulationen mehr stattfinden. Weil die bald vorzustellenden Ereignisbehandlungsmethoden im Ereignisverteilungs-Thread ausgeführt werden, gelten hier keine Zugriffsbeschränkungen. Das Single-Thread - Design des Swing-Frameworks wird erst dann zum relevanten Thema, wenn wir uns in Kapitel 15 mit Multithreading-Programmierung beschäftigen.

## *11.4 Bedienelemente (Teil 1)*

Das vorliegende Manuskript behandelt in zwei Portionen elementare Swing-Komponenten. Eine optische Präsentation *aller* Swing-Komponenten finden Sie im Java-Tutorial (Sun Microsystems 2009) über:

Creating a GUI with Swing > Using Swing Components >
A Visual Index to the Swing Components

### 11.4.1 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:

```
private JLabel lblText;
  . . .
lblText = new JLabel(labelPrefix + "0    ");
```

Für Textänderungen im Programmablauf verwendet man die **JLabel**-Methode **setText()**, z.B.:

```
lblText.setText(labelPrefix + numClicks);
```

Zur Steuerung der Textausrichtung dient die Methode **setHorizontalAlignment()**, z.B.:

```
status.setHorizontalAlignment(SwingConstants.CENTER);
```

Das zweite Label im Swing-Einstiegsbeispiel dient zur Anzeige von GIF-Dateien, die von **ImageIcon**-Objekten repräsentiert werden:

```
ImageIcon[] icons;
  . . .
icons = new ImageIcon[maxIcon];
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 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:

```
JLabel lblIcon;
  . . .
lblIcon = new JLabel(icons[0]);
```

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.

### 11.4.2 Befehlsschalter

Die Syntax zum Deklarieren bzw. Erzeugen eines Befehlsschalters mit Beschriftung bietet keinerlei Überraschungen:

```
private JButton cbCount;
  . . .
cbCount = new JButton("Ich mag Swing!");
```

Mit der **JButton**-Methode **setMnemonic()** kann eine **Alt**-Tastenkombination als Äquivalent zum Mausklick auf den Schalter festgelegt werden, z.B. **Alt+S**:

```
cbCount.setMnemonic(KeyEvent.VK_S);
```

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 11.2.2) zu richten, z.B.:

```
getRootPane().setDefaultButton(cbCount);
```

### 11.4.3 Untergeordnete Container

Eine **JPanel**-Komponente kann als untergeordneter Container für Swing-Komponenten dienen und dabei selbst in einem übergeordneten Container untergebracht werden, z.B. im Top-Level - Container eines **JFrame**-Fensters. Im unserem Beispiel wird ein **JPanel**-Behälter namens `panLeft` verwendet:

```
private JPanel panLeft;
  . . .
panLeft = new JPanel();
```

Um eine Komponente in einen Container aufzunehmen, benutzt man eine der zahlreichen **add()**-Überladungen, z.B.:

```
panLeft.add(cbCount);
panLeft.add(lblText);
```

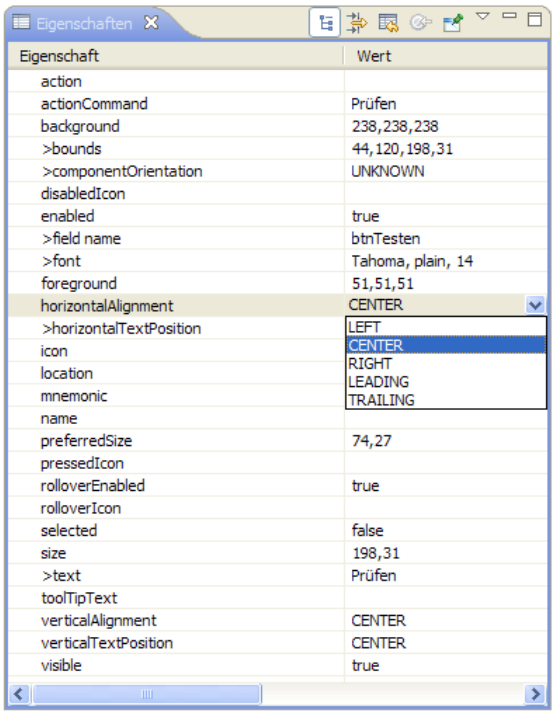
Für die Verteilung der im Container enthaltenen Komponenten auf die verfügbare Fläche ist der Layout-Manager des Containers verantwortlich (siehe Abschnitt 11.5).

### 11.4.4 Elementare Eigenschaften von Swing-Komponenten

In diesem Abschnitt werden **Component**-Methoden aufgelistet, die elementare Eigenschaften von Swing-Komponenten beeinflussen:

- **public void setAlignmentX(float** *horizAlignment***)**
  **public void setAlignmentY(float** *vertAlignment***)**
  Diese Methoden legt die von einer Komponente *gewünschte* horizontale (vertikale) Ausrichtung in der heimatlichen Container-Zelle fest. Bei einem von Wert 0,0 bevorzugt die Komponente den linken (oberen) Rand. Dem maximalen Wert von 1,0 entspricht eine Vorliebe für den rechten (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);
  ```
  Ob die gewünschte Orientierung Realität wird, hängt vom Layout-Manager des Containers ab, in dem sich eine Komponente befindet (vgl. Abschnitt 11.5).
- **public void setMinimumSize(Dimension** *ausdehnung***)**
  **public void setMaxmumSize(Dimension** *ausdehnung***)**
  **public void setPreferredSize(Dimension** *ausdehnung***)**
  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**) hat, z.B.:
  ```
  lblText.setMaximumSize(new Dimension(100, 50));
  ```
  Nach der Fixierung liefern die zugehörigen **get**-Methoden (z.B. **getPreferredSize()**) an Stelle des voreingestellten Verhaltens den festen Wert. Ob die gewünschten Größen Realität werden, hängt von dem für eine Komponente zuständigen Layout-Manager ab (vgl. Abschnitt 11.5).
- **public void setBackground(Color** *farbe***)**, **public void setForeground(Color** *farbe***)**
  Mit dieser Methode lässt sich die Hinter- bzw. Fordergrundgrundfarbe 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 generell verfügbaren Schriftarten (SansSerif, Serif, Monospaced) vermeidet man eine Abhängigkeit von der lokalen Systemausstattung. Werden die Schriftarten über Konstanten der Klasse **Font** angesprochen statt über Zeichenfolgen (z.B. `"Ariaal"`), sind Missverständnisse durch Tippfehler ausgeschlossen.
- **public void setVisible(boolean** *visible***)**
  Mit dieser Methode lässt sich eine Komponente (un)sichtbar machen.

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 Visual Editor in Eclipse präsentiert z.B. zu einer **JButton**-Komponente die folgende Eigenschaftstabelle:




Wer die Eigenschaften von Komponenten *zur Laufzeit* beeinflussen möchte, kommt allerdings um den Aufruf der **set**-Methoden nicht herum.

### 11.4.5 Zubehör für Swing-Komponenten

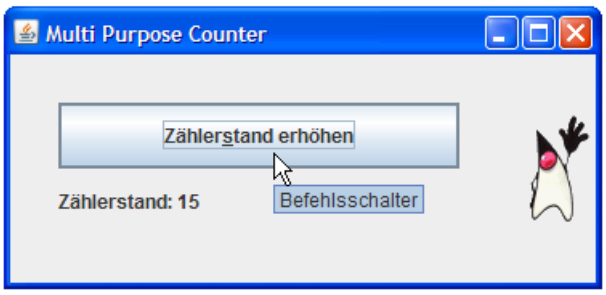
Es sind zahlreiche Möglichkeiten verfügbar, das optische Erscheinungsbild und die Bedienbarkeit von Swing-Komponenten zu verbessern.

#### *11.4.5.1 Tool-Tip - Text*

Mit der **JComponent**-Methode **setToolTipText()** kann man Tool-Tipps (QuickInfos) zu einzelnen Steuerelementen definieren, z.B.:

```
cbCount.setToolTipText("Befehlsschalter");
```

Diese erscheinen in einem PopUp-Fenster, wenn der Mauszeiger über einer betroffenen Komponente verharrt:

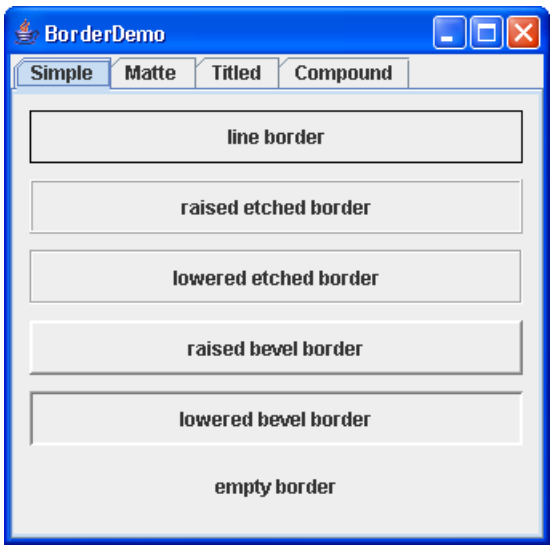



#### *11.4.5.2 Rahmen*

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 11.4.3), etwas „Luft“:

```
panLeft.setBorder(BorderFactory.createEmptyBorder(30, 30, 30, 30));
```

Den Quellcode zu der folgenden Rahmen-Musterschau:



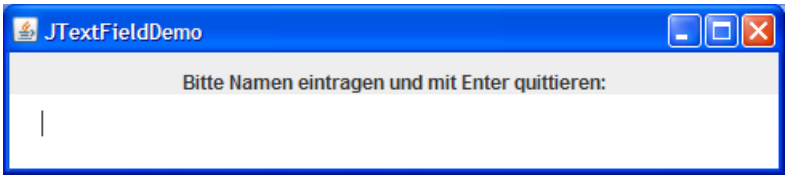

finden Sie über die Webseite:

http://java.sun.com/docs/books/tutorial/uiswing/components/border.html

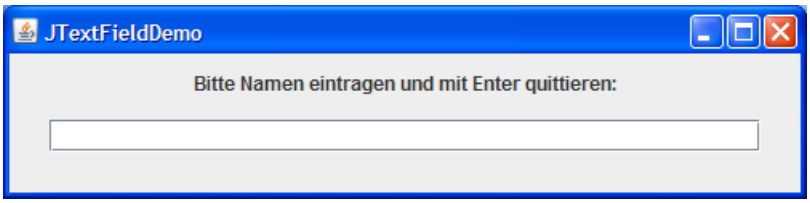
Obwohl die Methode **setBorder()** in der Klasse **JComponent** definiert ist, arbeitet sie nur bei **JPanel** und **JLabel** erwartungsgemäß (vgl. API-Dokumentation). Um andere Bedienelemente zu beranden, geht man am besten so vor:

- Bedienelement in einen **JPanel**-Container stecken.
- **setBorder()** auf den **JPanel**-Container anwenden.

Ein Rahmen um ein Texteingabefeld (**JTextField**, vgl. Abschnitt 11.7.1) wird von Swing wenig sinnvoll realisiert:




Bei einem Texteingabefeld im eingerahmten **JPanel**-Container erhält man das gewünschte Ergebnis:

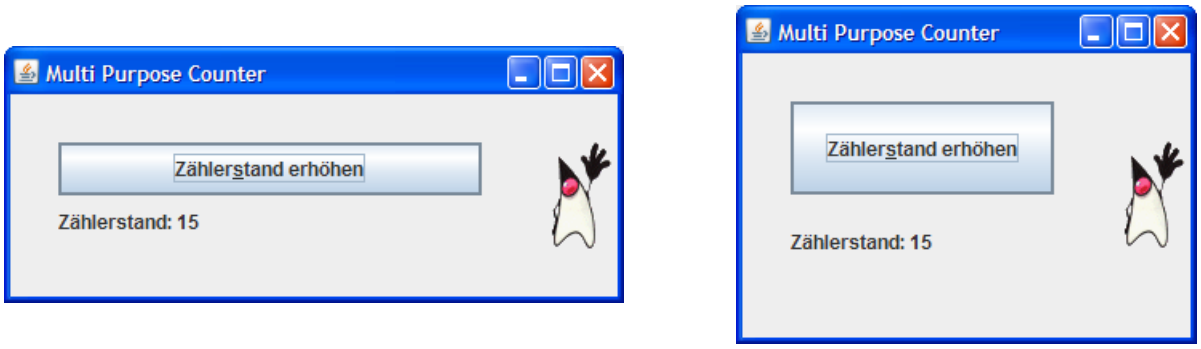



## *11.5 Die Layout-Manager der Container*

Der **Layout-Manager** eines Container 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 Orientierung, welche über die **Component**-Methoden **getPreferredSize()**, **getMinimumSize()**, **getAlignmentX()** etc. in Erfahrung zu bringen sind. Für das Innenleben von enthaltenen 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.

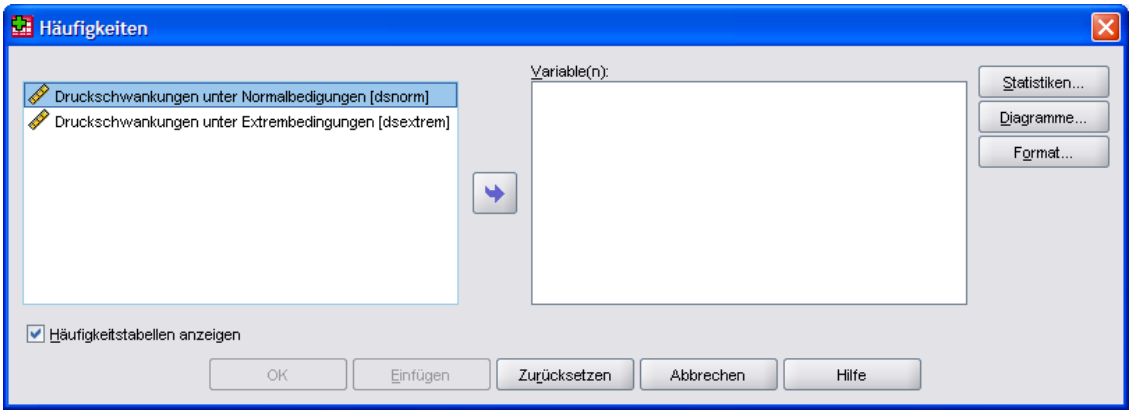
Durch Verschachteln von Containern, für die jeweils ein spezieller Layout-Manager engagiert werden kann, sollte sich fast jede Designidee verwirklichen lassen.

Bei einer Änderung der Container-Größe 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). Im SPSS 15 erschweren Dialogboxen mit fester Größe gelegentlich die Unterscheidung von Variablen mit identisch startender Bezeichnung:


Im Java-Programm SPSS 17 lässt sich die Breite der äquivalenten Dialogbox so wählen, dass die Bezeichnungen vollständig Platz finden:




Man kann aber auch auf die Dienste eines Layout-Managers verzichten und alle Layout-Details (Positionen und Größen) absolut festzulegen (siehe Abschnitt 11.5.5). Dann sollte mit dem folgenden Aufruf der **JFrame**-Methode **setResizable()** eine Änderung der Fenstergröße durch den Benutzer verhindert werden:

```
setResizable(false);
```

Einige Layout-Manager können in diesem Manuskript aus Zeitgründen nicht behandelt werden, z.B.:

- **GridBagLayout**
  Dieser Layout-Manager ist ebenso flexibel wie kompliziert. Im Unterschied zum **GridLayout** (vgl. Abschnitt 11.5.2) 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 graphischen Fenster-Designer *Matisse* in der Entwicklungsumgebung *NetBeans* eingeführt, eignet sich aber auch für das direkte Kodieren.

### 11.5.1 BorderLayout

In unserem Beispielprogramm bleiben bei nahezu beliebigen Veränderungen des Anwendungsfensters (siehe oben) die folgenden räumlichen Relationen erhalten:

- Das Text-Label befindet sich senkecht unter dem Befehlsschalter.
- Das Icon-Label erscheint rechts neben den beiden anderen Komponenten und ist vertikal zentriert.

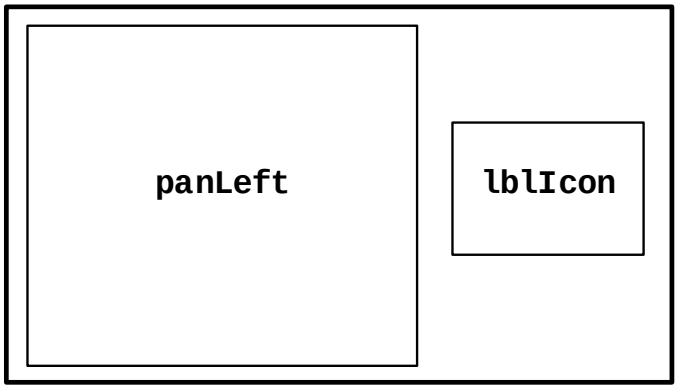
Eine wesentliche Voraussetzung für dieses Verhalten sind die beiden beteiligten Container und ihre Layout-Manager. Das Anwendungsfenster (Klasse `MPC`, abgeleitet aus **JFrame**) ist ein (schwergewichtiger) Top-Level - Container und enthält:

- den untergeordneten (leichtgewichtigen) Container `panLeft` aus der Klasse **JPanel**
- das **JLabel**-Objekt `lblIcon`

Diese beiden leichtgewichtigen Komponenten sind über **add()**-Methodenaufrufe an ihrem Standort gelangt:

```
cont.add(panLeft, BorderLayout.CENTER);
cont.add(lblIcon, BorderLayout.EAST);
```

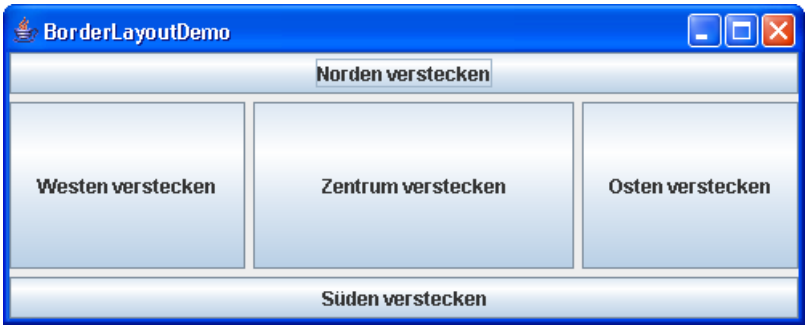
Die Aufrufe richten sich an die **JFrame**–Inhaltsschicht, ein über die Referenzvariable `cont` ansprechbares **Container**-Objekt. Es resultiert die folgende räumliche Anordnung:




Dafür sorgt der **Layout-Manager** der **JFrame**-Komponente `MPC` (genauer: der zugehörigen 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.

Welche Plätze ein **BorderLayout**-Objekt insgesamt für einen Container verwalten kann, zeigt folgendes Beispielprogramm, das an jeder möglichen Position einen Befehlsschalter enthält:



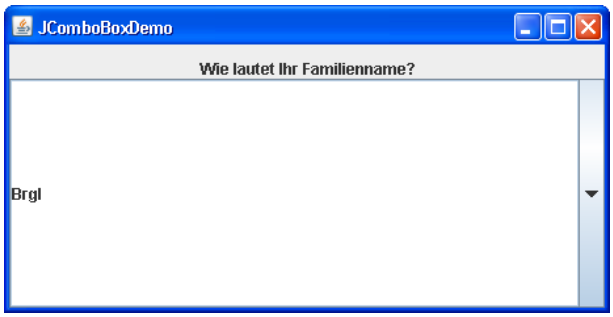

Für die Abstände zwischen den Komponenten (horizontal und vertikal jeweils 5 Pixel) wurde hier durch einen speziellen **BorderLayout**-Konstruktor gesorgt:

```
BorderLayout layout = new BorderLayout(5,5);
```

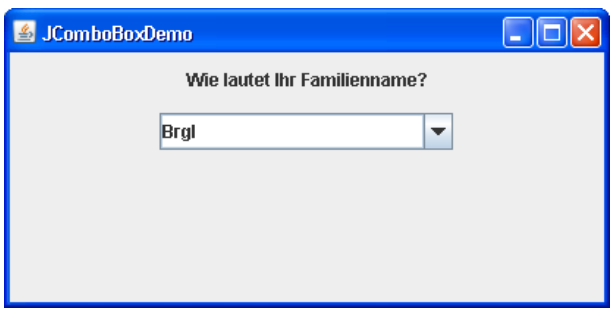
Welche Gestaltungsmöglichkeiten ein **BorderLayout** durch sein Verhalten bei teilweise unbesetzten Positionen, sollten Sie durch Probieren herausfinden. Es soll noch einmal daran erinnert werden, dass ein gelungenes Layout oft durch das hierarchische Schachteln von Containern mit jeweils passend gewähltem Layout-Manager zustande kommt.

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**, **JEditorPane**) 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 11.7.5):




Der bei einem **JPanel**-Container voreingestellte **FowLayout** - Manager (siehe Abschnitt 11.5.3) 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.:



### 11.5.2 GridLayout

Im Einstiegsbeispiel wird für das **JPanel**-Objekt `panLeft`, das den Befehlsschalter und das Textlabel 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 verwalten kann 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 freien 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 ($2 \times 4$) - **GridLayout** – Manager mit Zwischenabständen von jeweils 5 Pixeln zugeordnet:

```
getContentPane().setLayout(new GridLayout(2,4,5,5));
```

Der verfügbare Platz wird von einem **GridLayout**-Manager 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, behalten 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.:


### 11.5.3 FlowLayout

Das recht simple und betagte **FlowLayout**, dient bei vielen Containern (z.B. **JPanel**) als Voreinstellung (bei Verzicht auf die explizite Spezifikation eines Layout-Managers per Konstruktor oder **setLayout()**). Es ordnet die Komponenten nebeneinander an, bis ein „Zeilenumbruch“ erforderlich wird:



An Stelle der horizontalen Zentrierung ist alternativ auch eine links- bzw. rechtsbündige Anordnung möglich, 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.

### 11.5.4 BoxLayout

Ein **BoxLayout**-Manager kann die Komponenten im verwalteten Container vertikal (übereinander) oder horizontal (nebeneinander) anordnen, z.B.:


Er berücksichtigte dabei nach Möglichkeit die Wünsche der Komponenten bzgl. Größe und Ausrichtung. Bei horizontaler Anordnung findet im Unterschied zum **FlowLayout** kein „Zeilenumbruch“ statt.

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 gleich zweimal anzugeben hat, 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:

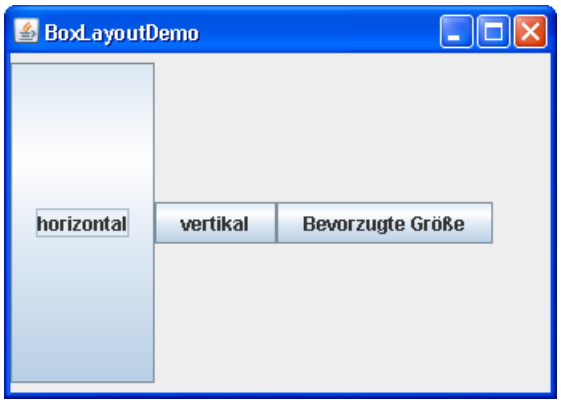
- **X_AXIS**
  Horizontale Anordnung von links nach rechts
- **Y_AXIS**
  Vertikale Anordnung von oben nach unten
- **LINE_AXIS**, **PAGE_AXIS**
  Diese Optionen unterstützen die Internationalisierung von Software und beachten die Komponentenorientierung des betroffenen Containers, welche die Anordnung von Textelementen regelt. Dazu besitzt die Klasse **Container** ein Mitgliedsobjekt der Klasse **ComponentOrientation**, das per **set**-Methode festgelegt und per **get**-Methode ermittelt werden kann. 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:

| Textanordnung des Containers | **Boxlayout**-Komponentenanordnung bei | |
|---|---|---|
| | **LINE_AXIS** | **PAGE_AXIS** |
| horizontal (zeilenweise), Wörter von links nach rechts (Westeuropa) | von links nach rechts | von oben nach unten |
| horizontal, (zeilenweise) Wörter von rechts nach links (Arabisch, Hebräisch) | von rechts nach links | |
| vertikal (von oben nach unten) Spalten von links nach rechts (Mongolei) | von oben nach unten | von links nach rechts |
| vertikal (von oben nach unten) Spalten von rechts nach links (Japan, China, Korea) | | von rechts nach links |

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 (siehe oben).

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 (siehe oben).

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);
```

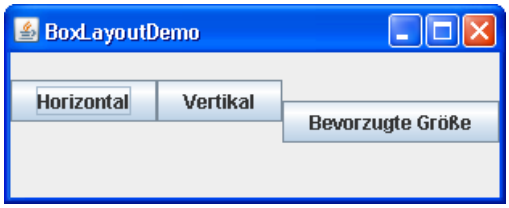
Sind alle Komponenten im Container so ausgerichtet, führt das vertikale **BoxLayout** zum folgenden Ergebnis:


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 untauglichen Ergebnissen, z.B.:




Die von **JComponent** abstammende Klasse **Box** realisiert einen Container mit voreingestelltem **BoxLayout** und bietet einige statische Methoden zur verbesserten Komponentenanordnung. Im Konstruktor ist die gewünschte Komponentenausrichtung anzugeben, z.B.:

```
boxContainer = new Box(BoxLayout.X_AXIS);
```

Mit der **Box**-Methode **createHorizontalStrut()** erhält man eine unsichtbare Komponente mit fester Breite. Durch Einfügen solcher Komponenten, z.B.:

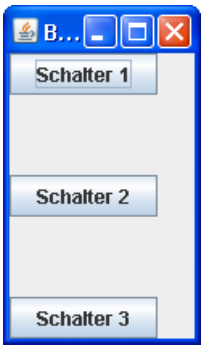
```
cb1 = new JButton("Schalter 1");
boxContainer.add(cb1);
boxContainer.add(Box.createHorizontalStrut(50));
cb2 = new JButton("Schalter 2");
boxContainer.add(cb2);
boxContainer.add(Box.createHorizontalStrut(50));
cb3 = new JButton("Schalter 3");
boxContainer.add(cb3);
```

schafft man feste Abstände zwischen horizontal angeordneten Komponenten:


Mit der **Box**-Methode **createVerticalStrut()** erhält man eine unsichtbare Komponente mit fester Höhe. Durch Einfügen solcher Komponenten, z.B.:

```
cb1 = new JButton("Schalter 1");
boxContainer.add(cb1);
boxContainer.add(Box.createVerticalStrut(50));
cb2 = new JButton("Schalter 2");
boxContainer.add(cb2);
boxContainer.add(Box.createVerticalStrut(50));
cb3 = new JButton("Schalter 3");
boxContainer.add(cb3);
```

schafft man feste Abstände zwischen vertikal angeordneten Komponenten:




Mit der **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.

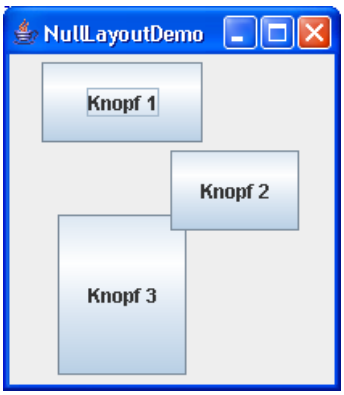
```
boxContainer.add(Box.createHorizontalGlue());
```

schafft man variable (elastische) Abstände zwischen horizontal angeordneten Komponenten:


Mit der **Box**-Methode **createVerticalGlue()** erreicht man dieselbe Elastizität in vertikaler Richtung.

### 11.5.5 Freies Layout

Man kann auf die Dienste eines Layout-Managers verzichten und die Positionen bzw. 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 **setBounds()** festlegen

Dies wird in folgendem Programmfragment demonstriert:

```
  .   .   .
class NullLayoutDemo extends JFrame {
  private JButton k1, k2, k3;

  NullLayoutDemo() {
    super("NullLayout-Demo");

    Container cont = getContentPane();
    cont.setLayout(null);

    k1 = new JButton("Knopf 1");
    k1.setBounds(20,5,100,50);
    cont.add(k1);

    k2 = new JButton("Knopf 2");
    k2.setBounds(100,60,80,50);
    cont.add(k2);

    k3 = new JButton("Knopf 3");
    k3.setBounds(30,100,80,100);
    cont.add(k3);

    setSize(210, 240);
    setVisible(true);
  }
  .   .   .
}
```

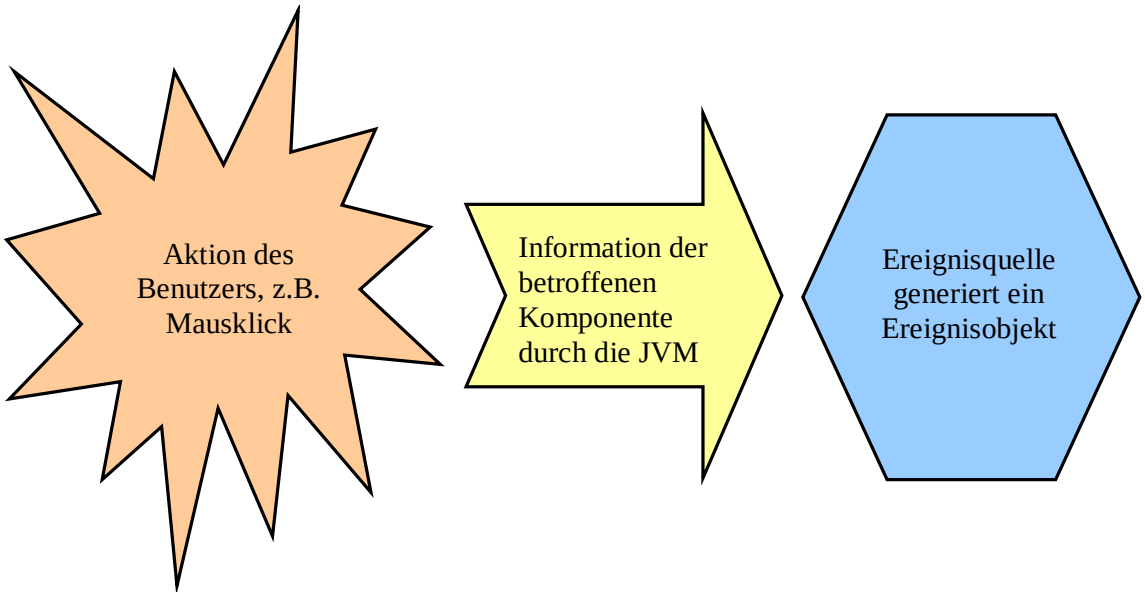
## *11.6 Ereignisbehandlung*

Die Besonderheit einer Komponente im Unterschied zu einem gewöhnlichen Objekt besteht neben ihrem optischen Auftritt in der Fähigkeit, **Ereignisse** (z.B. Mausklicks) zu erkennen und an interessierte andere Objekte weiterzuleiten, die dann durch Ausführen einer passenden Methode reagieren können.

### 11.6.1 Das Delegationsmodell

Bei der seit Java 1.1 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 11.3 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 auf das Schließkreuz in der Titelzeile zu einem Ereignis.

- **Ereignisobjekt**
  Zu jeder Ereignisart gehört in Java eine Ereignisklasse. Stellt die Quellkomponente ein Ereignis von bestimmter Art, 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 Ereignisobjekte aus verschiedenen Klassen generieren (z.B. **MouseEvent**, **ActionEvent**, **KeyEvent**). Über die Abstammungsverhältnisse der Ereignisklassen informiert Abschnitt 11.6.2.

  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* spezielle 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 11.6.2).

- **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.

  In vielen Programmen 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. In diesem Fall wären ge-

trennt agierende Ereignisbehandlungsmethoden unökonomisch. Außerdem fördert die Trennung von Benutzeroberfläche und Ereignisbehandlung die Wiederverwendbarkeit der Software.

### 11.6.2 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:

| Method Summary | |
|---|---|
| void | **windowActivated**(WindowEvent e)<br>Invoked when the Window is set to be the active Window. |
| void | **windowClosed**(WindowEvent e)<br>Invoked when a window has been closed as the result of calling dispose on the window. |
| void | **windowClosing**(WindowEvent e)<br>Invoked when the user attempts to close the window from the window's system menu. |
| void | **windowDeactivated**(WindowEvent e)<br>Invoked when a Window is no longer the active Window. |
| void | **windowDeiconified**(WindowEvent e)<br>Invoked when a window is changed from a minimized to a normal state. |
| void | **windowIconified**(WindowEvent e)<br>Invoked when a window is changed from a normal to a minimized state. |
| void | **windowOpened**(WindowEvent e)<br>Invoked the first time a window is made visible. |

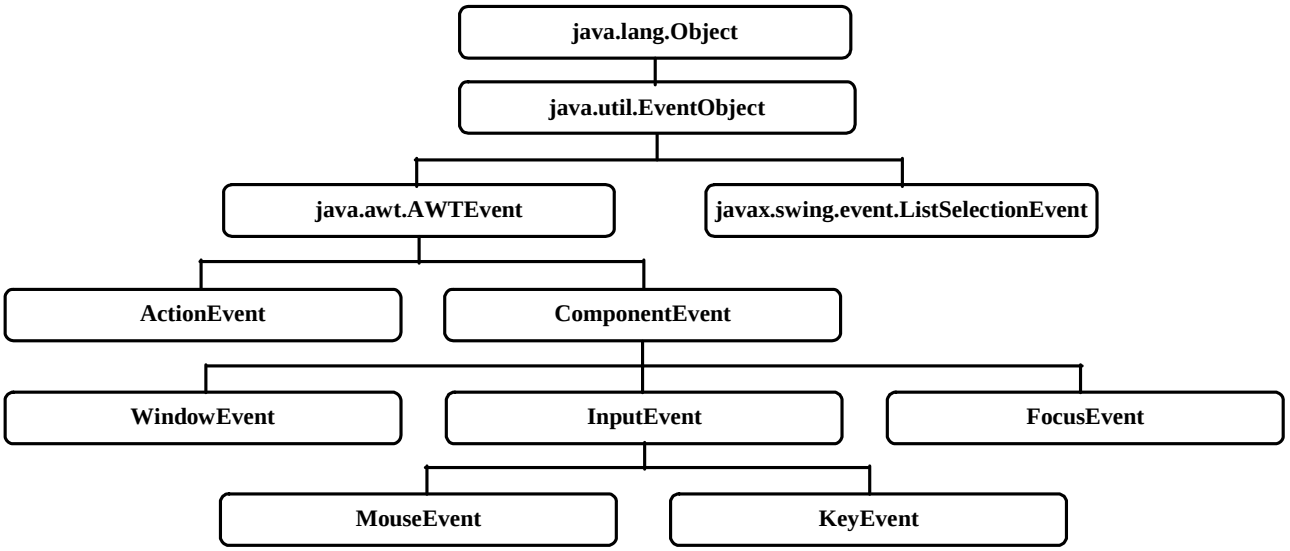
Aus der Anzahl und den Namen der Methoden wird klar, dass die Ereignisklasse **WindowEvent** bei *mehreren* Ereignisarten zum Einsatz kommt, z.B. beim Schließen und Verkleinern eines Fensters. Die Java-Designer haben sich für eine überschaubare Anzahl von Ereignisklassen entschieden, die jeweils für mehrere, zusammen gehörige Ereignisarten zuständig sind.

Die von einem Ereignis betroffene Komponente erfährt von der JRE 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

Das an die Event Handler übergebene Ereignisobjekt läst sich mit der Methode **getID()** zur Ereignisart befragen.

In der folgenden Abbildung sehen Sie die Abstammungsverhältnisse der im Kurs behandelten Ereignisklassen:




Klassen ohne Paketangabe gehören zum Paket **java.awt.event**.

### 11.6.3 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 bei Objekten der Klasse `MPC` (eine **JFrame**-Erweiterung) im Konstruktor für die Ereignisklasse **WindowEvent** mit der Methode **addWindowListener()** ein neu erzeugtes Objekt aus der Klasse `WC` als Empfänger eingetragen:

```
addWindowListener(new WC());
```

Damit die Klasse `WC` den Anforderungen an einen **WindowEvent**-Empfänger genügt, hat sie das Interface **WindowListener** (siehe Abschnitt 11.6.2) zu implementieren (direkt oder indirekt).

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** 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<br><br>addActionListener() |
| ComponentEvent | Position, Größe oder Sichtbarkeit einer Komponente haben sich verändert. | ComponentListener<br>ComponentAdapter<br>addComponentListener() |

| **Ereignisklasse** | **Mögliche Auslöser** | **Empfänger-Interface, zugeh. Adapterklasse, Registrierungsmethode** |
|---|---|---|
| WindowEvent | Der Benutzer (de)aktiviert, öffnet, schließt oder (de)ikonisiert ein Fenster. | WindowListener<br>WindowAdapter<br>addWindowListener() |
| MouseEvent[52] | Benutzer drückt eine Maustaste, während sich die Maus über einer Komponente befindet. | MouseListener<br>MouseAdapter<br>addMouseListener() |
| MouseEvent[1] | Der Benutzer bewegt die Maus über einer Komponente. | MouseMotionListener<br>MouseMotionAdapter<br>addMouseMotionListener() |
| KeyEvent | Der Benutzer drückt eine Taste, während eine Komponente den Eingabefokus besitzt. | KeyListener<br>KeyAdapter<br>addKeyListener() |
| ItemEvent | Das gewählte Item einer Liste oder der Zustand eines Umschalters (z.B. Kontrollkästchen) wechselt. | ItemListener<br><br>addItemListener() |
| FocusEvent | Eine Komponente erhält den Eingabefokus. | FocusListener<br>FocusAdapter<br>addFocusListener() |
| ListSelectionEvent | Der Benutzer wechselt in einer Liste den gewählten Eintrag. | ListSelectionListener<br><br>addListSelectionListener() |

Bei einer Ereignisquelle können zu einer Ereignisklasse auch *mehrere* Ereignisempfänger registriert werden, so dass Ereignisobjekte ggf. an mehrere Empfänger weitergeleitet werden.

### 11.6.4 Adapterklassen

In unserem Swing-Einstiegsbeispiel soll der **WindowEvent**-Empfänger zur Ereignisquelle `MPC` 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-Interfaces eine so genannte *Adapterklasse*. Diese implementiert das zugehörige Interface mit *leeren* Methoden. 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 Beispiel wird die Ereignisempfängerklasse `WC` aus der zum Interface **WindowListener** gehörigen Adapterklasse **WindowAdapter** abgeleitet:

```
private class WC extends WindowAdapter {
  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);
    }
}
```

[52] Ein **MouseEvent** wird ggf. an registrierte **MouseListener** *und* an registrierte **MouseMotionListener** weitergeleitet.

Es wird lediglich die tatsächlich benötigte Methode **windowClosing()** implementiert. Die `Ex`-Überschreibung fragt beim Benutzer nach, ob er allen Ernstes aufhören möchte (siehe Abschnitt 3.8 zur Klasse **JOptionPane**). Erst nach einer Bestätigung dieser Absicht, wird das Programm über die statische **System**-Methode **exit()** beendet.

Warum die Klasse `WC` den Zugriffsmodifikator **private** erhalten darf, erfahren Sie in Abschnitt 11.6.6.1.

### 11.6.5 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:

```
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 weiter läuft. 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()**:

```
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** 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**[53] hat nicht gegen eine Beendigung des Programms einzuwenden.

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 eine Referenz auf das Anwendungsfenster-Objekt, so dass der Garbage Collector vor dem Beenden der GUI-Threads keinen Anlass hat, das Anwendungsfenster zu beseitigen.

Um ein GUI-Programm per Anweisung zu beenden, verwendet man die Methode **System.exit()**, was eine **JFrame**-Komponente aufgrund des oben vorgestellten Methodenaufrufs

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

auch automatisch erledigen kann.

### 11.6.6 Optionen zur Definition von Ereignisempfängern

In diesem Abschnitt lernen Sie u.a. zwei neue Optionen zur Klassendefinition kennen, die *nicht nur* bei der Ereignisbehandlung eingesetzt werden können: innere und anonyme Klassen.

#### *11.6.6.1 Innere Klassen als Ereignisempfänger*

Wer den vollständigen Quellcode des Swing-Einstiegsbeispiels aufmerksam liest, wird feststellen, dass die `WC`–Klassendefinition im Rumpf der `MPC`-Klassendefinition steht, was `WC` zur **geschachtelten Klasse** (engl. *nested class*) bzw. zur **Mitgliedklasse** macht. Weil die Klasse `WC` *keinen* **static**-Modifikator erhalten hat, ist es auch eine **innere Klasse**:

```
class MPC extends JFrame {
  . . .
  private class WC extends WindowAdapter {
    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 `WC`-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.

Einige Eigenschaften von **Mitgliedsklassen** (geschachtelten Klassen):

- Der Compiler erzeugt auch für jede Mitgliedsklasse eine eigene **class**-Datei, in deren Namen die Bezeichner für die innere und die umgebende Klasse eingehen, so dass im Beispiel der Name **MPC$WC.class** resultiert.

---

[53] Diesen zentralen Bestandteil der Java-Sicherheitsarchitektur können wir aus Zeitgründen nicht behandeln.

- Mitgliedsklassen dürfen geschachtelt werden.
- Während für eigenständig definierte Klassen nur der Zugriffsmodifikator **public** erlaubt ist, können bei Mitgliedsklassen auch die Zugriffsmodifikatoren **private** und **protected** verwendet werden (vgl. Abschnitt 7.3.2). Im Beispiel erhält die innere Klasse `WC` wie alle anderen `MPC`-Mitglieder den Zugriffsmodifikator **private**.

Zusätzliche Eigenschaften von **inneren Klassen** (geschachtelt, aber nicht **static**):

- In den Methoden einer inneren Klasse kann man auf alle Felder und Methoden der äußeren Klasse zugreifen (auch auf die privaten).
- Um ein Objekt einer inneren Klasse zu erstellen, benötigt man ein Objekt der umgebenden Klasse. Wird zu einer inneren Klasse in einer berechtigten Fremdklasse ein Objekt benötigt, stehen zur Kreation zwei Möglichkeiten offen:
  - Ein Objekt der umgebenden Klasse kann in einer geeigneten Methode das gewünschte Objekt erzeugen und die Referenz abliefern. Im folgenden Beispiel steht für diesen Zweck die Methode `neueTasche()` bereit:

```
class Mantel {
  void tuWas() {
    System.out.print("Mantel");
  }
  Tasche neueTasche() {
    return new Tasche();
  }
  class Tasche {
    void tuWas() {
      Mantel.this.tuWas();
      System.out.println("tasche");
    }
  }
}

class Prog {
  public static void main(String[] args) {
    Mantel man = new Mantel();
    Mantel.Tasche tob = man.neueTasche();
    tob.tuWas();
  }
}
```

  - Fehlt in der umgebenden Klasse eine erzeugende Methode, wird der Konstruktor der inneren Klasse per **new**-Operator aufgerufen, wobei ebenfalls ein Objekt der umgebenden Klasse erforderlich ist. Um diese Technik im letzten Beispiel (trotz vorhandener Generatormethode) zu demonstrieren, ist die Zeile:

```
Mantel.Tasche tob = man.neueTasche();
```

    durch folgende Variante zu ersetzen:

```
Mantel.Tasche tob = man.new Tasche();
```

- Ein Objekt der inneren Klasse kann über die **this**-Referenz mit vorangestelltem Klassennamen auf seinen Geburtshelfer (auf das umgebende Objekt der äußeren Klasse) zugreifen, z.B.:

```
class Tasche {
  void tuWas() {
    Mantel.this.tuWas();
    System.out.println("tasche");
  }
}
```

Wird eine Mitgliedsklasse als **static** deklariert, handelt es sich *nicht* um eine innere Klasse, so dass z.B. *kein* Zugriff auf die Privatsphäre der umgebenden Klasse möglich ist. Es liegen *zwei Top-Level* - Klassen vor, wobei aber die **statische Mitgliedsklasse** einen Doppelnamen führen muss, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `class Mantel {`<br>`  static class Tasche {`<br>`  void tuWas() {`<br>`  System.out.println("Manteltasche");`<br>`  }`<br>`  }`<br>`}`<br>`class Prog {`<br>`  public static void main(String[] args) {`<br>`  Mantel.Tasche tob = new Mantel.Tasche();`<br>`  tob.tuWas();`<br>`  }`<br>`}` | `Manteltasche` |

### *11.6.6.2 Anonyme Klassen 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(labelPrefix + numClicks);
        if (iconInd < maxIcon-1)
           iconInd++;
        else
           iconInd = 0;
        lblIcon.setIcon(icon[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.

Wie bei den inneren Klassen haben wir auch bei einer solchen **anonymen Klasse** die Möglichkeit, in den Methoden auf Felder und Methoden der umgebenden Klasse zuzugreifen, was im Beispiel außerordentlich hilfreich ist.

Einige Eigenschaften von anonymen Klassen:

- Definition und Instantiierung 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 **ActionListener** angegeben und deren (einzige) Methode **actionPerformed()** implementiert.
- Es kann nur eine einzige Instanz erzeugt werden.
  Werden mehrere Instanzen benötigt, ist eine innere Klasse zu bevorzugen.
- Es sind keine Konstruktoren, keine statischen Methoden und keine statischen Felder erlaubt.
- Der Compiler erzeugt auch für eine anonyme Klasse eine eigene **class**-Datei, in deren Namen der Bezeichner für die umgebende Klasse eingeht, so dass im Beispiel der Name **MPC$1.class** resultiert.

Übrigens verwendet auch der im *Visual Editor* enthaltenen Quellcode-Generator anonyme Klassen zur Ereignisbehandlung, was schon in Abschnitt 4.8.6 zu beobachten war.

#### *11.6.6.3 Do-It-Yourself – Ereignisbehandlung*

Zur Behandlung der von Instanz-Komponenten oder vom Rahmenfenster generierten Ereignisse muss eine von **JFrame** abstammende Klasse nicht unbedingt *Fremd*klassen beauftragen, sondern kann den Job auch selbst übernehmen, sofern sie die erforderlichen Ereignis-Interfaces erfüllt. Dies wird in Abschnitt 11.6.7.2 an einem Beispielprogramm zum Umgang mit Mausereignissen demonstriert.

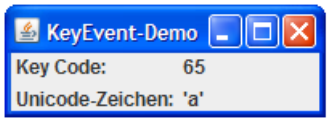
### 11.6.7 Tastatur- und Mausereignisse

#### *11.6.7.1 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 extends JFrame {
  private JLabel keyCode, uChar;

  KeyEventDemo() {
    super("KeyEvent-Demo");

    keyCode = new JLabel(" Key Code:");
    uChar = new JLabel(" Unicode-Zeichen:");
    getContentPane().add(keyCode, BorderLayout.NORTH);
    getContentPane().add(uChar, BorderLayout.SOUTH);

    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent e) {
        keyCode.setText(" Key Code:                "+e.getKeyCode());
      }
      public void keyTyped(KeyEvent e) {
        uChar.setText(" Unicode-Zeichen:  '"+e.getKeyChar()+"'");
      }
      public void keyReleased(KeyEvent e) {
        keyCode.setText(" Key Code:");
        uChar.setText(" Unicode-Zeichen:");
      }
    });

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(200, 70);
    setVisible(true);
  }

  public static void main(String[] args) {
    new KeyEventDemo();
  }
}
```

Der **KeyEvent**-Empfänger verarbeitet 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.

### *11.6.7.2 Die Klasse MouseEvent für Mausereignisse*

Im folgendem Beispiel wird der Umgang mit der Ereignisklasse **MouseEvent** sowie mit den (beiden!) zugehörigen Schnittstellen **MouseListener** und **MouseMotionListener** demonstriert. Außerdem wird gezeigt (wie in Abschnitt 11.6.6.3 angekündigt), dass eine von **JFrame** abstammende Klasse nicht unbedingt *Fremd*klassen mit der Ereignisbehandlung beauftragen muss, sondern den Job auch selbst übernehmen kann, sofern sie die erforderlichen Ereignis-Schnittstellen erfüllt:

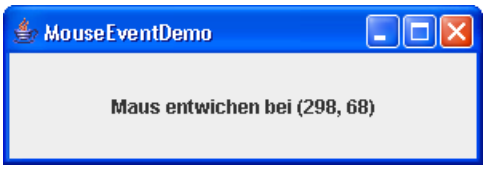
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MouseEventDemo extends JFrame implements MouseListener, MouseMotionListener {
   private JLabel status = new JLabel();

   MouseEventDemo() {
      super("MouseEvent-Demo");

      status.setHorizontalAlignment(SwingConstants.CENTER);
      getContentPane().add(status, BorderLayout.CENTER);

      addMouseListener(this);
      addMouseMotionListener(this);

      setDefaultCloseOperation(EXIT_ON_CLOSE);

      setSize(300, 100);
      setVisible(true);
   }

   public void mouseClicked(MouseEvent e) {
      status.setText("Mausklick bei ("+e.getX()+", "+e.getY()+")");
   }
   public void mousePressed(MouseEvent e) {
      status.setText("Maustaste gedrückt bei ("+e.getX()+", "+e.getY()+")");
   }
   public void mouseReleased(MouseEvent e) {
      status.setText("Maustaste losgelassen bei ("+e.getX()+", "+e.getY()+")");
   }
   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:

```
addMouseListener(this);
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.:

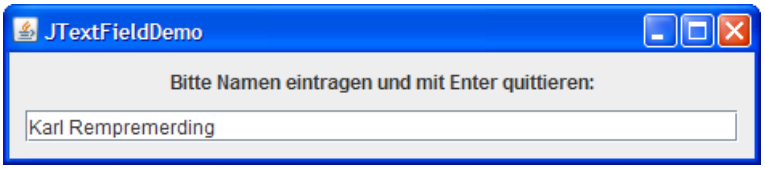



## *11.7 Bedienelemente (Teil 2)*

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 (Sun Microsystems 2009) alle benötigten Informationen finden.

### 11.7.1 Einzeiliges Texteingabefeld

Im Rahmen der **JOptionPane**-Klassenmethode **showInputDialog()** (vgl. Abschnitt 0) konnten Sie schon eine einzeilige Textkomponente besichtigen. In einem **JFrame**-Fenster kann dieses elementare Bedienelement mit Hilfe einer **JTextField**-Komponente implementiert werden, z.B.:




Der Quellcode des Beispielprogramms:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JTextFieldDemo extends JFrame {
   JTextField name = new JTextField(40);
   JLabel label = new JLabel("Bitte Namen eintragen und mit Enter quittieren:");
   final static String titel = "JTextField-Demo";

   JTextFieldDemo() {
      super(titel);
      Container cont = getContentPane();

      label.setHorizontalAlignment(JTextField.CENTER);
      label.setBorder(BorderFactory.createEmptyBorder(10, 0, 0, 0));
      cont.add(label, BorderLayout.NORTH);
```

```
        JPanel pan = new JPanel();
        pan.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
        pan.add(name);
        cont.add(pan, BorderLayout.CENTER);
        name.addActionListener(
           new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                   JOptionPane.showMessageDialog(((JComponent)e.getSource()).getParent(),
                    "Sie heißen "+name.getText(),titel,JOptionPane.INFORMATION_MESSAGE);
                }
             });

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
     }

     public static void main(String[] args) {
        new JTextFieldDemo();
     }
}
```

Die **JTextField**-Komponente wird gleich bei der Instanzvariablendeklaration erzeugt, wobei der verwendete Konstruktor die gewünschte Breite (in Spalten) erfährt:

```
JTextField name = new JTextField(40);
```

Um das Texteingabefeld zu beranden, kommt das in Abschnitt 11.4.5 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 im Zentrum des **JFrame** - Top-Level-Containers, der vom voreingestellten **BorderLayout** - Manager betreut wird:

```
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 Eingabefokus besitzt, wird ein **ActionEvent** ausgelöst. Im Beispiel präsentiert der im Rahmen einer anonymen Klasse (vgl. Abschnitt 0) 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);
```

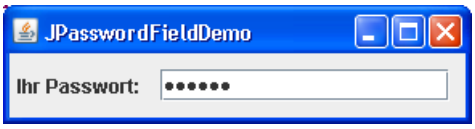
Rechtsbündige Textfelder sind z.B. bei der Erfassung von Zahlen zu bevorzugen.

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.

### 11.7.2 Einzeiliges Texteingabefeld für Passwörter

Zum Erfassen von **Passwörtern** steht die Swing-Komponente **JPasswordField** bereit, die im Unterschied zur Komponente **JTextField** für jedes eingegebene Zeichen einen Punkt anzeigt, z.B.:




Der Quellcode des Beispielprogramms:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JPasswordFieldDemo extends JFrame {
   private JPasswordField pw = new JPasswordField(16);
   private JLabel label = new JLabel("Ihr Passwort: ");
   private final static String titel = "JPasswordField-Demo";

   JPasswordFieldDemo() {
      super(titel);
      Container cont = getContentPane();

      pw.addActionListener(
         new ActionListener() {
               public void actionPerformed(ActionEvent e) {
                  JOptionPane.showMessageDialog(((JComponent)e.getSource()).getParent(),
                   "Ihr Passwort: " + new String(pw.getPassword()), titel,
                   JOptionPane.INFORMATION_MESSAGE);
               }
            });

      JPanel pan = new JPanel();
      pan.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
      pan.add(pw);
      cont.add(pan, BorderLayout.CENTER);

      label.setBorder(BorderFactory.createEmptyBorder(0, 5, 0, 0));
      cont.add(label, BorderLayout.WEST);

      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      pack();
      setVisible(true);
   }

   public static void main(String[] args) {
      new JPasswordFieldDemo();
   }
}
```

Die **JPasswordField**-Komponente wird im gleich bei der Instanzvariablendeklaration erzeugt, wobei der verwendete Konstruktor die gewünschte Breite (in Spalten) erfährt:

```
private JPasswordField pw = new JPasswordField(16);
```

Um das Passworteingabefeld zu beranden, kommt das in Abschnitt 11.7.1 beschriebene Verfahren zum Einsatz.

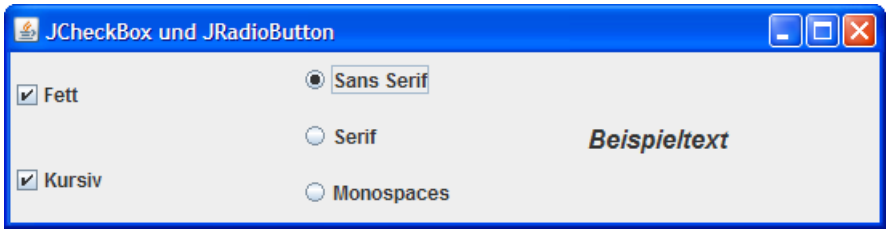
Das erfasste Passwort kann mit der **JPasswordField**-Methode **getPassword()** als **char**-Array extrahiert werden, was im Event Handler des Beispielprogramms zur „geschickten Kontrollausgabe“ geschieht:


### 11.7.3 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**

#### *11.7.3.1 Kontrollkästchen*

Im Beispielprogramm erhalten die beiden **JCheckBox**-Komponenten per Konstruktor eine Beschriftung:

```
private JPanel panCheck, panRadio;
private JCheckBox chkBold, chkItalic;
   . . .
Container cont = getContentPane();
cont.setLayout(new GridLayout(1, 3));
panCheck = new JPanel();
panCheck.setLayout(new GridLayout(0, 1));
cont.add(panCheck);

chkBold = new JCheckBox("Fett");
chkItalic = new JCheckBox("Kursiv");
panCheck.add(chkBold);
panCheck.add(chkItalic);
```

```
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. Dieser sitzt am linken Rand des **JFrame**-Fensters, das mit einem dreispaltigen **GridLayout** arbeitet.

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) {
    int oldStyle = lblBeispiel.getFont().getStyle(),
      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 Namensraum **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()**.

#### *11.7.3.2 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:

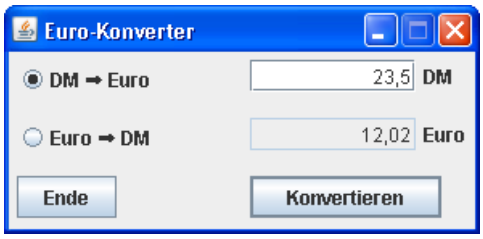
```
private JPanel panCheck, panRadio;
private JRadioButton rbSans, rbSerif, rbMono;
   . . .
panRadio = new JPanel();
panRadio.setLayout(new GridLayout(0, 1));
cont.add(panRadio);

rbSans = new JRadioButton("Sans Serif", false);
rbSerif = new JRadioButton("Serif", true);
rbMono = new JRadioButton("Monospaces", false);

panRadio.add(rbSans);
panRadio.add(rbSerif);
panRadio.add(rbMono);

rbGroup = new ButtonGroup();
rbGroup.add(rbSans);
rbGroup.add(rbSerif);
rbGroup.add(rbMono);

RadioHandler rbHandler = new RadioHandler();
rbSans.addItemListener(rbHandler);
rbSerif.addItemListener(rbHandler);
rbMono.addItemListener(rbHandler);
```

Im **ItemEvent**-Handler der Optionsschalter kommen die in Abschnitt 11.7.3.1 vorgestellten Schriftartenverwaltungsmethoden zum Einsatz:

```
private class RadioHandler implements ItemListener {
  public void itemStateChanged(ItemEvent e) {
    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));
  }
}
```

### 11.7.4 Standardschaltfläche und Eingabefokus

In diesem Abschnitt werden Techniken zur Verwaltung bzw. Kanalisierung von Tastaturereignissen behandelt, die von Bedeutung für die Bedienbarkeit eines Programms sind. Als Beispiel dient ein Programm zur Währungskonvertierung, das Sie als Übungsaufgabe erstellen sollen (siehe Abschnitt 11.8):




Für den mit **Konvertieren** beschrifteten Befehlsschalter (mit dem Referenzvariablennamen `cbKonvertieren`) sollte mit der schon in Abschnitt 11.4.2 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:

```
getRootPane().setDefaultButton(cbKonvertieren);
```

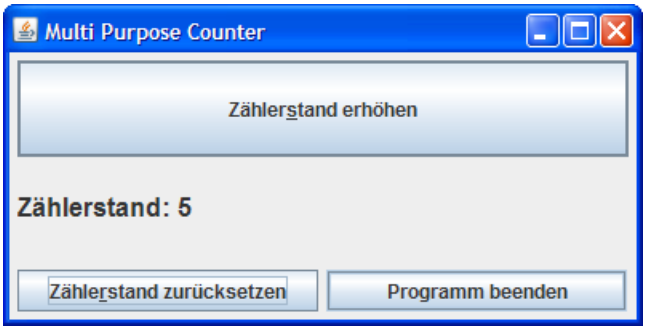
Ein **setDefaultButton()**-Aufruf ist an die **RootPane**-Schicht des **JFrame**-Fensters zu richten. Der Benutzer kann die **Standardschaltfläche** an einem betonten (breiten und dunklen) Rahmen erkennen (siehe oben) und manchen Griff zur Maus einsparen, was die Bedienung des Programms erleichtert.

Damit Benutzer nach dem Programmstart sofort den ersten zu konvertierenden DM-Betrag eingeben können, sollte die obere **JTextField**-Komponente (mit dem Referenzvariablennamen `tfEingabe`) über die **Component**-Methode **requestFocus()** den **Eingabefokus** anfordern:

```
tfEingabe.requestFocus();
```

Ein Textfeld mit Eingabefokus ist für die Benutzer an der blinkenden Einfügemarke zu erkennen.

Bei einer Schaltfläche mit Eingabefokus erscheint ein Zusatzrahmen um die Beschriftung. Im folgenden Beispiel hat der Schalter mit der Beschriftung **Zählerstand zurücksetzen** aktuell den Eingabefokus und reagiert daher auf die Leertaste, während die Standardschaltfläche mit **Zählerstand erhöhen** beschriftet ist und auf die **Enter**-Taste reagiert:




Benutzer können den Eingabefokus per Tabulatortaste an das nächste Bedienelement in der Fokussequenz übertragen. Ist für eine Komponente (aktuell) der Eingabefokus nicht sinnvoll, kann sie mit der **Component**-Methode **setFocusable()** aus der Fokussequenz herausgenommen werden, z.B.:

```
tfAusgabe.setFocusable(false);
```

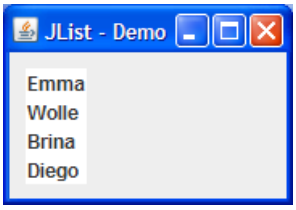
Mit der **Component**-Methode **transferFocus()** kann eine Komponente den Fokus an die nächste Station in der Fokussequenz weitergeben. Im Währungskonverterbeispiel (siehe oben) sollte diese Methode dazu genutzt werden, nach einer Konvertierung (**ActionEvent** beim Schalter `cbKonvertieren`) sofort die Eingabe des nächsten Betrags zu ermöglichen:

```
cbKonvert.transferFocus();
```

## 11.7.5 Listen

### *11.7.5.1 Einfach*

Mit einer Komponente aus der Klasse **JList** kann man dem Benutzer eine Liste mit markierbaren Einträgen präsentieren, z.B.:




Sind keine Listenänderungen im Programmablauf geplant, eignet sich beim Erstellen eines **JList**-Objekts die Konstruktorüberladung mit einem Parameter vom Typ **String[]**, z.B.

```
private JList listFest = new JList(new String[]{"Eimer", "Wand", "Brille"});
```

Ist Variabilität in der Listenzusammenstellung gefragt, stellt man dem **JList**-Objekt ein Objekt zu Seite, dessen Klasse das **ListModel**-Interface implementiert. Die beiden Objekte arbeiten als **Model-View** - Team zusammen, wobei …

- sich das **JList**-Objekt um die Anzeige und Benutzerinteraktion kümmert,
- das Model-Objekt die Listeneinträge verwaltet (z.B. Elemente ergänzt oder löscht)
- und beide über **ListDataEvent**-Ereignisse kommunizieren

Statt eine eigene Klasse zu definieren und dabei das **ListModel**-Interface zu implementieren, verwenden wir die Klasse **DefaultListModel**, die alle benötigten Kompetenzen besitzt:

```
private DefaultListModel modelLinks, modelRechts;
private JList listLinks, listRechts;
   . . .
modelLinks = new DefaultListModel();
modelLinks.addElement("Emma"); modelLinks.addElement("Wolle");
modelLinks.addElement("Brina"); modelLinks.addElement("Diego");
listLinks = new JList(modelLinks);
modelRechts = new DefaultListModel();
listRechts = new JList(modelRechts);
```

Zur Demonstration der Dynamik verwenden wir ein Beispielprogramm mit zwei Listen, wobei die linke Liste mögliche Teilnehmer eines Windhundrennens enthält, und die rechte Liste initial leer ist. Durch Markieren (Anklicken) eines Listeneintrags veranlasse der Benutzer dessen Wechsel in die jeweils anderer Liste, z.B.:


Wie die beiden Listen auf die von einem **BorderLayout**-Objekt verwaltete Inhaltsschicht des **JFrame**-Fensters gelangen, muss nach etlichen Layout-Beispielen nicht mehr erläutert werden.

Beim Markieren eines Eintrags initiiert der Benutzer beim betroffenen **JList**-Objekt ein **ListSelectionEvent**, über das ein registrierter **ListEventListener** durch Aufruf seiner Methode **valueChanged()** informiert wird. Im Beispielprogramm wird bei beiden **JList**-Objekten 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:

```
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)e.getSource()).getSelectedIndex() < 0)
      return;
    Object wahl = ((JList)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() == false)
  return;
```

Neben der im Beispiel durch folgenden Methodenaufruf

```
listLinks.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

benutzten Einfachauswahl beherrscht die Klasse **JList** auch:

- **ListSelectionModel.SINGLE_INTERVAL_SELECTION**
  Es kann eine Teilmenge hintereinander liegende Listeneinträge gewählt werden.
- **ListSelectionModel.SINGLE_INTERVAL_SELECTION**
  Es kann eine beliebige Teilmenge der Listeneinträge gewählt werden. Dies ist die Voreinstellung.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

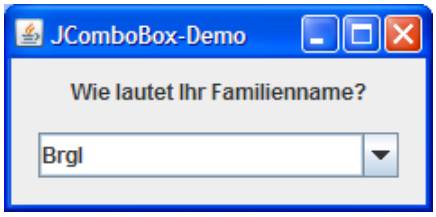
**…\BspUeb\Swing\JList**

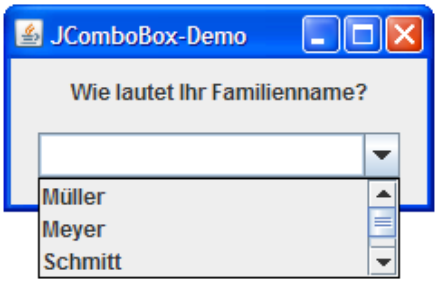
#### *11.7.5.2 Kombiniert*

Die **JComboBox**-Komponente bietet eine Kombination aus einem einzeiligen Textfeld und einer Liste, wobei normalerweise nur *ein* Element sichtbar ist. Um seine Wahl zu treffen, hat der Benutzer zwei Möglichkeiten:

- den Namen der gewünschten Option eintragen und mit **Enter** quittieren
- die versteckte Liste aufklappen und die gewünschte Option markieren

Im folgenden Programm wird die Angabe des Familiennamens durch eine Liste mit den häufigsten Namen erleichtert:




Per Voreinstellung funktioniert ein **JComboBox**–Bedienelement als reine Aufklapp(engl.: *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:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JComboBoxDemo extends JFrame {
  private JComboBox name;
  private JLabel label;
  private final String[] auswahl = {"Müller", "Meyer", "Schmitt", "Schulze"};
  private final static String titel = "JComboBox-Demo";

  JComboBoxDemo() {
    super(titel);
    Container cont = getContentPane();

    label = new JLabel("Wie lautet Ihr Familienname?");
    label.setHorizontalAlignment(JTextField.CENTER);
    label.setBorder(BorderFactory.createEmptyBorder(10, 0, 0, 0));
    cont.add(label, BorderLayout.NORTH);

    name = new JComboBox(auswahl);
    name.setMaximumRowCount(3);
    name.setEditable(true);
    name.setSelectedItem("");

    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);

    name.addItemListener(
      new ItemListener() {
          public void itemStateChanged(ItemEvent e) {
            if(e.getStateChange() == ItemEvent.SELECTED) {
              JComboBox cb = (JComboBox) e.getSource();
              JOptionPane.showMessageDialog(cb.getParent(),
                "Sie heißen "+cb.getSelectedItem(), titel,
                JOptionPane.INFORMATION_MESSAGE);
            }
          }
        });

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pack();
    setVisible(true);
  }
```

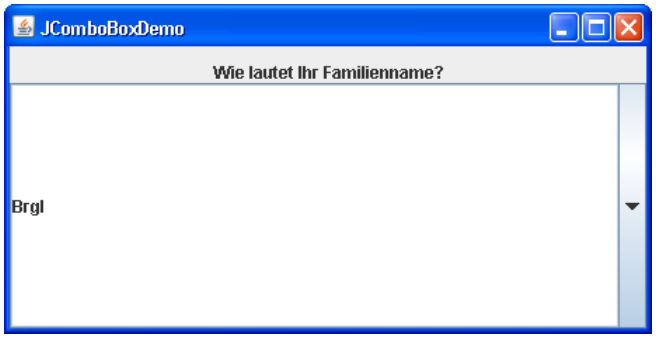
```
    public static void main(String[] args) {
      new JComboBoxDemo();
    }
}
```

Im Beispiel wird das **JComboBox**-Bedienelement aus folgenden Gründen in einen **JPanel**-Container gesteckt, der seinerseits im Zentrum des **BorderLayouts** zum **JFrame** - Top-Level-Container landet:

- Der voreingestellte **FowLayout** - Manager des **JPanel**-Containers respektiert die bevorzugte Größe des **JComboBox**-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 **BorderLaout**-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**-Bedienelement festgelegen (siehe Abschnitt 11.7.1 zum Rand bei Bedienelementen).

Ein **JComboBox**-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. Im Beispielprogramm wird der Empfänger nur bei einem positiven Auswahlereignis aktiv und ruft zur Diagnose die **ItemEvent**-Methode **getStateChange()** auf:

```
if(e.getStateChange() == ItemEvent.SELECTED) {
  . . .
}
```

Über die Methode **setMaximumRowCount()** wird einem **JComboBox**-Objekt mitgeteilt, wie viele Einträge es maximal anzeigen soll, z.B.:

```
name.setMaximumRowCount(3);
```

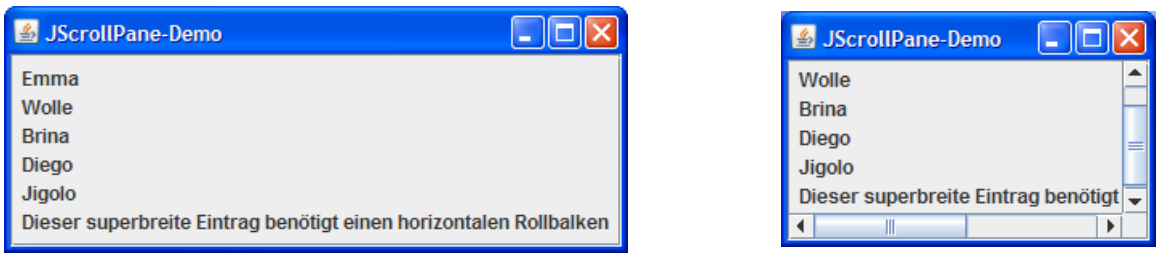
Sind mehr Einträge vorhanden, erscheint automatisch ein vertikaler Rollbalken (siehe Beispiel).

### 11.7.6 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 extends JFrame {
 private JList liste;

 JScrollPaneDemo() {
  super("JScrollPane-Demo");
  liste = new JList(new String[] {"Emma","Wolle","Brina","Diego","Jigolo",
    "Dieser superbreite Eintrag benötigt einen horizontalen Rollbalken"});
  liste.setBackground(this.getBackground());
  JPanel panel = new JPanel();
  panel.add(liste);
  getContentPane().add(new JScrollPane(panel),java.awt.BorderLayout.CENTER);
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  pack(); setVisible(true);
 }

 public static void main(String[] args) {
  new JScrollPaneDemo();
 }
}
```

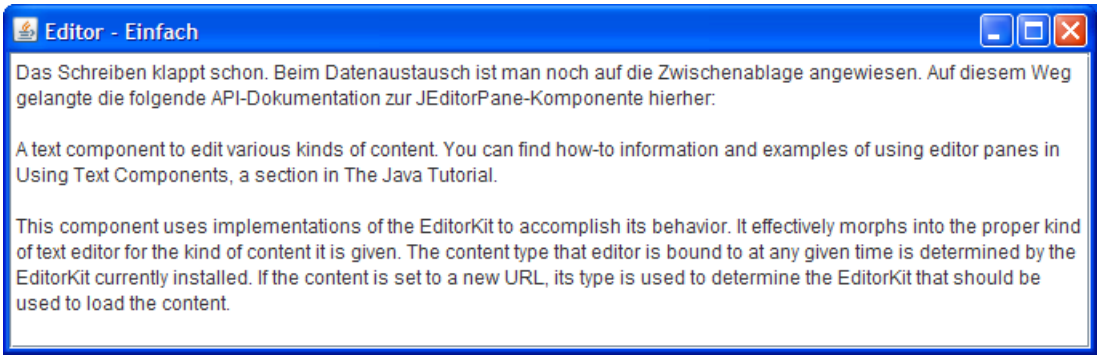
wird ein **JList**-Objekt durch ein **JScrollPane**-Objekt bei Bedarf mit Rahmen versorgt:




Es können nicht nur atomare Komponenten mit Rollbalken versorgt werden, sondern auch untergeordneter Container mit diversen Bestandteilen.

### 11.7.7 Ein (fast) kompletter Editor als Swing-Komponente

Ein Objekt aus der Klasse **JEditorPane** 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:

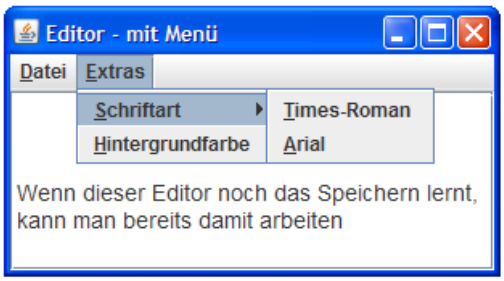
```
import javax.swing.*;

public class Editor extends JFrame{
   public Editor() {
      super("Editor - Einfach");

      JEditorPane text = new JEditorPane();
      getContentPane().add(new JScrollPane(text),java.awt.BorderLayout.CENTER);

      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(300, 200);
      setVisible(true);
   }
   public static void main(String[] arg) {
      new Editor();
   }
}
```

Um die **JEditorPane**-Komponente mit **Rollbalken** zu versorgen, wird sie in eine **JScrollPane**-Komponente verpackt.

Wir werden in den nächsten Abschnitten an unserem Editor noch weitere Verbesserungen vornehmen.

### 11.7.8 Menüzeile, Menü und Menüitem

Um das Potential der **JEditorPane**-Komponente besser auszuschöpfen, erweitern wir das in Abschnitt 11.7.7 begonnene Programm um ein Datei- und ein Extras-Menü:




#### *11.7.8.1 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 11.2.2 zum Aufbau eines Top-Level - Containers):

```
private JMenuBar menuBar;
   . . .
menuBar = new JMenuBar();
setJMenuBar(menuBar);
```

#### *11.7.8.2 Menü*

Die Menüs werden als Komponenten der Klasse **JMenu** erstellt, bei Bedarf mit einer **Alt**-Tastenkombination zum schnellen Öffnen versehen und dann auf der Menüzeile gesetzt oder in ein Untermenü aufgenommen (siehe unten):

```
private JMenu fileMenu, toolsMenu, fontMenu;
    . . .
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.:

```
toolsMenu = new JMenu("Extras");
toolsMenu.setMnemonic(KeyEvent.VK_E);
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 GIF-Datei passender Größe oder findet ein legal verwendbares Exemplar. Die Firma Sun Microsystems stellt einige Icons (meist in einer Größe von 16 × 16 bzw. 24 × 24 Pixeln) im JDK und auf der folgenden Webseite zur Verfügung:

  **http://java.sun.com/developer/techDocs/hi/repository/**

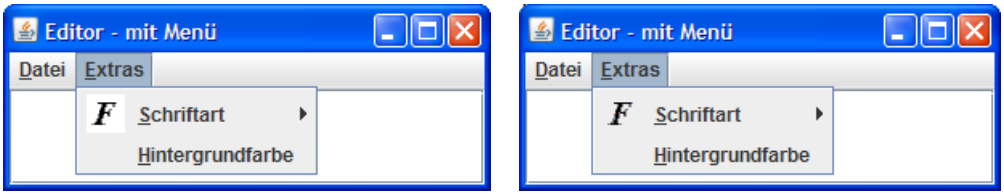
  Wer selbst kreativ werden möchte, findet auf der folgenden Webseite Empfehlungen zur Icon-Gestaltung:

  **http://java.sun.com/products/jlf/ed2/book/HIG.Graphics3.html**

  Für das `fontMenu` soll ein bescheidenes Icon (mit 24 × 24 Pixeln) genügen:

- In der Regel ist es sinnvoll, die Hintergrundfarbe eines Icons transparent zu setzen, z.B.:

  Dies kann z.B. mit diversen kostenlosen Programmen geschehen, unter Windows z.B. mit **IrfanView**.

- Schließlich wird aus dem Icon ein **ImageIcon**-Objekt erzeugt und per **AbstractButton**-Methode **setIcon()** dem Menü zugeordnet, z.B.:

  ```
  fontMenu.setIcon(new ImageIcon("font.gif"));
  ```

Analog werden wir gleich das Menüitem **Datei > Öffnen** verzieren.

### *11.7.8.3 Menüitem*

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 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 Menüitems aufnehmen, z.B.:

```
private JMenuItem timesItem;
  . . .
timesItem = new JMenuItem("Times-Roman");
timesItem.setMnemonic(KeyEvent.VK_T);
  Menu.add(timesItem);
```

Für jedes Menüitem 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));
     }
   });
```

Um ein Menüitem mit einem Icon zu verzieren, geht man genauso vor wie bei einem Menü (siehe Abschnitt 11.7.8.2). Hier erhält das Editor-Menüitem **Datei > Öffnen** ein Icon unter Verwendung der Datei **open.gif** aus dem JDK 6.0:

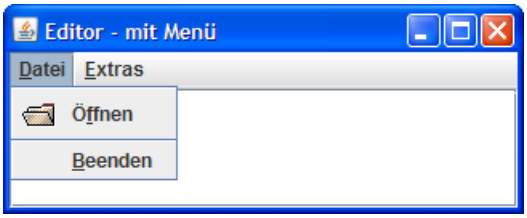
```
openItem.setIcon(new ImageIcon("open.gif"));
```

#### *11.7.8.4 Separatoren*

Um eine Trennlinie zwischen zwei
Menün 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:




### 11.7.9 Standarddialog zur Dateiauswahl

Im **ActionEvent**-Handler zum Menüitem

**Datei > Öffnen**

unseres Editors

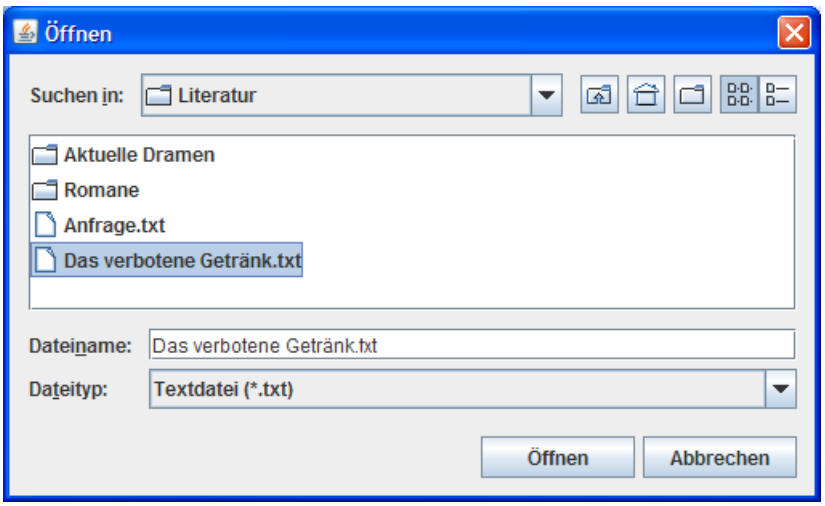
```
openItem.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
       getFile();
       if (file != null) readText();
     }
   });
```

wird eine private Methode `getFile()` 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 getFile() {
   FileNameExtensionFilter filter =
     new FileNameExtensionFilter("Textdatei (*.txt)", "txt");
   JFileChooser fc = new JFileChooser();
   fc.addChoosableFileFilter(filter);
   if (fc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
     file = fc.getSelectedFile();
}
```

Mit einem Objekt der Klasse **FileNameExtensionFilter** und der **JFileChooser**-Methode **addChoosableFileFilter()** 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. Mit den beteiligten Klassen werden wir uns bald 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 11.8) sollen Sie mit Hilfe dieser Klasse einen **ActionListener** zum Menüitem **Extras > Hintergrundfarbe** realisieren.

### 11.7.10 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 11.4.2), doch sind auch andere Komponenten erlaubt (z.B. **JTextField**).

Damit eine Symbolleiste vom Benutzer per Maus abgerissen und problemlos neu positioniert werden, sollte sie zu einem Container mit **BorderLayout** gehören (vgl. Abschnitt 11.5.1):

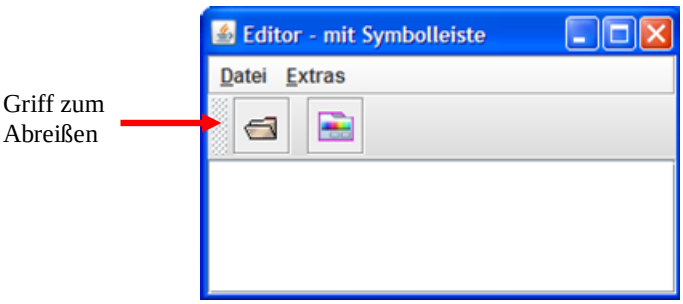
- Die über Symbolleistenschalter veranlassten Aktionen sollten die Komponente im **BorderLayout**-Zentrum betreffen.
- Die Symbolleiste sollte an der Peripherie starten (Norden, Osten, Süden oder Westen).

Neben den vier Randzonen steht dem Benutzer dann als Ablagevariante auch das eigenständige Fenster zur Verfügung.

Über die **JToolBar**-Methode **setFloatable()** lässt sich eine Symbolleiste fixieren, z.B.:

```
toolBar.setFloatable(false);
```

In der Regel erhalten Symbolleistenschalter ein Icon (z.B. in der Größe 24 × 24), doch sind auch Beschriftungen möglich. Unser Editor, den wir seit Abschnitt 11.7.7 als Demonstrationsbeispiel benutzen, soll per Symbolleiste das Öffnen einer Textdatei und die Wahl einer Hintergrundfarbe erlauben, wobei es sich um eine etwas willkürliche und zufällige Zusammenstellung ohne Anspruch auf vorbildliche Bedienungslogik handelt:




Die Symbolleisten-Komponente erhält per Konstruktor eine Titelzeilenbeschriftung für den selbständigen Zustand und startet (mit der voreingestellten horizontalen Orientierung) im Norden des **JFrame**-Layouts:

```
JToolBar toolBar = new JToolBar("Editor");
getContentPane().add(toolBar, BorderLayout.NORTH);
```

Um Menüzeile und Symbolleiste optisch zu trennen, erhält die Menüzeile einen Rahmen:

```
menuBar.setBorder(BorderFactory.createEtchedBorder());
```

Die beiden Schaltflächen können das Icon und die **ActionEvent**-Empfänger jeweils vom funktionsgleichen Menüitem übernehmen, z.B.:

```
JButton openButton = new JButton(new ImageIcon("open.gif"));
openButton.addActionListener(openItem.getActionListeners()[0]);
toolBar.add(openButton);
```

Wie bei Menüeinträgen kann auch bei Symbolleistenelementen ein Separator zur Gruppierung verwendet werden, z.B.:

```
toolBar.addSeparator();
```

Die Erstellung eines beschrifteten Symbolleistenschalters

```
JButton exitButton = new JButton("Beenden");
exitButton.addActionListener(exitItem.getActionListeners()[0]);
toolBar.add(exitButton);
```

ist ebenso langweilig wie das Aussehen (hier im frei schwebenden Zustand):

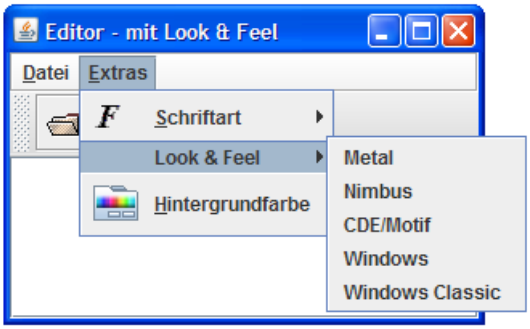
## *11.8 Weitere Swing-Techniken*

### 11.8.1 Look & Feel umschalten

Wer den in Abschnitt 11.7.9 gezeigten Dateiauswahldialog lieber im Windows-Design erleben möchte, kann die Swing-Option nutzen, das Look & Feel einer Anwendung zwischen folgenden Alternativen umzuschalten (Stand: Java 6, Update 16):

- **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 - klassisch**

Bevor man ein solches Look & Feel - Menü präsentieren kann,




ist etwas Arbeit angesagt.

Neben einem **JMenu**-Objekt und einem **JMenuItem**-Array zur Aufnahme der aktuell verfügbaren Look & Feel - Optionen deklarieren wir einen Array der Klasse **LookAndFeelInfo**, die als statische Mitgliedklasse (vgl. Abschnitt 11.6.6.1) im Rumpf der Klasse **UIManager** definiert ist:

```
private JMenu lafMenu;
private JMenuItem[] lafItems;
private UIManager.LookAndFeelInfo[] lafs;
```

Im Rahmenfensterkonstruktor werden die Objekte generiert und initialisiert:

```
lafMenu = new JMenu("Look & Feel");
lafs = UIManager.getInstalledLookAndFeels();
lafItems = new JMenuItem[lafs.length];
UIActionHandler uah = new UIActionHandler();
for (int i = 0; i< lafs.length; i++) {
   lafItems[i] = new JMenuItem(lafs[i].getName());
   lafItems[i].addActionListener(uah);
   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 JRE ab, so dass unser Editor von zukünftigen Neuerungen profitieren wird.

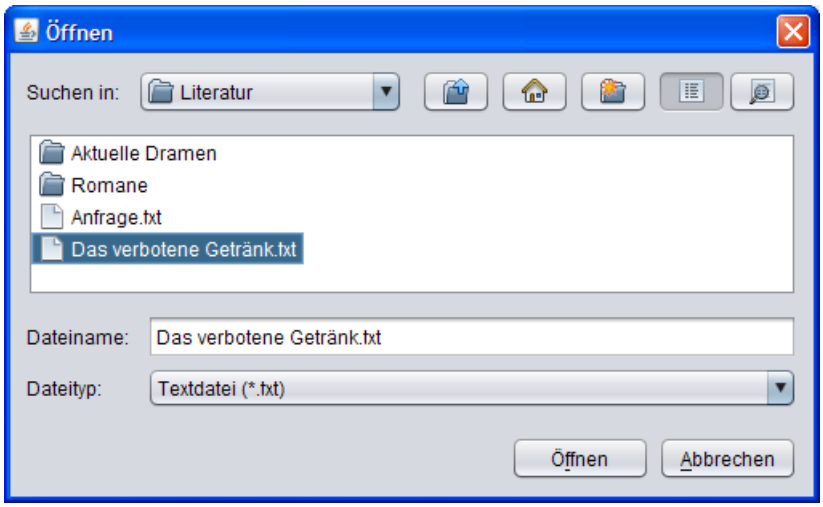
Für den **ActionListener** zu den Look & Feel - Menüitems definieren wir eine interne Klasse:

```
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);
        } 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. Diese Referenz ist in der inneren Klasse `UIActionHandler` über das Schlüsselwort **this** mit vorangestelltem Klassennamen verfügbar:

```
SwingUtilities.updateComponentTreeUI(Editor.this);
```

Bei aktivem **Nimbus** - 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)**

### 11.8.2 Zwischenablage

Mit Hilfe der System-Zwischenablage (vom Betriebssystem verwaltet) können Anwender auf bequeme Weise Daten zwischen Anwendungen austauschen 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 **JEditorPane**) erlauben dem Benutzer den Zwischenablagentransfer von Zeichenfolgen über die systemüblichen

Tastenkombinationen und bieten dem Programmierer über die Methoden **cut()**, **copy()** und **Paste()** einen bequeme Schnittstelle 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 11.8.3).

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. **text/plain**, **image/jpeg**).

Damit ein Objekt den gesamten Zwischenablageninhalt repräsentieren kann, muss seine Klasse die Methoden der Schnittstelle **Transferable** implementieren:

- **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.
- **public DataFlavor[] getTransferDataFlavors()**
  Man erhält einen Array der Klasse **DataFlavor** und kennt damit alle in der Zwischenablage vorhandenen Formate.

Für den Zugriff auf die Zwischenablage verwendet man ein Objekt der Klasse **Clipboard**, dessen Adresse mit Hilfe der Klasse **Toolkit** zu ermitteln ist:

```
private Clipboard clip;
private Transferable clipContent;
  . . .
clip = Toolkit.getDefaultToolkit().getSystemClipboard();
```

Das **Toolkit**-Objekt beherrscht u.a. die folgenden Methoden:

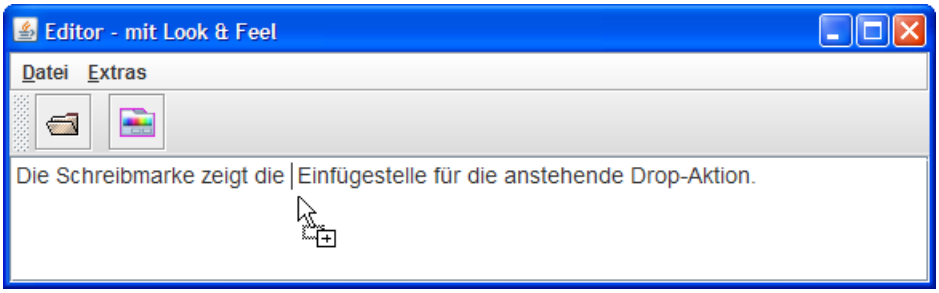
- **public Transferable getContents(Object** *requestor***)**
  Über diese Methode verschafft man sich ein Objekt, das die gesamte Zwischenablage (mit allen vorhandenen Formaten) repräsentiert und die Methoden der Schnittstelle **Transferable** beherrscht. Weil der **getContents()**-Parameter derzeit 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 über Änderungen der verfügbaren Zwischenablagendatenformate informiert werden, registriert man beim **Clipboard**-Objekt einen Ereignisempfänger, der das Interface **FlavorListener** erfüllt, also die Methode **flavorsChanged()** implementiert.

### 11.8.3 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 festlegen, 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 Programmierer realisieren.

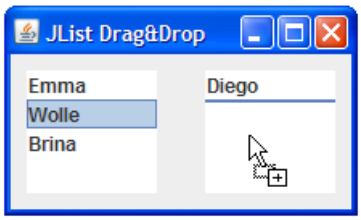
Etliche Klassen (u.a. **JTextField** und **JEditorPane**) beherrschen die Empfängerrolle bereits ab Fabrik, z.B.:




Es genügt ein einfacher Aufruf der Methode **setDragEnabled()**, um auch die Quellfunktionalität zu aktivieren, z.B. beim Editor, den wir seit Abschnitt 11.7.7 sukzessive ausbauen:

```
text.setDragEnabled(true);
```

Auch bei der Klasse **JList** 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. Wir erstellen ein Beispielprogramm mit zwei Listen, das ein Verschieben und Kopieren von Einträgen zwischen den Listen oder innerhalb einer Liste erlaubt:





Außerdem ist sogar der **String**-Dateiaustausch mit anderen Java- und Windows-Programmen möglich.

Im Vergleich zum ähnlichen Beispielprogramm in Abschnitt 11.7.5.1 ist jede **JList**-Komponente in eine **JScrollPane**-Komponente verpackt, die bei Bedarf Rollbalken beisteuert und außerdem dafür sorgt, dass auch eine leere Liste sichtbar und ablagefähig bleibt.

Ein **JList**-Objekt benötigt einen Transporthelfer aus einer geeigneten Spezialisierung der Klasse **javax.swing.TransferHandler**. Im Beispielprogramm kommt ein Objekt aus der von **TransferHandler** abgeleiteten inneren Klasse `ListTransferHandler` zum Einsatz, deren Methoden anschließend erläutert werden:

```
private class ListTransferHandler extends TransferHandler {
  private JList quellListe, zielListe;
  private int quellIndex, zielIndex;
. . .
}
```

Beide **JList**-Objekte engagieren denselben Transporthelfer:

```
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.

#### *11.8.3.1 TransferHandler-Methoden für die Drag-Rolle*

Über eine **JList**-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**-Objekt implementiert:

```
protected Transferable createTransferable(JComponent quelle) {
  quellListe = (JList) 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++;
    quellListe = (JList) quelle;
    ((DefaultListModel) quellListe.getModel()).remove(quellIndex);
  }
}
```

Weil das Verschieben in andere Java-Programme (z.B. Eclipse) ebenfalls klappt, lassen sich Listeneinträge entfernen. Ist das Ablageziel ein Windows-Programm, wird jedoch unabhängig vom Benutzerwunsch kopiert.

#### *11.8.3.2 TransferHandler-Methoden für die Drop-Rolle*

Über ein **JList**-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 false;
  else
    return true;
}
```

Bei der Spezifikation von unterstützten Datentypen kommen Konstanten (Felder mit den Modifikatoren **public, static und final**) der in Abschnitt 11.8.2 vorgestellten Klasse **DataFlavor** zum Einsatz. Wie das Beispiel zeigt, wird für deren Namen leider das Camel-Casing verwendet.

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) info.getComponent();

   DefaultListModel listModel = (DefaultListModel) 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**-Objekt die bestmögliche Drop-Rückmeldung für den Benutzer zu erhalten (z.B. Anzeige der Einfügestelle), setzt man den **DropMode** wie im folgenden Beispiel:

```
listLinks.setDropMode(DropMode.ON_OR_INSERT);
listRechts.setDropMode(DropMode.ON_OR_INSERT);
```

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

**…\BspUeb\Swing\Drag & Drop**

## *11.9 Übungsaufgaben zu Kapitel 11*

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Zu jeder Event Id gehört 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 Fenster 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:

- 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**.

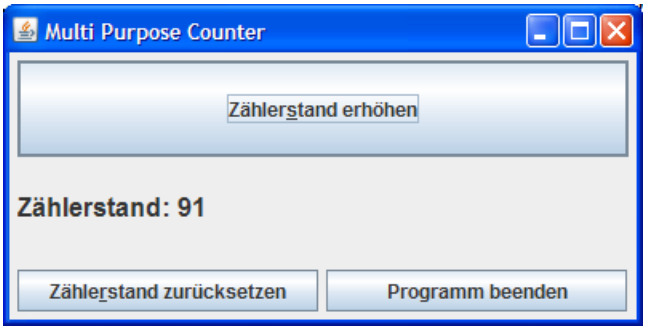
4) Zu einer anonymen Klasse lässt sich nur *eine* Instanz erzeugen. Es ist aber durchaus möglich, ein solches Objekt z.B. als **ActionListener** für mehrere Befehlsschalter zu verwenden, indem der zuerst versorgte Schalter mit **getActionListeners()** nach dem zuständigen Ereignisempfänger befragt und 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 9 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 11.3, z.B. mit folgender Bedienoberfläche:



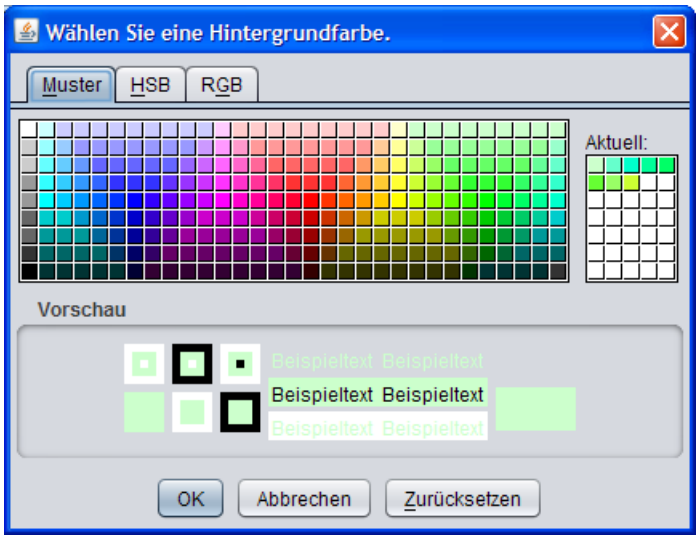

Bieten Sie statt der verspielten Icon-Anzeige Schalter an, um den Zählenstand zurück zu setzen und das Programm zu beenden. Verhindern Sie mit der Methode **setMinimumSize()** der **JFrame**-Basisklasse **java.awt.Window**, dass beim Verkleinern des Rahmenfensters Bedienelemente verschwinden. Über die **JComponent**-Methode **setFont()** können Sie die Schriftart der Zählerstandsanzeige beeinflussen, z.B.:

```
lblCounter.setFont(new Font(Font.SANS_SERIF, Font.BOLD, 16));
```

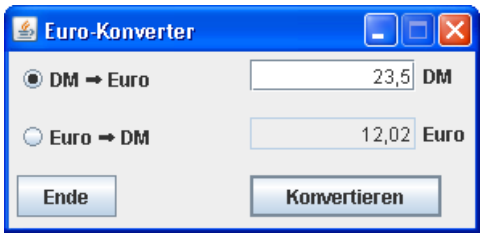
9) Realisieren Sie für das Editor-Beispielprogramm ausgehend vom Entwicklungsstand in

**…\BspUeb\Swing\Editor\E1 (mit Menü)**

einen **ActionListener** zum Menüitem **Extras > Hintergrundfarbe** unter Verwendung der Klasse **JColorChooser**, die einen attraktiven Standarddialog zur Farbauswahl bietet:




10) Erstellen Sie den in Abschnitt 11.7.4 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.

# 12 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, z.B.:

- Primitive Werte (Typ **byte**, **int**, **double** etc.) oder ganze Objekte in eine Datei auf der Festplatte schreiben bzw. von dort lesen
- Zeichen auf den Bildschirm schreiben bzw. von der Tastatur entgegen nehmen

Vorausblick auf zwei verwandte Themen:

- In einem späteren Kapitel werden Sie mit den Netzwerkverbindungen weitere, außerordentlich wichtige Datenquellen bzw. -senken kennen lernen und dabei von Ihren Kenntnissen über die generelle Datenstromtechnik profitieren.
- Im Kapitel über Datenbankprogrammierung werden anspruchsvolle Datenverwaltungstechniken vorgestellt, die sich auch im Netzwerk- und Mehrbenutzerkontext bewähren. Dabei überlassen wir aber 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 von Dateien mit wahlfreiem Zugriff (siehe z.B. Ullenboom 2009). Während die anschließend behandelten Methoden eine Datei unidirektional vorwärts durchlaufen, ist beim wahlfreien Zugriff auch eine Rückwärtsbewegung möglich.

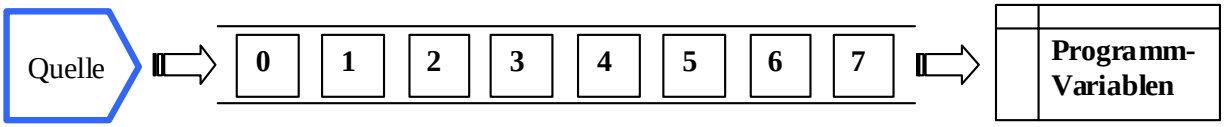
Die behandelten Java-Klassen zur Datenein- und -ausgabe befinden sich im Paket **java.io**, das folglich von jedem betroffenen Programm importiert werden sollte, z.B.:

```
import java.io.*;
```

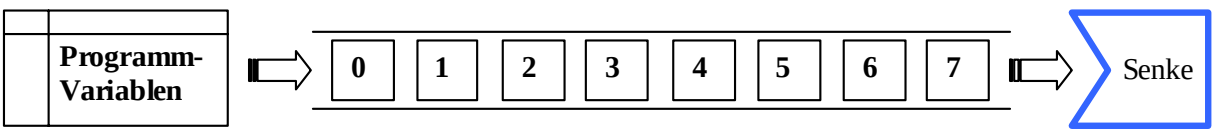
## *12.1 Grundlagen*

### 12.1.1 Datenströme

In Java wird die sequentielle Datenein- und -ausgabe über so genannte *Ströme* (engl. *Streams*) abgewickelt. Ein Programm liest Bytes[54] aus einem **Eingabestrom**, der aus einer Datenquelle (z.B. Datei, Eingabegerät, Netzwerkverbindung) gespeist wird:




Ein Programm **schreibt** Bytes in einen **Ausgabestrom**, der die Werte von Programmvariablen zu einer Datensenke befördert (z.B. Datei, Ausgabegerät, Netzverbindung):




In der Regel kommen *externe* Quellen bzw. Senken zum Einsatz (Dateien, Geräte, Netzwerkverbindungen). Gelegentlich werden aber programminterne Objekte per Datenstromtechnik angesprochen

[54] Wenn in Abschnitt 12 der Namensteil *Byte* auftaucht, ist keine Java-Wrapper-Klasse gemeint, sondern eine 8 Bit umfassende Informationseinheit der Datenverarbeitung.

(z.B. **byte**-Arrays, **String**-Objekte). Zu den internen Strömen gehören auch die so genannten *Pipes* zum Kommunikation zwischen verschiedenen Threads, mit denen wir uns in diesem Manuskript aus Zeitgründen nicht beschäftigen können.

Ziel des Datenstromkonzepts ist es, Ein- und Ausgaben möglichst unabhängig von den Besonderheiten konkreter Datenquellen und –senken formulieren zu können.

Ein- bzw. Ausgabeströme werden in Java-Programmen durch Objekte aus geeigneten 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.

### 12.1.2 Beispiel

Das folgende Programm schreibt einen **byte**-Array in eine Datei und liest die Daten anschließend wieder zurück:

Quellcode

```
import java.io.*;

class EaEinstieg {
   public static void main(String[] args) throws IOException {
      String name = "demo.txt";
      byte[] arr = {0,1,2,3,4,5,6,7};

      FileOutputStream fos = null;
      try {
         fos = new FileOutputStream(name);
         fos.write(arr);
      } finally {
         if (fos != null)
            fos.close();
      }

      FileInputStream fis = null;
      try {
         fis = new FileInputStream(name);
         fis.read(arr);
         for (int i : arr)
            System.out.println(i);
      } finally {
         if (fos != null)
            fis.close();
      }
   }
}
```

Ausgabe

```
0
1
2
3
4
5
6
7
```

Zum Schreiben wird das Ausgabestromobjekt `fos` aus der Klasse **FileOutputStream** erzeugt und mit einer Datei verbunden. Nachdem es seine Schreibarbeit in der Methode **write()** verrichtet hat, wird die Datei per **close()**-Aufruf freigegeben. Damit diese Freigabe unter allen Umständen stattfindet, wird sie in einem **finally**-Block vorgenommen.

Zum Lesen wird das Eingabestromobjekt `fis` aus der Klasse **FileInputStream** erzeugt und mit der zuvor gefüllten Datei verbunden. Nach dem Lesen per **read()**-Methode gibt das Programm die nicht mehr benötigte Datei in einem **finally**-Block per **close()**-Aufruf wieder frei.

Die beiden Stromklassen-Konstruktoren sowie die Methoden **write()** und **read()** werfen potentiell von **IOException** abstammende, also behandlungs- bzw. deklarationspflichtige Ausnahmen (vgl. Abschnitt 9.5). Weil es Beispielprogramm solche Fehler nicht reparieren, sondern bestenfalls in einer eigenen Meldung beschreiben könnte, verzichtet es auf eine Behandlung und beschränkt sich

auf eine **throws**-Klausel im Definitionskopf. Mit den **try-finally** -Anweisungen wird aber dafür gesorgt, dass die **close()**-Aufrufe auf jeden Fall stattfinden.

### 12.1.3 Klassifikation der Stromverarbeitungsklassen

Das Paket **java.io** enthält vier abstrakte Basisklassen, von denen die für uns relevanten Stromverarbeitungsklassen abstammen:

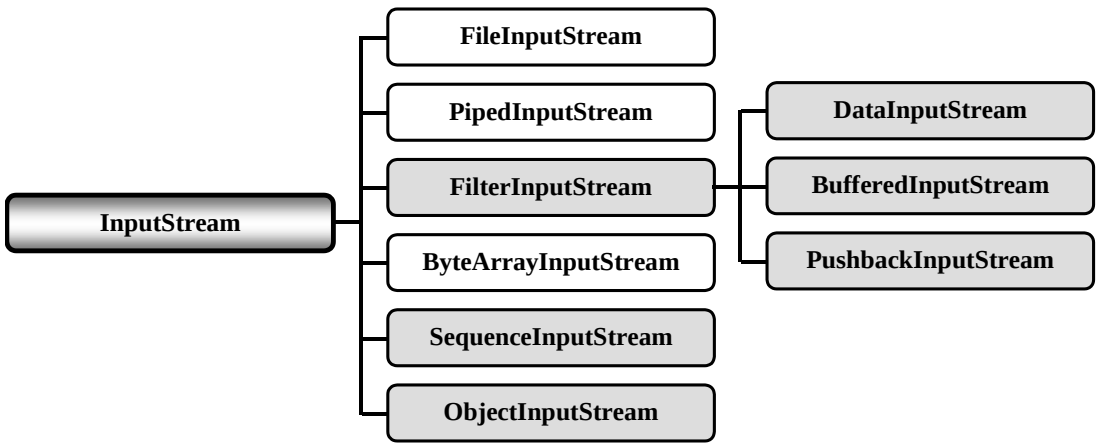
- **InputStream** und **OutputStream**
  Die Klassen aus den zugehörigen Hierarchien verarbeiten **Ströme mit Bytes** als Elementen. Byteströme werden für das Lesen und Schreiben von **binären Daten** verwendet (z.B. Bitmap-Graphiken).
- **Reader** und **Writer**
  Die Klassen aus den zugehörigen Hierarchien verarbeiten **Zeichenströme**, die aus einer Folge von Zeichen in einer bestimmten Kodierung (z.B. Unicode oder ASCII) bestehen. Landet ein Zeichenstrom in einer Datei, kann diese anschließend mit jedem Texteditor bearbeitet werden, der die verwendete Kodierung beherrscht.

Während die byteorientierten 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 **direktem Kontakt zu Datenquellen bzw. –senken**. Sollen z.B. Bytes aus einer Datei gelesen werden, kommt ein Objekt der Klasse **FileInputStream** zum Einsatz.
- **Filterklassen**
  Sie dienen zum **Filtern** bzw. **Transformieren** von Eingabe- bzw. Ausgabeströmen. 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:



### 12.1.4 Aufbau und Verwendung der Filterklassen

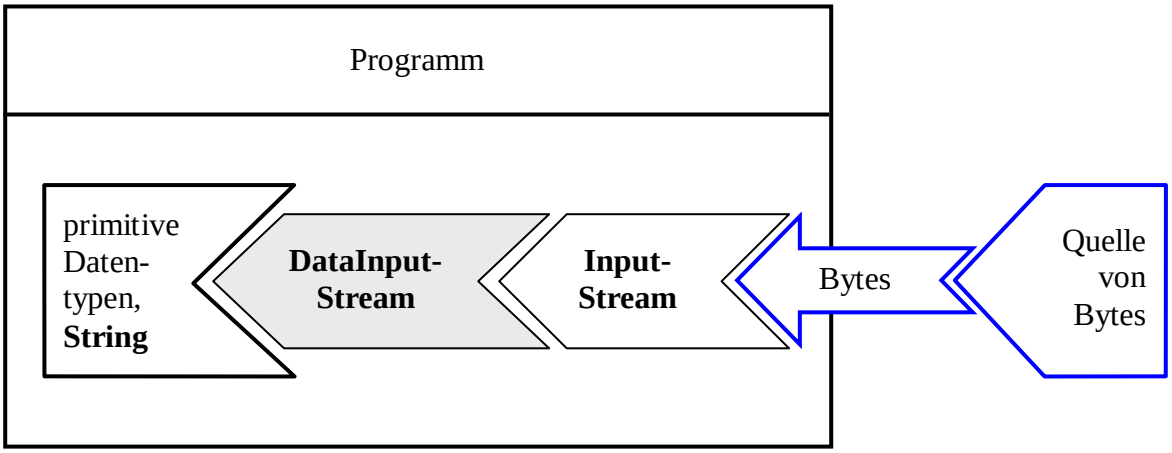
Eine Filter- bzw. Transformationsklasse baut auf einer Ein- bzw. Ausgabeklasse auf und stellt Methoden für eine erweiterte Funktionalität zur Verfügung. Wie diese Zusammenarbeit organisiert wird, betrachten wir am Beispiel der Eingabetransformationsklasse **DataInputStream** aus der **InputStream**-Hierarchie. Diese Klasse besitzt …

- eine Instanzvariable vom Typ **InputStream**,
- in ihrem Konstruktor einen Parameter vom Typ **InputStream**, dessen Aktualwert 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, z.B.:

```
FileInputStream fis = new FileInputStream(name);
DataInputStream dis = new DataInputStream(fis);
```

Die Transformationsleistung eines **DataInputStream**–Objekts besteht darin, Werte primitiver Datentypen aus einer **byte**-Sequenz passender Länge zusammenzusetzen. Der Filter nimmt also Bytes entgegen und liefert z.B. **int**-Werte (bestehend aus jeweils 4 Bytes) aus. Aus dem elementaren Bytestrom wird ein Strom, dem Daten von primitivem Typ entnommen werden können:




Wird für ein **DataInputStream**–Objekt die **close()**-Methode aufgerufen, dann leitet es diese Botschaft an das verbundene **InputStream**-Objekt weiter.

Im folgenden Beispielprogramm kooperiert die Klasse **DataInputStream** mit der Eingabeklasse **FileInputStream**, um **int**-Werte aus einer Datei zu lesen. Zuvor werden diese **int**-Werte in dieselbe Datei geschrieben, wobei die Ausgabetransformationsklasse **DataOutputStream** und die Ausgabeklasse **FileOutputStream** kooperieren. Hier zerlegt der Filter die **int**-Werte in einzelne Bytes und schiebt sie in den Ausgabestrom.

| Quellcode | Ausgabe |
|---|---|

```
import java.io.*;

class DataInOutputStream {
  public static void main(String[] egal) throws IOException {
    String name = "demo.txt";
    int[] arr = {1024,2048,4096,8192};

    DataOutputStream dos = null;
    try {
      dos = new DataOutputStream(new FileOutputStream(name));
      for (int i=0; i<arr.length; i++)
        dos.writeInt(arr[i]);
    } finally {
      if (dos != null)
        dos.close();
    }

    DataInputStream dis = null;
    try {
      dis = new DataInputStream(new FileInputStream(name));
      for (int i=0; i<arr.length; i++)
        System.out.println(arr[i] = dis.readInt());
    } finally {
      if (dis != null)
        dis.close();
    }
  }
}
```

```
1024
2048
4096
8192
```

Am Beispiel **DataInputStream** sollen noch einmal wichtige Merkmale einer Transformations- bzw. Filterklasse zusammengefasst werden:

- Die Klasse **DataInputStream** besitzt eine Instanzvariable vom Typ **InputStream**, über die der Kontakt zu einer Datenquelle hergestellt wird. Diese wird im Konstruktor initialisiert.
- 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 verwendet man die Eingabeklasse **FileInputStream**.
- Ein Aufruf der **DataInputStream**-Methode **close()** wird an das verbundene **InputStream**-Objekt durchgereicht.

### 12.1.5 Zum guten Schluss

Ist ein Datenstromobjekt mit einer externen Quelle oder Senke verbunden, ist eine Ressource (z.B. Datei oder Netzwerkverbindung) belegt, die für andere Prozesse nicht mehr (uneingeschränkt) zur Verfügung steht. Nach der Programmbeendigung sind die 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 von Dateien).

Außerdem setzen viele Ausgabestromklassen 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 oder Schreiben von Daten nicht mehr zu gebrauchen.

Einen expliziten **close()**-Aufruf zu unterlassen, hat oft 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 Bedarf zum Einsatz kommt. Erst recht sind Zeitpunkt und Reihenfolge der Aufrufe für verschiedene Objekte ungewiss. Im Java-Tutorial (Sun Microsystems 2009) 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**() nicht aufgerufen (siehe z.B. Abschnitt 12.3.1.4).

Um allen Problemen aus dem Weg zu gehen, schließt man jeden Strom so früh wie möglich durch einen expliziten **close()**-Aufruf. Bei manchen programminternen Quellen oder Senken (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. Es kann allerdings der (mehr oder weniger unwahrscheinliche) Fall auftreten, dass nach dem erfolgreichen Öffnen eines 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 Ausgabestrom bliebe offen.

Damit ein **close()**-Aufruf auch bei Ausnahmefehlern unter allen Unständen ausgeführt wird, sollte er in der Regel in einem **finally**-Block erfolgen (vgl. Abschnitt 9.2.1.2).

## *12.2 Verwaltung von Dateien und Verzeichnissen*

Der Umgang mit Dateien und Verzeichnissen (z.B. Erstellen, auf Existenz prüfen, Löschen, Attribute lesen und setzen) wird in Java 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};
  . . .
  }
}
```

### 12.2.1 Verzeichnis anlegen

Zunächst legen wir das Verzeichnis

**U:\Eigene Dateien\Java\FileDemo\AusDir**

an:

```
String dname = "U:/Eigene Dateien/Java/FileDemo/AusDir";
File dir = new File(dname);
if (dir.exists()) {
    if (dir.isDirectory())
      System.out.println("Das Verzeichnis "+dname+" existiert bereits.");
    else {
      System.out.println(dname+" existiert, ist aber kein Verzeichnis.");
} else
 if (dir.mkdirs())
  System.out.println("Verzeichnis "+dname+" erstellt");
 else {
  System.out.println("Verzeichnis "+dname+" 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 Rückwärts-Trennstrich in Java eine Escape-Sequenz einleitet, muss unter Windows zwischen Pfadbestandteilen entweder der gewöhnliche Trennstrich oder ein verdoppelter Rückwärts-Trennstrich 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).

### 12.2.2 Dateien explizit erstellen

Zwar wird z.B. beim Erzeugen eines **FileOutputStream**-Objekts eine verknüpfte 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 = dname+"/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.

### 12.2.3 Informationen über Dateien und Ordner

Neben **isDirectory()** kennen **File**-Objekte noch weitere Informationsmethoden, z.B.:

- **String getAbsolutePath()**
  Ermittelt den absoluten Pfadnamen
- **long lastModified()**
  Ermittelt den Zeitpunkt der letzten Änderung, gemessen in Millisekunden seit dem 1. Januar 1970 (00:00:00 GMT)
- **long length()**
  Stellt die Größe einer Datei in Bytes fest
- **boolean canWrite()**
  Prüft, ob das Programm schreibend auf eine Datei zugreifen darf
- **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 Aenderung:   " + time);
System.out.println("  Groesse in Bytes:   " + f.length());
System.out.println("  Schreiben moeglich: " + 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:   28.01.10 17:59
  Groesse in Bytes:   3
  Schreiben moeglich: true
```

Für die formatierte Ausgabe der **lastModified()**-Rückgabe sorgen ein **Date**- und ein **DateFormat**-Objekt.

### 12.2.4 Atribute ändern

Man kann etliche Attribute von Dateien oder Ordnern ändern, z.B.:

- **boolean setLastModified(long** *time***)**
  Legt für eine Datei oder einen Ordner den Zeitpunkt der letzten Änderung neu fest
- **boolean setWritable(boolean** *writable***)(**
  Setzt oder entfernt den Schreibschutz

Hier werden die Anforderungen an ein **File**-Objekt gerichtet, das eine Datei repräsentiert:

```
Date d = null;
DateFormat df = DateFormat.getInstance();
try {
  d = df.parse("24.01.00 16:15");
} catch (Exception e) {}
f.setLastModified(d.getTime());
f.setWritable(false);
```

### 12.2.5 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("\nDateien 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
```

### 12.2.6 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(dname+"/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.

### 12.2.7 Löschen

Mit der **File**-Methode **delete()** löscht man eine Datei oder einen Ordner, z.B.:

```
if (fn.delete())
   System.out.println("Datei "+fn.getName()+" geloescht");
else {
   System.out.println("Fehler beim Loeschen der Datei "+fn.getName());
   System.exit(1);
}

if (dir.delete())
   System.out.println("Verzeichnis "+dir.getName()+" geloescht");
else {
   System.out.println("Fehler beim Loeschen des Ordners "+dir.getName());
   System.exit(1);
}
```

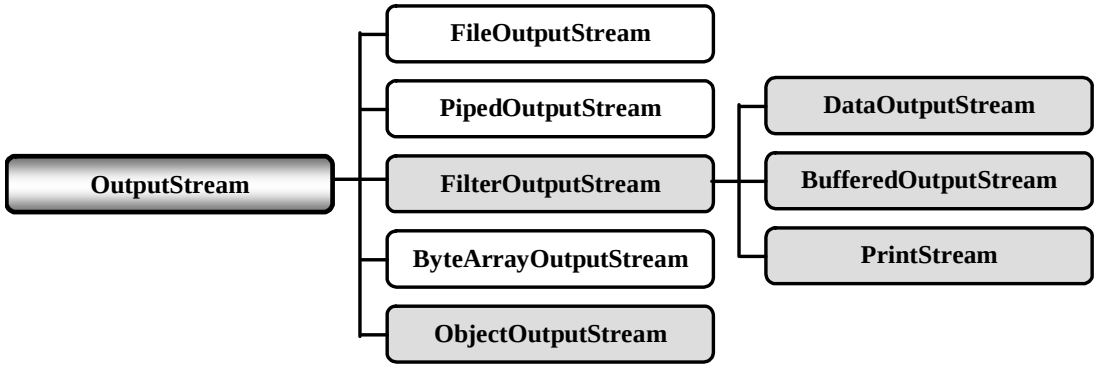
## *12.3 Klassen zur Verarbeitung von Byteströmen*

In Java 1.0 stammten *alle* Ein-/Ausgabeklassen von **InputStream** oder **OutputStream** ab. Während sich diese Klassen zur Ein- und Ausgabe von primitiven Datenwerten und Objekten bewährten, machte die Behandlung von Unicode-Zeichen vor allem beim Internationalisieren von Java-Software Probleme. Mit Java 1.1 wurden daher zur Verarbeitung von Textdaten die neuen Basisklassen **Reader** und **Writer** mit ihren Klassenhierarchien eingeführt. Für andere Ein-/Ausgabeprobleme sind aber nach wie vor die von **InputStream** oder **OutputStream** abstammenden byteorientierten Klassen adäquat. An einigen Stellen (z.B. bei der Standardausgabe) haben die alten Lösungen zur Zeichenverarbeitung überlebt.

### 12.3.1 Die OutputStream-Hierarchie

#### *12.3.1.1 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:



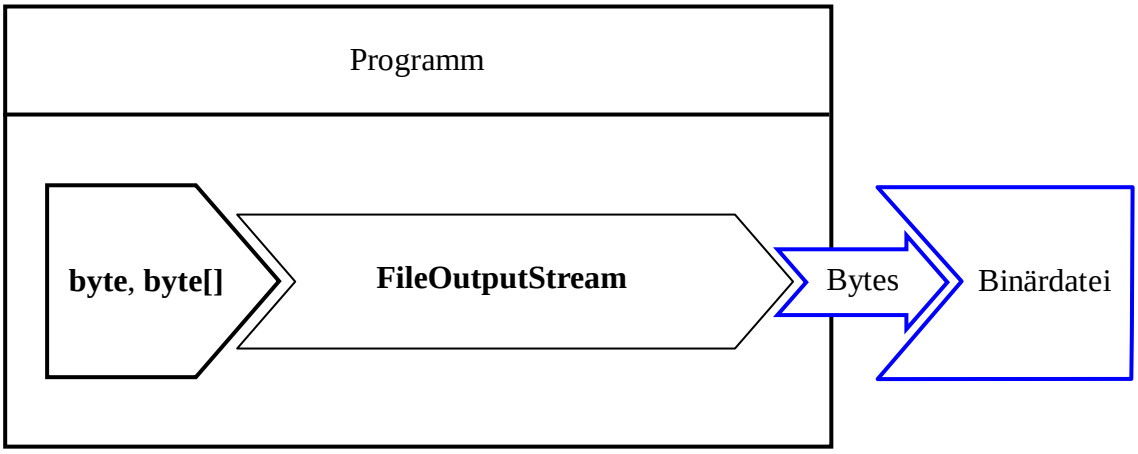

Mit der Transformationsklasse **ObjektOutputStream** können komplette Objekte in einen byteorientierten Ausgabestrom geschrieben werden. Sie wird zusammen mit ihrem Gegenstück **ObjektInputStream** in Abschnitt 12.6 behandelt.

Die folgenden **OutputStream**-Unterklassen werden in diesem Manuskript *nicht* näher beschrieben:

- **PipedOutputStream**
  Objekte dieser Klasse schreiben Bytes in eine Pipe, die zur Kommunikation zwischen Threads dient.
- **ByteArrayOutputStream**
  Objekte dieser Klasse schreiben Bytes in einen **byte**-Array, also in eine programminterne Datensenke.

#### *12.3.1.2 FileOutputStream*

Ein **FileOutputStream**-Objekt ist mit einer Datei verbunden, die vom Konstruktor im Schreibmodus geöffnet und nötigenfalls automatisch erstellt wird. Die drei verfügbaren **write()**-Methoden befördern die Inhalte von **byte**–Variablen oder –Arrays in die Ausgabedatei:




Im **FileOutputStream**-Konstruktor wird die anzusprechende Datei über ein **File**-Objekt (siehe Abschnitt 12.2) 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*)

Der Konstruktor wirft eine (behandlungspflichtige) 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.

Weil **FileOutputStream**-Objekte nur **byte**–Variablen oder –Arrays befördern können, werden sie oft mit Filterobjekten (z.B. aus der Klasse **DataOutputStream**) kombiniert, die reichhaltigere Ausgabemethoden bieten (siehe unten). Im folgenden Beispielprogramm zum Kopieren von Dateien: ist diese Einschränkung jedoch irrelevant:

```
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)
     byte[] buffer = new byte[buflen];
     int nread;
     long zeit, total = 0;
     FileInputStream  fis = null;
     FileOutputStream fos = null;
     try {
        zeit = System.currentTimeMillis();
        fis = new FileInputStream(quelle);
        fos = new FileOutputStream(ziel);
        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. "+
             "(Benoetigte Zeit: "+zeit+" Millisekunden.)");
     } catch (IOException e) {
        System.out.println(e.getMessage());
     } finally {
        if (fis != null) try {fis.close();} catch (IOException ioe) {}
        if (fos != null) try {fos.close();} catch (IOException ioe) {}
     }
  }
}
```

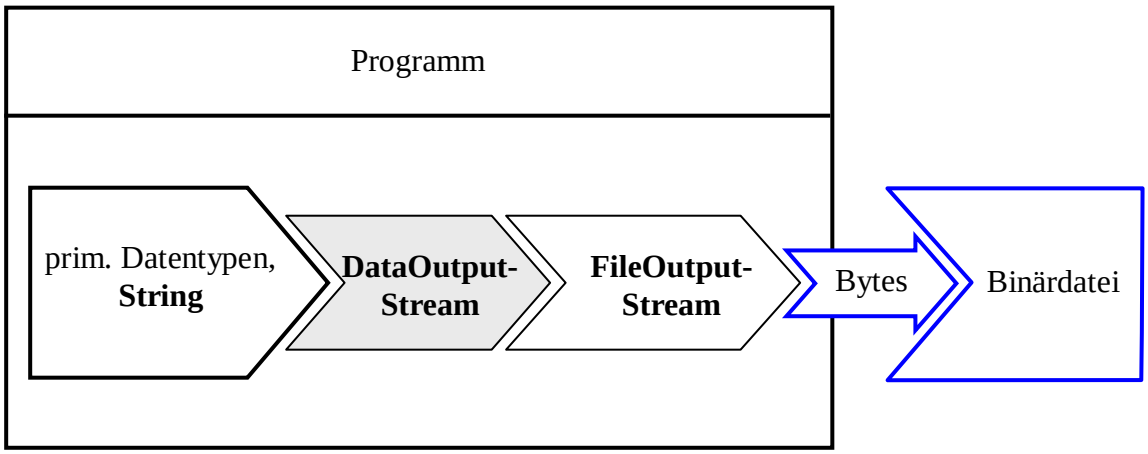
In der zentralen **while**-Schleife wird mit der **FileInputStream**–Methode **read()** aus der Quelldatei jeweils ein Megabyte oder aber die per **available()**-Aufruf ermittelte Restmenge (vgl. Abschnitt 12.3.2.2) gelesen und anschließend von der **FileOutputStream**–Methode **write()** in die Zieldatei 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.)
```

Weil sich die **main()**-Methode *nicht* per **throws**-Klausel der Pflicht zur Behandlung von **IOException**-Objekten entzieht, sind entsprechende **catch**-Blöcke erforderlich. Davon sind auch die **close()**-Aufrufe im **finally**-Block betroffen, so dass eingeschachtelte **try-catch**-Anweisungen erforderlich sind. Auf eine Behandlung der bei einem **close()**-Aufruf möglichen Ausnahmen darf hier aber verzichtet werden.

#### *12.3.1.3 DataOutputStream*

Mit einem Objekt aus der Transformationsklasse **DataOutputStream** lassen sich die Werte primitiver Datentypen sowie **String**-Objekte über einen **OutputStream** (z.B. einen **FileOutputStream**) in eine Datensenke befördern:




Ein **DataOutputStream** beherrscht diverse Methoden zum Schreiben primitiver Datenwerte (**writeInt()**, **writeDouble()**). Mit **writeUTF()** steht auch eine Methode zur Ausgabe von Zeichen bereit, wobei eine *modifizierte* Variante der UTF-8 - Kodierung (vgl. Abschnitt 12.4.1.2) zum Einsatz kommt. Diese Methode ist angemessen, sofern die resultierenden Zeichen später mit der **DataInputStream**-Methode **readUTF()** wieder eingelesen werden sollen (vgl. Abschnitt 12.3.2.3). Für universell verwendbare Textdateien ist die Klasse **OutputStreamWriter** mit einstellbarer und normkonformer Kodierung weit besser geeignet.

Im folgenden Beispielprogramm wird ein **DataOutputStream** vor einen **FileOutputStream** geschaltet und dann beauftragt, Daten vom Typ **int**, **double** und **String** zu schreiben:

```
import java.io.*;
class DataOutputStreamDemo {
  public static void main(String[] egal) {
    String name = "demo.dat";
    DataOutputStream dos = null;
    DataInputStream dis = null;

    try {
      dos = new DataOutputStream(new FileOutputStream(name));
      dos.writeInt(4711);
      dos.writeDouble(Math.PI);
      dos.writeUTF("DataOutputStream-Demo");
    } catch (Exception e) {
      System.out.println(e.getMessage());
    } finally {
      if (dos != null)
        try {dos.close();} catch (IOException ioe) {};
    }

    try {
      dis = new DataInputStream(new FileInputStream(name));
      System.out.println("readInt()-Ergebnis:\t"+dis.readInt()+
                         "\nreadDouble()-Ergebnis:\t"+dis.readDouble()+
                         "\nreadUTF()-Ergebnis:\t"+dis.readUTF());
    } catch (Exception e) {
      System.out.println(e.getMessage());
    } finally {
      if (dis != null)
        try {dis.close();} catch (IOException ioe) {};
    }
  }
}
```

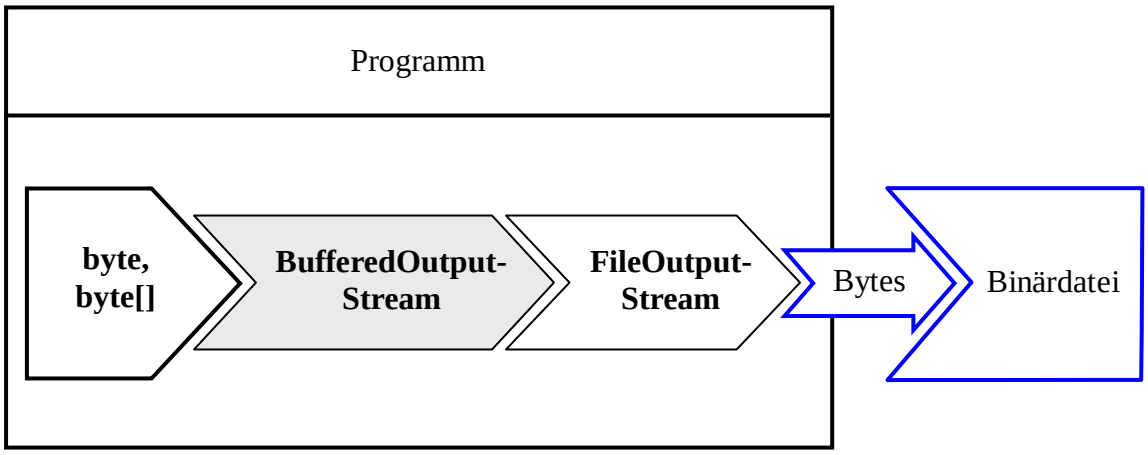
Ein **DataInputStream** holt in Kooperation mit einem **FileInputStream** die Werte zurück (vgl. Abschnitt 12.3.2):

```
readInt()-Ergebnis:      4711
readDouble()-Ergebnis:   3.141592653589793
readUTF()-Ergebnis:      DataOutputStream-Demo
```

#### *12.3.1.4 BufferedOutputStream*

Zur Beschleunigung von Ein- oder Ausgaben setzt man oft Transformationsklassen ein, die durch das Zwischenspeichern von Daten die Anzahl der (oft 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 12.1.4). Optional kann die voreingestellte Puffergröße von 512 Bytes geändert werden:

- **public BufferedOutputStream(OutputStream** *out*)
- **public BufferedOutputStream(OutputStream** *out***, int** *size*)

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 = 500000;
  public static void main(String[] args) {
    long time;
    FileOutputStream fos = null;
    try {
      time = System.currentTimeMillis();
      fos = new FileOutputStream("noBuffer.dat");
      for (int i = 1; i <= ANZAHL; i++)
        fos.write(0);
      System.out.println("Zeit fuer die ungepufferte Ausgabe: " +
          (System.currentTimeMillis() - time));
    } catch (Exception e) {
      System.out.println(e.getMessage());
    } finally {
      if (fos != null)
        try {fos.close();} catch (IOException ignored) {};
    }
```

```
      BufferedOutputStream bos = null;
      try {
        time = System.currentTimeMillis();
        fos = new FileOutputStream("Buffer.dat");
        bos = new BufferedOutputStream(fos, 8192);
        for (int i = 1; i <= ANZAHL; i++)
          bos.write(i);
        System.out.println("Zeit fuer die gepufferte Ausgabe:   " +
            (System.currentTimeMillis() - time));
      } catch (Exception e) {
        System.out.println(e.getMessage());
      } finally {
        if (bos != null)
          try {bos.close();} catch (IOException ignored) {};
      }
    }
}
```

Durch den Einsatz des **BufferedOutputStream**-Objekts (mit einer Puffergröße von 8192 Bytes) kann der Zeitaufwand beim Schreiben erheblich reduziert werden. Schon beim Schreiben auf eine lokale Festplatte treten erhebliche Unterscheide auf (Angaben in Millisekunden):

```
Zeit fuer die ungepufferte Ausgabe: 2672
Zeit fuer die gepufferte Ausgabe:   62
```

Ist das Ziel eine Freigabe im lokalen Netzwerk, steigt der Zeitaufwand für die ungepufferte Ausgabe dramatisch an:

```
Zeit fuer die ungepufferte Ausgabe: 317160
Zeit fuer die gepufferte Ausgabe:   63
```

Wegen des erheblichen Performanzvorteils sollte also ein Ausgabepuffer eingesetzt werden, wenn zahlreiche Schreibvorgänge mit jeweils kleinem Volumen stattfinden. Das `FileCopy`-Beispielprogramm in Abschnitt 12.3.1.2 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 liegenden **OutputStream** schließt, wie ein Blick in den Quellcode der **BufferedOutputStream**–Basisklasse **FilterOutputStream** zeigt:[55]

```
public void close() throws IOException {
  try {
    flush();
  } catch (IOException ignored) {
  }
  out.close();
}
```

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 12.1.5).

[55] 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.

Das obige Programm schreibt exakt 500.000 Bytes in beide Ausgabedateien. Entfernt man die **finally**-Klausel mit

```
bos.close();
```

dann befinden sich nach Programmende in der Ausgabedatei **Buffer.dat** nur 499712 (= 61 · 8192) Bytes!

#### *12.3.1.5 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 12.1.3 sind bei der Zeichenstromverarbeitung eigentlich 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 12.4.1.2). 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.[56] Das Kodierungsproblem bei der Textausgabe per **PrintStream**-Objekt kennen Sie bereits aus den vergeblichen Versuchen, unter Windows Umlaute auf die Konsole zu schreiben. Wir kommen am Ende dieses Abschnitts noch mal darauf zurück.

Ein **PrintStream**-Objekt kann mit Hilfe seiner vielfach überladenen Methoden **print()** und **println()** Werte mit beliebigem Datentyp ausgeben, z.B.:

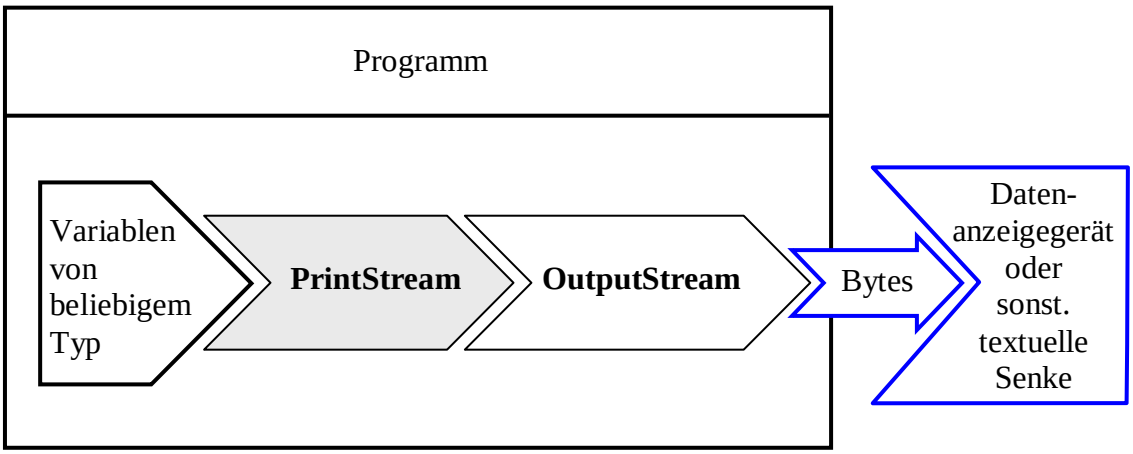
| Quellcode | Ausgabe |
|---|---|
| `class PrintStreamConsole {`<br>`  public static void main(String[] args) {`<br>`  PrintStreamConsole wob = new PrintStreamConsole();`<br>`  System.out.println("Ein PrintStream kann Variablen\n"`<br>`  +"bel. Typs verarbeiten.\n"`<br>`  +"Z.B. die double-Zahl\n"+"  "+Math.PI`<br>`  +"\noder auch das Objekt\n"+"  "+wob);`<br>`  }`<br>`}` | `Ein PrintStream kann Variablen`<br>`bel. Typs verarbeiten.`<br>`Z.B. die double-Zahl`<br>`  3.141592653589793`<br>`oder auch das Objekt`<br>`  PrintStreamDemo@16f0472` |

Seit Java 5.0 (alias 1.5) ist auch die sehr leistungsfähige **PrintStream**-Methode **printf()** zur formatierten Ausgabe verfügbar, die schon in Abschnitt 3.2.2 dargestellt wurde.

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 geschützten **try**-Block zu setzen.

Generell kann man die **PrintStream**-Arbeitsweise folgendermaßen darstellen, wobei an Stelle der abstrakten Klasse **OutputStream** eine konkrete Unterklasse stehen muss:

[56] Wird z.B. ein Ausnahmeobjekt über die Methode **printStackTrace()** beauftragt, die Aufrufsequenz auszugeben, dann landet diese im Fehlerausgabestrom.



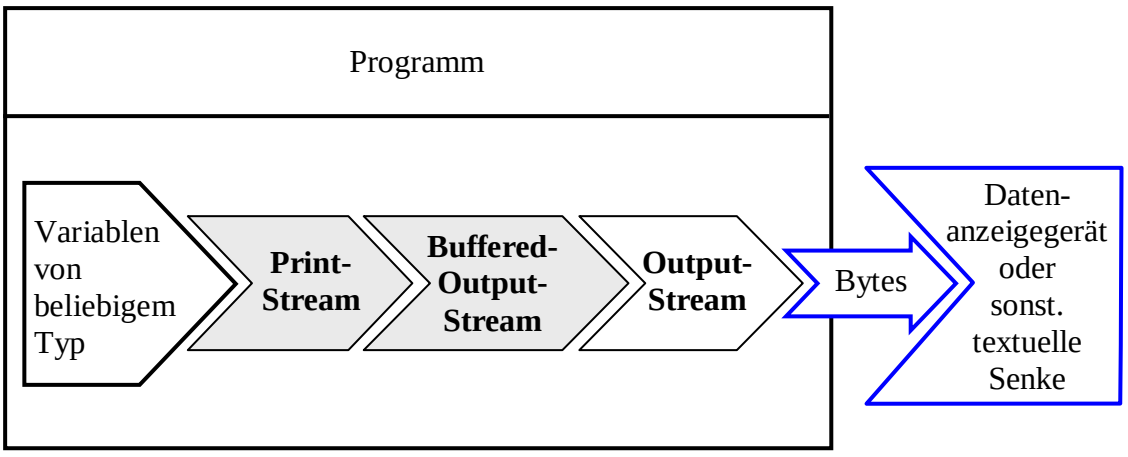

Im nächsten Beispiel ist zu sehen, wie mit Hilfe der Transformationsklasse **PrintStream** Werte primitiver Datentypen in eine *Textdatei* geschrieben werden (über einen **FileOutputStream**):

```
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 eines **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 12.3.1.4).
- **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()**-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 Ausgabe von *Textdaten* übersetzen **PrintStream**-Objekte den Java-internen Unicode unter Verwendung der plattformspezifischen Standardkodierung in Bytes. Unter Windows arbeitet die JRE mit der ANSI-Kodierung. Allerdings verwendet Windows diesen Zeichensatz nur bei grafikorientierten Anwendungen, während im Konsolenfenster eine 8-Bit-ASCII-Erweiterung zum Einsatz kommt. Welche Konsequenzen sich daraus ergeben, zeigt die unglückliche Ausgabe von Umlauten im Konsolenfenster unter Windows, z.B.:[57]

| Quellcode | Ausgabe |
|---|---|
| `class Prog {`<br>`  public static void main(String[] args) {`<br>`  System.out.println("Ganz schön übel!");`<br>`  }`<br>`}` | `Ganz sch÷n ³bel!` |

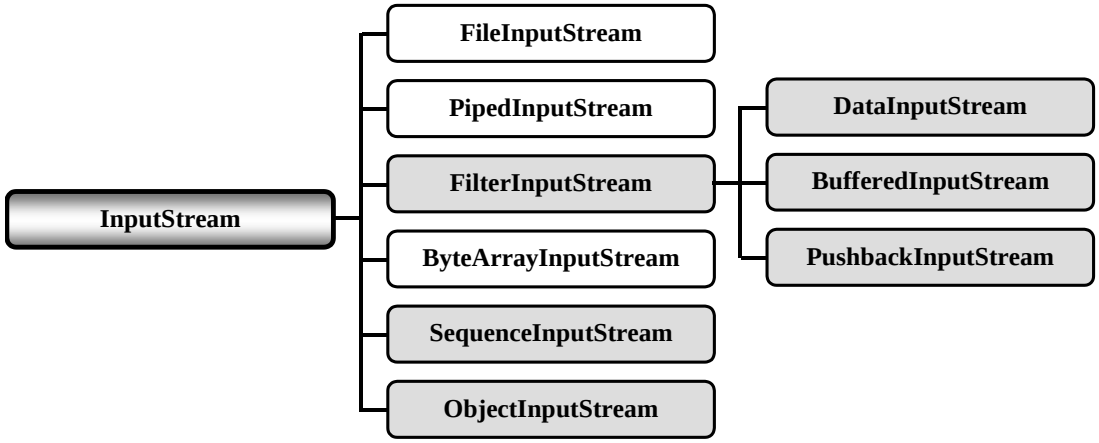
Bei der Textausgabe ist in der Regel die modernere und flexiblere Klasse **PrintWriter** zu bevorzugen (siehe Abschnitt 12.4.1). Mit ihrer Hilfe werden wir in Abschnitt 12.4.1.6 das Umlautproblem in der Windows-Konsole lösen. Vermutlich werden also der per **System.out** ansprechbare Standardausgabestrom und der per **System.err** ansprechbare Standardfehlerausgabestrom die einzigen **PrintStream**-Objekte in Ihren Java-Programmen bleiben.

### 12.3.2 Die InputStream-Hierarchie

#### *12.3.2.1 Überblick*

Um Ihnen das Blättern zu ersparen, wird die schon in Abschnitt 12.1.3 gezeigte Abbildung zur **InputStream**-Hierarchie wiederholt (Eingabeklassen mit weißem Hintergrund und Eingabe*transformations*klassen mit grauem Hintergrund):

[57] Auf der Eclipse-Konsole erscheint die erwünschte Ausgabe, beim normalen Start via **java**-Werkzeug jedoch nicht.



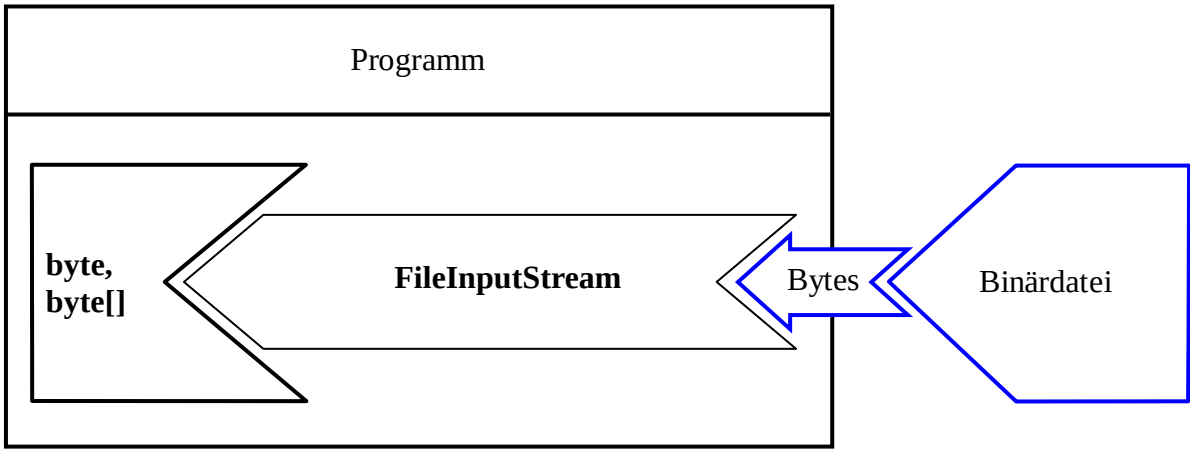

Mit der Transformationsklasse **ObjectInputStream** werden wir uns in Abschnitt 12.6 näher beschäftigen. Sie ermöglicht das Einlesen kompletter Objekte aus einem Bytestrom.

Die folgenden Klassen können nur kurz vorgestellt werden:

- **PipedInputStream**
  Objekte dieser Eingabeklasse lesen Bytes aus einer Pipe, die zur Kommunikation zwischen Threads dient.
- **ByteArrayInputStream**
  Objekte dieser Eingabeklasse lesen Bytes aus einem **byte**-Array.
- **BufferedInputStream**
  Analog zum **BufferedOutputStream** (siehe Abschnitt 12.3.1) realisiert diese Eingabetransformationsklasse einen Zwischenspeicher, um das Lesen aus einem Eingabestrom zu beschleunigen.
- **PushbackInputStream**
  Diese Transformationsklasse bietet Methoden, um aus einem Eingabestrom entnommene Bytes wieder zurück zu stellen.
- **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.

#### *12.3.2.2 FileInputStream*

Wie man mit einem **FileInputStream** Bytes aus einer Datei liest, wurde schon im Abschnitt 12.3.1.2 (über den **FileOutputStream)** gezeigt.



Im **FileInputStream**-Konstruktor kann die anzusprechende Datei über ein **File**-Objekt (siehe Abschnitt 12.2) 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**.

Eine Auswahl der **FileInputStream**-Methoden:

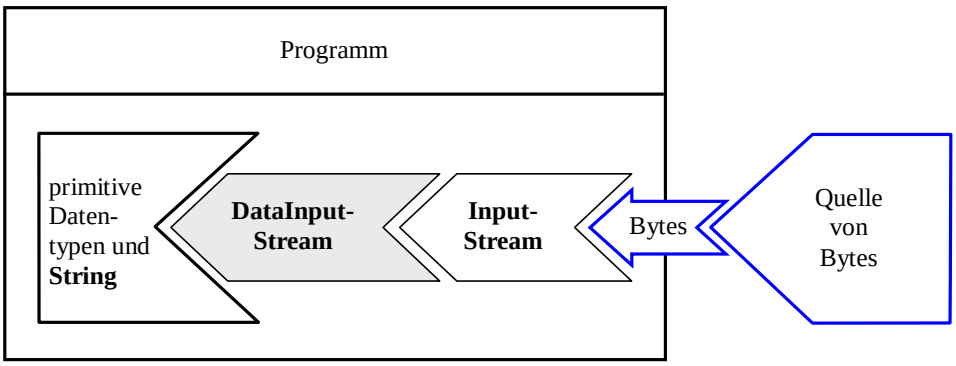
- **int read()**
  Als Rückgabewert (vom Typ **int**!) erhält man das nächste Byte (mögliche Werte von 0 bis 255) oder den Wert -1, falls das Dateiende erreicht ist.
- **int read(byte[] b)**
  Diese Methode überträgt maximal **b.length** Bytes aus der Eingabedatei in den per Parameter angegebenen Array. 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. Nach meinen Beobachtungen mit Java 1.6 (Update 17) unter Windows liefert **available()** stets die Dateigröße zurück, was bei einer größeren Datei recht lange dauern kann (z.B. über eine Minute bei 100 MB).

Im folgenden Codefragment kommen die drei beschriebenen Methoden zum Einsatz:

```
FileInputStream fis = null;
  . . .
int anfang;
byte[] rest = new byte[10];
fis = new FileInputStream(name);
anfang = fis.read();
System.out.println("Verfuegbar: "+fis.available());
System.out.println("Gelesen:    "+fis.read(rest));
```

#### *12.3.2.3 DataInputStream*

Die Transformationsklasse **DataInputStream** liest primitive Datentypen sowie **String**-Objekte aus einem Bytestrom und ist uns zusammen mit ihrem Gegenstück **DataOutputStream** schon in Abschnitt 12.3.1.3 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, wo-

bei eine *modifizierte* Variante der UTF-8 - Kodierung (vgl. Abschnitt 12.4.1.2) vorausgesetzt wird. Diese Methode ist angemessen, sofern die Zeichen mit der **DataOutputStream**-Methode **writeUTF()** geschrieben worden sind (vgl. Abschnitt 12.3.1.3). Für Textdateien mit einer anderen Kodierung ist die Klasse **InputStreamReader** mit einstellbarer und normkonformer Kodierung weit besser geeignet.

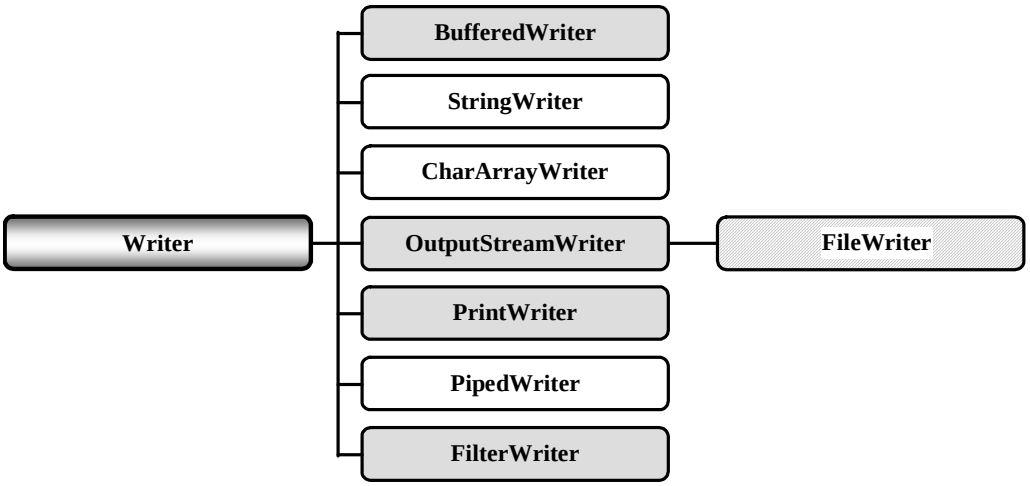
## *12.4 Klassen zur Verarbeitung von Zeichenströmen*

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 (z.B. wegen der unproblematischen Internationalisierung) zu bevorzugen.

### 12.4.1 Die Writer-Hierarchie

#### *12.4.1.1 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** eine später noch zu besprechende Besonderheit aufweist, ist sie mit schraffiertem Hintergrund gezeichnet.

Bei den folgenden **Writer**-Subklassen beschränken wir uns auf kurze Hinweise:

- **StringWriter** und **CharArrayWriter**
  **StringWriter** schreiben in einen dynamisch wachsenden **StringBuffer**. Im folgenden Beispiel werden die auszugebenden Zeichen von einem **PrintWriter** geliefert:

| Quellcode | Ausgabe |
|---|---|
| `import java.io.*;`<br>`class StringWriterDemo {`<br>`  public static void main(String[] args) {`<br>`  StringWriter sw = new StringWriter();`<br>`  PrintWriter pw = new PrintWriter(sw);`<br>`  for (int i = 1; i <= 5; i++)`<br>`  pw.println("Zeile " + i);`<br>`  System.out.println(sw.toString());`<br>`  }`<br>`}` | `Zeile 1`<br>`Zeile 2`<br>`Zeile 3`<br>`Zeile 4`<br>`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.

#### *12.4.1.2 Brückenklasse OutputStreamWriter*

Die Klasse **OutputStreamWriter** überträgt die von Java intern verwendeten Unicode-Zeichen in einen Byte-Strom, wobei unterschiedliche Kodierungsschemas (engl. *encodings*) zum Einsatz kommen können. Weil ein **OutputStreamWriter** einen Zeichenstrom in einen Bytestrom überführt, spricht die API-Dokumentation hier von einer **Brückenklasse**.

Die folgenden Kodierungsschemas (bezeichnet nach den Vorschriften der IANA (***I**nternet **A**ssigned **N**umbers **A**uthority*)) werden von allen Java-Implementationen unterstützt:

<table>
<tr><th>IANA-Bezeichnung</th><th>Beschreibung</th></tr>
<tr><td>US-ASCII</td><td>7-bit-ASCII-Code<br>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>Erweiterter ASCII-Code (ISO Latin-1)<br>Bei den Unicode-Zeichen \u0000 bis \u00FF wird das niederwertige Byte ausgegeben, ansonsten ein Fragezeichen (0x3F).</td></tr>
<tr><td>UTF-8</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>Für alle Unicode-Zeichen werden 16 Bit in Big-Endian - Reihenfolge ausgegeben: Das höherwertige Byte zuerst.<br>In Java ist diese Reihenfolge generell voreingestellt (auch bei anderen Datentypen).<br>Beim großen griechischen Delta (\u0394) wird ausgegeben: 03 94</td></tr>
<tr><td>UTF-16LE</td><td>Für alle Unicode-Zeichen werden 16 Bit in Little-Endian - Reihenfolge ausgegeben: Das niederwertige Byte zuerst.<br>Bei großen griechischen Delta (\u0394) wird ausgegeben: 94 03</td></tr>
</table>

Unter Windows verwendet Java folgendes Kodierungsschema:

| IANA-Bezeichnung | Beschreibung |
|---|---|
| **Cp1252** | Windows Latin-1 (ANSI)<br>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. |

Auf der Webseite

http://java.sun.com/j2se/1.5.0/docs/guide/intl/encoding.doc.html

werden weitere ab Java 5.0 (alias 1.5) unterstützte Kodierungsschemas aufgelistet.

Auf der Webseite

http://www.ingrid.org/java/i18n/encoding/mapping.html

findet sich die Unicode-Abbildung für zahlreiche Kodierungsschemas.

Bei folgenden Überladungen des **OutputStreamWriter**-Konstruktor kann das gewünschte Kodierungsschema über seinen IANA-Namen oder über ein **Charset**-Objekt angegeben werden:

- **public OutputStreamWriter(OutputStream** *out***, String** *charsetName*)
- **public OutputStreamWriter(OutputStream** *out***, Charset** ***cs***)

Statt die in drei Überladungen verfügbare **OutputStreamWriter**-Methode **write()** direkt zu verwenden, spricht man sie in der Regel über einen vorgeschalteten **PrintWriter** an (siehe Abschnitt 12.4.1.4). Diese Klasse verfügt analog zur älteren Klasse **PrintStream** (siehe Abschnitt 12.3.1.5) über Ausgabemethoden für praktisch alle Datentypen und bietet u.a. folgende Vorteile:

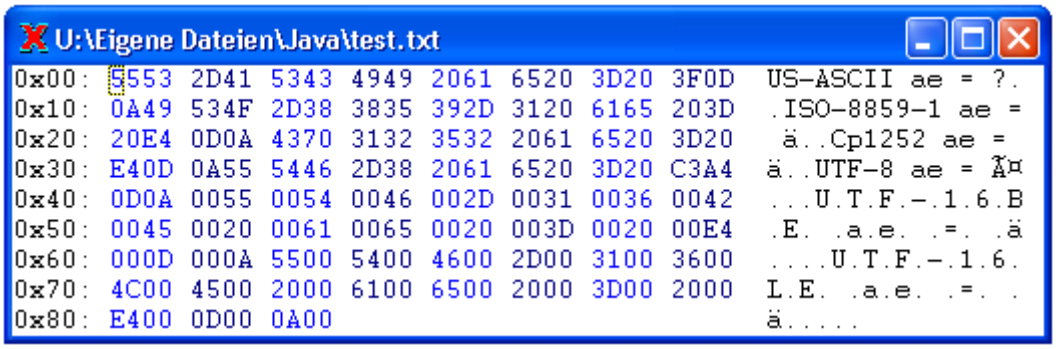
- Freie Wahl des Kodierungsschemas durch die Kooperation mit einem **OutputStreamWriter**
- Besseres **autoFlush**-Verhalten (siehe unten)
- Die **println()**-Methoden produzieren eine plattformspezifische Zeilentrennung, statt (wie bei der Klasse **StreamWriter**) ein NewLine-Zeichen (\n) zu schreiben.

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.

```
import java.io.*;

class OutputStreamWriterDemo {
  public static void main(String[] args) throws IOException {
    String[] encodings = {"US-ASCII", "ISO-8859-1", "Cp1252", "UTF-8",
                          "UTF-16BE", "UTF-16LE"};
    FileOutputStream fos = null;
    OutputStreamWriter osw;
    PrintWriter pw = null;
    try {
      fos = new FileOutputStream("test.txt");
      for (int i = 0; i < 6; i++) {
          osw = new OutputStreamWriter(fos, encodings[i]);
        pw = new PrintWriter(osw, true); // 2. Parameter: autoFlush
        pw.println(encodings[i] + " ae = ä");
      }
    } finally {
      if (pw != null)
        pw.close();
    }
  }
}
```

In einem Hex-Editor[58] sieht die Ergebnisdatei folgendermaßen aus:

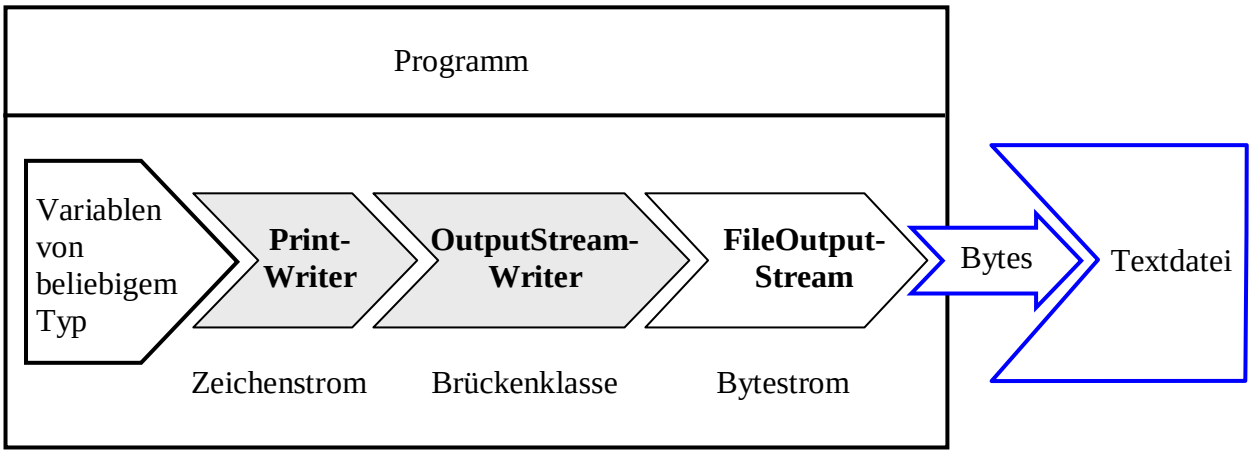
[58] Der hier verwendete **HexWizard** wird auf folgender Webseite kostenlos angeboten:
http://www.handycheats.de/seiten/hexwizard.html




Für das Unicode-Zeichen \u00E4 wird jeweils ausgegeben:

| Kodierungsschema | Byte(s) in der Ausgabe |
|---|---|
| US-ASCII | 3F |
| ISO-8859-1 | E4 |
| Cp1252 | E4 |
| UTF-8 | C3 A4 |
| UTF-16BE | 00 E4 |
| UTF-16LE | E4 00 |

Das Beispielprogramm arbeitet mit folgender Datenstrom-Architektur:




**OutputStreamWriter** (und damit auch **FileWriter**, siehe Abschnitt 12.4.1.5) 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.
- **Autoflush** eines vorgeschalteten **PrintWriter**s (siehe Abschnitt 12.4.1.4) per Konstruktor-Parameter aktivieren
  Dann wird der Puffer bei jedem Aufruf der **PrintWriter**-Methoden **println()**, **printf()** oder **format()** entleert.

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()**-Aufruf und keine Pufferentleerung statt.

#### *12.4.1.3 BufferedWriter*

Am Ende von Abschnitt 12.4.1.2 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 **PrintWriter**s und eines **FileWriter**s (siehe Abschnitt 12.4.1.5) zweimal jeweils 10.000 Zeilen in eine Textdatei geschrieben, zunächst ohne und dann mit zwischengeschaltetem **BufferedWriter**:

```
import java.io.*;

class BufferedWriterDemo {
  public static void main(String[] args) throws IOException {
    final long ANZAHL = 10000;
    PrintWriter pw = null;
    try {
      pw = new PrintWriter(new FileWriter("test.txt"));
      long time = System.currentTimeMillis();
      for (int i = 1; i <= ANZAHL; i++)
        pw.println("Zeile " + i);
      System.out.println("Benoetigte Zeit ohne BufferedWriter: "
          +(System.currentTimeMillis()-time));
    } finally {
      if (pw != null)
        pw.close();
    }

    try {
    pw = new PrintWriter(new BufferedWriter(
      new FileWriter("test.txt"),16384));
    long time = System.currentTimeMillis();
    for (int i = 1; i <= ANZAHL; i++)
      pw.println("Zeile " + i);
    System.out.println("Benoetigte Zeit mit BufferedWriter: "
        +(System.currentTimeMillis()-time));
    } finally {
      if (pw != null)
        pw.close();
    }
  }
}
```

Trotz der eingangs erwähnten impliziten Byte-Pufferung durch den **FileWriter** fällt der Gewinn durch den **BufferedWriter**-Einsatz recht deutlich aus:

```
Benoetigte Zeit ohne BufferedWriter: 63
Benoetigte Zeit mit BufferedWriter: 15
```

#### *12.4.1.4 PrintWriter*

Die schon mehrfach im Vorgriff benutzte Transformationsklasse **PrintWriter** wird nun endlich offiziell vorgestellt. Sie besitzt diverse **print()**- bzw. **println()**-Überladungen, um Variablen belie-

bigen Typs in Textform auszugeben (z.B. in eine Datei oder auf die Konsole). 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 12.3.1.5). Seit der Java-Version 5.0 (alias 1.5) beherrschen **PrintWriter**-Objekte auch die weitgehend funktionsgleichen Methoden **printf()** und **format()** zur formatierten Ausgabe.

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:

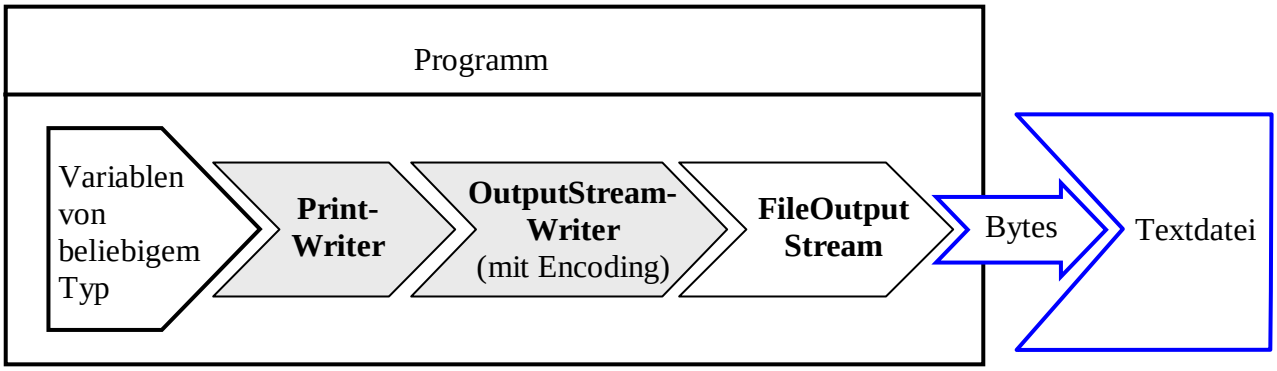
- **public PrintWriter(OutputStream** *out***)**
- **public PrintWriter(OutputStream** *out***, boolean** *autoFlush***)**
- **public PrintWriter(Writer** *out***)**
- **public PrintWriter(Writer** *out***, boolean** *autoFlush***)**

Wird ein **PrintWriter** auf einen Ausgabestrom aus der **OutputStream**-Hierarchie gesetzt, dann kommt automatisch ein übersetzender **OutputStreamWriter** mit dem voreingestellten Kodierungsschema (und ein **BufferedWriter**) zum Einsatz (siehe Abschnitt 12.4.1.2).

Letztlich übergibt ein **PrintWriter** alle Ausgabedaten als 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 12.4.1.2). 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.[59] Dies ist bei einer *interaktiven* Anwendung unbedingt erforderlich, sollte aber bei einer Dateiausgabe aus Performanzgründen vermieden werden.

Gibt man im **PrintWriter**-Konstruktor 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 12.4.1.2). 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:

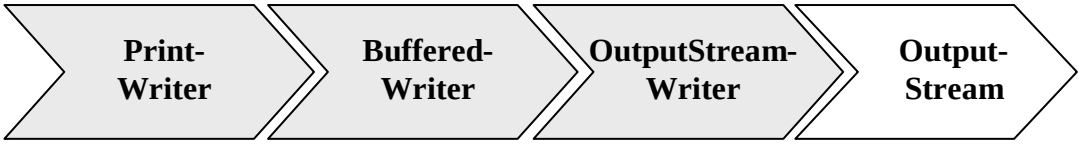



[59] Bei der Klasse **PrintStream** (siehe Abschnitt 12.3.1.5) bewirkt der **autoFlush**-Wert **true** eine automatische Pufferentleerung nach dem Schreiben eines **byte**-Arrays oder Newline-Zeichens (\n) sowie nach der Ausführen einer **println()**-Überladung.

Wird beim Erstellen eines **PrintWriter**s dem Konstruktor ein **OutputStream**-Abkömmling als Aktualparameter übergeben, dann nimmt der Konstruktor insgeheim einen **BufferedWriter**, der Zeichen zwischenspeichert(siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**), und einen **OutputStreamWriter**, der Bytes zwischenspeichert (siehe Abschnitt 12.4.1.2), in Betrieb:[60]

```
public PrintWriter(OutputStream out, boolean autoFlush) {
  this(new BufferedWriter(new OutputStreamWriter(out)), autoFlush);
  . .
}
```

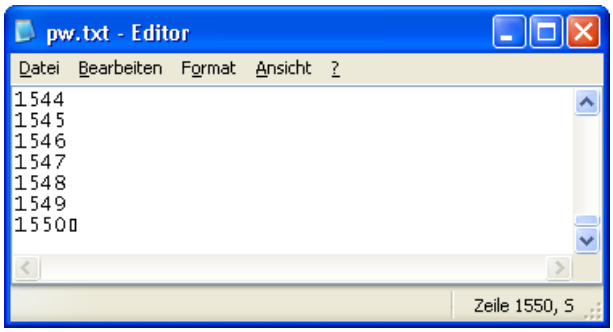
Damit arbeitet insgesamt folgendes Gespann:




Vor dem Terminieren eines **PrintWriter**-Objekts müssen die Zwischenspeicher unbedingt entleert werden, z.B. per **close()**-Aufruf:

```
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:



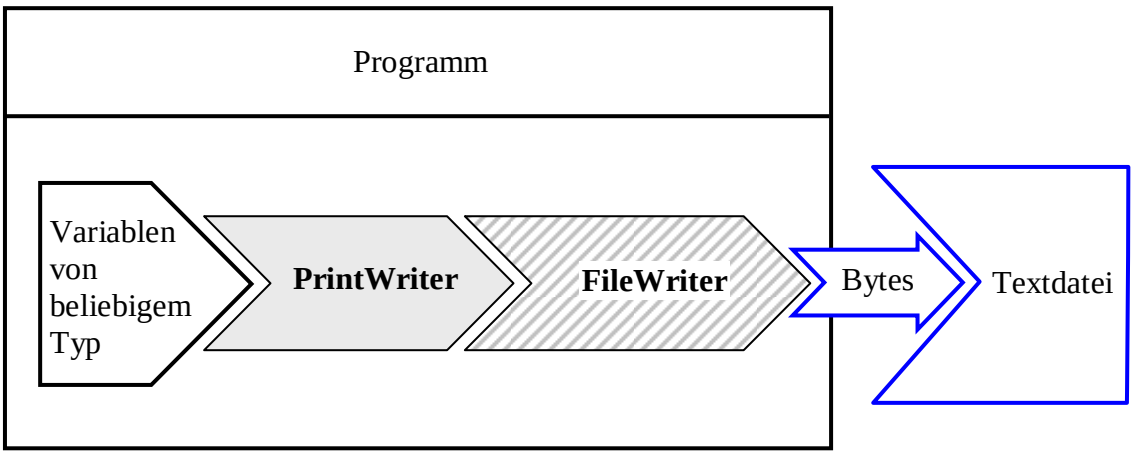

Abweichend vom Aufbau der **OutputStream**-Hierarchie ist **PrintWriter** übrigens *nicht* von **FilterWriter** abgeleitet.

---

[60] 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.

#### *12.4.1.5 FileWriter*

Die von **OutputStreamWriter** abgeleitete Klasse **FileWriter** wird oft an einen **PrintWriter** zur bequemen Ausgabe in eine Textdatei angehängt, wenn das voreingestellte Kodierungsschema (unter Windows: *Cp1252* alias *Windows Latin 1*) akzeptabel ist:




Abgesehen von der impliziten Pufferung im Zusammenhang mit der Transkodierung (vgl. Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**) sollte sich dieses Gespann genauso verhalten wie die Kombination aus einem **PrintStream** und einem **FileOutputStream**, die ebenfalls mit dem voreingestellten Kodierungsschema arbeitet.

In obiger Darstellung der **Writer**-Hierarchie (siehe Abschnitt 12.4.1.1) 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 bequemen 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***)**
- **public FileWriter(File** *file***, boolean** *append***)**
- **public FileWriter(String** *fileName***)**
- **public FileWriter(String** *fileName***, boolean** *append***)**

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 12.4.1.2).

Bei der folgenden Variante des Beispielprogramms aus Abschnitt 12.4.1.4 kommt das am Anfang des aktuellen Abschnitts dargestellte **PrintWriter**-**FileWriter** - Gespann zum Einsatz:

```
import java.io.*;

class FileWriterDemo {
  public static void main(String[] args) throws IOException {
    FileWriter fw = null;
    PrintWriter pw = null;
    try {
      fw = new FileWriter("fw.txt");
      pw = new PrintWriter(fw);
      for (int i = 1; i <= 3000; i++) {
        pw.println(i);
      }
    } finally {
      if (pw != null) pw.close();
    }
  }
}
```

#### *12.4.1.6 Umlaute in Java-Konsolenanwendungen unter Windows*

Mit Hilfe der Brückenklasse **OutputStreamWriter** lässt sich endlich ein kleines Problem lösen, das uns während des ganzen Kurses begleitet hat:

- Windows arbeitet in Konsolenfenstern mit einer 8-Bit-Variante des ASCII-Zeichensatzes („IBM-ASCII“), in grafikorientierten Anwendungen hingegen mit dem ANSI-Zeichensatz.
- Die JRE arbeitet unter Windows grundsätzlich mit dem ANSI-Zeichensatz (genauer: mit dem Kodierungsschema *Cp1252* alias *Windows Latin 1*, siehe Abschnitt 12.4.1.2). Infolgedessen werden Umlaute in Java-Konsolenanwendungen unter Windows falsch dargestellt.

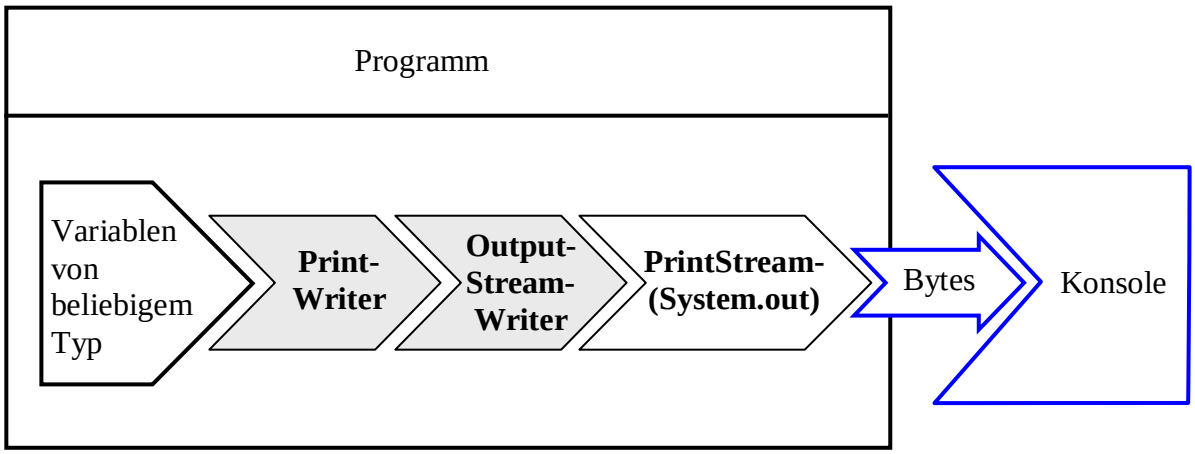
Ein **OutputStreamWriter** mit dem Kodierungsschema *Cp850* (IBM-ASCII) ermöglicht die korrekte Darstellung der Umlaute:[61]

| Quellcode | Ausgabe |
|---|---|
| `import java.io.*;`<br>`class Cp850 {`<br>`  public static void main(String[] ars) throws IOException {`<br>`  OutputStreamWriter osw = null;`<br>`  if (System.getProperty("file.encoding").equals("Cp1252"))`<br>`  osw = new OutputStreamWriter(System.out, "Cp850");`<br>`  else`<br>`  osw = new OutputStreamWriter(System.out);`<br><br>`  PrintWriter pw = new PrintWriter(osw, true);`<br>`  pw.println("Der Ärger war nicht nötig.");`<br>`  }`<br>`}` | `Der Ärger war nicht nötig.` |

Er erhält über die statische Variable **System.out** als Ausgabestrom ein **PrintStream**-Objekt, das mit der Konsole verbunden ist. Um die Plattformunabhängigkeit des Programms nicht einzuschränken, wird das spezielle Kodierungsschema nur dann verwendet, wenn der **System**-Methodenaufruf **getProperty("file.encoding")** *Cp1252* als Standardkodierungsschema zurückmeldet.

Es liegt in jedem Fall die folgende Datenstrom-Architektur vor:

[61] Weil die Entwicklungsumgebung Eclipse 3.x bei der Konsolenausgabe eine automatische Kodierungsanpassung vornimmt, führt nun eine doppelte Korrektur zur fehlerhaften Ausgabe von Umlauten.




Die Darstellung ist leicht vereinfacht, denn zwischen dem **PrintStream**-Filterobjekt und der Konsole agieren noch (vgl. Abschnitt 12.3.1.5):

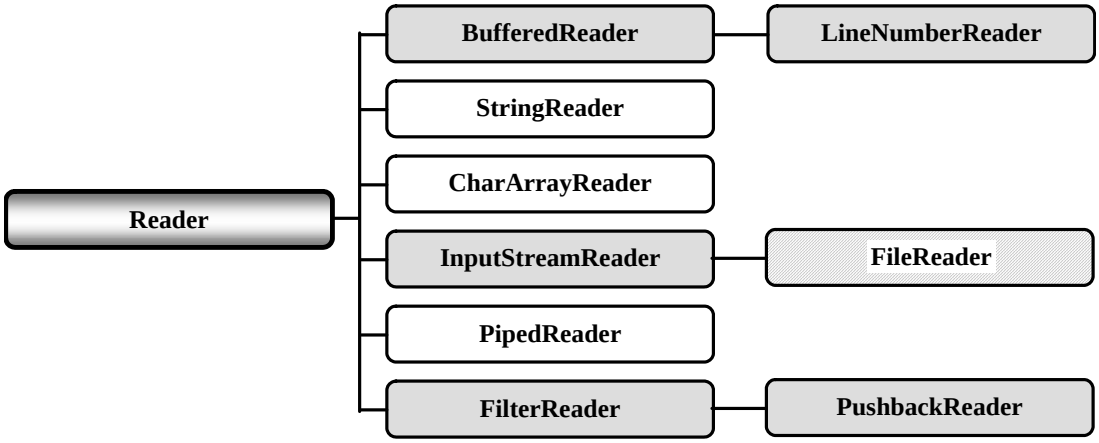
- ein **BufferedOutputStream**
- ein **FileOutputStream**

### 12.4.2 Die Reader-Hierarchie

In der **Reader**–Hierarchie finden sich diverse Klassen zur Verarbeitung von zeichenorientierten *Eingabe*strömen.

#### *12.4.2.1 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 Filterfunktion besitzt, ist sie mit schraffiertem Hintergrund gezeichnet.




Bei den meisten **Reader**-Unterklassen beschränken wir uns auf kurze Hinweise:

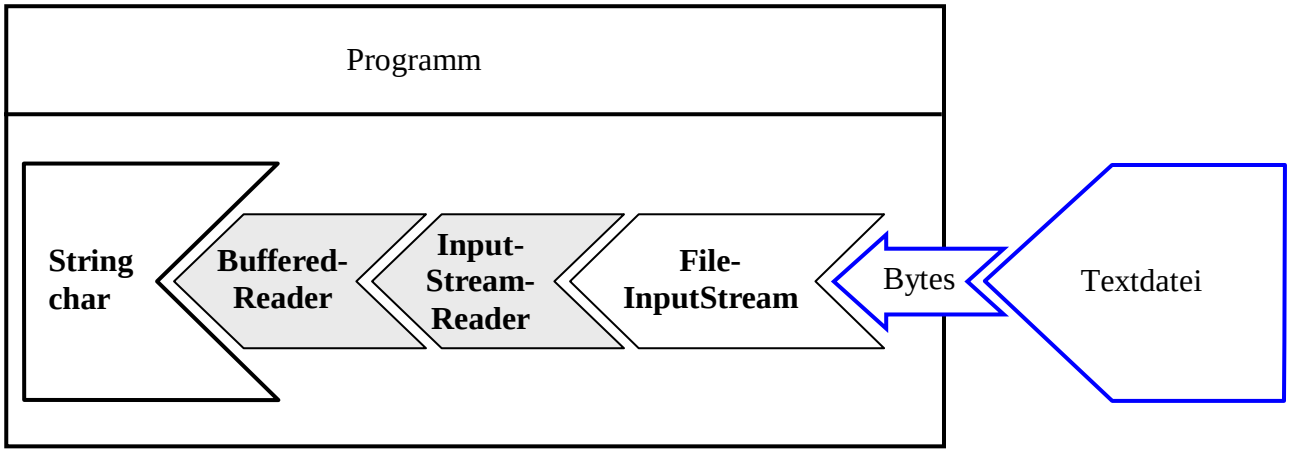
- **LineNumberReader**
  Dieser gepufferte Zeicheneingabestrom erweitert seine Basisklasse um Methoden zur Verwaltung von Zeilennummern.
- **StringReader** und **CharArrayReader**
  **StringReader** bzw. **CharArrayReader** lesen aus einem **String** bzw. **char**-Array.
- **PipedReader**
  Objekte dieser 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.

Wer eine Möglichkeit zum komfortablen Einlesen von *numerischen* Daten aus Textdateien sucht, sollte sich in Abschnitt 12.5 die (unmittelbar aus **java.lang.Object** abgeleitete) Klasse **Scanner** ansehen.

#### *12.4.2.2 Brückenklasse InputStreamReader*

Die Brückenklasse **InputStreamReader** ist das Gegenstück zur Klasse **OutputStreamReader**, wandelt also Bytes unter Verwendung eines einstellbaren Kodierungsschemas (vgl. Abschnitt 12.4.1.2) in Unicode-Zeichen:




Wie beim **OutputStreamReader** findet zur Beschleunigung der Konvertierung automatisch eine Pufferung des Byte-Stroms statt.

#### *12.4.2.3 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 der Plattform zum Einsatz (bei Windows: *Cp1252* alias *Windows Latin 1*). Wer ein alternatives Kodierungsschema benötigt, muss auf die **FileReader**-Bequemlichkeit verzichten, einen **InputStreamReader** mit passendem Kodierungsschema erstellen und mit einem **FileInputStream** kombinieren.

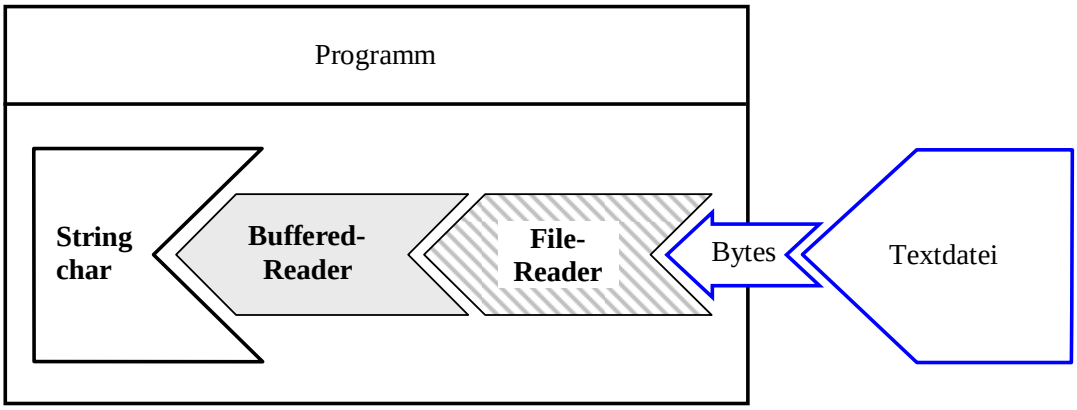
In der Regel setzt man vor den **FileReader** noch einen **BufferedReader**, der für eine Beschleunigung sorgt und außerdem die bei Dateien mit Zeilenstruktur sehr nützliche Methode **readLine()** zum Einlesen einer kompletten Zeile bietet, z.B.:

| Quellcode | Ausgabe |
|---|---|

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

class FileReaderDemo {
   public static void main(String[] args) throws IOException {
      ArrayList<String> als = new ArrayList<String>();
      BufferedReader br = null;
      try {
         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));
      } finally {
         if (br != null)
            br.close();
      }
   }
}
```

```
Zeile 100
```

Das Beispielprogramm arbeitet mit folgender Datenstrom-Architektur:




Die bei einem **BufferedReader** voreingestellte Puffergröße von 8192 Zeichen lässt sich per Konstruktor ändern.

## *12.5 Primitive Typen und Zeichenketten aus Textdateien lesen*

Seit Java 5.0 (alias 1.5) erleichtert die unmittelbar von **java.lang.Object** abstammende Klasse **Scanner** (Paket **java.util**) das Einlesen von primitiven Datentypen und Zeichenketten aus Textdateien, wobei speziell das Einlesen *numerischer* Daten im Vergleich zu den früheren Lösungen erleichtert wird.

Ein Scanner zerlegt den Eingabestrom aufgrund eines frei wählbaren Trennzeichenmusters in Bestandteile, *Tokens* genannt, auf die man mit diversen Methoden sequentiell zugreifen kann, z.B.:

- **public int nextInt()**
  Versucht, das nächste Token **int**-Wert zu interpretieren, und wirft bei Misserfolg eine **InputMismatchException**.

- **public double nextDouble()**
  Versucht, das nächste Token als **double**-Wert zu interpretieren, und wirft bei Misserfolg eine **InputMismatchException**.
- **public String next()**
  Holt das nächste Token vom Scanner.

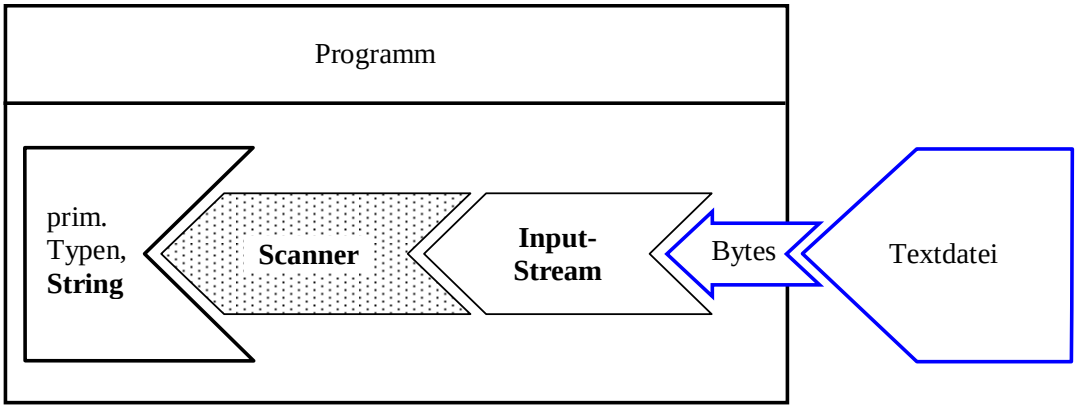
Ob noch ein Token vorhanden und vom gewünschten Typ ist, kann mit einer entsprechenden Methode festgestellt werden, z.B.:

- **public boolena 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 *WhiteSpace*-Zeichen steht die **Scanner**-Methode **useDelimiter()** zur Verfügung

Im **Scanner**-Konstruktor wird u.a. ein beliebiges Objekt aus der **InputStream**-Hierarchie als Datenquelle akzeptiert (z.B. ein **FileInputStream**), wobei optional sogar ein Kodierungsschema angegeben werden kann.

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 zunächst Zahlen und **String**-Objekte aus einer Textdatei:

```
import java.util.*;

class ScannerDatei {
  public static void main(String[] args) {
    Scanner input = null;
    try {
      input = new Scanner(new java.io.FileInputStream("daten.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(Exception e) {
         System.out.println(e);
      } finally {
         if (input != null)
            input.close();
      }
   }
}
```

Mit der Eingabedaten

```
4711    3,1415926
Nicht übel!
13      9,99
```

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 geeignet wechseln, z.B.

```
   . .
input = new Scanner(new java.io.FileInputStream("daten.txt"));
input.useLocale(Locale.US);
   . .
```

Wir haben den Einsatz der Klasse **Scanner** für die bequeme 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 ScannerKonsole {
   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.out.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 public boolean status;
   static public String  errdes = "";
  .   .   .
  .   .   .
   static public int gint() {
      int eingabe = 0;
      Scanner input = new Scanner(System.in);
      try {
         eingabe = input.nextInt();
         status = true;
         errdes = "";
      } catch(Exception e) {
         status = false;
         errdes = "Eingabe konnte nicht konvertiert werden.";
         System.out.println("Falsche Eingabe!\n");
      }
      return eingabe;
   }
  .   .   .
  .   .   .
}
```

## 12.6 Objektserialisierung

Nach vielen Mühen und lästigen Details kommt nun als Belohnung für die Ausdauer eine angenehm einfache Lösung für eine wichtige Aufgabe. Wer objektorientiert programmiert, möchte natürlich auch objektorientiert speichern und laden. Erfreulicherweise können Objekte tatsächlich meist genau so wie primitive Datentypen in einen Byte-Strom geschrieben bzw. von dort gelesen werden. Die Übersetzung eines Objektes (mit all seinen Instanzvariablen) in einen Bytestrom 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 der EA-Behandlung (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.

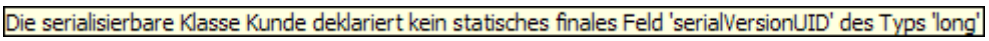
Für das Schreiben von Objekten ist die byteorientierte Ausgabetransformationsklasse **ObjectOutputStream** zuständig, für das Lesen die byteorientierte Eingabetransformationsklasse **ObjectInputStream**. Ein komplexes Objektensemble in einen Bytestrom zu verwandeln bzw. von dort zu rekonstruieren, ist eine komplexe Aufgabe, die aber automatisiert abläuft.

Wir demonstrieren die Serialisation mit Hilfe der folgenden Klasse `Kunde`:

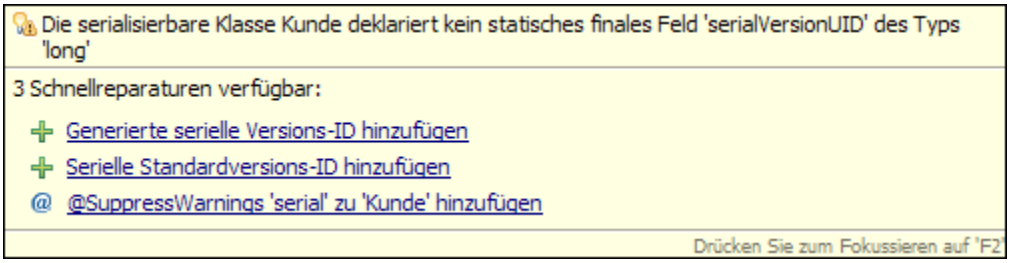
```
import java.io.*;
import java.math.BigDecimal;

public class Kunde implements Serializable {
  private static final long serialVersionUID = -79874726308638208 8L;

  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.Einkaeufe: " + nkaeufe +
                       ", Aussenstaende: " + aussen+ "\n");
  }
}
```

Für die Summe der Außenstände eines Kunden wird zur Vermeidung von Rundungsungenauigkeiten (vgl. Abschnitt 3.5.4) an Stelle einer Gleitkommavariablen ein Objekt der Klasse **BigDecimal** verwendet.

Unsere Entwicklungsumgebung Eclipse reklamiert die fehlende **serialVersionUID**:




Eine solche Versionsangabe wird beim Serialisieren automatisch mit gespeichert und beim Deserialisieren überprüft, um Fehler durch veraltete, inkompatible Objektversionen zu verhindern. Akzeptiert man den aus dem Popup-Menü zur Warnung die Schnellreparatur **Generierte serielle Versions-ID hinzufügen**,




dann wird der Mangel bequem und professionell behoben:

```
private static final long serialVersionUID = -79874726308638208 8L;
```

Die Eclipse-Warnung ist Ihnen sich nicht neu. Wir haben sie immer dann gesehen (und ignoriert), wenn eine selbst definierte Klasse die Schnittstelle **Serializable** von ihrer Basisklasse übernommen hatte.

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();
    ObjectOutputStream oos = null;
    try {
      oos = new ObjectOutputStream(new FileOutputStream("test.bin"));
      oos.writeInt(1);
      oos.writeObject(kunde);
    } finally {
      if (oos != null) oos.close();
    }

    ObjectInputStream ois = null;
    try {
      ois = new ObjectInputStream(new FileInputStream("test.bin"));
      int anzahl = ois.readInt();
      Kunde unbekannt = (Kunde) ois.readObject();
      System.out.printf("Fall Nr. %d:\n\n", anzahl);
      unbekannt.prot();
    } finally {
      if (ois != null) ois.close();
    }
  }
}
```

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 ...

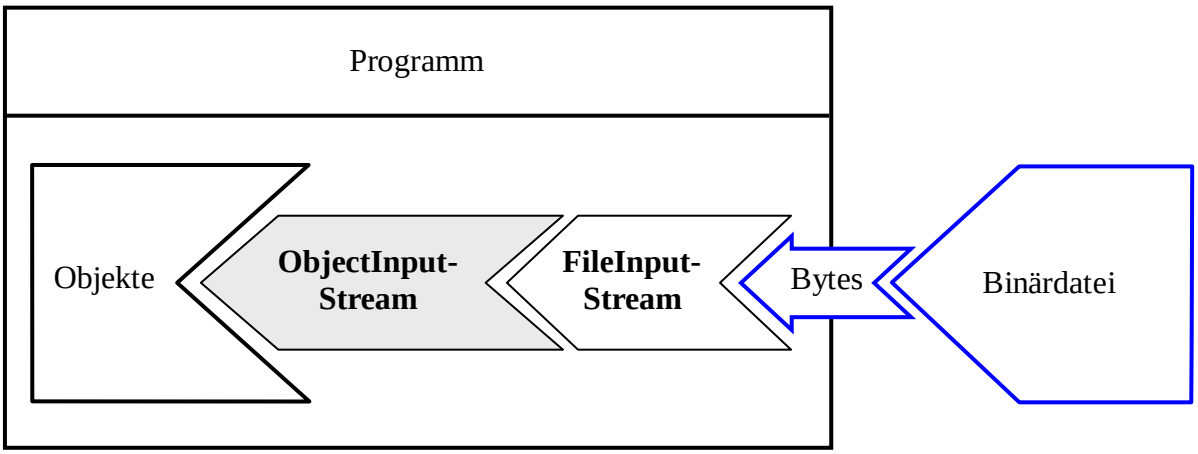
- Variablen, die aus Sicherheitsgründen nicht in eine Datei gespeichert werden sollen,
- Variablen, deren temporärer Charakter ein Speichern nutzlos macht.

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

1 Fall eingelesen:

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:




## *12.7 Empfehlungen zur Verwendung der EA-Klassen*

Weil die Ein-/Ausgabe - Behandlung in Java durch die Vielzahl beteiligter Klassen auf Anfänger etwas unübersichtlich wirkt, folgt in diesem Abschnitt eine rezeptartige Beschreibung wichtiger Spezialfälle.

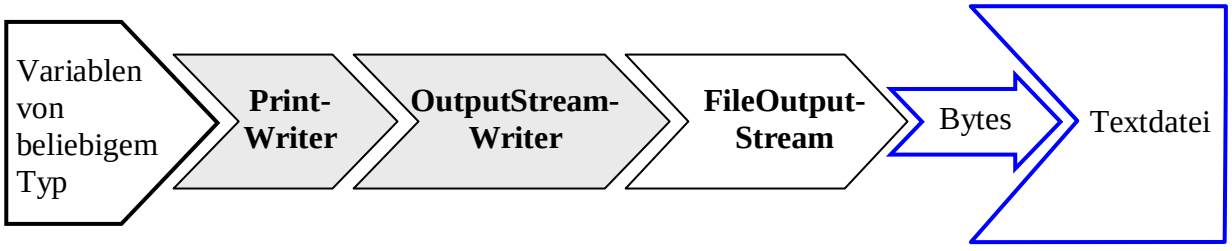
### 12.7.1 Ausgabe in eine Textdatei

Um Textdaten (Datentypen **String**, **char**) oder die Zeichenfolgen-Repräsentationen beliebiger andere Datentypen in eine Datei zu schreiben, verwendet man in der Regel einen **FileWriter** (siehe Abschnitt 12.4.1.5) in Kombination mit einem **PrintWriter** (siehe Abschnitt 12.4.1.4), der bequeme Ausgabemethoden bietet (z.B. **println()**, **printf()**).


Beispiel:

```
PrintWriter pw = new PrintWriter(new FileWriter("test.txt"));
pw.println("Diese Zeile landet in der Datei test.txt.");
```

Wenn bei der Ausgabe von Textdaten das voreingestellte Kodierungsschema ungeeignet ist, ersetzt man den **FileWriter** durch die Kombination aus einem **OutputStreamWriter** mit wählbarer Kodierung (siehe Abschnitt 12.4.1.2) und einem **FileOutputStream** (siehe Abschnitt 12.3.1.2).

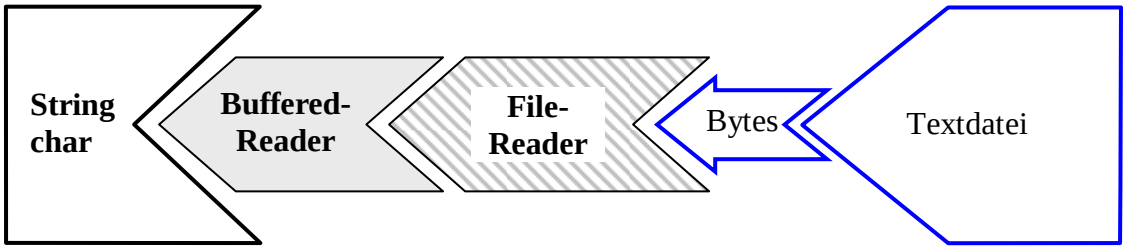



Die **PrintWriter**-Methode **printf()** ermöglicht eine flexible Formatierung der Ausgabe.

### 12.7.2 Textdaten einlesen

Um Textdaten aus einer Datei zu lesen, verwendet man in der Regel ein Objekt aus der Klasse **FileReader** (siehe Abschnitt 12.4.2.3). Meist schaltet man hinter den **FileReader** noch einen **Buffe-**

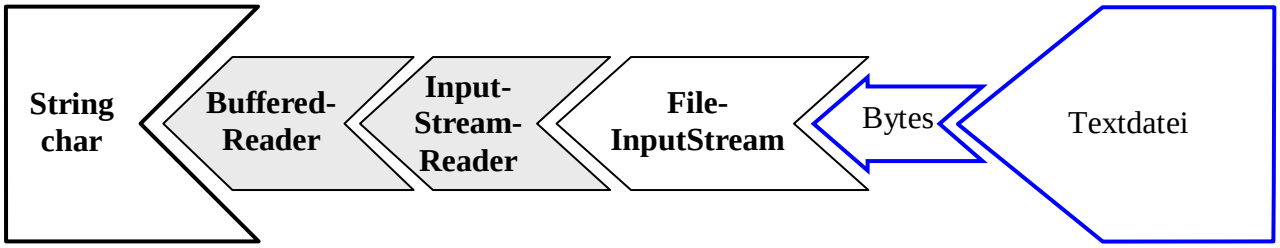
**redReader**, der die Anzahl der Dateizugriffe reduziert und die bei Dateien mit Zeilenstruktur sehr nützliche Methode **readLine()** bietet.




Beispiel:

```
BufferedReader br = new BufferedReader(new FileReader("test.txt"));
String s = br.readLine();
```

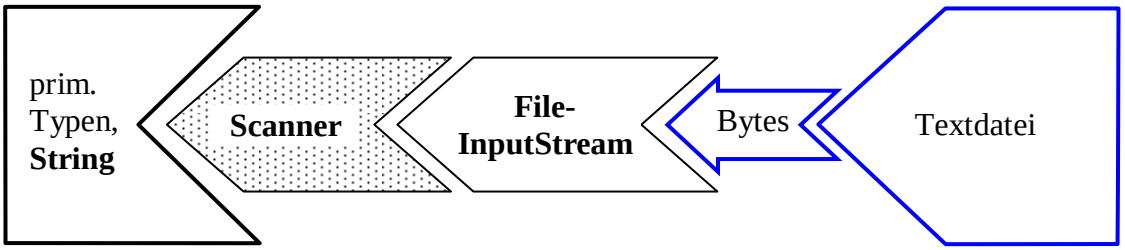
Wer mit dem voreingestellten Kodierungsschema unzufrieden ist, ersetzt den **FileReader** durch die Kombination aus einem **InputStreamReader** mit wählbarer Kodierung und einem **FileInputStream**.




Zum Lesen von Zeichenfolgen kommt auch die relativ neue Klasse **Scanner** in Frage (siehe Abschnitte 12.5 und 12.7.2), die den Eingabestrom aufgrund wählbarer Trenneichen in Bestandteile (Tokens) zerlegen kann und die Wahl eines Kodierungsschemas erlaubt.

### 12.7.3 Primitive Datentypen aus einer Textdatei lesen

Um primitive Datentypen aus einer Textdatei zu lesen, kann man seit Java 5 (alias 1.5) ein Objekt aus der Klasse **Scanner** in Verbindung mit einem **FileInputStream** verwenden (siehe Abschnitt 12.5):




Beispiel:

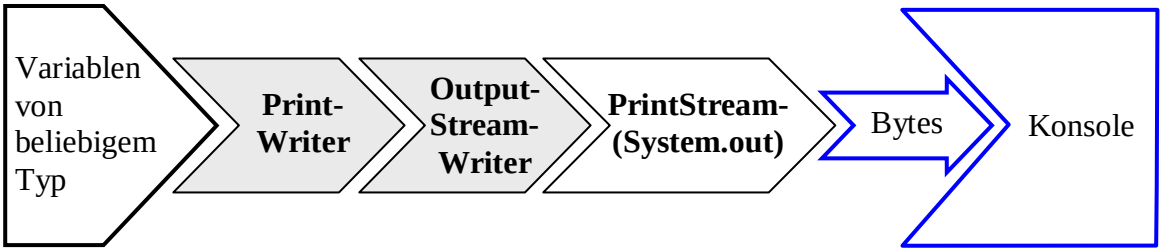
```
Scanner input = new Scanner(new java.io.FileInputStream("daten.txt"));
double a = input.nextDouble();
int b = input.nextInt();
```

### 12.7.4 Ausgabe auf die Konsole

In den meisten Fällen genügt es, die Standardausgabe über die automatisch initialisierte Referenzvariable **System.out** mit **PrintStream**-Methoden anzusprechen.
Beispiel:

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

In Abschnitt 12.4.1.6 wird beschrieben, wie unter Windows eine perfekte Konsolenausgabe mit korrekter Darstellung der Umlaute zu bewerkstelligen ist:[62]



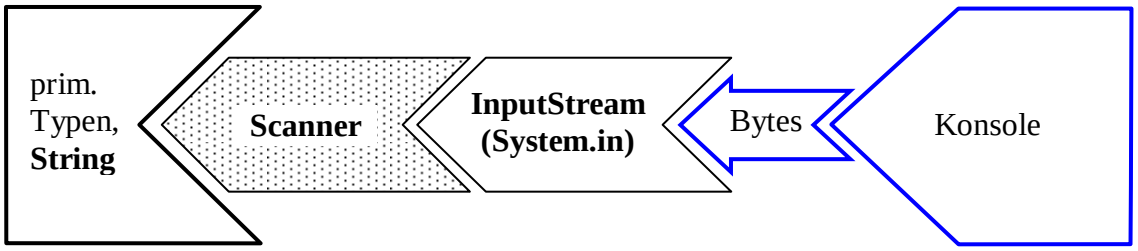

Weil der **OutputStreamWriter** die durch Wandlung von Unicode-Zeichen entstehenden Bytes automatisch puffert (vgl. Abschnitt 12.4.1.2), sollte beim **PrintWriter** die **autoFlush**-Option eingeschaltet werden.

Beispiel:

```
OutputStreamWriter osw = new OutputStreamWriter(System.out, "Cp850");
PrintWriter pw = new PrintWriter(osw, true);
pw.println("Hallo Wörld");
```

### 12.7.5 Eingabe von der Konsole

In Abschnitt 12.5 wird beschrieben, wie man Tastatureingaben mit Hilfe der Klasse **Scanner** entgegen nimmt:

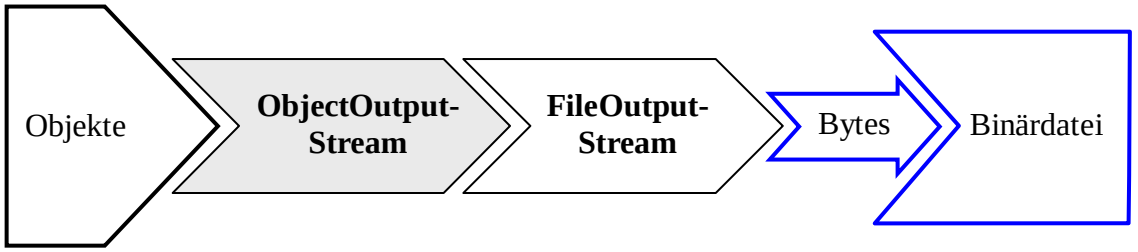



Beispiel:

```
Scanner input = new Scanner(System.in);
System.out.print("Ihr Alter: ");
int alter = input.nextInt();
```

### 12.7.6 Objekte schreiben und lesen

Um Objekte in eine Datei zu schreiben oder aus einer zu Datei lesen, verwendet man die in Abschnitt 12.6 vorgestellten Klassen **ObjectOutputStream** und **ObjectInputStream**:



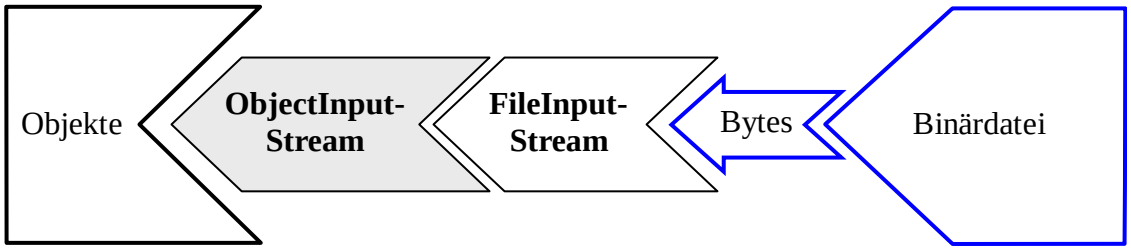

Beispiel:

---

[62] Die Darstellung ist leicht vereinfacht, denn zwischen dem **PrintStream**-Filterobjekt und der Konsole agieren noch:

- ein **BufferedOutputStream**
- ein **FileOutputStream**

```
ObjectOutputStream oos = new ObjectOutputStream(
                           new FileOutputStream("test.bin"));
oos.writeObject(demobj);
```

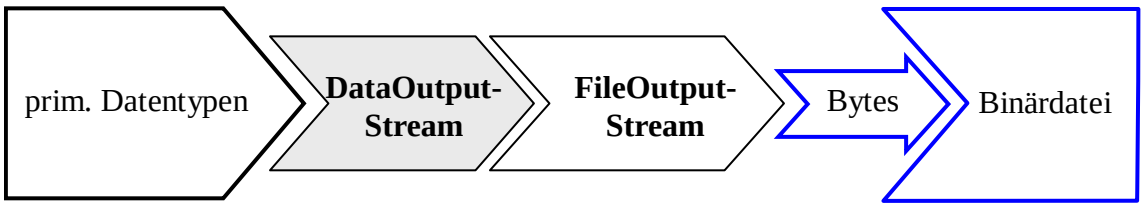



Beispiel:

```
ObjectInputStream ois = new ObjectInputStream(
                          new FileInputStream("test.bin"));
Kunde unbekannt = (Kunde) ois.readObject();
```

### 12.7.7 Primitive Datentypen in eine Binärdatei schreiben

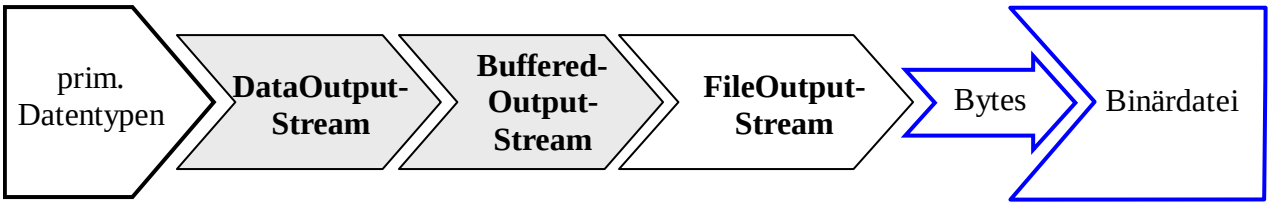
Um primitive Datentypen (z.B. **int**, **double**) binär in eine Datei zu schreiben, verwendet man ein Filterobjekt aus der Klasse **DataOutputStream** in Kombination mit einem Ausgabeobjekt aus der Klasse **FileOutputStream**:




Beispiel:

```
DataOutputStream dos = new DataOutputStream(new FileOutputStream("demo.dat"));
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:



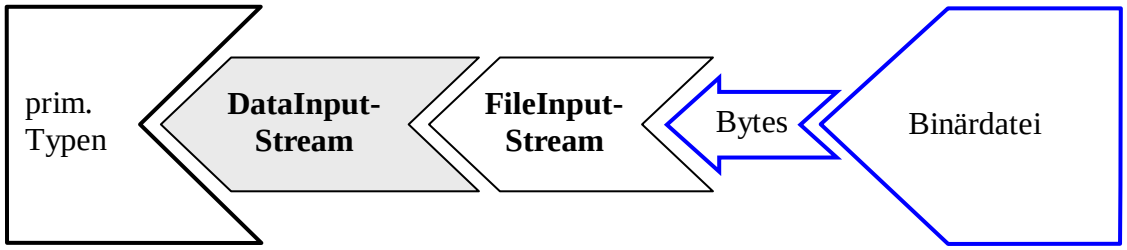

Hier wird ein Puffer mit 16384 Bytes Kapazität verwendet:

```
DataOutputStream dos = new DataOutputStream(
                         new BufferedOutputStream(
                           new FileOutputStream("demo.dat"), 16384));
```

Ein Puffer muss auf jeden Fall vor dem Programmende entleert werden, z.B. durch den generell empfehlenswerten Aufruf der **close()**-Methode des **DataOutputStream**-Objekts.

### 12.7.8 Primitive Datentypen aus einer Binärdatei lesen

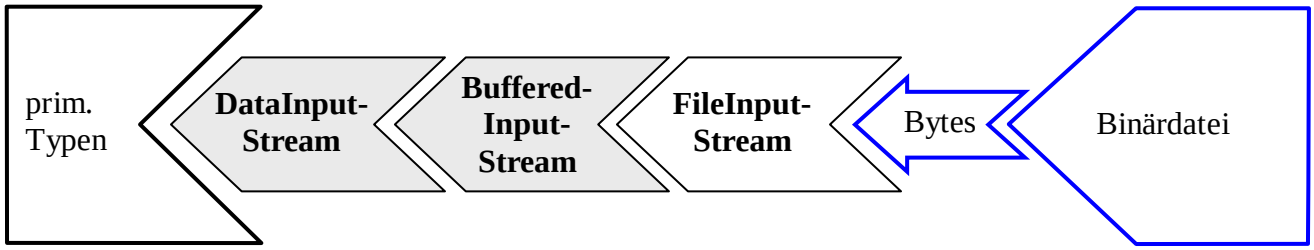
Um primitive Datentypen (z.B. **int**, **double**) aus einer Binärdatei zu lesen, verwendet man ein Filterobjekt aus der Klasse **DataInputStream** in Kombination mit einem Eingabeobjekt aus der Klasse **FileInputStream**:




Beispiel:

```
DataInputStream dis = new DataInputStream(new FileInputStream("demo.dat"));
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:




Hier wird ein Puffer mit 16384 Bytes Kapazität verwendet:

```
DataInputStream dis = new DataInputStream(
                        new BufferedInputStream(
                          new FileInputStream("demo.dat"), 16384));
```

## *12.8 Herabgestufte Methoden*

Im Laufe der Evolution des Java-E/A-Systems mussten einige vormals gebräuchliche Methoden als *deprecated* (dt.: *herabgestuft*, *veraltet*, *unerwünscht*, *missbilligt*) eingestuft werden. Sie sind aus Kompatibilitätsgründen noch vorhanden, doch wird von ihrer Verwendung abgeraten. Ein Beispiel ist die Methode **readLine()** der Klasse **DataInputStream**, die in Java 1.0 verwendet wurde, um Zeichenketten von der Tastatur einlesen, z.B.:

| Quellcode | Ausgabe |
|---|---|
| `import java.io.*;`<br>`class DeprecatedDemo {`<br>`  public static void main(String[] args) throws IOException {`<br>`  String wort;`<br>`  DataInputStream dis = new DataInputStream(System.in);`<br>`  System.out.print("Wort eingeben: ");`<br>`  wort = dis.readLine();`<br>`  System.out.println("Sie gaben ein: " + wort);`<br>`  }`<br>`}` | `Wort eingeben: egal`<br>`Sie gaben ein: egal` |

Als byteorientierte Eingabetransformationsklasse kann **DataInputStream** mit beliebigen **InputStream**-Abkömmlingen kooperieren. In Java ist die Konsoleneingabe als Objekt aus der Klasse **InputStream** realisiert, das über die statische Instanzvariable **System.in** angesprochen werden kann.

Außerdem besitzt die Klasse **DataInputStream** eine Methode **readLine()**, und man kann erwarten, mit einem auf **System.in** aufgesetzten **DataInputStream**-Objekt Zeichenketten von der Tastatur einlesen zu können. Der Beispieldialog klappt auch erwartungsgemäß, aber die folgende Compiler-

Warnung erinnert uns daran, dass man Zeichenketten ja eigentlich *nicht* mit byteorientierten Strömen einlesen soll:

```
Note: DeprecatedDemo.java uses or overrides a deprecated API.
Note: Recompile with -deprecation for details.
```

In der Online-Dokumentation zur **DataInputStream**-Methode **readLine()** erhalten wir genauere Informationen und Empfehlungen:

> **Deprecated.** *This method does not properly convert bytes to characters. As of JDK 1.1, the preferred way to read lines of text is via the* `BufferedReader.readLine()` *method. Programs that use the* `DataInputStream` *class to read lines can be converted to use the* `BufferedReader` *class by replacing code of the form:*
>
> *DataInputStream d = new DataInputStream(in);*
>
> *with:*
>
> *BufferedReader d*
>
> *= new BufferedReader(new InputStreamReader(in));*

Diese Kritik richtet sich nur gegen die Methode **readLine()**, keinesfalls gegen die immer noch wichtige Klasse **DataInputStream**, mit der wir oben erfolgreich Werte primitiver Datentypen aus einer Binärdatei (!) gelesen haben.

## 12.9 Übungsaufgaben zu Kapitel 12

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 Ausgabe von Textdaten verwenden die von **Writer** abstammenden Klassen stets die UTF-8 – Kodierung.

2) Die **FileInputStream**-Methode **read()** versucht, ein Byte aus der angeschlossenen Datei zu lesen. Warum verwendet sie den Rückgabetyp **int**?

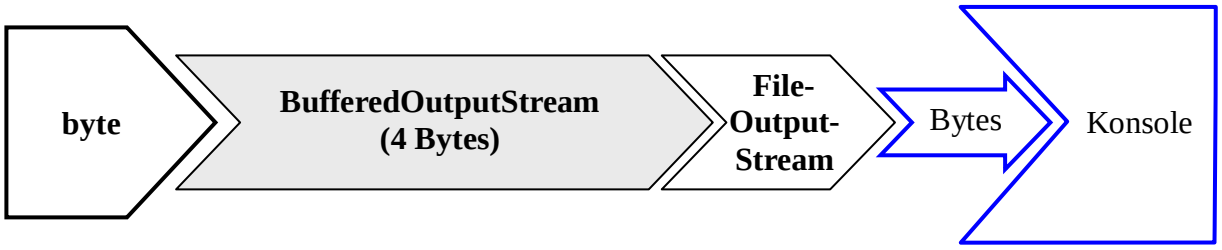
3) Erweitern Sie das Editor-Beispielprogramm in Abschnitt 11.7 um die Möglichkeit, den bearbeiteten Text zu sichern.

4) Erstellen Sie ein Programm zur Demonstration der Ausgabepufferung. Um mitverfolgen zu können, wie beim Überlaufen des Puffers Daten weitergeleitet werden, sollte Sie als Senke die Konsole verwenden.

Wie Sie aus dem Abschnitt 12.3.1.5 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.:

```
□□□□4567
```

Sollten sich bei Programmende noch Bytes im Puffer befinden, müssen diese per **flush()** oder **close()** vor dem Untergang bewahrt werden.

5) 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 {
    PrintWriter pw = null;
    try {
      long time = System.currentTimeMillis();
      FileOutputStream fos = new FileOutputStream("pw.txt");
      pw = new PrintWriter(fos, true);
      for (int i = 1; i < 30000; i++) {
        pw.println(i);
      }
      System.out.println("Zeit: " + (System.currentTimeMillis()-time));
    } finally {
      if (pw != null)
        pw.close();
    }
  }
}
```

6) Als byteorientierte Eingabetransformationsklasse kann **DataInputStream** in Kooperation mit beliebigen **InputStream**-Abkömmlingen die Werte primitiver Datentypen lesen. In Java zeigt die

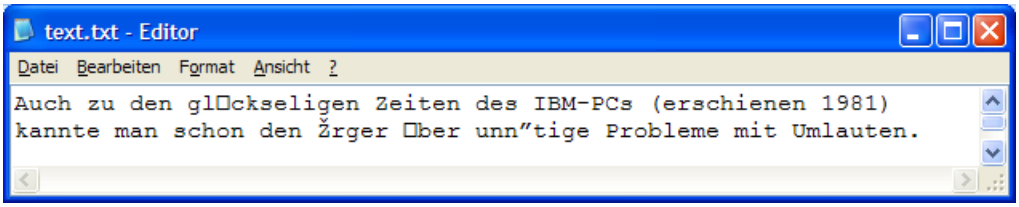
Referenzvariable **System.in** auf ein Objekt aus der Klasse **InputStream** (genauer: auf ein Objekt aus einer Klasse, die von der abstrakten Klasse **InputStream** abstammt). Nun kann man hoffen, mit einem **DataInputStream**-Objekt, das auf **System.in** aufsetzt, Werte primitiver Datentypen von der 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.*;`<br>`class DisSysIn {`<br>`  public static void main(String[] args)`<br>`  throws IOException {`<br>`  short zahl;`<br>`  DataInputStream dis = new DataInputStream(System.in);`<br>`  System.out.print("Zahl eingeben: ");`<br>`  zahl = dis.readShort();`<br>`  System.out.println("Gelesen: " + zahl);`<br>`  }`<br>`}` | `Zahl eingeben: 0`<br>`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?

7) Schreiben Sie ein Programm, das den Text




in der Datei

**...\BspUeb\EA\ASCII-Text\text.txt**

einlesen und korrekt in einem **JOptionPane**-Meldungsfenster darstellen kann:


8) 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 die Daten aus einer Textdatei nach folgendem Muster lesen 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.

# 13 Applets

Java war eine Zeit lang vor allem dazu gedacht, das WWW (*World Wide Web*) mit kleinen „Prográmmchen“ (*Applet* = Verkleinerungsform des englischen Worts *Application*) aufzupolieren, die auf einem Webserver bereitgestellt, via Internet schnell zum lokalen Rechner transportiert und dort in einem Browser-Fenster ausgeführt werden können. Auf diese Weise lassen sich HTML-Seiten attraktiv (z.B. multimedial), dynamisch und interaktiv gestalten. Die Anwender können einen prinzipiell unbegrenzten Fundus an Software nutzen, ohne diese lokal installieren zu müssen. Zwei Beispiele für Java-Applets (mit dem Spiel Tic-Tac-Toe bzw. einer Demonstrationen zur linearen Regressionsanalyse der Statistik) waren schon in Abschnitt 1.2.4 zu sehen.

Die ursprüngliche Bevorzugung von Applets gegenüber den Java-Applikationen, die wir im bisherigen Kursverlauf ausschließlich kennen gelernt haben, kommt z.B. darin zum Ausdruck, dass einige Java-Versionen lang die Sound-Ausgabe ausschließlich in Applets möglich war. Mittlerweile hat sich die Lage jedoch deutlich gewandelt:

- Es gibt einige alternative Möglichkeiten zur dynamischen und interaktiven Gestaltung von Webseiten (z.B. Ajax, JavaScript, Flash, JavaFX, Silverlight, ActiveX). JavaFX wurde von Sun Microsystems als eingeschränkte, auf die Entwicklung von *Rich Internet Applications* (RIA) spezialisierte und leichter erlernbare Java-Variante entwickelt. Vermutlich wird hier der Umgang mit dem Swing-GUI und mit dem Java2D-API (vgl. Abschnitt 14.1) erleichtert.
- Java hat sich inzwischen zu einer vollwertigen, weitgehend universell einsetzbaren Programmiersprache entwickelt, die wegen ihrer modernen Konzeption hohes Ansehen geniest und eine zunehmende Verbreitung erlebt. Java-Applikationen haben mittlerweile eine größere Bedeutung als Applets. Außerdem ist die Java Enterprise Edition (JEE) zu einer festen Größe auf dem Markt für unternehmensweite oder serverorientierte Lösungen geworden, und die so genannten *Servlets* der JEE spielen heutzutage eine wichtigere Rolle als die Applets. Wir werden uns im weiteren Kursverlauf noch ausführlich mit Servlets und einigen weiteren Aspekten der JEE beschäftigen. Schließlich ist auch noch die Java Micro Edition (JME) für Kommunikationsgeräte mit beschränkter Leistung zu nennen.

Damit haben Applets zwar ihre ursprüngliche Bedeutung verloren, doch bieten sie nach wie vor interessante Möglichkeiten für viele Szenarien und stellen damit einen Zusatznutzen der Programmiersprache Java dar.

Wir konzentrieren uns im aktuellen Abschnitt auf die Besonderheiten von Applets hinsichtlich Funktionsweise und Laufzeitumgebung. Multimediale Applet-Beispiele lernen Sie in Abschnitt 14 kennen.

## *13.1 Kompatibilität*

Durch die Browser-Einbettung ergeben sich zwar einige Besonderheiten bei Applets, doch bleiben wir bei derselben Programmiersprache, können also die Syntaxregeln und den größten Teil der Java-Klassen auch bei Applets verwenden. Über Möglichkeiten, eine Java-Applikation mit geringem Aufwand in ein Java-Applet umzuwandeln, informiert z.B. Krüger & Stark (2007, Abschnitt 40.3).

Damit ein Java-Applet ausgeführt werden kann, muss der WWW-Browser mit einer virtuellen Java-Maschine kooperieren, die auf entsprechendem Versionsstand ist. Während man bei einer Java-*Applikation* die JVM problemlos mitliefern und mitinstallieren kann, ist man bei einem Applet auf die beim Benutzer vorhandene Ausführungsumgebung angewiesen. Man darf davon ausgehen, dass weit über 90% alle Internet-Benutzer über einen Browser mit Java-Unterstützung verfügen. Leider gibt es wenige Informationen zur Verbreitung der verschiedenen Java-Versionen. Wer im Sinne maximaler Kompatibilität eine möglichst niedrige Java-Version voraussetzen möchte, benötigt ei-

nen passenden Compiler. In unserer Entwicklungsumgebung Eclipse 3 kann man über das Kontextmenü zu einem (Applet)-Projekt mit

**Eigenschaften > Java-Compiler**

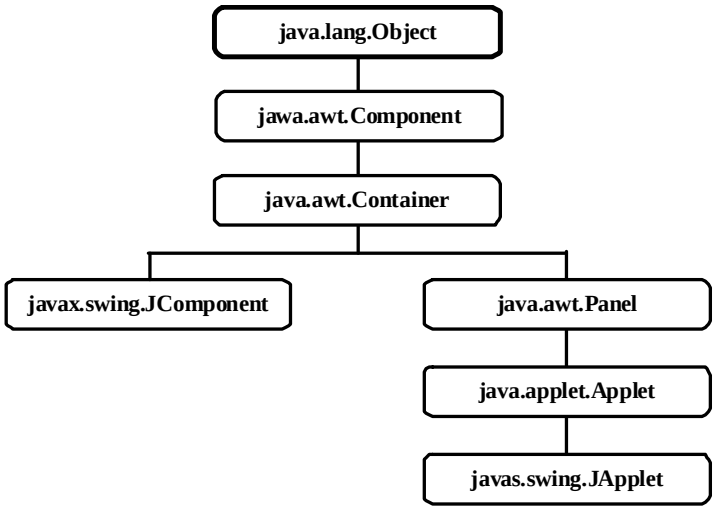
die Java-Version 1.3 als niedrigste **Konformitätsstufe** einstellen.

Im hauseigenen *Intranet* einer Organisation lässt sich eine aktuelle Laufzeitumgebung für Applets auf den Klientenrechnern leicht herstellen, indem dort die Java Runtime Environment (JRE) der Firma Sun Microsystems installiert wird (siehe Abschnitt 13.6).

## *13.2 Stammbaum der Applet-Basisklasse*

Wie eine Java-*Anwendung* besteht auch ein Java-*Applet* aus mindestens einer Klassendefinition. Von der beim Start anzugebenden Hauptklasse eines Applets wird automatisch ein Objekt erzeugt, das als Top-Level - Container fungiert. Die Hauptklasse wird aus **java.applet.Applet** (bei Beschränkung auf AWT-Technik) oder aus **javax.swing.JApplet** (bei Verwendung von Swing-Komponenten) abgeleitet.

Um zu wissen, welche Methoden und Variablen die Klasse **(J)Applet** von ihren Vorfahren erbt, lohnt sich ein Blick auf den Stammbaum:




Insbesondere ist ein Applet also eine *Komponente* und folglich eine potentielle Ereignisquelle. Während eine Java-Anwendung auch „ereignislos“ und textorientiert entworfen werden kann, ist ein Applet stets grafik- und ereignisorientiert.

Die in der Klasse **(J)Applet** im Vergleich zu einer Java-*Anwendung* definierten Zusatzkompetenzen beziehen sich vor allem auf die Kooperation mit dem Browser, in dessen Kontext ein Applet abläuft. Wichtige Kooperationsmethoden werden gleich mit Hilfe des folgenden Beispiels erläutert, das auf der Klasse **java.applet.JApplet** basiert:

```
import java.awt.*;
import javax.swing.*;

public class Life extends JApplet {
   private int startAnzahl, stopAnzahl;
   private JLabel lblStatus, lblStatusWert, lblStart, lblStartAnzahl,
                  lblStop, lblStopAnzahl;
```

```
    public void init() {
       lblStatus = new JLabel("Status");
       lblStatusWert = new JLabel("Neu initialisiert");
       lblStart = new JLabel("Start-Aufrufe");
       lblStartAnzahl = new JLabel("0");
       lblStop = new JLabel("Stop-Aufrufe");
       lblStopAnzahl = new JLabel("0");
       setLayout(new GridLayout(3,2));
       add(lblStatus); add(lblStatusWert);
       add(lblStart); add(lblStartAnzahl);
       add(lblStop); add(lblStopAnzahl);
    }

    public void start() {
       startAnzahl++;
       lblStartAnzahl.setText(String.valueOf(startAnzahl));
    }

    public void stop() {
       stopAnzahl++;
       lblStatusWert.setText("Altlast");
       lblStopAnzahl.setText(String.valueOf(stopAnzahl));
       JOptionPane.showMessageDialog(null, "Stop");
    }

    public void destroy() {
       JOptionPane.showMessageDialog(null, "Destroy");
    }
}
```

Weil hier kritische Ereignisse im Leben eines Applets sichtbar gemacht werden sollen, hat die Klasse den Namen `Life` erhalten.

Während die Startklasse einer Anwendung ohne den Modifikator **public** auskommt, kann die Startklasse eines Applets nicht darauf verzichten.

## *13.3 Applet-Start via Browser oder Appletviewer*

Wesentliche Eigenart eines Applets ist seine Einbettung in eine HTML-Seite, wobei traditionell ein so genanntes **applet**-Tag verwendet wird, das im `Life`-Beispiel z.B. folgendermaßen aussehen kann:

```
<html>
<head>
<title>Demo-Applet Life</title>
</head>
<applet code="Life.class" width=250 height=100>
</applet>
</body>
</html>
```

Bei den Attributen des **applet**-Tags beschränkt sich das Beispiel auf die wichtigsten Angaben:

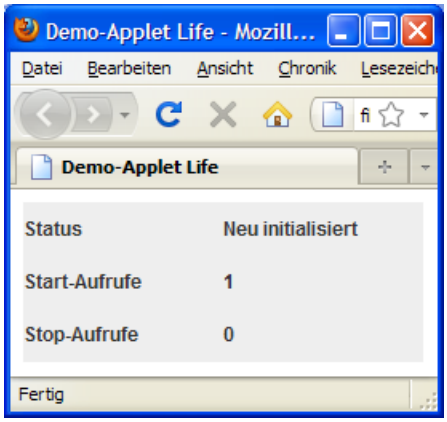
- **code**
  Name der Bytecodedatei zur Hauptklasse
- **width**, **height**
  Die Breite und Höhe der Appletfläche in Pixeln

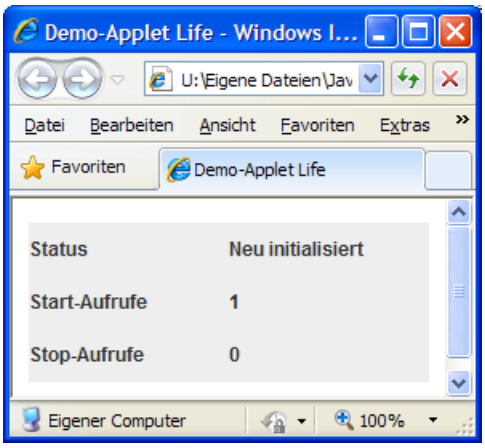
Das **World Wide Web Consortium** (W3C) deklariert seit der HTML-Spezifikation 4 das **applet**-Tag als *deprecated* (herabgestuft, veraltet) und empfiehlt, stattdessen das **object**-Tag zu verwenden.

In der Dokumentation zu Java 5.0[63] empfiehlt Sun Microsystems jedoch, das **applet**-Tag weiterhin zu verwenden:

> *Note: The HTML specification states that the `applet` tag is deprecated, and that you should use the `object` tag instead. However, the specification is vague about how browsers should implement the `object` tag to support Java applets, and browser support is currently inconsistent. Sun therefore recommends that you continue to use the `applet` tag as a consistent way to deploy Java applets across browsers on all platforms.*

Um ein Applet in einem Browser auszuführen, öffnet man das zugehörige HTML-Dokument über das **Datei**-Menü, per Doppelklick oder Drag-&-Drop, z.B.:



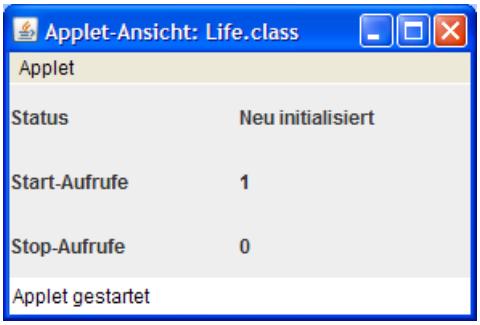




Die Rolle des Browsers beim Starten und Ausführen eines Java-Applets kann auch vom JDK-Hilfsprogramm **Appletviewer** übernommen werden. In der Entwicklungsphase ist der Appletviewer sogar zu bevorzugen, weil die übliche Cache-Strategie der Browser das Testen der aktuellen Applet-Version oft erschwert.

Man kann den Appletviewer in einem Konsolenfenster starten und dabei den Namen der HTML-Datei als Kommandozeilenparameter übergeben, z.B.:

```
U:\Eigene Dateien\Java\Life>appletviewer Life.htm
```

Das `Life` - Applet sieht im Fenster des Appletviewers unmittelbar nach dem Start so aus:



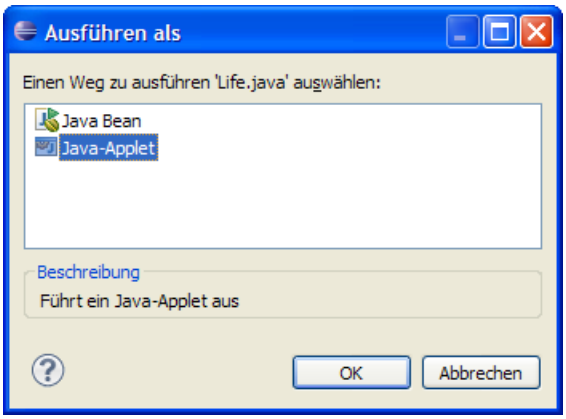

In der Entwicklungsumgebung Eclipse 3 wählt man beim ersten Start eines Applets z.B. die Option

**Ausführen als > Java-Applet**

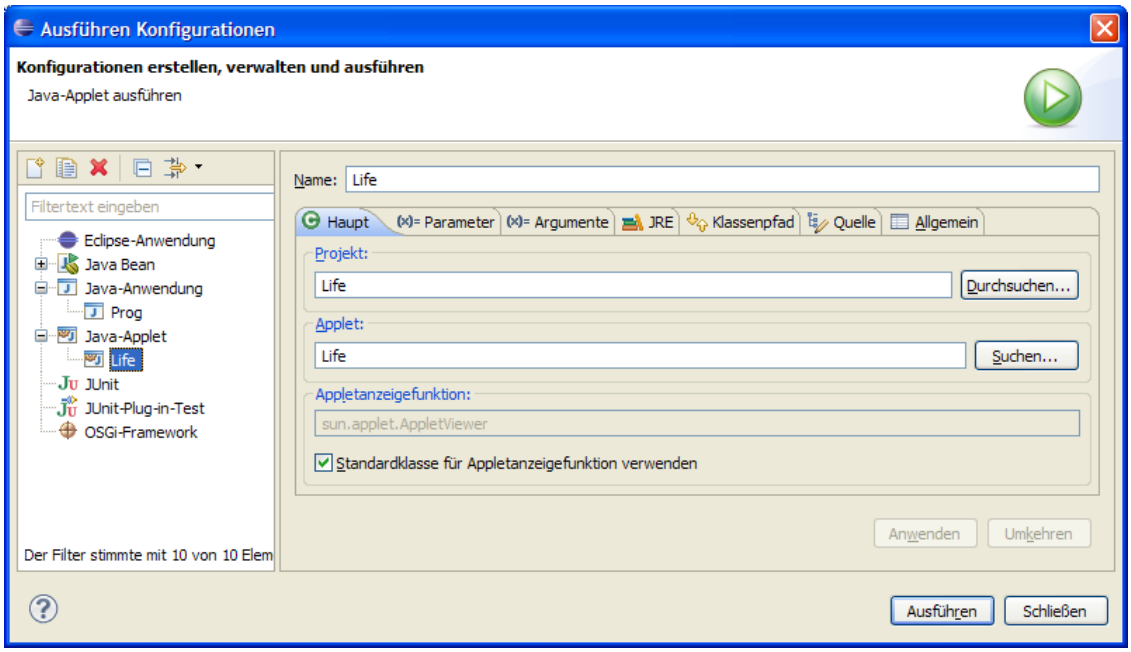
aus dem Kontextmenü zur Hauptklasse im Paket-Explorer oder wählt nach einem Mausklick auf den **Ausführen**-Schalter die passende Option in folgender Dialogbox:

---

63 http://java.sun.com/j2se/1.5.0/docs/guide/plugin/developer_guide/using_tags.html




Es entsteht eine neue Startkonfiguration, die anschließend bei einem Mausklick auf den **Ausführen**-Schalter zum Einsatz kommt, wenn das zugehörige Projekt aktiv ist:




Eclipse verwendet den Appletviewer und erzeugt automatisch eine umrahmende HTML-Datei, die im Projektordner zu finden ist, während das aus Eclipse gestartete Applet läuft. Wie man sieht, verwendet auch Eclipse 3 noch das **applet**-Tag:

```
<html>
<body>
<applet code=Life.class width="200" height="200" >
</applet>
</body>
</html>
```

Über weitere Attribute des **applet**-Tags und sonstige Optionen beim Einbinden von Applets auf HTML-Seiten informiert z.B. Münz (2007).

## 13.4 Methoden für kritische Lebensereignisse

Wer das Applet in Abschnitt 13.2 vor dem Hintergrund unserer bisherigen Erfahrungen mit Java-Anwendungen näher betrachtet, wird sicher die **main()**-Methode vermissen, über die eine Java-*Anwendung* von der JRE in Gang setzt wird. Bei einem Applet läuft die Startphase anders ab:

- Der Browser (oder Appletviewer) erzeugt automatisch ein Objekt der im **applet**-Tag angegebenen Hauptklasse, die unbedingt als **public** deklariert werden muss.
- Anschließend ruft der Browser (oder Appletviewer) die **(J)Applet**-Methode **init()** auf, um das neue Objekt zu initialisieren. Bei Applets verwendet man zum Initialisieren keinen Konstruktor, weil vor dem **init()** - Aufruf keine vollständige Laufzeitumgebung garantiert ist, so dass z.B. das Laden von Bildern oder Audio-Clips misslingen könnte.
- Nach **init()** ruft der Browser (oder Appletviewer) die **start()**-Methode des Applets auf. Nun erscheint das Applet auf der Bildfläche, wird also zum Zeichnen seiner Komponenten aufgefordert und über Ereignisse informiert. Anders als bei einer Swing-*Anwendung* (vgl. Abschnitt 11.3) ist ein Aufruf der **Component**-Methode **setVisible()** weder erforderlich noch sinnvoll, um ein Applet in Szene zu setzen.

In unserem Beispiel wählt **init()** eine Hintergrundfarbe, erstellt die **Label**-Komponenten und fügt sie in den Top-Level - Container ein, der von einem **GridLayout** verwaltet wird. Die Methoden **start()** und **stop()** erhöhen im Beispiel jeweils eine Zählvariable, welche die Anzahl ihrer bisherigen Aufrufe festhält, z.B.:


Damit der Browser (oder Appletviewer) die genannten Methoden aufrufen kann, müssen sie alle als **public** definiert werden. Sollten Sie anderes planen, wird schon der Compiler protestieren: Die Methoden sind in den Basisklassen als **public** definiert, und beim Überschreiben dürfen generell keine Zugriffsrechte eingeschränkt werden.

Nach Abschluss der Initialphase muss ein Applet zusätzlich mit folgenden Methodenaufrufen durch den Browser rechnen:

- Der Browser kann die **stop()**-Methode eines Applets ausführen, um es vorübergehend zu desaktivieren, z.B. weil das Browserfenster in die Taskleiste beordert worden ist. Dabei bleiben alle Ressourcen des Applets erhalten, so dass es später fortgesetzt werden kann, wobei die **start()**-Methode erneut ausgeführt wird. Im Unterschied zum Appletviewer machen aktuelle Browser (Firefox 3, Internet Explorer 8) wenig Gebrauch von der **stop()**-Methode und rufen sie nur vor dem Beenden des Applets auf. Dementsprechend kommt die **Start()**-Methode nur in der Initialphase zum Einsatz.
- Schließt der Benutzer das Fenster bzw. die Registerkarte mit dem Applet, dann ruft der Browser die Methode **destroy()** auf, wobei das Applet ggf. Ressourcen freigegeben sollte.

## 13.5 Sonstige Methoden für die Applet-Browser - Kooperation

Anschließend werden noch einige Methoden für die Interaktion zwischen Applet und Browser vorgestellt. Wer sich für die Interaktion zwischen mehreren, simultan aktiven, Applets interessiert, kann sich z.B. bei Krüger & Stark (2007, Abschnitt 40.2) informieren.

### 13.5.1 Parameterwerte aus der HTML-Datei übernehmen

Ein Applet-Programmierer kann seinem Kunden (dem Web-Designer) eine Möglichkeit schaffen, das Verhalten des Applets über Parameter zu steuern. Zur Vereinbarung von konkreten Parameterausprägungen im HTML-Code dienen **param**-Tags, z.B.:

```
<html>
<head>
<title>Parameterübernahme aus dem HTML-Quelltext</title>
</head>
<body>
<applet code="Param.class" width=200 height=50>
<param name = "Par1" value = "3">
</applet>
</body>
</html>
```

Pro **param**-Tag ist ein **name-value** - Paar erlaubt:

- **name**
  Unter diesem Namen ist ist der Paramter im Applet ansprechbar.
- **value**
  Das Applet erhält den Wert stets als **String**.

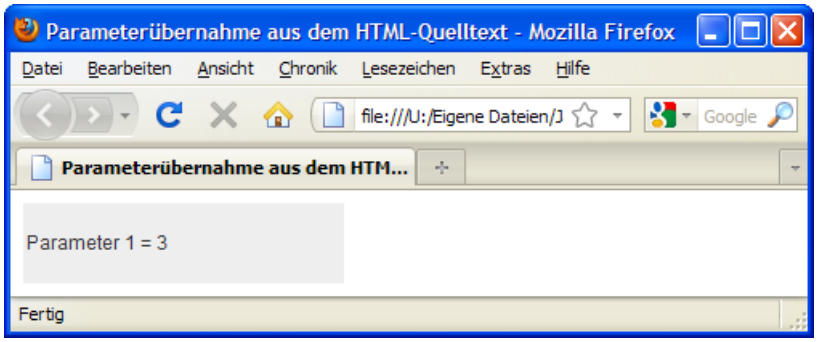
Ein Applet wird seine Steuerparameter in der Regel schon in der **init()**-Methode ermitteln, z.B.:

```
public class Param extends javax.swing.JApplet {
  public void init() {
    add(new java.awt.Label("Parameter 1 = " + getParameter("Par1")));
  }
}
```

Die zuständige **Applet**-Methode **getParameter()** liefert nur Strings, so dass eventuell mit den entsprechenden Methoden der Wrapper-Klassen noch eine Konvertierung vorzunehmen ist, z.B.:

```
int pari1 = Integer.parseInt(par1);
```

Im Beispiel ist eine solche Wandlung nicht erforderlich:

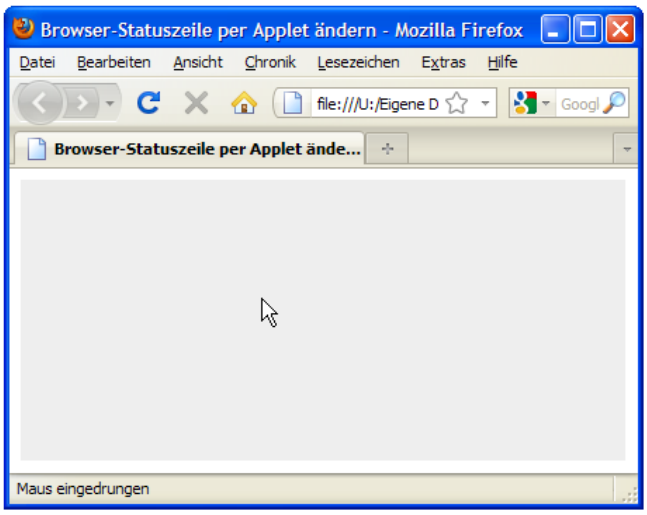



### 13.5.2 Browser-Statuszeile ändern, Ereignisbehandlung

Über seine Methode **showStatus()** hat ein Applet Zugriff auf die Statuszeile des Browserfensters. In folgendem Beispiel werden dort Besuche der Maus dokumentiert:

```
import java.awt.event.*;

public class Statuszeile extends javax.swing.JApplet {
   public void init() {
      addMouseListener(new MouseDetector());
   }

   private class MouseDetector extends MouseAdapter {

      public void mouseEntered(MouseEvent e) {
         showStatus("Maus eingedrungen");
      }

      public void mouseExited(MouseEvent e) {
         showStatus("Maus entwischt");
      }
   }
}
```

Wie das Beispiel zeigt, ist die in Abschnitt 11.6 vorgestellte Ereignisbehandlung auch bei Applets verwendbar:




### 13.5.3 Andere Webseiten öffnen

Über die Möglichkeit, aus einem Applet heraus HTML-Seiten zu öffnen, kann man z.B. ein Frameset mit Navigationszone erstellen:


Im oberen Frame (`navigation` genannt) wird die Datei **Navigation.htm** geöffnet, die das Applet **Appligator.class** aufruft. Im unteren Frame (`content` genannt) wird initial die Datei **Con-**

**tent1.htm** (mit der animierten Java-Tasse) geöffnet. Das gesamte Frameset ist in der Datei **Navi-Demo.htm** enthalten:

```
<html>
<head>
<title>Navigation per Applet</title>
</head>
<frameset rows="*,*">
  <frame name="navigation" src="Navigation.htm">
  <frame name="content" src="Content1.htm">
  <noframes>
  <body>
  <p>Diese Seite verwendet Frames. Frames werden von Ihrem Browser aber nicht
  unterstützt.</p>
  </body>
  </noframes>
</frameset>
</html>
```

In der Datei **Navigation.htm** wird mit dem **center**-Tag für einen zentrierten Auftritt des Applets gesorgt:

```
<html>
<body>
<center>
<applet code="Appligator.class" width=200 height=30>
</applet>
</center>
</body>
</html>
```

In der `Appligator`-Klassendefinition werden gemäß obiger Verabredung die Befehlsschalter über AWT-Komponenten realisiert. An Stelle der Swing-Klasse **JButton** kommt die AWT-Klasse **Button** zum Einsatz, was aber keine weiteren syntaktischen Konsequenzen hat.

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.*;

public class Appligator  extends JApplet  implements ActionListener {
  private JButton duke = new JButton("Duke");
  private JButton java = new JButton("Java");

  public void init() {
    add(java, BorderLayout.WEST);
    add(duke, BorderLayout.EAST);
    getContentPane().setBackground(Color.WHITE);
    java.addActionListener(this);
    duke.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    URL url;
    try {
      if (e.getSource() == java)
        url = new URL(getCodeBase(), "Content1.htm");
      else
        url = new URL(getCodeBase(), "Content2.htm");
      getAppletContext().showDocument(url, "content");
    }
    catch (Exception ex) {//Ausnahmebehandlung
    }
  }
}
```

Die **JApplet**-Komponente fungiert als Top-Level - Container (vgl. Abschnitt 11.2.2) und verwendet dabei per Voreinstellung das **BorderLayout** (vgl. Abschnitt 11.5.3). In der **init()**-Methode des Beispiels werden die beiden **JButton**-Komponenten über die **Container**-Methode **add()** aufgenommen. Ein **JApplet**-Container ist (anders als ein **Applet**-Container) wie ein **JFrame**-Container mehrschichtig aufgebaut (vgl. Abschnitt 11.2.2), und alle Kind-Komponenten sollten in der Inhaltsschicht untergebracht werden. Dass die **add()**-Aufrufe trotzdem wie im folgenden Beispiel

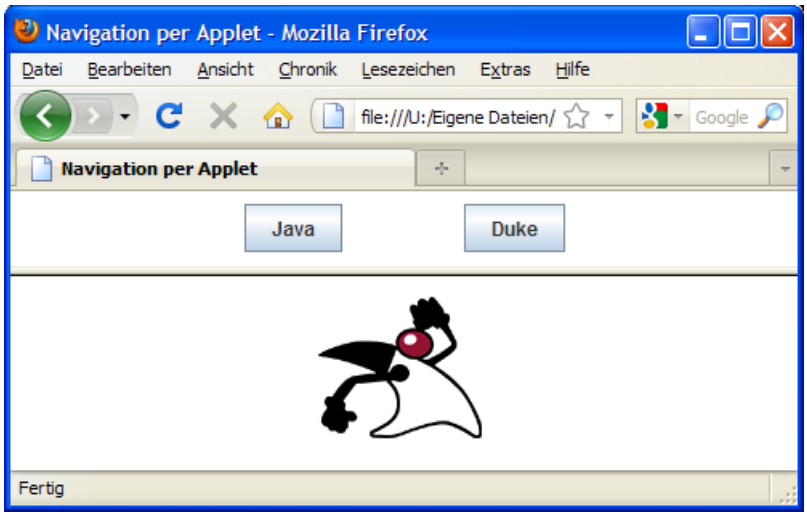
```
add(java, BorderLayout.WEST);
```

direkt an die **JApplet**-Komponente gerichtet werden können, liegt an den seit Java 5 vorhandenen Bequemlichkeitsüberschreibungen in der Klasse **JApplet**.[64]

Die Klasse `Appligator` ist als **ActionEvent**-Empfänger für die Befehlsschalter gerüstet, weil sie das Interface **ActionListener** implementiert. In der Methode **actionPerformed()** wird je nach betätigtem Schalter ein **URL**-Objekt zur passenden HTML-Datei erzeugt (**Content1.htm** oder **Content2.htm**). Mit der Klasse **URL**, die Internet-Adressen repräsentiert, werden wir uns im Zusammenhang mit der Netzwerkprogrammierung noch näher beschäftigen (siehe Kapitel 16). Im verwendeten **URL**-Konstruktor wird die neue Adresse aus dem Namen der HTML-Datei und der URL zum Verzeichnis mit dem Applet zusammengesetzt. Letzteres liefert die **Applet**-Methode **getCodeBase()**.

Über **getAppletContext()** wird ein Objekt erreicht, welches den Browser vertritt. An dieses Objekt richtet sich die Botschaft **showDocument()**, die als Parameter die URL der anzuzeigenden HTML-Seite sowie den Zielframe enthält. Wie alternative Ausgabeziele (z.B. ein neues Fenster bzw. Registerblatt des Browsers) angesprochen werden können, ist in der API-Dokumentation zur **Applet-Context**-Methode **showDocument()** zu erfahren.

Abschließend soll noch der Zustand nach einem Mausklick auf den Schalter **Duke** gezeigt werden:



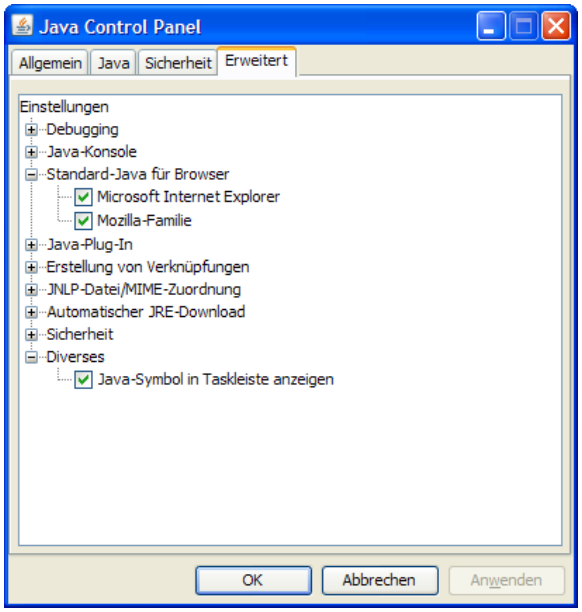

---

[64] Wer neugierig darauf ist, wie in der **JApplet**-Klassendefinition die **add()**-Aufrufe weitergeleitet werden, sollte die Datei **JApplet.java** öffnen und einen Blick auf die Methode **addImpl()** werfen:

```
protected void addImpl(Component comp, Object constraints, int index) {
    if(isRootPaneCheckingEnabled()) {
        getContentPane().add(comp, constraints, index);
    } else {
        super.addImpl(comp, constraints, index);
    }
}
```

## 13.6 Das Java-Browser-Plugin

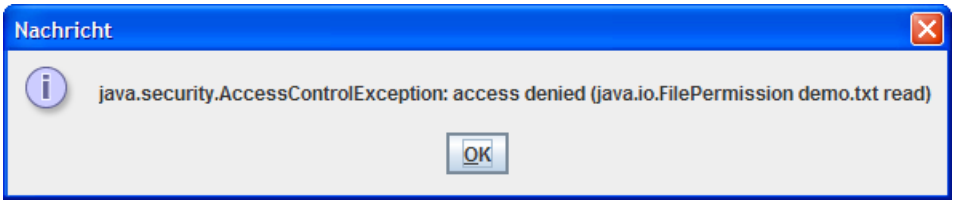
Zusammen mit der Java Runtime Environment (JRE) oder mit dem entsprechenden Java Development Kit (JDK) der Firma Sun Microsystems wird per Voreinstellung für die angetroffenen Browser aus der Mozilla-Familie (z.B. Firefox) und für Microsofts Internet Explorer das **Java-Plugin** installiert. Es sorgt dafür, dass vom **applet**-Tag die JVM der Firma Sun Microsystems angesprochen wird. Unter Windows landet außerdem in der Systemsteuerung das **Java(TM)-Bedienungsfeld**. Hier kann man z.B. einstellen, ob das Plugin als Java-Laufzeitumgebung für die unterstützten Browser verwendet werden soll:




Ein aktives Java-Plugin macht sich per Voreinstellung im Infobereich der Windows-Taskleiste (neben der Uhr) durch ein Java-Symbol bemerkbar, z.B.:


## 13.7 Das Sicherheitsmodell für Applets

Ein via Browser gestartetes Applet wird in einem abgeschotteten **Sandkasten** ausgeführt und hat gegenüber einer Java-Anwendung deutlich reduzierte Standardrechte. Es darf weder lesend noch schreibend auf lokale Dateien zugreifen




und ist bei Netzwerkverbindungen auf den eigenen Herkunfts-Server beschränkt.

Eclipse 3 legt bei Applets allerdings automatisch im Projektordner die Datei **java.policy.applet** an und genehmigt hier für den Start aus Eclipse volle Zugriffsrechte:

```
/* AUTOMATICALLY GENERATED ON Tue Apr 16 17:20:59 EDT 2002*/
/* DO NOT EDIT */

grant {
  permission java.security.AllPermission;
};
```

Nähere Informationen zum Java-Sicherheitsmodell finden Sie z.B. auf der Webseite

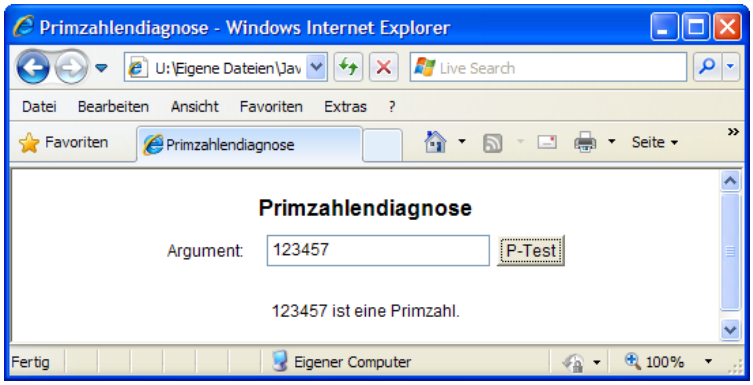
http://java.sun.com/developer/technicalArticles/Security/applets/

## 13.8 Übungsaufgaben zu Kapitel 13

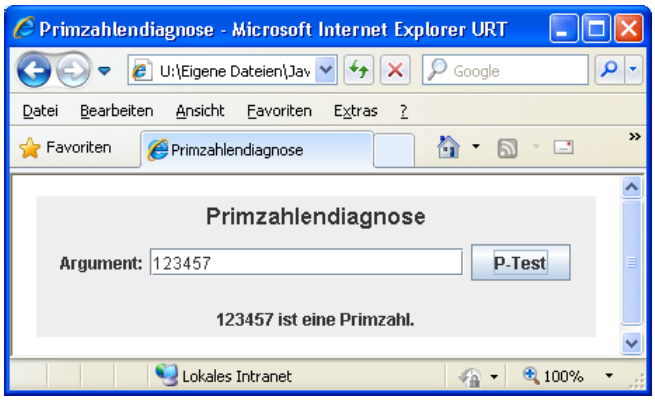
1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Unter den RIA-Lösungen (*Rich Internet Application*) sind die Applets eine oft unterschätzte Option.
2. Ein Applet wird aus Sicherheitsgründen in einer sogenannten Sandbox ausgeführt und darf z.B. per Voreinstellung nicht auf das lokale Dateisystem zugreifen.
3. Bei Applets sollte man keine Konstruktoren zum Initialisieren neuer Objekte verwenden.
4. Bei Applets sollte man aus Kompatibilitätsgründen auf das Swing-Toolkit verzichten.

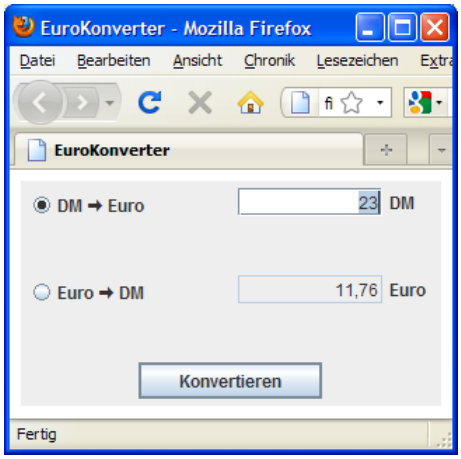
2) Erstellen Sie ein Applet zur Primzahlendiagnose, wahlweise mit AWT-GUI




oder mit Swing-GUI:




3) Erstellen Sie eine Applet-Variante zum Euro-DM - Konverter, den Sie als Übungsaufgabe zu Abschnitt 11 entwickelt haben. Verwenden Sie dabei ein Swing-GUI, z.B.:



# 14 Multimedia

Dieser (noch entwicklungsbedürftige) Abschnitt enthält elementare Hinweise zu 2D-Grafik und Sound.

## *14.1 2D-Grafik*

Die bereits behandelten GUI-Komponenten bieten etliche Möglichkeiten zur individuellen Gestaltung der Benutzeroberfläche. Allerdings ist oft eine freie Grafikausgabe unumgänglich, die von Java mit zahlreichen Methoden zur Ausgabe von

- Vektorgrafik (z.B. Linien, Ellipsen, Rechtecke),
- Pixelgrafik und
- Text

unterstützt wird. Wir haben bereits in der Einleitung dieses Applet


von der Webseite (http://www.math.csusb.edu/faculty/stanton/m262/index.html) bewundert, das die lineare Regressionsanalyse der Statistik mit Hilfe von 2D-Grafiken erläutert.

Beim Swing-GUI kann man auf alle Objekte einer von **JComponent** abstammenden Klasse zeichnen. Die Top-Level - Container - Klasse **JFrame** hat einen alternativen Stammbaum und sollte *nicht* direkt bemalt werden. Stattdessen fügt man eine **JPanel**-Komponente ein und verwendet diese als Zeichenoberfläche.

Weil bei den AWT-Komponenten die Wirtsplattform wesentlich beteiligt ist, gelten hier Einschränkungen: Ein Bemalen der Oberfläche ist nur möglich bei den Top-Level - Containern (**Frame**, **Dialog**, **Window**, **Applet**) sowie bei der Komponente **Canvas**.

Auch bei der Grafikausgabe bietet das Swing-API wesentliche Verbesserungen im Vergleich zum älteren AWT-API (z.B. die automatische Doppelpufferung gegen das Flackern bei der Bildschirmausgabe).

Selbstverständlich ist der Bildschirm nicht das einzig mögliche Ausgabegerät für 2D-Grafik in Java. Als hoch relevante Alternative wird auch die Druckausgabe unterstützt, wobei eine Behandlung im Kurs aus Zeitgründen leider nicht möglich ist.

Wie es bei einem erfolgreichen und traditionsreichen Software-System nicht anders zu erwarten, hat in Java auch bei der Grafikausgabe eine Entwicklung stattgefunden, so dass teilweise ursprüngliche und aktuelle Lösungen koexistieren. Wir behandeln zunächst die Grundlogik der Graphikausgabe und verwenden dabei die ursprüngliche Ausgabetechnik. Sie ist weniger leistungsfähig, aber auch etwas einfacher als das aktuelle Java2D-API, das später vorgestellt wird.

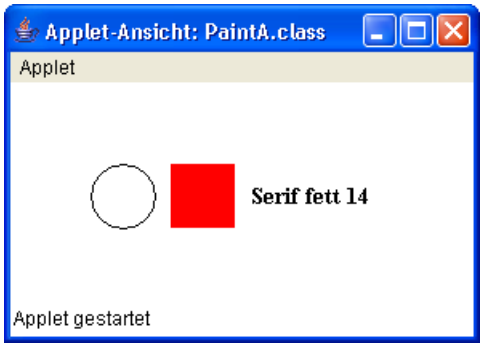
### 14.1.1 Organisation der Graphikausgabe

#### *14.1.1.1 Die Klasse Graphics*

Als erstes Beispiel verwenden wir ein AWT-Applet, weil hier die Grafikausgabe besonders einfach zu realisieren ist:

```
import java.awt.*;
import java.applet.*;

public class PaintAWT extends Applet {
   public void init() {
      setFont(new Font(Font.SERIF, Font.BOLD, 14));
   }

   public void paint(Graphics g) {
      g.drawOval(50, 50, 40, 40);
      g.drawString("Serif fett 14", 150, 75);
      g.setColor(Color.RED);
      g.fillRect(100, 50, 40, 40);
   }
}
```

Im Appletviewer (vgl. Abschnitt 13.3) siehe die Ausgabe so aus:




Für die Grafikausgabe auf der Oberfläche einer Komponente ist ein Objekt aus einer potentiell plattformabhängigen Klasse zuständig, die aus der abstrakten Klasse **java.lang.Graphics** abgeleitet wird.[65] Ein **Graphics**-Objekt bietet zahlreiche Ausgabemethoden für graphische Elemente wie Linien, Rechtecke, Ovale, Polygone, Texte etc. (siehe API-Dokumentation) und verwaltet die dabei relevanten Kontextinformationen, z.B.

- Position und Größe der rechteckigen Zeichenfläche
- aktuelle Zeichenfarbe
- aktuelle Schriftart

Wie das Beispiel demonstriert und der Abschnitt 14.1.1.2 ausführlich erläutert, bringt man Arbeitsaufträge an **Graphics**-Objekte am besten in den Methoden unter, die vom Laufzeitsystem automatisch aufgerufen werden, wenn eine Komponente ihre Oberfläche neu zeichnen soll. Je nach GUI-Toolkit ist für diesen Zweck die **paint()**-Methode (AWT) bzw. die **paintComponent()**-Methode (Swing) einer Komponente zu überschreiben. Diesen Methoden wird beim Aufruf ein **Graphics**-Objekt per Parameter mitgeliefert.

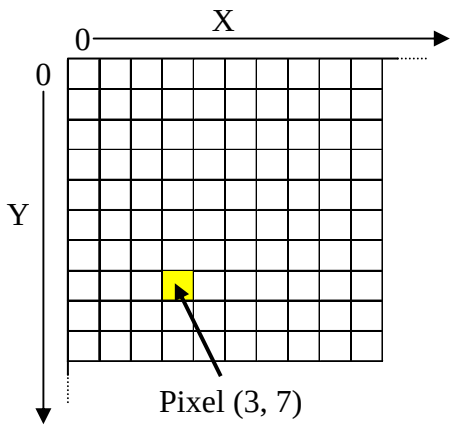
Die Grafikausgabe basiert per Voreinstellung auf einem Koordinatensystem mit dem Ursprung (0,0) in der linken oberen Ecke der betroffenen Komponente, wobei die X-Werte von links nach rechts, und die die Y-Werte von oben nach unten wachsen:

---

[65] Um welche Klasse es sich konkret handelt, kann man über die **Object**-Methode **getClass()** ermitteln:

```
g.drawString(g.getClass().toString(), 50,50);
```

Bei Java 6.0 stellt man z.B. unter Mac OS X 10.4.11 und Windows XP die **K**lasse **sun.java2d.SunGraphics2D** fest.




Aus der Bindung an die Pixel-Matrix des Ausgabegeräts resultiert eine störende Auflösungsabhängigkeit, die im Java2D-API überwunden wird (siehe Abschnitt 14.1.3.3).

#### *14.1.1.2 System-initiierte Grafikausgabe*

Eine Komponente muss aus verschiedenen Anlässen neu gezeichnet bzw. aktualisiert werden, z.B.:

- Das umgebende Rahmenfensters wird erstmals angezeigt oder kehrt aus dem minimierten Zustand zurück.
- Ein zuvor überdeckter Teil der Komponente wird sichtbar.
- Änderungen im Programm (im dargestellten Modell) erfordern eine Aktualisierung der Anzeige.

In bisherigen GUI-Programmen mussten wir uns nicht damit beschäftigen, wie das Laufzeitsystem die Aktualisierung der eventuell mehrstufig verschachtelten Komponenten organisiert. Sobald sich ein Programm nicht auf Standardkomponenten beschränkt, sondern „freie“ Zeichnungen vornimmt, wird jedoch ein Blick hinter die Kulissen relevant:

- Die Aktualisierung beginnt mit der „umfassendsten“ änderungsbedürftigen Komponente und wird jeweils mit den enthaltenen Komponenten fortgesetzt.
- Weil die GUI-Ausgabemethoden im selben Thread (Ausführungsfaden, siehe unten) ablaufen wie die Ereignisbehandlungsmethoden, gilt:
  - Während eine Ereignisbehandlungsmethode abläuft, ist keine GUI-Ausgabe möglich.
  - Während eine GUI-Ausgabemethode abläuft, ist keine Ereignisbehandlung möglich.

  Genau genommen konkurrieren die GUI-Ausgabemethoden nicht mit den Ereignisbehandlungsmethoden, sondern es *sind* welche, ausgelöst vom Ereignis **PaintEvent**.

Einer Komponente wird vom Laufzeitsystem automatisch eine **paint()**- bzw. **paintComponent()**-Botschaft zugestellt, sobald ihre Oberfläche neu zu zeichnen ist, z.B. nach der Rückkehr eines Fensters aus der Windows-Taskleiste oder aus dem Macintosh-Dock. Alle von **java.awt.Component** abgeleiteten Klassen verfügen über eine **paint()**-Methode mit folgendem Definitionskopf:

**public void paint(Graphics** *g***)**

Das zum Zeichnen erforderliche **Graphics**-Objekt wird per Parameter geliefert.

Bei **paint()** handelt es sich um eine typische **Callback**-Methode, die vom Programm bereit gehalten und vom Laufzeitsystem aufgerufen wird.

Um die Bemalung einer Komponente zu beeinflussen, definiert man eine eigene Klasse mit geeigneter Basisklasse und überschreibt die Callback-Methode zur Grafikausgabe. Beim AWT-GUI wird die Methode **paint()** selbst überschrieben, beim komplexeren und leistungsfähigeren Swing-GUI hingegen meist die von **paint()** aufgerufene **JComponent**-Methode **paintComponent()**.

**protected void paintComponent(Graphics *g*)**

#### *14.1.1.3 Grafikausgabe im Swing-Toolkit*

Bisher wurde ein AWT-Applet als Beispiel verwendet, weil seine Einfachheit die volle Konzentration auf die grundlegende Organisation der Grafikausgabe begünstigt hat. Auch im weit wichtigeren Swing-Toolkit erfolgt die Grafikausgabe nach der oben beschriebenen Logik. Allerdings ist zu beachten:

- Während auch bei einer **JApplet**-Komponente nichts gegen die direkte Verwendung als Zeichenoberfläche einzuwenden ist, sollte man keinesfalls direkt auf eine **JFrame**-Komponente zeichnen, sondern eine **JPanel**-Komponente als Zeichenoberfläche einfügen.
- Bei einer vom **JComponent** abstammenden Klasse sollte man an Stelle der **Component**-Methode **paint()** die **JComponent**-Methode **paintComponent()** überschreiben, die von **paint()** aufgerufen wird (siehe unten).

Um das folgende Trivialprogramm


zu realisieren, leiten wir eine eigene Klasse von **JPanel** ab und überschreiben die Methode **paintComponent()**:

```
import javax.swing.*;
import java.awt.*;

class PaintSwing extends JFrame {
  public PaintSwing() {
    super("Grafik im Swing-Anwendungen");
    MyPanel pan = new MyPanel();
    getContentPane().add(pan, BorderLayout.CENTER);
    setSize(300, 125);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args) {
    new PaintSwing();
  }

  class MyPanel extends JPanel {
    public void paintComponent(Graphics g) {
      super.paintComponent(g);
      g.setFont(new Font(Font.SERIF, Font.BOLD, 14));
      g.drawOval(50, 20, 40, 40);
      g.drawString("Serif fett 14", 150, 45);
      g.setColor(Color.RED);
      g.fillRect(100, 20, 40, 40);
    }
  }
}
```

Durch die mit Swing eingeführten Neuerungen (z.B. Komponenten-Umrahmungen, wählbares *Look & Feel*, automatische Doppelpufferung gegen das Flackern bei der Bildschirmausgabe) wurden einige Erweiterungen der Grafikausgabe erforderlich. Die **paint()**-Implementation der Klasse **javax.swing.JComponent** ruft nacheinander folgende Methoden auf:

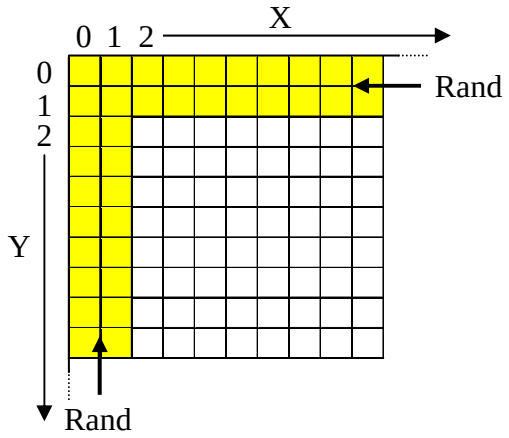
- **paintComponent()**
  Bei den meisten Swing-Komponenten wird das *Look & Feel* (das Erscheinungsbild und das Interaktionsverhalten) durch ein separates Objekt aus der Klasse **ComponentUI** implementiert, und ein **paintComponent()**–Aufruf setzt einige Aktivitäten des **ComponentUI**-Objekts in Gang. Daher sollte bei einer **paintComponent()**-Überschreibung zu Beginn die Basisklassenvariante aufgerufen werden.
- **paintBorder()**
  Zeichnet den Rahmen der Komponente.
- **paintChildren()**
  Zeichnet die enthaltenen Komponenten.

Sofern die bei Swing-Komponenten voreingestellte **Doppelpufferung** nicht abgeschaltet wurde, erhalten **paintComponent()**, **paintBorder()** und **paintChildren()** ein *Offscreen* - **Graphics**-Objekt. Erst die fertig gezeichnete Fläche gelangt auf den Bildschirm, so dass kein lästiges Flackern auftreten kann.

Für individuelle Grafikausgaben mit dem Swing-Toolkit sollte man so vorgehen:

- Eigene Klasse aus **JPanel** abgeleiten
  Zwar unterscheidet sich die Klasse **JPanel** seit der Java-Version 1.4 nur noch unwesentlich von ihrer Basisklasse **JComponent**, jedoch wird sie als Zeichnungsgrundlage in der Regel weiterhin bevorzugt. Ein **JFrame** - Top-Level - Container sollte keinesfalls bemalt werden.
- Methode **paintComponent()** überschreiben
  Wird paintComponent() überschrieben, bleibt die oben beschriebene, wesentlich von der **JComponent**-Methode **paint()** realisierte Logik der Swing-Grafikausgabe erhalten (z.B. Doppelpufferung, Zeichnen von eventuell enthaltenen Komponenten).
- Methode **super.paintComponent()** aufrufen
  Zu Beginn der **paintComponent()**-Überschreibung sollte unbedingt die Basisklassenvariante aufgerufen werden, weil dort das zur Look & Feel - Implementation eingeteilte Objekt (siehe oben) seine Aufträge erhält und z.B. bei einer Komponente mit dem **opaque**-Wert **true** den Hintergrund deckend zeichnet.
- Ggf. **paintBorder()** überschreiben
  Wer die Randzone einer Komponente individualisieren möchte, kann zusätzlich **paintBorder()** überschreiben, wobei zu Beginn wiederum die Basisklassenvariante aufgerufen werden sollte.

Beim Bemalen einer Swing-Komponente ist der ggf. vorhandene **Rand** zu berücksichtigen. So wird z.B. durch einen 2 Pixel breiten Rand der Punkt (2,2) zur linken oberen Ecke der verfügbaren Zeichenfläche, und bei Breite und Höhe gehen jeweils 4 Pixel verloren:



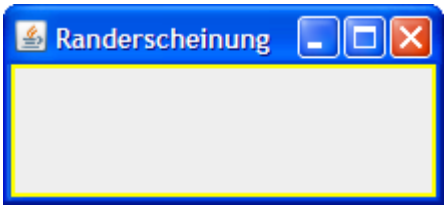
Über die aktuelle Größe der Zeichenfläche sowie der Ränder kann man sich mit den folgenden **JComponent**-Methoden informieren:

- **int getWidth()**
  Man erhält die Breite der Komponente in Pixeln.
- **int getHeight()**
  Man erhält die Höhe der Komponente in Pixeln.
- **Insets getInsets()**
  Man erhält ein **Insets**-Objekt, das mit einen vier (öffentlichen!) **int**-Feldern **left**, **right**, **top** und **bottom** über die Randbreiten informiert.

Im folgenden Codesegment wird für eine aus **JPanel** abgeleitete Komponente, die einen 2 Pixel breiten gelben Rand besitzt, mit der **Graphics**-Methode **drawRect()** versucht, die dem Rand innen liegend benachbarten Pixel rot einzufärben:

```
MyPanel pan = new MyPanel();
pan.setBorder(BorderFactory.createLineBorder(Color.YELLOW, 2));
getContentPane().add(pan, BorderLayout.CENTER);
. . .
class MyPanel extends JPanel {
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.RED);
    g.drawRect(0, 0, getWidth(), getHeight());
  }
}
```

Im ersten Versuch ist keine **drawRect()**-Ausgabe zu erkennen:

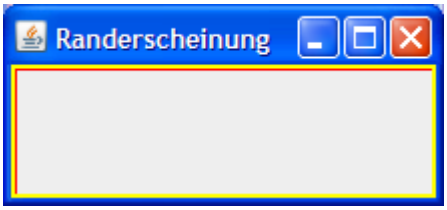



Sie wird vom ignorierten Rand überdeckt, den die *nach* **paintComponent()** aufgerufene Methode **paintBorder()** zeichnet (siehe Abschnitt 14.1.1.3).

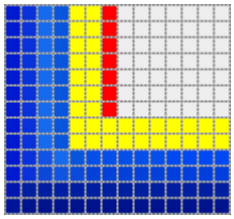
Beim folgenden **drawRect()**-Aufruf werden die Ränder (fast korrekt) berücksichtigt:

```
g.drawRect(getInsets().left, getInsets().top,
           getWidth()-getInsets().left-getInsets().right,
           getHeight()-getInsets().top-getInsets().bottom);
```

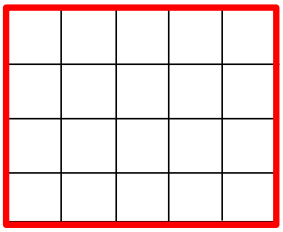
Am rechten und am unteren Rand ist allerdings die rote Linie immer noch verdeckt:



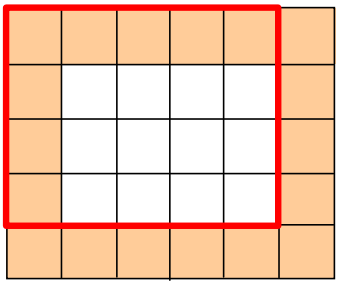

In der Vergrößerung der unteren linken Ecke ist das Problem besser zu erkennen:



Das Resultat des zweiten Versuchs ist aus dem Verhalten der **Graphics**-Objekte beim Zeichnen von Umrisslinien zu erklären. Wird z.B. per **drawRect()** ein Rechteck mit der Breite 5 und der Höhe 4 angefordert, dann legt der virtuelle Zeichenstift Strecken mit den geforderten Längen zurück



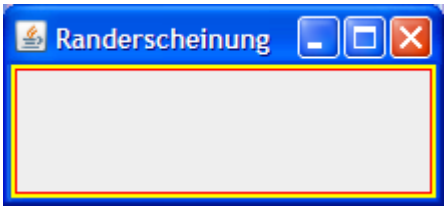
und zeichnet dabei mit einer Breite von einem Pixel *rechts neben* bzw. *unter* der Linie, so dass die folgende Figur mit einer Breite von 6 und einer Höhe von 5 Pixeln entsteht:



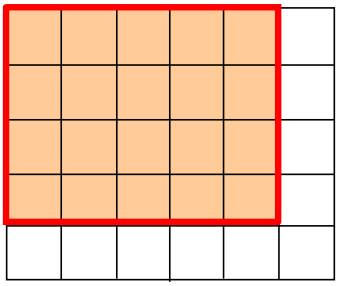
Bei Berücksichtigung dieses „Vergrößerungseffekts“:

```
g.drawRect(getInsets().left, getInsets().top,
           getWidth()-getInsets().left-getInsets().right-1,
           getHeight()-getInsets().top-getInsets().bottom-1);
```

erhält man das gewünschte Ergebnis:



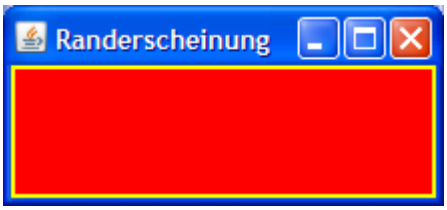

Bei *gefüllten* Figuren entstehen keine „Extrapixel“. Ein per **fillRect()** angefordertes Rechteck mit der Breite 5 und der Höhe 4 ist exakt von dieser Größe:



Der Methodenaufruf

```
g.fillRect(getInsets().left, getInsets().top,
           getWidth()-getInsets().left-getInsets().right,
           getHeight()-getInsets().top-getInsets().bottom);
```

liefert:




#### *14.1.1.4 Vergebliche Bemühungen*

Im nächsten Beispielprogramm (wiederum mit Swing-GUI) soll die Rolle der Callback-Methoden **paint()** bzw. **paintComponent()** noch einmal demonstriert werden, wobei ein Befehlsschalter ins Spiel kommt. Wie bereits erwähnt, sollte ein **JFrame**-Container generell nicht bemalt werden. Das Patentrezept für diese Lage kennen Sie schon aus dem letzten Beispiel: Für die Grafikausgaben kommt eine **JPanel**-Komponente zum Einsatz. Um individuelle Grafikausgaben in einer **paintComponent()**-Überschreibung vornehmen zu können, leiten wir eine eigene Klasse von **JPanel** ab:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Vergeblich extends JFrame implements ActionListener {
   private JButton cbEi;
   private MyPanel panEier;

   Vergeblich() {
      super("Vergeblich");
      Container cont = getContentPane();
      cbEi = new JButton("Ei legen");
      cbEi.addActionListener(this);
      cont.add(cbEi, BorderLayout.NORTH);
      panEier = new MyPanel();
      cont.add(panEier, BorderLayout.CENTER);
      setSize(300, 200);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setVisible(true);
   }

   public void actionPerformed(ActionEvent e) {
      panEier.getGraphics().drawOval(150, 50, 70, 40);
   }

   public static void main(String[] args) {
      new Vergeblich();
   }

   class MyPanel extends JPanel {
      public void paintComponent(Graphics g) {
         super.paintComponent(g);
         g.drawOval(80, 50, 40, 40);
      }
   }
}
```

Die bei jedem **PaintEvent** (d.h. bei Renovierungsbedarf) vom Laufzeitsystem aufgerufene **paintComponent()**-Methode unserer Klasse `MyPanel` malt ein Ei auf die Komponentenoberfläche, das folglich stets vorhanden ist. Die Ereignisbehandlungsmethode **actionPerformed()** zum Befehlsschalter des Beispielprogramms malt ebenfalls ein Ei, wählt aber den falschen Weg. Sie besorgt sich das **Graphics**-Objekt zur Panel-Komponente und veranlasst eine direkte Grafikausgabe. Ein Mausklick auf den Schalter hat zunächst den gewünschten Effekt, doch währt die Freude nur kurz:


Nach dem Start | Unmittelbar nach einem Mausklick auf den Schalter | Nach einer Größenänderung des Fensters

Was dauerhaft auf einer Komponente sichtbar sein soll, muss über deren **paint()**- bzw. **paintComponent()**-Methode gemalt werden.

Von der direkten Grafikausgabe ist in der Regel auch aus anderen Gründen ebenso abzuraten wie vom direkten Aufruf der **paint()**- bzw. **paintComponent()**-Methode (siehe Fowler 2003). Nun ist es aber keinesfalls ungewöhnlich, dass ein Programm oder Applet seine Oberfläche (möglichst unverzüglich) aktualisieren möchte. Im nächsten Abschnitt wird ein geeignetes Verfahren beschrieben.

#### *14.1.1.5 Programm-initiierte Aktualisierung*

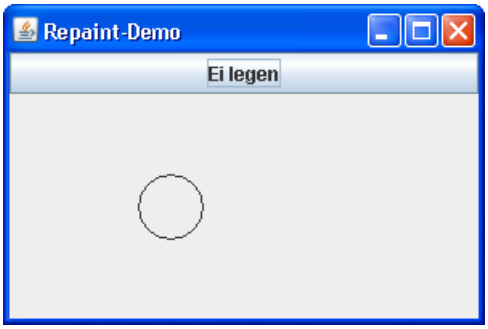
Ein Programm oder Applet kann jederzeit die Aktualisierung der Grafikanzeige veranlassen, indem es die Methode **repaint()** für die betroffene Komponente aufruft. Diese Methode stellt ein **PaintEvent** in die Ereigniswarteschlange, das baldmöglichst vom zuständigen Thread (Ausführungsfaden, siehe unten) bearbeitet wird und zum Aufruf der **paint()**- bzw. **paintComponent()**-Methode der betroffenen Komponente führt.

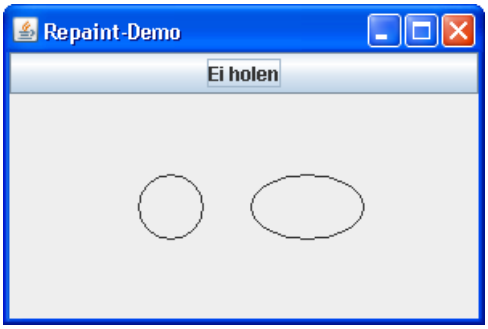
In der folgenden Variante des Eier-Programms aus dem letzten Abschnitt werden alle Eier in der **paint()**-Methode gemalt, beim rechten Ei in Abhängigkeit von der booleschen Instanzvariablen `eida`. Die Befehlsschalter-Ereignisroutine macht keine Grafikausgabe mehr, sondern verändert den Wert der Instanzvariablen `eida` und ruft dann die **repaint()**-Methode der zu bemalenden Komponente auf:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class RepaintDemo extends JFrame implements ActionListener {
  private boolean eida = false;
  private JButton cbEi;
  private MyPanel mpEier;

  RepaintDemo() {
    super("Repaint-Demo");
    Container cont = getContentPane();
    cbEi = new JButton("Ei legen");
    cbEi.addActionListener(this);
    cont.add(cbEi, BorderLayout.NORTH);
    mpEier = new MyPanel();
    cont.add(mpEier, BorderLayout.CENTER);
    setSize(300, 200);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }
```

```
    public void actionPerformed(ActionEvent e) {
      eida = !eida;
      if (eida)
        cbEi.setText("Ei holen");
      else
        cbEi.setText("Ei legen");
      mpEier.repaint();
    }

    public static void main(String[] args) {
      new RepaintDemo();
    }

    class MyPanel extends JPanel {
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawOval(80, 50, 40, 40);
        if (eida)
          g.drawOval(150, 50, 70, 40);
      }
    }
  }
```

Das Ei kann praktisch verzögerungsfrei per Mausklick gelegt und geholt werden:




Der aktuelle Zustand des Geleges wird unter allen Umständen (z.B. nach der Rückkehr aus der Taskleiste) korrekt angezeigt.

Trotz einiger Unterschiede bei der Grafikausgabe gilt für das AWT- wie für das Swing-Toolkit (vgl. Fowler 2003):

- Um die Anzeige einer Komponente zu aktualisieren, ruft man deren **repaint()**-Methode auf.
- Man sollte die **paint()**- bzw. **paintComponent()**-Methode nicht direkt aufrufen.

An Stelle der parameterfreien **repaint()**-Methode ist zur Beschleunigung der Grafikausgabe bei komplexen Komponenten eine Überladung mit Angabe des tatsächlich aktualisierungsbedürftigen Rechtecks zu bevorzugen:

**void repaint(int x, int y, int width, int height)**

Wenn allerdings eine Animation mit schnell wechselnder Oberflächenbemalung ablaufen soll, ist ausnahmsweise die direkte Grafikausgabe sinnvoll. Man ermittelt (z.B. in einer Ereignisbehandlungsmethode) bei der zu gestaltenden Komponente das zuständige Graphics-Objekt mit der Methode **getGraphics()**, um es anschließend mit Grafikausgaben zu beauftragen. Gleichzeitig ist dafür zu sorgen, dass die **paint()**- bzw. **paintComponent()**-Methode beim nächsten System-initiierten Aufruf ein aktuelles Bild zeichnet.

### 14.1.2 Elementare Methoden und Hilfsmittel bei der Grafikausgabe

Beim Einstiegsbeispiel in Abschnitt 14.1.1 wird die Klasse der individuell zu bemalenden Komponente aus **Applet** abgeleitet. In der **paint()**-Überschreibung kommen die **Graphics**-Methoden **drawOval()**, **drawString()**, **setColor()** und **fillRect()** zum Einsatz. Z.B. wird mit **drawOval()** an der Bildschirmposition (50, 50) ein Oval mit einer Höhe und Breite von 40 Pixeln gezeichnet, also ein Kreis mit diesem Durchmesser. Weil die Schriftart des Applets unverändert bleiben soll, kann sie schon in der **init()**-Methode mit **setFont()** festgelegt werden.

Sobald das Applet seine Oberfläche neu zeichnen muss, ruft das Laufzeitsystem die **paint()**-Methode auf und übergibt ein **Graphics**-Objekt. Auf eine Erläuterung der **Graphics**-Methoden zum Zeichen von elementaren Grafikelementen (Linien, Figuren) wird verzichtet, weil die aus **Graphics** abgeleitete Klasse **Graphics2D** leistungsfähigere Alternativen bereit hält (siehe Abschnitt 14.1.3).

#### *14.1.2.1 Farben*

Mit der **Graphics**-Methode **setColor()** kann die aktuelle Zeichenfarbe gesetzt werden, wobei ein Objekt der Klasse **Color** zu übergeben ist. Java verwendet ein **ARGB-Farbmodell** mit einem Alpha- (Transparenz-) Kanal sowie den Farbkanälen Rot, Grün und Blau. In der Klasse **Color** stehen u.a. die beiden folgenden Konstruktoren zur Verfügung:

- **public Color(int** *red***, int** *green***, int** *blue***, int** *alpha***)**
  Für alle vier Kanäle stehen die Ausprägungen von 0 bis 255 zur Verfügung, wobei der Alpha-Wert 0 für Transparenz und der Alpha-Wert 255 für komplette Deckung steht, z.B.:
  ```
  g.setColor(new Color(150, 0, 150, 255));
  ```
- **public Color(float** *red***, float** *green***, float** *blue***, float** *alpha***)**
  Für alle vier Kanäle stehen die Ausprägungen von 0,0 bis 1,0 zur Verfügung, wobei der Alpha-Wert 0,0 für Transparenz und der Alpha-Wert 1,0 für komplette Deckung steht, z.B.:
  ```
  g.setColor(new Color(0.7f, 0.0f, 0.7f, 0.5f));
  ```

Statt eine eigene Farbe zu mixen, kann man über Konstanten der Klasse **Color** auf 13 Standardfarben zugreifen, z.B.:[66]

```
g.setColor(Color.RED);
```

**Color**-Objekte sind ebenso unveränderlich wie **String**-Objekte.

#### *14.1.2.2 Schriftarten*

Mit der Methode **setFont()** kann man die Schriftart für spätere Textausgaben festlegen, wobei ein **Font**-Objekt zu übergeben ist, z.B.:

```
g.setFont(new Font(Font.SERIF, Font.BOLD, 14));
```

Eine Methode **setFont()** steht sowohl in der Klasse **java.awt.Component** als auch in der Klasse **java.awt.Graphics** zur Verfügung. Wir haben die Klasse **Font** schon in Abschnitt 11.7.3 verwendet und lernen jetzt noch einige Details kennen.

Im Beispiel wird der folgende **Font**-Konstruktor verwendet:

**public Font(String** *family***, int** *style***, int** *size***)**

---

[66] Bis zur Java-Version 1.3 waren abweichend von der üblichen Bezeichnungsweise die statischen und finalisierten Referenzvariablen zu den **Color**-Objekten für die Standardfarben *klein* zu schreiben (z.B. `Color.red`). Seit der Version 1.4 sind auch groß geschriebene Variablennamen vorhanden (z.B. `Color.RED`).

Im ersten Parameter ist der Name einer Schriftfamilie zu übergeben, wobei die verfügbare Auswahl vom lokalen Betriebssystem abhängt. Für die folgenden Schriftfamilien garantiert Java eine generelle Verfügbarkeit durch geeignete Abbildung auf lokal vorhandene Schriften:

- Monospaced
- SansSerif
- Serif
- Dialog
- DislogInput

Um Tippfehler im **String**-Parameter für die Schriftfamilie zu vermeiden, sollte man für die Namen der Standardschriften Konstanten der Klasse **Font** verwenden (siehe obiges Beispiel).

Auch zur Spezifikation der Schriftauszeichnung im zweiten Parameter bieten sich Konstanten der Klasse **Font** an. Hier wird eine schnörkellose, ***fett-kursive*** Schrift durch additive Kombination von zwei Stilkonstanten angefordert:

```
g.setFont(new Font(Font.SANS_SERIF, Font.BOLD+Font.ITALIC, 16));
```

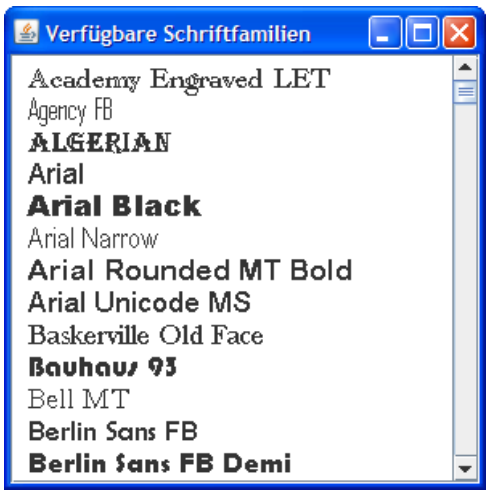
Im letzten Konstruktor-Parameter wird die Schriftgröße in Punkten zu 1/72 Zoll angegeben.

**Font**-Objekte sind ebenso unveränderlich wie **String**- oder **Color**-Objekte.

An eine Liste der lokal vorhandenen Schriftartfamilien gelangt man über ein Objekt der Klasse **GraphicsEnvironment**, das Informationen über verfügbare Schriften und Grafikausgabegeräte (Bildschirm, Drucker) bereit hält. Die **paintComponent()**-Überladung der folgenden **JPanel**-Ableitung schreibt alle Familiennamen mit der **Graphics**-Methode **drawString()** auf die Komponentenoberfläche:

```
private class FontPanel extends JPanel {
 public void paintComponent(Graphics g) {
  super.paintComponent(g);
  GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
  String[] fontFamilies = ge.getAvailableFontFamilyNames();
  int i = 0;
  for (String s : fontFamilies) {
  i++;
  Font font = new Font(s, Font.PLAIN, 18);
  g.setFont(font);
  g.drawString(s, 10, i*20);
  }
  setPreferredSize(new Dimension(100, i*20));
 }
}
```

Für das folgende Beispielprogramm



wurde ein Objekt der eben definierten Klasse `FontPanel` durch eine **JScrollPane**-Hülle (vgl. Abschnitt 11.7.6) mit Rollbalken ausgestattet und dann auf dem **BorderLayout**-Zentrum der **JFrame**-Inhaltsschicht untergebracht:

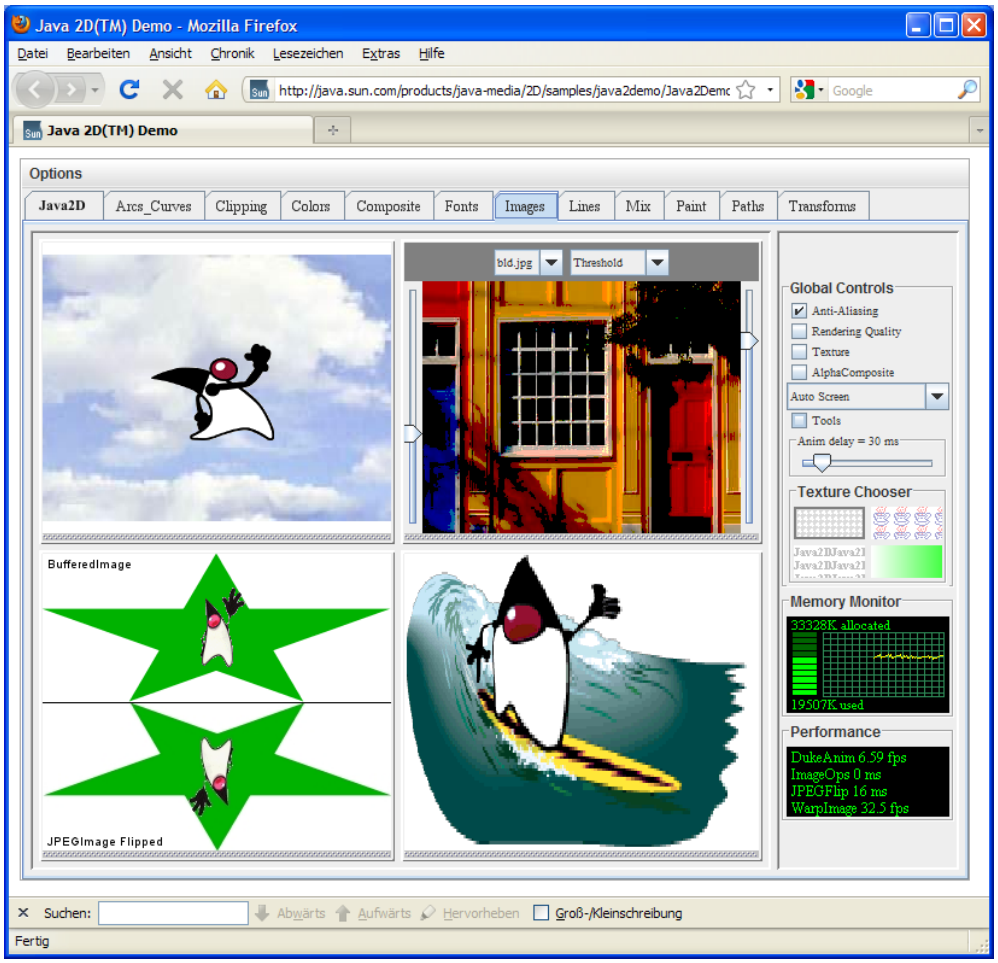
```
getContentPane().add(new JScrollPane(pan), BorderLayout.CENTER);
```

### 14.1.3 Das Java 2D API

Für anspruchsvolle Grafikausgaben sollten die Klassen und Methoden aus dem **Java 2D API** verwendet werden. Auf der Webseite

http://java.sun.com/products/java-media/2D/samples/java2demo/Java2Demo.html

bietet die Firma Sun Microsystems ein Applet an, das die Java2D-Optionen sehr beeindruckend demonstriert, z.B.:




Anschließend können aus Zeitgründen nur wenige Optionen angesprochen werden. Eine ausführlichere Darstellung bietet das Java-Tutorial (Sun Microsystems 2009).

#### *14.1.3.1 Die Klasse Graphics2D*

Etliche Nachteile der **Graphics**-Ausgabemethoden sind im Java2D-API überwunden, z.B.:

- Feste Linienstärke Eins
- Beschränkung auf monochrom gefüllte Flächen
- Beschränkung auf Gerätekoordinaten (Pixel), Auflösungsabhängigkeit

Das einer **paintComponent()**-Methode beim Aufruf übergebene **Graphics**-Objekt stammt eigentlich aus der abgeleiteten Klasse **Graphics2D**[67], und genau ein solches Objekt wird benötigt, um die Java2D-Optionen nutzen zu können. Wir müssen lediglich mit der übergebenen Referenz eine Typwandlung vornehmen, z.B.:

```
Graphics2D g2 = (Graphics2D) g;
```

#### *14.1.3.2 Die Schnittstellen Paint und Stroke*

Während ein **Graphics**-Objekt Flächen nur monochrom füllen kann und zur Farbwahl die Methode **setColor()** anbietet, lassen sich nun über die Methode **setPaint()** diverse Färbemittel wählen (z.B. Farbverläufe, Texturen), wobei der Methode ein Objekt vom Typ einer Klasse zu übergeben ist, die das Interface **Paint** erfüllt, z.B.:

```
g2.setPaint(new GradientPaint(0,0,Color.RED, 100, 50,Color.ORANGE, true));
```

Ebenso flexibel ist die Wahl eines Zeichenwerkzeugs für Umrisslinien mit der **Graphics2D**-Methode **setStroke()**, der ein Objekt vom Typ einer Klasse zu übergeben ist, die das Interface **Stroke** erfüllt, z.B.:

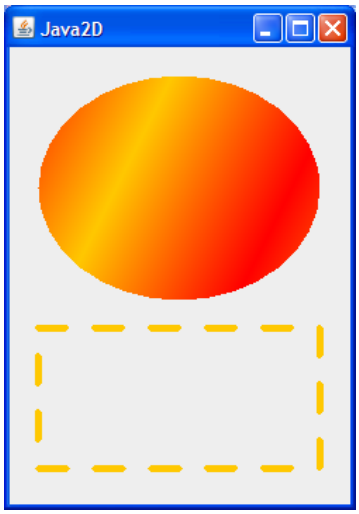
```
g2.setStroke(new BasicStroke(5, BasicStroke.CAP_ROUND,
  BasicStroke.JOIN_ROUND, 1, dash, 0));
```

Im folgenden Programm

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;

class SwingGraphics2D extends JFrame {
  public SwingGraphics2D() {
    super("Java2D in Swing");
    MyPanel pan = new MyPanel();
    getContentPane().add(pan, BorderLayout.CENTER);
    setSize(250, 360);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args) {
    new SwingGraphics2D();
  }

  class MyPanel extends JPanel {
    public void paintComponent(Graphics g) {
      super.paintComponent(g);

      Graphics2D g2 = (Graphics2D) g;
      g2.setPaint(new GradientPaint(0, 0, Color.RED,
        100, 50,Color.ORANGE, true));
      g2.fill(new Ellipse2D.Double(20, 20, 200, 160));

      g2.setPaint(Color.ORANGE);
      float[] dash = {20};
      g2.setStroke(new BasicStroke(5, BasicStroke.CAP_ROUND,
        BasicStroke.JOIN_ROUND, 1, dash, 0));
      g2.draw(new Rectangle2D.Double(20, 200, 200, 100));
    }
  }
}
```

[67] Wie die Klasse **Graphics** ist auch die Klasse **Graphics2D** abstrakt und wird plattformspezifisch konkretisiert.

wird ein **Graphcics2D**-Objekt über die Methode **fill()** beauftragt, eine Ellipse mit dem oben definierten Farbverlauf zu füllen. Außerdem entsteht über die Methode **draw()** ein Rechteck mit dem oben definierten Linientyp:




#### *14.1.3.3 Transformationen*

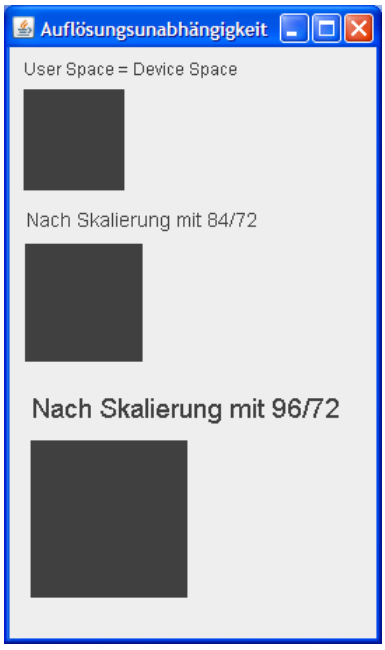
Im Java2D-API verwendet der Programmierer logische Koordinaten mit einer Einheit von 1/72 Zoll (*User Space*), die durch eine Transformation in die in Gerätekoordinaten eines Bildschirms oder Druckers übersetzt werden (*Device Space*). Per Voreinstellung ist allerdings die identische Transformation eingestellt, und die im Java2D-API versprochene Auflösungsunabhängigkeit stellt sich nicht automatisch ein. Ist die Auflösung (Größe eines Pixels) bekannt, kann eine Skalierungstransformation vorgenommen werden, so dass die logischen Koordinaten korrekt in Pixel umgerechnet werden. Mit dem folgenden Methodenaufruf bringt man die Vermutung des Betriebssystems über die Auflösung in Erfahrung:

```
int dpi = Toolkit.getDefaultToolkit().getScreenResolution();
```

Der folgende **scale()**-Aufruf nimmt eine Transformation in Betrieb, welche die logischen Koordinaten (Einheit von 1/72 Zoll) in Gerätekoordinaten umsetzt:

```
g2.scale(dpi/72.0 , dpi/72.0);
```

Bei einer vermuteten Auflösung > 72 dpi führt die Transformation zu einer Vergrößerung, z.B.:



Moderne Bildschirme arbeiten meist mit einer Auflösung oberhalb von 72 dpi, und aktuelle Windows-Versionen unterstellen 96 dpi. Liegt die Unterstellung des Betriebssystems zu hoch, schießt die zur Normalisierung gedachte Transformation über das Ziel hinaus. Bei einem LCD-Display mit einer tatsächlichen Pixelgröße von 0,303mm und einer resultierenden Auflösung von ca. 84 dpi wird ein Quadrat mit der logischen Kantenlänge 72 (= ein Zoll = 2,54 cm) mit einer Breite von 2,9 cm angezeigt. Erst die korrekte Skalierung

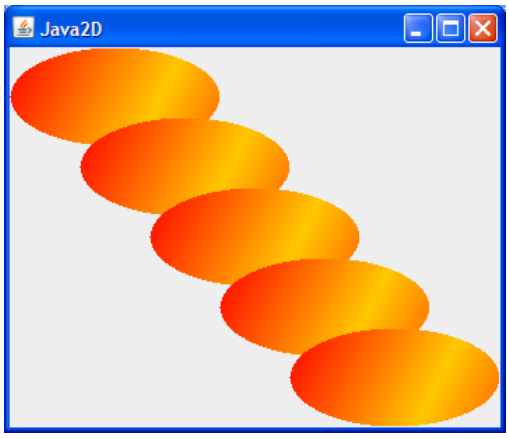
```
g2.scale(84/72.0 , 84/72.0);
```

sorgt auf dem Bildschirm für die erwartete Kantenlänge.

Neben der Auflösungsanpassung gibt es weitere Gründe für eine Transformation beim Übergang von logischen Koordinaten in Gerätekoordinaten, und ein **Graphics2D**-Objekt beherrscht neben **scale()** auch noch die Methoden **translate()** und **rotate()**, die eine Verschiebung bzw. Rotation bewirken. Im folgenden Beispiel wird eine Ellipse wiederholt gezeichnet bei zwischenzeitlicher Verschiebung des Ursprungs:

```
for (int i = 0; i < 5; i++) {
  g2.translate(50, 50);
  g2.fill(new Ellipse2D.Double(0, 0, 150, 70));
}
```

Das Ergebnis:




### *14.1.3.4 Verbesserte Schriftausgabe*

Ein **Graphics2D**-Grafikkontext erlaubt über ein Objekt der Klasse **RenderingHints** u.a. das Aktivieren der **Kantenglättung** (engl. *Antialiasing*[68]) für die Textausgabe:

```
RenderingHints rh = new RenderingHints(
     RenderingHints.KEY_TEXT_ANTIALIASING,
     RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
g2.setRenderingHints(rh);
```

Von den beiden folgenden Schriftproben ist die linke geglättet:

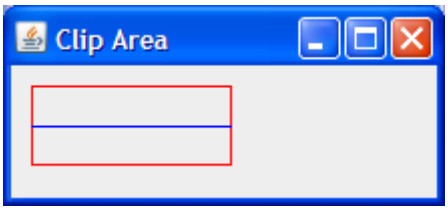

[68] Vermutlich stammt der Begriff *Antialiasing* aus der Signalverarbeitung, wo eine Signalverfälschung durch eine zu geringe Abtastfrequenz als *Alias-Effekt* bezeichnet wird. Analog kann z.B. eine Treppe nur als Ersatz (Alias) für eine schräge Gerade gelten. Versuche, die Abweichung vom Original zu mildern, bezeichnet man als *Antialiasing*.

### 14.1.4 Clip-Bereiche

Bisher haben wir die gesamte Zeichenfläche einer Komponente zur Grafikausgabe genutzt. Über die **Graphics**-Methode **SetClip()** lässt sich ein eingeschränkter Bereich (engl. *clip area*) definieren, auf den die Grafikausgabe anschließend beschränkt bleiben soll. Neben *rechteckigen* Clip-Bereichen, auf die wir uns beschränken wollen, sind mit Hilfe von **Shape**-Objekten auch beliebig geformte Clip-Bereiche möglich. Im folgenden Codesegment wird ein Ausgabebereich per **setClip()** definiert und per **drawRect()** markiert (rotes Rechteck):

```
pg.setClip(10,10, 100, 40);
pg.setColor(Color.RED);
pg.drawRect(10,10,100-1,40-1);
pg.setColor(Color.BLUE);
pg.drawLine(0, 30, pan.getWidth(), 30);
```

Die mit **drawLine()** gezeichnete Linie ist nur innerhalb des Clip-Bereichs zu sehen:




### 14.1.5 Rastergrafik

Mit der Klasse **BufferedImage** (Paket **java.awt.image**) unterstützt das Java 2D API die Darstellung und Bearbeitung von Rastergrafiken, die z.B. mit Hilfe der Klasse **ImageIO** (Paket **javax.imageio**) aus einer Datei importiert werden können:

```
private BufferedImage biLand;
  . . .
try {
  biLand = ImageIO.read(new File("land.jpg"));
} catch (IOException e) {}
```

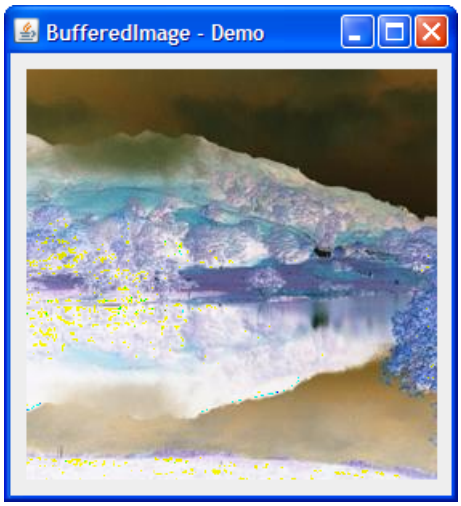
Die folgende **paintComponent()**-Überladung der Klasse `ImagePanel` zeichnet das Bild im **BufferedImage**-Objekt `bi` (Größe: 256 × 256 Pixel) zweimal mit verschiedenen Überladungen der **Graphics**-Methode **drawImage()**:

```
private class ImagePanel extends JPanel {
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawImage(biLand, 10, 10, null);
    g.drawImage(biLand, 276, 10, 676, 266, 0, 0, 256, 256, null);
  }
}
```

Zunächst werden die Bildkoordinaten unverändert als Benutzerkoordinaten (in den *User Space*) übernommen. In der zweiten **drawImage()**-Überladung wird das Bild horizontal auf eine Breite von 400 Pixeln gedehnt:


Mit Hilfe des letzten Parameters der verwendeten **drawImage()**-Überladungen kann sich ein Programm über Veränderungen bei einem asynchron geladenen Bild informieren lassen. Weil diese Option selten relevant ist, verwendet man meist den Aktualparameterwert **null**.

Eine **drawImage()**-Überladung der Klasse **Graphics2D** bietet die Möglichkeit, **Filter** auf **BufferedImage**-Objekte anzuwenden. Hier wird ein Vorschlag aus der Java - Tutorial (Sun Microsystems 2009) umgesetzt:




Um ein **BufferedImage**-Objekt im Speicher zu *ändern*, verschafft man sich ein **Graphics(2D)**-objekt

```
private BufferedImage biEB;
   . . .
Graphics2D g2 = (Graphics2D) biEB.getGraphics();
g2.setPaint(Color.RED);
g2.setStroke(new BasicStroke(5));
g2.draw(new Rectangle2D.Double(1440, 276, 400, 350));
```

und gibt die Renovierungsarbeiten in Auftrag. Hier wird zur Hervorhebung eines wichtigen Bildbereichs ein Rechteck eingezeichnet:


Die oben bereits zum Lesen von Bilddateien verwendete Klasse **ImageIO** beherrscht auch das Schreiben, wobei als Dateiformate u.a. JPEG, PNG, GIF und BMP unterstützt werden, z.B.:

```
private BufferedImage biEB;
   . . .
try {
   ImageIO.write(biEB, "jpg", new File("EmmaBodoMod.jpg"));
} catch (IOException e) {}
```

## 14.2 Sound

In folgendem Applet wird eine Musikbox realisiert, die derzeit sieben Stücke vorspielen kann, aber beliebig ausbaufähig ist:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;

public class MusicBox extends JApplet implements ActionListener {
   private JButton cbStart = new JButton("Start");
   private JButton cdChange = new JButton("Wechseln");
   private JButton cbStop = new JButton("Stopp");
   private JLabel label = new JLabel("");
   private final static int MAXCLIP = 5;
   private int actclip = 0;
   private boolean musicOn;
   private AudioClip[] clips = new AudioClip[MAXCLIP];
   private String[] titles = new String[MAXCLIP];

   public void init() {
      cbStart.addActionListener(this);
      cdChange.addActionListener(this);
      cbStop.addActionListener(this);

      add(cbStart, BorderLayout.WEST);
      add(cdChange, BorderLayout.CENTER);
      add(cbStop, BorderLayout.EAST);
      add(label, BorderLayout.SOUTH);

      java.net.URL cb = getCodeBase();
      clips[0] = getAudioClip(cb,"chirp1.au");
      titles[0] = "Vogel-Geschwister (.au)";
      clips[1] = getAudioClip(cb,"elise.mid");
```

```
        titles[1] = "Für Elise (mid)";
        clips[2] = getAudioClip(cb,"trippygaia1.mid");
        titles[2] = "Trippygaia (mid)";
        clips[3] = getAudioClip(cb,"BoogieWoogie.mid");
        titles[3] = "Boogie Woogie Blues (mid)";
        clips[4] = getAudioClip(cb,"spacemusic.au");
        titles[4] = "Spacemusic (au)";
        label.setText(titles[0]);
    }

    public void stop() {
        clips[actclip].stop();
    }

    public void start() {
        if (musicOn)
            clips[actclip].loop();
    }

    public void destroy () {
        clips[actclip].stop();
    }

    public void actionPerformed(ActionEvent e) {
        clips[actclip].stop();
        if (e.getSource() == cbStop)
            musicOn = false;
        else {
            if (e.getSource() == cdChange) {
                if (actclip < MAXCLIP-1)
                    actclip++;
                 else
                    actclip = 0;
                label.setText(titles[actclip]);
            } else  // Source == start
                musicOn = true;
            if (musicOn)
                clips[actclip].loop();
        }
    }
}
```

Die **(J)Applet**-Methode **getAudioClip()** erstellt aus einer Audiodatei ein Objekt aus einer Klasse[69], die das Interface **AudioClip** erfüllt. Im Beispiel werden die Audiodateien mit Hilfe eines URL-Objekts im Pfad des Applets lokalisiert (vgl. Abschnitt 13.5.3):

```
java.net.URL cb = getCodeBase();
clip[0] = getAudioClip(cb,"chirp1.au");
```

Jedes **AudioClip**-Objekt repräsentiert ein Musikstück und bietet u.a. die folgenden Methoden:

- **play()**
  Einmal spielen
- **loop()**
  Endlos spielen
- **stop()**
  Wiedergabe beenden

In Bezug auf den Abschnitt 14.1 ist die Bemerkung angebracht, dass im Sound-Applet (wie in früheren GUI-Anwendungen und -Applets) keine **paint()**-Methode überschrieben werden muss. Auf der Oberfläche des Applets befinden sich nur Komponenten, und diese verstehen sich selbst zu zeichnen, wenn sie vom umgebenden Container dazu aufgefordert werden.

---

[69] Für besonders Neugierige: Es handelt sich um die Klasse **sun.applet.AppletAudioClip**.

Die Bedienungsoberfläche des Applets hat noch kein disko-taugliches Design, sieht aber zumindest auf einem Mac recht passabel aus:


Obwohl die beschrieben Sound-Technik offenbar für Applets konzipiert wurde, ist sie auch in Java-Applikationen anwendbar, z.B.:

```
import java.awt.*;
import javax.swing.*;
public class Die extends JPanel {
  private java.applet.AudioClip clip;
    . . .
  public Die() {
    java.net.URL clipURL = getClass().getResource("werfen.wav");
    clip = java.applet.Applet.newAudioClip(clipURL);
  }
  . . .
}
```

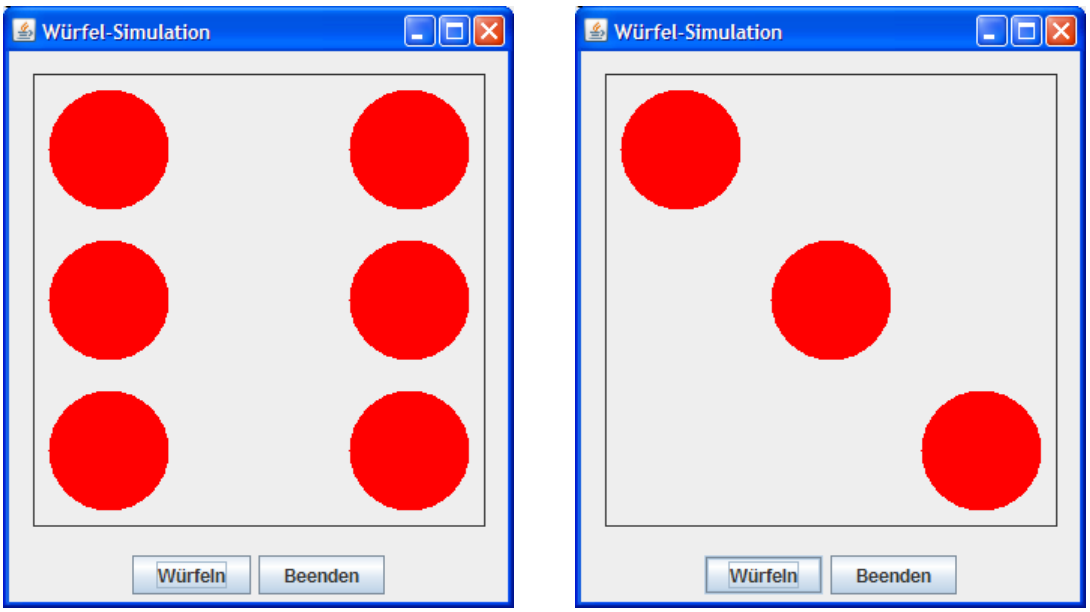
Für anspruchsvollere Aufgaben bei Wiedergabe und Aufzeichnung von Audio- und Videoströmen eignet sich das Java Media Framework (JMF), das leider seit einigen Jahren nicht mehr weiterentwickelt wird (siehe Eidenberger & Divotkey 2004).

## *14.3 Übungsaufgaben zu Kapitel 14*

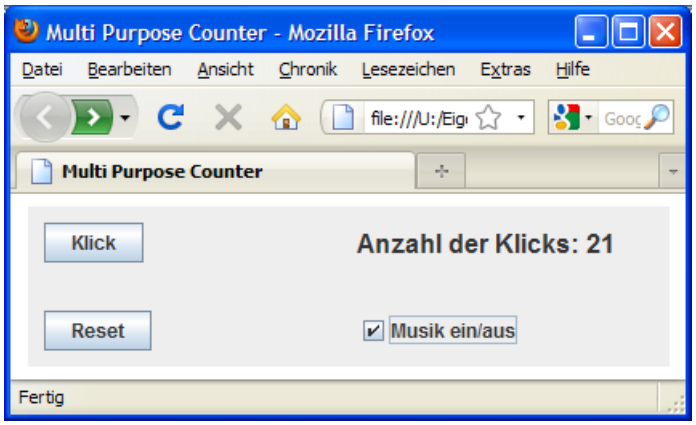
1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Der Ursprung des Java-Koordinatensystems (Punkt 0,0) befindet sich in der linken oberen Ecke der Zeichenfläche.
2. Die direkte Grafikausgabe in einer Ereignisbehandlungsmethode (außerhalb der Methode **paint()** bzw. **paintComponent()** für die System-initiierte Grafikausgabe) ist grundsätzlich unsinnig.
3. Im Java2D-API liegt allen Größen und Positionsangaben die Einheit 1/72 Zoll zugrunde.
4. Bei einem **Font**-Objekt kann man lediglich Größe und Farbe ändern, Schriftfamilie und Schriftschnitt liegen hingegen fest.

2) Erstellen Sie ein Swing-Programm, das einen virtuellen Würfel realisiert:



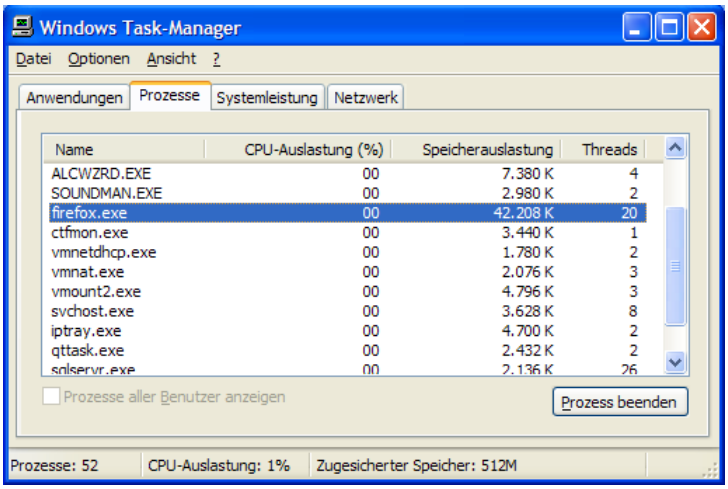

3) Erstellen Sie ein Applet, das zum Zählen von Ereignissen beliebiger Art über einen Schalter sowie über ein Label mit aktueller Anzeige des Zählerstands verfügt. Dem zuständigen Sachbearbeiter soll seine monotone Tätigkeit durch eine nach Belieben ein- und ausschaltbare Hintergrundmusik erleichtert werden, so dass sich ungefähr folgende Benutzeroberfläche ergibt:



# 15 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 mit deren Fenstern arbeiten. Wie unter Windows ein Blick in die Prozessliste mit dem Task-Manager zeigt, sind z.B. bei einer typischen Firefox-Sitzung ca. 20 Threads aktiv, wobei die Anzahl ständig schwankt, z.B.:




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) in Teilaufgaben zerlegt werden, um die CPU-Kerne auszulasten und Zeit zu sparen. Mittlerweile (2010) sind 4 Kerne guter Standard und dank Intels Hyper-Threading - Technologie sieht das Betriebssystem bei einer Quad-Core - CPU in der Regel sogar 8 Kerne. Der Klarheit halber soll betont werden:

- 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 als auf einem Single-Core - Rechner laufen
- Die Möglichkeit zum (Quasi-)parallelbetrieb mehrerer Programmfunktionen ist unabhängig von der CPU-Architektur in vielen Situationen für die Anwender sehr nützlich.
- Die zu verwendenden Techniken der Multithreading-Programmierung hängen *nicht* von der Anzahl der CPU-Kerne ab.

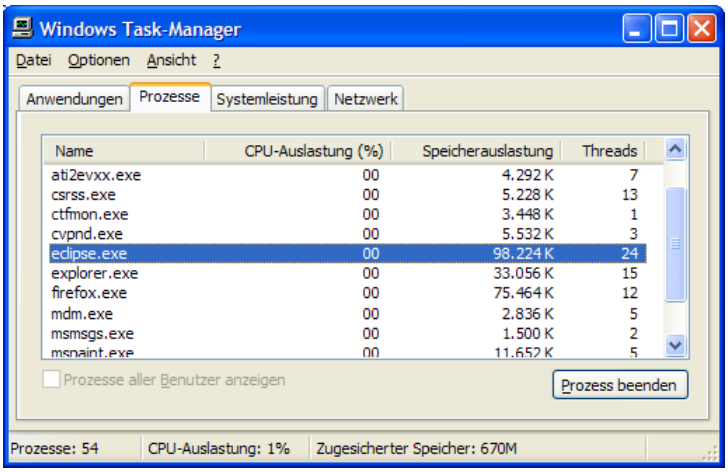
Beim Multithreading ist allerdings eine sorgfältige Einsatzplanung erforderlich, denn:

- Thread-Wechsel sind mit einem gewissen Zeitaufwand verbunden und sollten daher nicht zu häufig stattfinden.
- Das Laufzeitsystem wird durch die Verwaltung von Threads zusätzlich belastet.
- In der Regel erfordert das Synchronisieren von Threads einige Aufmerksamkeit beim Programmierer (siehe Abschnitt 15.2). Hier kann es zu Fehlern kommen, die zudem aufgrund variabler Ergebnisse 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, wohingegen jeder Thread als selbständiger Kontrollfluss bzw. Ausführungsfaden aber einen eigenen Stack-Speicher benötigt.

Vor der Einführung von Java gehörte die Unterstützung von Threads zur Guru-HighTech-Programmierung, weil man die Single-Thread-Architektur von Programmiersprachen wie C/C++ 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. Die Java-Anwendung Eclipse 3 verteilt ihre Tätigkeit auf 20-30 Threads:




## *15.1 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 in kleinen Beispielen zwar das Grundprinzip gut veranschaulichen, aber den Nutzen nicht nachweisen.

### 15.1.1 Die Klasse Thread

Ein Thread ist in Java als Objekt der gleichnamigen Klasse bzw. einer Unterklasse realisiert. Im ersten Beispiel werden die Klassen `ProThread` und `KonThread` aus der Klasse **Thread** abgeleitet. Sie sollen einen Produzenten und einen Konsumenten modellieren, die gemeinsam auf einen Lagerbestand einwirken, der von einem Objekt der Klasse `Lager` gehütet wird:

```
class Lager {
  private int bilanz;
  private int anz;
  private final int MANZ = 20;

  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()+
      " ergaenzt\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);
  }
}
```

Das folgende Hauptprogramm erzeugt ein `Lager`-Objekt mit initialem Bestand

```
class ProKonDemo {
  public static void main(String[] args) {
    Lager lager = new Lager(100);
    ProThread pt = new ProThread(lager);
    KonThread kt = new KonThread(lager);
    pt.start();
    kt.start();
  }
}
```

und generiert dann ein `ProThread`- sowie ein `KonThread`-Objekt. Weil beide Threads mit dem `Lager`-Objekt kooperieren sollen, erhalten sie als Konstruktor-Parameter eine entsprechende Referenz.

Anschließend werden die beiden Threads vom Zustand **new** durch Aufruf ihrer **start()**-Methode in den Zustand **ready** gebracht:

```
pt.start();
kt.start();
```

Von der **start()**-Methode eines Threads wird seine **run()**-Methode aufgerufen, welche die im Thread auszuführenden Anweisungen enthält. Eine aus **Thread** abgeleitete Klasse muss also die **run()**-Methode überschreiben, z.B.:

```
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 {
        sleep((int) (1000 + Math.random()*3000));
      } catch(InterruptedException ie) {}
    }
  }
}
```

Im Beispiel 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 15.5.1).

Sobald seine **run()**-Methode abgearbeitet ist, endet ein Thread. Er befindet sich dann im Zustand **terminated** und kann *nicht* erneut gestartet werden.

Im Beispiel ergänzt der `ProThread` innerhalb einer **while**-Schleife das Lager um eine zufallsbestimmte Menge. Er spricht über die per Konstruktor 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:

```
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 **Fehler! Verweisquelle konnte nicht gefunden werden.** näher zu beschreibende **Thread**-Methode **interrupt()** wird oft dazu eingesetzt, einen per **sleep()** in den Schlaf oder per **wait()** (siehe Abschnitt 15.2.2) in den Wartezustand geschickten Thread wieder zu aktivieren. Im Beispielprogramm ist allerdings keine Unterbrechungsaufforderung an einen schlafenden Thread enthalten.

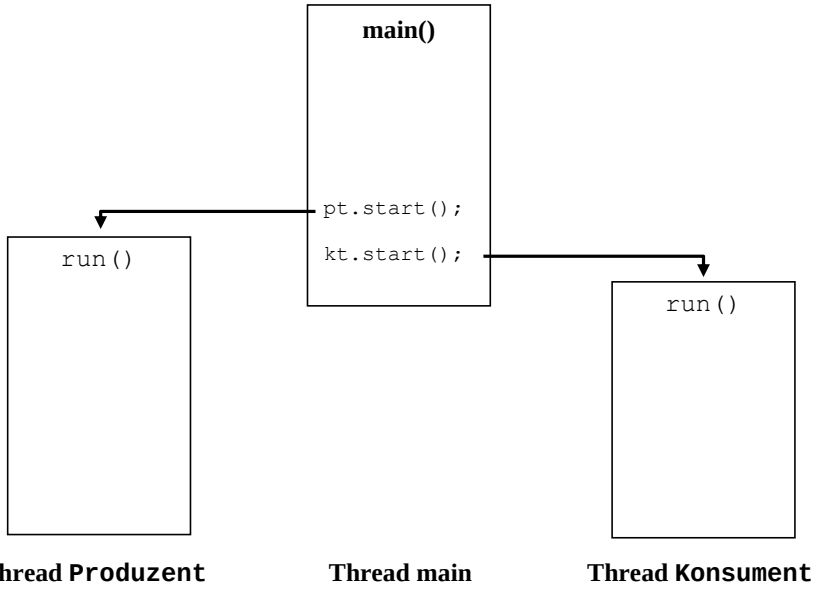
Zum `ProThread`-Konstruktor ist noch anzumerken, dass im Aufruf des Superklassen-Konstruktors ein Thread-Name festgelegt wird.

Der Konsumenten-Thread 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 {
        sleep((int) (1000 + Math.random()*3000));
      } catch(InterruptedException ie) {}
    }
  }
}
```

In beiden **run()**-Methoden wird vor jedem Schleifendurchgang geprüft, ob das Lager noch offen ist. Nach Dienstschluss des Lagers (im Beispiel: nach 20 Arbeitsgängen) enden beide **run()**-Methoden und damit auch die Threads.

Auch der automatisch zur Startmethode 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:[70]




Wenn die drei Benutzer-Threads abgeschlossen sind, endet auch das Programm.

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:

[70] 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.

- 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. Dazu besorgen Sie sich mit der statischen **Thread**-Methode **currentThread()** eine Referenz auf den aktuell ausgeführten Thread 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 der beiden Threads:

```
Der Laden ist offen (Bestand = 100)

Nr. 1:    Produzent ergaenzt      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 ergaenzt      32     um 12:43:37 Uhr. Stand: 75
Nr. 5:    Konsument entnimmt      42     um 12:43:38 Uhr. Stand: 33
Nr. 6:    Produzent ergaenzt      44     um 12:43:39 Uhr. Stand: 77
Nr. 7:    Konsument entnimmt      63     um 12:43:41 Uhr. Stand: 14
Nr. 8:    Produzent ergaenzt      42     um 12:43:42 Uhr. Stand: 56
Nr. 9:    Konsument entnimmt      99     um 12:43:43 Uhr. Stand: -43
Nr. 10:   Produzent ergaenzt      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 ergaenzt      90     um 12:43:47 Uhr. Stand: 28
Nr. 14:   Produzent ergaenzt      47     um 12:43:48 Uhr. Stand: 75
Nr. 15:   Konsument entnimmt      101    um 12:43:51 Uhr. Stand: -26
Nr. 16:   Produzent ergaenzt      42     um 12:43:51 Uhr. Stand: 16
Nr. 17:   Konsument entnimmt      79     um 12:43:52 Uhr. Stand: -63
Nr. 18:   Produzent ergaenzt      22     um 12:43:53 Uhr. Stand: -41
Nr. 19:   Konsument entnimmt      90     um 12:43:56 Uhr. Stand: -131
Nr. 20:   Produzent ergaenzt      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 Multi-Thread - Programmablauf.

In Abschnitt 15.2 werden Techniken zur Koordination bzw. Synchronisation von Threads vorgestellt, mit denen man fehlerhafte Anzeigen und Schlimmeres verhindern kann.

### 15.1.2 Das Interface Runnable

Als Basis für einen eigenständigen Kontrollfluss haben wir eben 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. Es verlangt lediglich eine parameterfreie Methode **run()** mit Rückgabetyp **void**. Diese Flexibilität ist sehr zu begrüßen, weil Java bekanntlich

keine Mehrfachvererbung unterstützt, eine Klasse aber beliebig viele Schnittstellen unterstützen kann.

Wir verwenden allerdings weiterhin das Produzenten-Konsumenten - Beispiel aus Abschnitt  und ersetzen die **Thread**-Ableitung `ProThread`

```
class ProThread extends Thread {
   . . .
}
```

durch die Klasse `Produzent`, die das Interface **Runnable** erfüllt:

```
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) {}
    }
  }
}
```

Bei dieser Vorgehensweise wird der funktionsäquivalente Umbau demonstriert. 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 entfällt die Vergabe eines Thread-Namens.
- Beim Aufruf der statischen **Thread**-Methode **sleep()** in der `Produzent`-Methode **run()** ist der Klassenname anzugeben.

Beim *Erzeugen* eines Ausführungsfadens wird aber 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.

## 15.2 Threads synchronisieren

### 15.2.1 Monitore und synchronisierte Bereiche

Am Anfang des eben wiedergegebenen Ablaufprotokolls 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 aber 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 erhöht 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 15.7.2).

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 (vgl. Abschnitt 15.5.2). 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 das Objekt beteiligt ist, welches die Klasse in der JVM repräsentiert (siehe Abschnitt 15.2.4). Solche Klassenobjekte haben lassen sich mit der **Object**-Methode **getClass()** ermitteln und dann z.B. mit **getName()** nach dem Namen der repräsentierten Klasse befragen.

In unserem Beispiel sollten die `Lager`-Methoden `offen()`, `ergaenze()` und `liefere()` als **synchronized** deklariert werden, z.B.:

```
synchronized void ergaenze(int add) {
  bilanz += add;
  anz++;
  System.out.println("Nr. "+anz+":\t"+Thread.currentThread().getName()+
    " ergaenzt\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 ergaenzt      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 ergaenzt      58     um 14:54:34 Uhr. Stand: 118
Nr. 5:    Konsument entnimmt      70     um 14:54:35 Uhr. Stand: 48
Nr. 6:    Produzent ergaenzt      13     um 14:54:35 Uhr. Stand: 61
Nr. 7:    Konsument entnimmt      74     um 14:54:38 Uhr. Stand: -13
Nr. 8:    Produzent ergaenzt      11     um 14:54:38 Uhr. Stand: -2
Nr. 9:    Konsument entnimmt      65     um 14:54:40 Uhr. Stand: -67
Nr. 10:   Produzent ergaenzt      26     um 14:54:41 Uhr. Stand: -41
Nr. 11:   Konsument entnimmt      71     um 14:54:42 Uhr. Stand: -112
Nr. 12:   Produzent ergaenzt      9      um 14:54:43 Uhr. Stand: -103
Nr. 13:   Konsument entnimmt      8      um 14:54:45 Uhr. Stand: -111
Nr. 14:   Produzent ergaenzt      100    um 14:54:46 Uhr. Stand: -11
Nr. 15:   Konsument entnimmt      5      um 14:54:47 Uhr. Stand: -16
Nr. 16:   Produzent ergaenzt      43     um 14:54:48 Uhr. Stand: 27
Nr. 17:   Konsument entnimmt      44     um 14:54:51 Uhr. Stand: -17
Nr. 18:   Produzent ergaenzt      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.

Weil bei einem Konstruktor der Modifikator **synchronized** nicht erlaubt ist, sollte man in dieser Methode keine Referenz zum entstehenden Objekt veröffentlichen, wenn interferierende Zugriffe aus anderen Threads zu befürchten sind.

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()+
      " ergaenzt\t"+add+"\tum "+formZeit()+" Uhr. Stand: "+bilanz);
  }
}
```

Nach dem Schlüsselwort **synchronized** ist in 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* zurück gegeben. Folglich ist die **sleep()**-Methode in synchronisierten Bereichen zu vermeiden.

### 15.2.2 Koordination per wait(), notify() und notifyAll()

Mit Hilfe der **Object**-Methoden **wait()** und **notify()** können negative Lagerbestände in unserem Produzenten-Lager-Konsumenten - Beispiel verhindert werden: Trifft eine Konsumenten-Anfrage auf einen unzureichenden Lagerbestand, dann wird der Thread mit der Methode **wait()** in den Zustand **waiting** versetzt (vgl. Abschnitt 15.5.2). Die Methode **wait()** kann nur in einem synchronisierten Bereich, aufgerufen werden, z.B.:

```
synchronized void liefere(int sub) {
  while (bilanz < sub)
    try {
      System.out.println(Thread.currentThread().getName()+
        " muss warten: Keine "+sub+" Einheiten vorhanden.");
      wait();
    } catch (InterruptedException ie) {
        System.err.println(ie);
    }

  bilanz -= sub;
  anz++;
  System.out.println("Nr. "+anz+":\t"+Thread.currentThread().getName()+
    " entnimmt\t"+sub+"\tum "+formZeit()+" Uhr. Stand: "+bilanz);
}
```

Dem Konsumenten-Thread wird der Monitor entzogen, so dass der Produzenten-Thread freie Bahn hat, den synchronisierten Block zu betreten und den Lagerzustand aufzubessern.

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 JRE-Implementation überlassen und erfolgt potentiell willkürlich.
- **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()+
    " ergaenzt\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 ergaenzt      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 ergaenzt      36     um 15:21:25 Uhr. Stand: 87
Konsument muss warten: Keine 92 Einheiten vorhanden.
Nr. 4:    Produzent ergaenzt      10     um 15:21:27 Uhr. Stand: 97
Nr. 5:    Konsument entnimmt      92     um 15:21:27 Uhr. Stand: 5
Nr. 6:    Produzent ergaenzt      39     um 15:21:29 Uhr. Stand: 44
Nr. 7:    Konsument entnimmt      24     um 15:21:29 Uhr. Stand: 20
Nr. 8:    Produzent ergaenzt      6      um 15:21:31 Uhr. Stand: 26
Nr. 9:    Produzent ergaenzt      30     um 15:21:32 Uhr. Stand: 56
Konsument muss warten: Keine 62 Einheiten vorhanden.
Nr. 10:   Produzent ergaenzt      66     um 15:21:35 Uhr. Stand: 122
Nr. 11:   Konsument entnimmt      62     um 15:21:35 Uhr. Stand: 60
Nr. 12:   Produzent ergaenzt      35     um 15:21:36 Uhr. Stand: 95
Konsument muss warten: Keine 97 Einheiten vorhanden.
Nr. 13:   Produzent ergaenzt      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 ergaenzt      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 ergaenzt      17     um 15:21:46 Uhr. Stand: 29
Konsument muss warten: Keine 74 Einheiten vorhanden.
Nr. 19:   Produzent ergaenzt      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 *irgend etwas* passiert 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 **while**-Schleife aufgerufen werden, deren Bedingungsteil die kritische Prüfung enthält (siehe oben).

Ein Thread kann (wie bei **sleep()**, siehe Abschnitt 15.2.2) per **InterruptedException** aus dem Wartezustand gerissen werden, weil ihm per **interrupt()** aus einem anderen Thread ein Unterbrechungssignal zugestellt wurde. Im Beispiel wartet die Methode `liefere()` nach einer Protokollausgabe im **catch**-Block unbeirrt weiter auf einen ausreichenden Lagerzustand. In anderen Fällen

kann nötig 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.

### 15.2.3 Explizite Lock-Objekte

Im bisherigen Verlauf von Kapitel 15 wurden Klassen und Methoden vorgestellt, die schon in der ersten Java-Version vorhanden waren. In der Version 5.0 (alias 1.5) wurde Java um neue Optionen zur Parallelverarbeitung erweitert, die mehr Flexibilität, teilweise aber auch mehr Verantwortung für Programmierer mit sich bringen. Man findet die neuen Klassen und Schnittstellen im Paket **java.util.concurrent** sowie in den untergeordneten Paketen.

Im Vergleich zur Synchronisation von Methoden bietet die das Interface ***Lock*** aus dem Namensraum **java.util.concurrent.locks** implementierenden Klassen (z.B. **ReentrantLock**) u.a. folgende Vorteile:

- Per Synchronisation werden Monitore (implizite Locks) blockorientiert erworben und (automatisch) beim Verlassen des Blocks zurückgegeben. Bei expliziten **Lock**-Objekten ist der Gültigkeitsbereich nicht an einen Block gebunden. Allerdings ist der Programmierer nun für die Freigabe der Locks verantwortlich (per **unlock()**-Aufruf).
- Mit der **Lock**-Methode **tryLock()** lässt sich feststellen, ob ein Lock zu haben ist. Alternativ kann man noch eine maximale Wartezeit angeben.

Über die Klasse **ReentrantReadWriteLock** ) kann man *einen* Writer-Thread, aber beliebig viele Reader-Threads zulassen und auf diese Weise unnötige Blockaden vermeiden.

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 15.2.2 beschriebenen Thread-Koordinierung.

### 15.2.4 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

```
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();
  }
}
```

sind zwei **ReentrantLock**-Objekte `lock1` und `lock2` im Spiel, und zwei Threads versuchen, 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.
```

### 15.2.5 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 in Ruhe 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) {
      return;
    }
    for (int i = 0; i < 5; i++)
      System.out.println("Thread 2, i = "+i);
  }
}
```

Wie **sleep()** und **wait()** (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**) reagiert auch **joint()** bei einer **interrupt()**-Aufforderung an den passiven Thread mit einer **InterruptedException**.

Ist der per **join()** angesprochene Thread bereits terminiert, hat der Aufruf keinen Effekt.

Nach dem Start der beiden Threads

```
class JoinDemo {
  public static void main(String[] args) {
    Thread1 t1 = new Thread1();
    Thread2 t2 = new Thread2(t1);
    t1.start();
    t2.start();
  }
}
```

arbeiten sie nacheinander:

```
Thread 1, i = 0
Thread 1, i = 1
Thread 1, i = 2
Thread 1, i = 3
Thread 1, i = 4
Thread 2, i = 0
Thread 2, i = 1
Thread 2, i = 2
Thread 2, i = 3
Thread 2, i = 4
```

Ohne Koordination per **join()** resultiert eine schlecht vorhersehbare Sequenz:

```
Thread 1, i = 0
Thread 1, i = 1
Thread 2, i = 0
Thread 2, i = 1
Thread 2, i = 2
Thread 2, i = 3
Thread 2, i = 4
Thread 1, i = 2
Thread 1, i = 3
Thread 1, i = 4
```

In einer alternativen **join()**-Überladung kann man die maximale Wartezeit in Millisekunden angeben, z.B.

```
t1.join(5000);
```

## *15.3 Threads und Swing*

### 15.3.1 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-Methode *nicht* synchronisiert sind. Man hat auf eine Synchronisation verzichtet, weil Performanzprobleme bis hin zur blockierten Oberfläche auftreten könnten.

Daher muss (bis auf einige Ausnahmen, siehe unten) der Zugriff auf die Swing-Komponenten dem bei GUI–Anwendungen bzw. –Applets 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[71]), weil sonst die Benutzeroberfläche 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 15.3.3 und 15.3.4).

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.

Analoge Regeln gelten übrigens auch für andere GUI-Frameworks, z.B. für das zur Windows-Programmierung mit C# häufig verwendete WinForms-Framework (siehe z.B. Baltes-Götz 2009).

---

[71] Diese Empfehlung stammt von der Webseite:
http://java.sun.com/developer/technicalArticles/javase/swingworker/index.html#EDT

### 15.3.2 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 (nämlich im Thread main), kann es zu Multithreading-Problemen wie Deadlocks kommen.

Statt der bisher im Skript benutzten Swing-Initialisierung

```
import javax.swing.*;
import java.awt.*;

class SwingNotThreadSafe extends JFrame {
   SwingNotThreadSafe() {
      super("Swing-Init. not Thread-Safe");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      JLabel lblText = new JLabel("Swing-Init. not Thread-Safe");
      lblText.setHorizontalAlignment(SwingConstants.CENTER);
      getContentPane().add(lblText, BorderLayout.CENTER);
      setSize(290, 100);
      setVisible(true);
   }

   public static void main(String args[]) {
      new SwingNotThreadSafe();
   }
}
```

muss streng genommen über ein **Runnable**-Objekt dafür gesorgt werden, dass die Methode **setVisible(true)** im EDT abläuft, z.B.:

```
import javax.swing.*;
import java.awt.*;

class SwingIniThreadSafe extends JFrame {
   SwingIniThreadSafe() {
      super("Swing-Init. Thread-Safe");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      JLabel lblText = new JLabel("Swing-Init. Thread-Safe");
      lblText.setHorizontalAlignment(SwingConstants.CENTER);
      getContentPane().add(lblText, BorderLayout.CENTER);
      setSize(290, 100);
      setVisible(true);
   }

   public static void main(String args[]) {
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            new SwingIniThreadSafe();
         }
      });
   }
}
```

Über die Methode **invokeLater()** der Klasse **SwingUtilities** wird dafür gesorgt, dass die **run()**-Methode der anonymen Klasse und damit der Fensterkonstruktor baldmöglichst im EDT abläuft. Bei dieser Technik sind nur minimale Änderungen gegenüber der bisherigen Vorgehensweise erforderlich. Wenn der Fensterkonstruktor Parameter benötigt, muss aber etwas mehr Aufwand betrieben werden, weil das Interface **Runnable** einen parameterlose **run()**-Definitionskopf vorschreibt.

Eine analoge Programmiertechnik verwendet übrigens auch der im Eclipse-Plugin *Visual Editor* enthaltene Quellcode-Generator. Der in Abschnitt 4.8 beschriebene visuelle Entwurf einer Swing-Anwendung zum Kürzen von Brüchen führt zu der folgenden **main()**-Methode:

```
public static void main(String[] args) {
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      BK thisClass = new BK();
      thisClass.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      thisClass.setVisible(true);
    }
  });
}
```

Nach Muller & Walrath (2000) ist die oben skizzierte Konfliktkonstellation sehr unwahrscheinlich, und wir haben bisher in bester Gesellschaft (z.B. mit Deitel & Deitel 2005, Krüger & Stark 2007, Mössenböck 2005, Ullenboom 2009) ungestraft auf die perfekt Thread-Sicherheit unsere Swing-Programme verzichtet. In Zukunft werden wir diese Nachlässigkeit aber abstellen. Wir haben nun genügend Wissen zur Verfügung, um die systemseitig im Einsatz befindlichen Threads (z.B. **main**, EDT) zu verstehen, und um aufwändige Algorithmen in eigene Benutzer- bzw. Hintergrund-Threads auszulagern.

Das Java-Tutorial (SUN Microsystems 2009) empfiehlt auch für Swing-Applets analoge Initialisierungstechniken. Allerdings ist nach Muller & Walrath (2000) die GUI-Gestaltung in der **init()**-Methode (vgl. Abschnitt 13.4) unproblematisch, obwohl diese *nicht* im EDT ausgeführt wird. Weil ein Applet nicht vor Rückkehr der **init()**-Methode angezeigt und vom EDT mit Nachrichten beschickt wird, kann es nicht zu Konflikten kommen.

### 15.3.3 Swing-Komponenten aus Hintergrund-Threads modifizieren

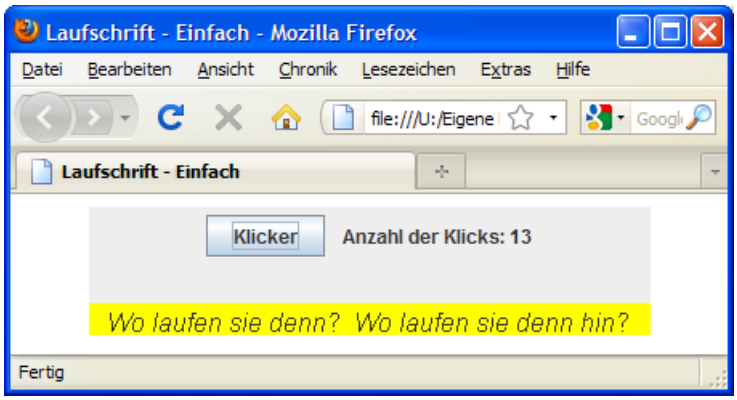
Das Swing-API bietet etliche Möglichkeiten, Änderungen von Komponenten auf sichere Weise aus beliebigen Threads zu veranlassen:

- **Thread-sichere Methoden**
  Als Ausnahme von der generellen Regel finden sich bei einigen Swing-Komponenten doch Thread-sichere Methoden, z.B.: **JTextComponent.setText()**.
- **repaint()**
  Die angesprochene GUI-Komponente wird gebeten, baldmöglichst neu zu zeichnen.
- **invokeLater()**, **invokeAndWait()**
  Mit diesen statischen **SwingUtilities**-Methoden 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 0 war schon die Thread-sichere Swing-Initialisierung über die Methode **invokeLater()** zu sehen.
- **Ereignisempfängerlisten ändern**

Im folgenden Beispiel implementiert eine von **JApplet** abstammende Klasse das Interface **Runnable**:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Laufschrift extends JApplet implements Runnable{
 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;  // akt. Schreibpos, Textbreite
 private final String LABELPREFIX = "  Anzahl der Klicks: ";
 private JLabel lab;
 private int numClicks = 0;

 public void init() {
  lab = new JLabel(LABELPREFIX + "0    ");
  JButton butt = new JButton("Klicker");
  JPanel pan = new JPanel();
  pan.add(butt); pan.add(lab);
  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 = getFontMetrics(gp.getFont()).stringWidth(TXT);
  gp.setPreferredSize(new Dimension(0, 20)); // Höhe Laufschriftband
  add(gp, BorderLayout.SOUTH);

  (new Thread(this)).start();
 }

 public void run() {
  while (true) {
   try {
    Thread.sleep(PAUSE);
   } catch(InterruptedException ie) {}
   if (x-SPRUNG + tb < 0)
    x = getSize().width;
   else
    x -= SPRUNG;
   gp.repaint();
  }
 }

 private class GraphicsPanel extends JPanel {
  public void paintComponent(Graphics g){
   super.paintComponent(g);
   g.drawString(TXT, x, 18);
  }
 }
}
```

Während ein Hintergrund-Thread damit beschäftigt ist, einen Text kontinuierlich über die Applet-Fläche laufen zu lassen, interagiert der Ereignisverteilungs-Thread mit dem Benutzer, der z.B. vorbeifahrende Autos per Mausklick zählen kann:




Bei der Kommunikation zwischen den Threads kommt die **repaint()**-Methode zum Einsatz:

- Der Hintergrund-Thread erzeugt bzw. überarbeitet relevante Daten für das GUI (er erneuert das Modell). Im Beispiel wird die x-Koordinate der Schrift-Ausgabeposition neu berechnet.
- Dann wird die betroffene GUI-Komponente (als Ansicht bzw. View) durch einen Aufruf ihrer **repaint()**-Methode gebeten, die Oberfläche zu erneuern, was sie unter Berücksichtigung des aktuellen Modells tut.

Ein Swing-Initialisierungsproblem im Sinn von Abschnitt 15.3.1 ist nach Muller & Walrath (2000) bei Applets *nicht* zu befürchten, wenn

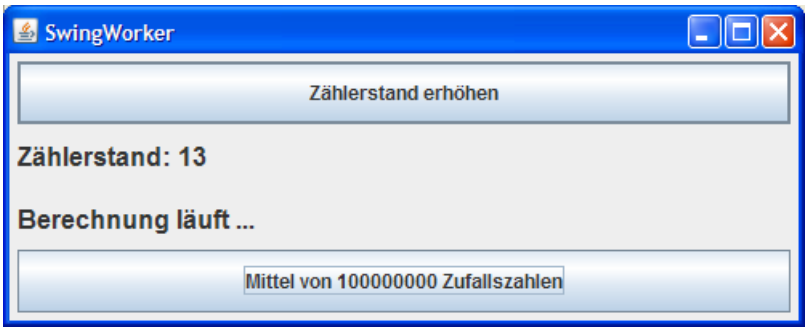
- die GUI-Gestaltung in der **init()**-Methode stattfindet,
- beim Applet auf die (hier überflüssigen) Methoden **show()** und **setVisible()** verzichtet wird.

### 15.3.4 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 perfekt 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> {
  private double d;

  public Double doInBackground() {
    for (int i = 0; i < ANZ; i++)
      d += Math.random();
    return new Double(d/ANZ);
  }

  public void done() {
    try {
      lblRandom.setText("Mitterwert der Zufallszahlen: "+get());
    } catch (Exception e) {
      JOptionPane.showMessageDialog(null, "Berechnung gescheitert");
    }
  }
}
```

Die im Hintergrund-Thread auszuführende Methode **doInBackground()** berechnet die monströse Zahl und liefert sie 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:[72]

```
cbWorker = new JButton("Mittel von "+ANZ+" Zufallszahlen");
cbWorker.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    lblRandom.setText("Berechnung läuft ...");
    RandomNumberCruncher worker = new RandomNumberCruncher();
    worker.execute();
  }
});
```

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.

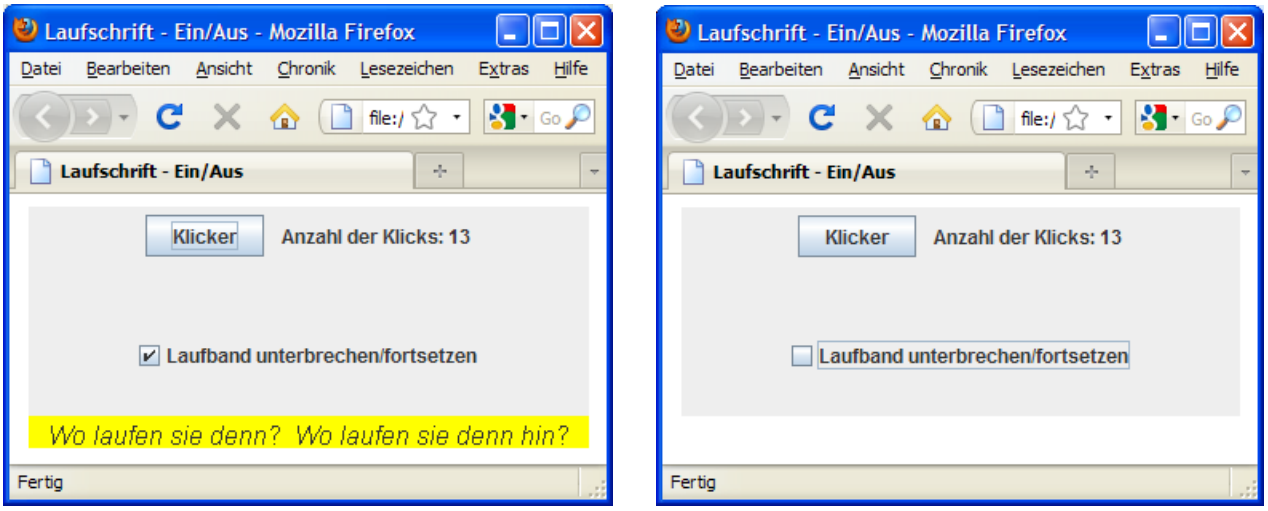
## *15.4 Andere Threads unterbrechen, fortsetzen oder beenden*

Zum Unterbrechen und Reaktivieren eines anderen Threads sollten die früher vorgesehenen **Thread**-Methoden **suspend()** und **resume()** *nicht* mehr verwendet werden. Ein **suspend()**-Aufruf kann leicht zu einem Deadlock (deutsch: zu einer *Systemverklemmung*) führen, weil ein suspendierter Thread die in seinem Besitz befindlichen Monitore behält. Statt dessen wird eine Inter-Thread - Kommunikation über die **Object**-Methoden **wait()** und **notify()** empfohlen, die wir schon in Abschnitt 15.2.2 kennen gelernt haben. Dabei wird dem Ziel-Thread ggf. über eine von ihm regelmäßig zu beobachtende Variable signalisiert, dass er sich durch einen **wait()**-Aufruf in den Zustand **waiting for monitor** begeben sollte. In seinem Besitz befindliche Monitore werden dabei automatisch zurück gegeben, so dass kein Deadlock droht.

---

[72] Mit der 64-Bit - Version der JRE 6.0 Update 18 unter Windows 7 (64 Bit) macht der **SwingWorker<Double, Void>** Probleme ab dem dritten Aufruf, während mit der 32-Bit JRE unter demselben Betriebssystem keine Fehler auftreten.

In folgender Variante `LaufschriftEinAus`des Swing-Applets aus Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.** wird ein Kontrollkästchen angeboten, um aus dem Ereignisverteilungs-Thread der Bedienoberfläche (siehe Abschnitt 15.3) den Laufschrift-Thread zu unterbrechen bzw. fortzusetzen:




In der booleschen Instanzvariablen `lbAnAus` des Applets wird der aktuelle Schaltzustand des Kontrollkästchens gespeichert. Wird in der **run()**-Methode des Laufschrift-Threads der `lbAnAus`-Wert **false** angetroffen, dann begibt sich der Thread per **wait()**-Aufruf freiwillig in den Wartezustand.

Wie Sie aus Abschnitt 15.2.2 bereits wissen, darf die Methode **wait()** nur in einem synchronisierten Bereich aufgerufen werden. Die **run()**-Methode eines Threads als **synchronized** zu deklarieren, kann nicht sinnvoll sein. Weil diese Methode bis zum **Thread**-Ende aktiv bleibt, wäre der synchronisierte Block in dieser Zeit besetzt. Als Alternative zum **synchronized**-Modifikator für Methoden steht in Java der **synchronized**-Block zur Verfügung, wobei statt einer kompletten Methode nur eine einzelne Blockanweisung in den synchronisierten Bereich aufgenommen wird, z.B.:

```
public void run() {
  while (true) {
    try {
      Thread.sleep(PAUSE);
      synchronized(this) {
        if (!lban)
          wait();
      }
    } catch(InterruptedException ie) {}
    if (x-SPRUNG + tb < 0)
      x = getSize().width;
    else
      x -= SPRUNG;
    gp.repaint();
  }
}
```

Nach dem Schlüsselwort **synchronized** wird in runden Klammern ein geeignetes Objekt als Monitor (Lock-Objekt) für die Synchronisation angegeben. Im Beispiel übernimmt das **JApplet**-Objekt diese Rolle selbst.

In einer per **sleep()**-Methode ausgelösten Ruhephase werden im Besitz eines Threads befindliche Monitore *nicht* zurück gegeben. Folglich ist die **sleep()**-Methode in synchronisierten Bereichen zu vermeiden.

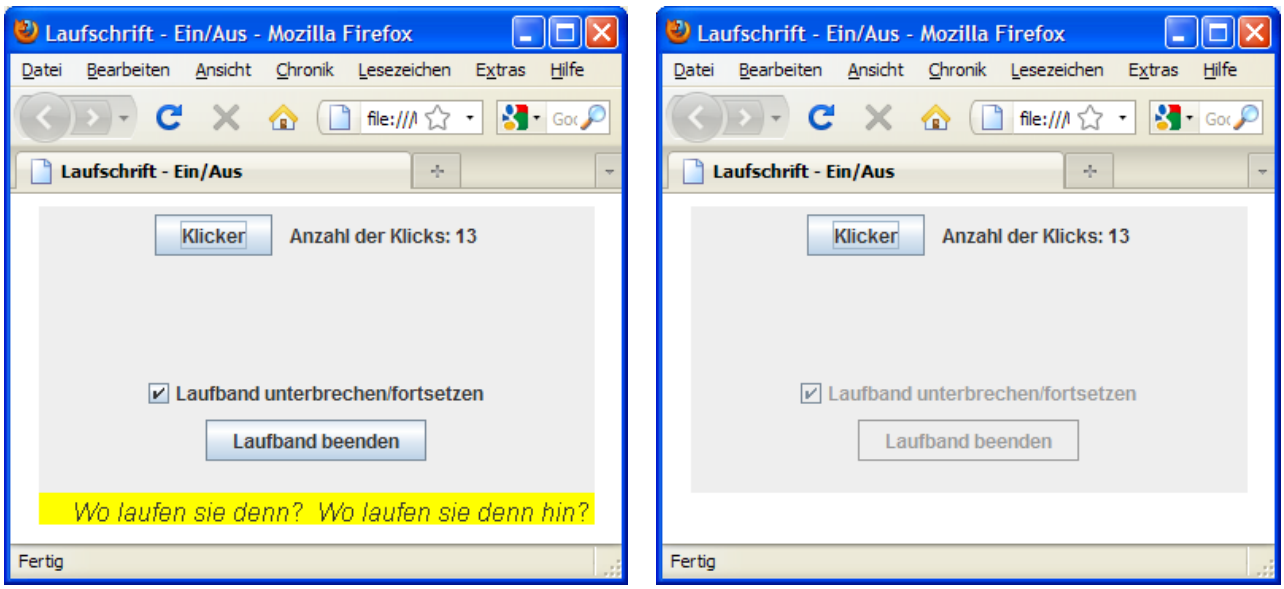
Um den wartenden Laufschrift-Thread aus dem Ereignisverteilungs-Thread zu reaktivieren, wird in der Ereignisbehandlungsmethode **itemStateChanged()** zum Kontrollkästchen die **notify()**-Anweisung ausgeführt. Diese Anweisung muss zum selben synchronisierten Bereich gehören wie **wait()**-Anweisung, so dass wiederum das **JApplet**-Objekt zur Synchronisation zu verwenden ist. Dabei sollte ein möglichst kleiner **synchronized**-Block gebildet werden, um andere Monitor-Anwärter nicht unnötig zu behindern, z.B.:

```
public synchronized void itemStateChanged(ItemEvent e) {
   lban = !lban;
   if (lban)
      synchronized(this) {notify();}
   else
      repaint();
}
```

Zum **Stoppen** eines anderen Threads sollte man die unerwünschte (herabgestufte) **Thread**-Methode **stop()** nicht mehr verwenden, weil sie aufgrund einer konzeptionellen Schwäche zu Stabilitätsproblemen führen kann: Ein gestoppter Thread verliert seine Monitor-Besitzrechte, so dass unter seinem Schutz befindliche Objekte beschädigt werden können. Statt der rabiaten **stop()**-Methode sollte ein auf freundliche Kooperation angelegtes Verfahren benutzt werden:

- 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 schlafenden oder wartenden Thread führt der **interrupt()**-Aufruf zu einer **InterruptedException**, und der Thread entscheidet im Exception Handler über das weitere Vorgehen (siehe unten).

Zur Demonstration des Verfahrens erhält das Applet mit Laufband (siehe Abschnitte **Fehler! Verweisquelle konnte nicht gefunden werden.** und 15.4) noch einen Schalter zum (irreversiblen) Beenden des Laufband-Threads:




Bei einem Klick auf den Schalter wird aus dem Ereignisverteilungs-Thread der Bedienoberfläche (siehe Abschnitt 15.3) für den Laufschrift-Thread das Interrupt-Signal gesetzt:

```
cbLbBeenden.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    chkLaufband.setEnabled(false);
    cbLbBeenden.setEnabled(false);
    threadLb.interrupt();
  }
});
```

In der **while**-Schleife seiner **run()**-Methode prüft der Laufband-Thread, ob sein Interrupt-Signal gesetzt ist, und beendet ggf. seine Tätigkeit per **return**:

```
public void run() {
  while (true) {
    if (Thread.currentThread().isInterrupted()) {
      lbAnAus = false;
      gp.repaint();
      return;
    }
    try {
      Thread.sleep(PAUSE);
      synchronized(this) {
        if (!lbAnAus)
          wait();
      }
    } catch(InterruptedException ie) {Thread.currentThread().interrupt();}
    if (x-SPRUNG + tb < 0)
      x = getSize().width;
    else
      x -= SPRUNG;
    gp.repaint();
  }
}
```

Angewandt auf einen per **sleep()**, **wait()** oder **join()** in den Wartezustand versetzten Thread hat **interrupt()** folgende Effekte:

- Der Thread wird sofort in den Zustand **ready** versetzt, auch wenn die **sleep()**-Zeit noch nicht abgelaufen ist.
- Es wird eine **InterruptedException** geworfen, und das Interrupt-Signal wird *aufgehoben* (auf **false** gesetzt).

Es kann daher sinnvoll sein, **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 keine von den eben genannten Bedingungen zutrifft, wird das Interrupt-Signal gesetzt. Das kann also auch einem Thread passieren, der gerade auf einen Monitor wartet.

## 15.5 Thread-Lebensläufe

In diesem Abschnitt wird zunächst die Vergabe von Arbeitsberechtigungen für konkurrierende Threads behandelt. Dann fassen wir unsere Kenntnisse über die verschiedenen Zustände eines Threads und über Anlässe für Zustandswechsel zusammen.

### 15.5.1 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 keinesfalls der korrekte Ablauf eines Pogramm davon abhängig sein, dass sich die Rechenzeitvergabe an Threads in einem strengen Sinn an den Prioritäten orientiert.

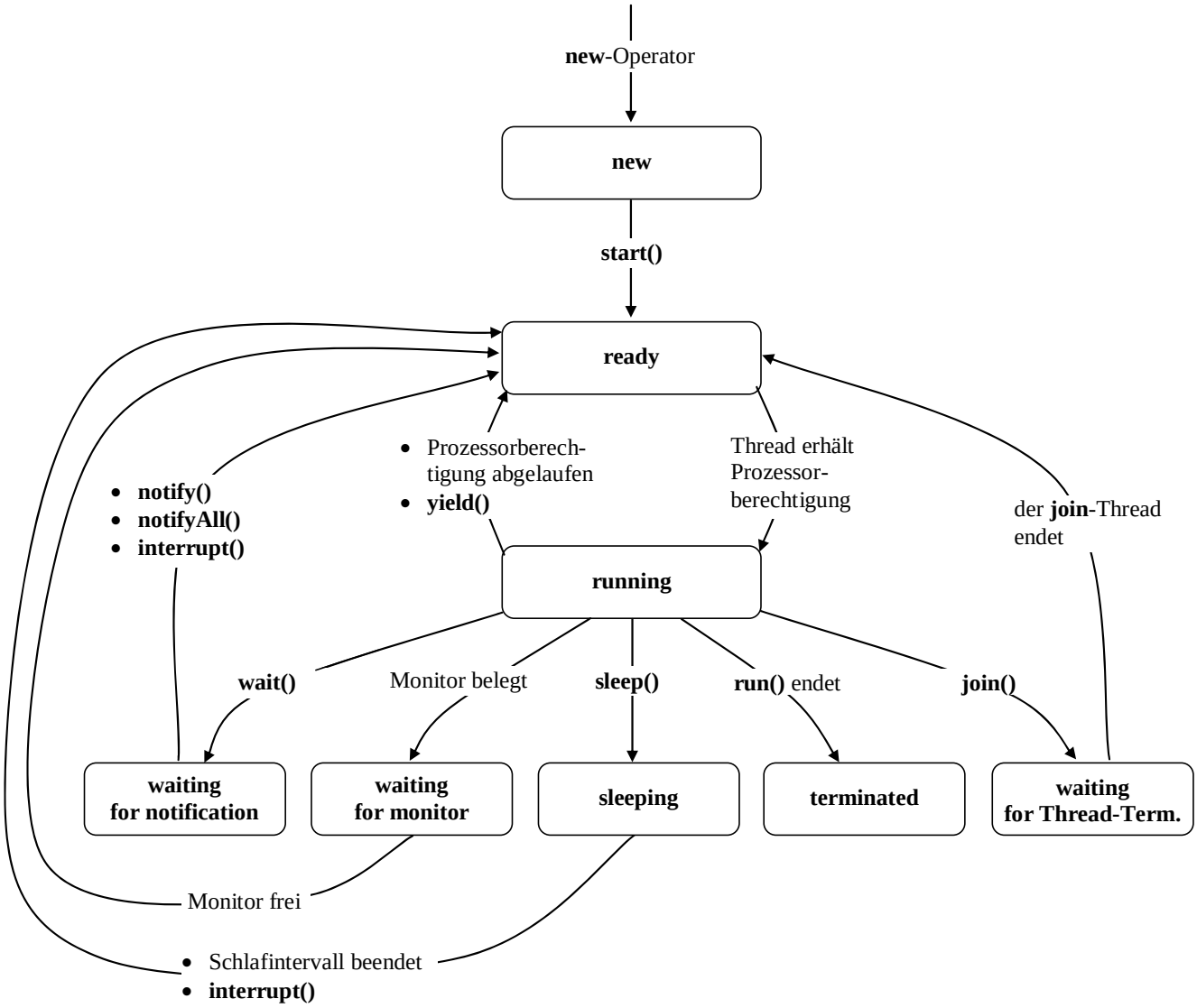
Weil der JVM-Scheduler eng mit dem Wirtsbetriebssystem zusammenarbeiten muss, besteht leider bei der Verteilung von Rechenzeit auf mehrere Threads mit *gleicher* Priorität *keine vollständige* Plattformunabhängigkeit. Auf einigen Plattformen (z.B. Windows) kommt das **preemtive Zeitscheibenverfahren** 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 Warteschlage der rechenwilligen Threads einreihen.

### 15.5.2 Zustände von Threads

In der folgenden Abbildung nach Deitel & Deitel (1999, S. 738) werden wichtige Thread-Zustände und Anlässe für Zustandsübergänge dargestellt:




## *15.6 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 sollten keinesfalls (etwa aufgrund der Präsentation am Ende des Kapitels) als Lösungen für besonders komplexe Anwendungsfälle verstanden werden. Sie eignen sich vielmehr für die bequeme und leistungsfähige Multithreading-Routine.

### 15.6.1 Eine einfache und effektive 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** *runnableObj***)**

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 über einen **ThreadPool-Executor** ein Objekt dieser Klasse erzeugen.

In einer Variante unseres Produzenten-Konsumenten - Beispiels 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 drei Zugriffen:

```
class KonThread extends Thread {
  private Lager pl;
  private int custID;
  final private int MANZ = 3;
  private int nr;

  KonThread(Lager pl, int custID) {
    super ("Kunde Nr "+custID);
    this.custID = custID;
    this.pl = pl;
  }

  public void run() {
    while (++nr < MANZ) {
      pl.liefere((int) (5 + Math.random()*100), custID);
      try {
        sleep((int) (1000 + Math.random()*3000));
      } catch(InterruptedException ie){
        System.err.println(ie);
      }
    }
    System.out.println("\nDer Kunde "+custID+" hat keine Wuensche mehr.\n");
  }
}
```

Wie das folgende Ablaufprotokoll zeigt, versorgt z.B. der Thread **pool-1-thread-1** nacheinander die Kunden 1 und 4:

```
Der Laden ist offen (Bestand = 1000)

Nr. 1:  Produzent ergaenzt                      542  um 01:12:23 Uhr. Stand: 1542
Nr. 2:  pool-1-thread-1 (Kunde 1) entnimmt       100  um 01:12:23 Uhr. Stand: 1442
Nr. 3:  pool-1-thread-1 (Kunde 1) entnimmt       27   um 01:12:24 Uhr. Stand: 1415
Nr. 4:  pool-1-thread-2 (Kunde 2) entnimmt       6    um 01:12:26 Uhr. Stand: 1409
Nr. 5:  Produzent ergaenzt                      524  um 01:12:26 Uhr. Stand: 1933
Nr. 6:  pool-1-thread-2 (Kunde 2) entnimmt       13   um 01:12:27 Uhr. Stand: 1920

Der Kunde 1 hat keine Wuensche mehr.

Der Kunde 2 hat keine Wuensche mehr.

Nr. 7:  pool-1-thread-2 (Kunde 3) entnimmt       9    um 01:12:29 Uhr. Stand: 1911
Nr. 8:  Produzent ergaenzt                      599  um 01:12:29 Uhr. Stand: 2510
Nr. 9:  Produzent ergaenzt                      530  um 01:12:31 Uhr. Stand: 3040
Nr. 10: pool-1-thread-1 (Kunde 4) entnimmt       41   um 01:12:32 Uhr. Stand: 2999
Nr. 11: pool-1-thread-2 (Kunde 3) entnimmt       28   um 01:12:32 Uhr. Stand: 2971
Nr. 12: Produzent ergaenzt                      564  um 01:12:33 Uhr. Stand: 3535
Nr. 13: Produzent ergaenzt                      581  um 01:12:34 Uhr. Stand: 4116
Nr. 14: pool-1-thread-3 (Kunde 5) entnimmt       9    um 01:12:35 Uhr. Stand: 4107
Nr. 15: pool-1-thread-1 (Kunde 4) entnimmt       49   um 01:12:35 Uhr. Stand: 4058

Der Kunde 3 hat keine Wuensche mehr.

Der Kunde 4 hat keine Wuensche mehr.

Nr. 16: Produzent ergaenzt                      564  um 01:12:36 Uhr. Stand: 4622
Nr. 17: pool-1-thread-3 (Kunde 5) entnimmt       58   um 01:12:37 Uhr. Stand: 4564

Der Kunde 5 hat keine Wuensche mehr.

Nr. 18: Produzent ergaenzt                      503  um 01:12:40 Uhr. Stand: 5067
Nr. 19: Produzent ergaenzt                      570  um 01:12:43 Uhr. Stand: 5637
Nr. 20: Produzent ergaenzt                      586  um 01:12:45 Uhr. Stand: 6223

Der Produzent macht Feierabend.
```

Den vollständigen Quellcode finden Sie im Ordner

**...\BspUeb\Multithreading\ProKon\Threadpool**

### 15.6.2 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, wobei im Vergleich zum Interface **Runnable** (vgl. Abschnitt 15.1.2) einige Unterschiede bestehen:

Im Unterschied zur **Runnable**-Methode **run()**, die ebenfalls in einem eigenen Thread ausgeführt werden kann, liefert die **Callable**-Methode **call()** einen Rückgabewert. Eine analoge Option ist uns übrigens schon bei der Klasse **SwingWorker** begegnet (vgl. Abschnitt 15.3.4). 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 9.5) 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:

```
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 schon im Abschnitt 15.6.1 kennen gelernt haben, z.B.:

```
ExecutorService es = Executors.newSingleThreadExecutor();
```

Hier entsteht ein 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 = 20000000;
    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:

```
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 (Exception e) {}
  }
  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 einer das Interface **Future<Double>** implementierenden Klasse. Über die Methode **isDone()** kann man sich bei diesem Objekt über die Fertigstellung des Auftrags informieren.

Seine Methode **get()** liefert schließlich die **call()**-Rückgabe an den **submit()**-Aufrufer oder leitet ein von **call()** geworfenes Ausnahmeobjekt weiter. Bei einem gelungenen Aufruf (ohne **TimeoutException**) liefert das Beispielprogramm die Ausgabe:

```
Warten auf call()-Ende (14.03.2010 01:58:32)
Warten auf call()-Ende (14.03.2010 01:58:33)
Warten auf call()-Ende (14.03.2010 01:58:34)
Warten auf call()-Ende (14.03.2010 01:58:35)
Warten auf call()-Ende (14.03.2010 01:58:36)

Mittelwert der Zufallszahlen: 0.500071458147535
```

Der vom Exekutor verwaltete Thread (mit dem Namen **pool-1-thread-1**) verbleibt nach Abschluss des **call()**-Aufrufs in Lauerstellung 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();
```

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

**...\BspUeb\Multithreading\Callable**

### 15.6.3 Threadpools mit Timer-Funktionalität

Mit der (im Kurs 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. Soll dabei ein Threadpool zum Einsatz kommen, bietet sich die Klasse **ScheduledThreadPoolExecutor** im Paket **java.util.concurrent** an. Bei einer periodisch auszuführenden Aufgabe werden zeitliche Überschneidungen von aufeinanderfolgenden Episoden verhindert. Bevor die nächste Wiederholung gestartet wird, muss also die vorherige beendet sein. Folglich genügt bei einer einzelnen periodisch auszuführenden Aufgabe die ältere Klasse **Timer**, die nur einen Thread verwendet. Die Klasse **ScheduledThreadPoolExecutor** arbeitet mit einem Threadpool, wobei aber im Unterschied zur Klasse **ThreadPoolExecutor** ein fester Umfang verwendet wird. Sobald mehrere wiederholt und mit individuellem Zeitplan auszuführende Aufgaben vorliegen, können die Pool-Threads flexibel eingesetzt werden.

Im folgenden Beispiel wird ein **ScheduledThreadPoolExecutor**-Objekt über die statische **Executors**-Methode **newScheduledThreadPool()** erzeugt:

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

class ScheduledThreadPoolExecutorDemo {
  public static void main(String[] args) {
    ScheduledThreadPoolExecutor es =
      (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(3);
    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 (Exception ignored) {}
    es.shutdown();
  }
}
```

Es erhält über seine Methode **scheduleAtFixedRate()** drei Aufträge erteilt, wobei jeweils mit individueller Initialverzögerung und Periode 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()+
         ", Pool-Threads: "+es.getActiveCount());
    try {Thread.sleep(500*nr);
    } catch (Exception ignored) {}
  }
}
```

Wie das folgende Ablaufprotokoll zeigt, variiert die Anzahl der aktiven Threads, wobei das Maximum drei nicht überschritten werden kann (auch nicht bei einer Pool-Kapazität größer als drei):

```
ScheduledRunner 1, Zeit: 02:25:27, Pool-Threads: 1
ScheduledRunner 1, Zeit: 02:25:28, Pool-Threads: 1
ScheduledRunner 1, Zeit: 02:25:29, Pool-Threads: 1
ScheduledRunner 2, Zeit: 02:25:29, Pool-Threads: 2
ScheduledRunner 1, Zeit: 02:25:30, Pool-Threads: 2
ScheduledRunner 3, Zeit: 02:25:30, Pool-Threads: 2
ScheduledRunner 1, Zeit: 02:25:31, Pool-Threads: 2
ScheduledRunner 2, Zeit: 02:25:31, Pool-Threads: 3
ScheduledRunner 1, Zeit: 02:25:32, Pool-Threads: 2
```

Die Methode **main()** fordert den Exekutor nach fünf Sekunden Fleißarbeit per **shutdown()**-Methode auf, seine Tätigkeit einzustellen. Daraufhin werden keine neuen Ausführungen mehr begonnen, so dass die Pool-Threads nach einiger Zeit enden.

Die im Paket **javax.swing** enthaltenen Klasse **Timer** unterscheidet sich von den oben vorgestellten Lösungen in folgenden Punkten:

- Einfache Integration in Swing-Anwendungen
- Ereignisgesteuerte Ausführung
  Weil die einmalig oder regelmäßig zu einer vorbestimmten Zeit auszuführenden Aufgaben von Ereignisbehandlungsmethoden, also im Event Dispatch Thread (EDT), ausgeführt werden, sind zugunsten einer flüssig regierenden Bedienoberfläche nur Aufgaben mit sehr geringem Zeitaufwand sinnvoll (z.B. Aktualisieren einer Zeitanzeige).

## *15.7 Sonstige Thread-Themen*

### 15.7.1 Daemon-Threads

Neben den bisher behandelten Benutzer-Threads kennt Java noch so genante Daemon-Threads, die im Hintergrund 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.

Mit der Thread-Methode **setDeamon()** lässt sich auch ein Benutzer-Thread dämonisieren, was vor dem Aufruf seiner **start()**-Methode geschehen muss.

Um das Terminieren von Daemon-Threads braucht man sich in der Regel nicht zu kümmern, denn ein Java-Programm oder -Applet endet, sobald ausschließlich Daemon-Threads vorhanden sind.

### 15.7.2 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 2009, Abschnitt 11.7). Dies wird durch den Modifikator **volatile** verhindert, z.B.:

```
private volatile long counter;
```

Außerdem verhindert die **volatile**-Deklaration Speicherzugriffsoptimierungen der Laufzeitumgebung durch Zwischenspeicherung (Cache-Strategien). Somit ist sicher gestellt, dass jede Wertänderung sofort allen Threads zur Verfügung steht. Dieser Effekt des Modifikators ist potentielle bei Feldern von beliebigem Typ von Bedeutung, z.B. bei einem Objekt der Klasse **StringBuffer**:

```
private volatile StringBuffer sbAdress;
```

## *15.8 Übungsaufgaben zu Kapitel 15*

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Ein Java-Programm endet zusammen mit seinem Thread **main**.
2. Ein Java-Programm endet, wenn all seine Threads beendet sind.
3. Ein terminierter Thread kann *nicht* mehr neu gestartet werden.
4. Ein Thread im Zustand **sleeping** gibt alle Monitore zurück.

2) Das folgende Programm startet einen Thread aus der Klasse `Schnarcher`, lässt ihn 5 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(5000);
    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 die **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:

```
public 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)?

4) Erstellen Sie eine Swing-Anwendung, die den Effekt der Prioritäten auf das Verhalten zweier Threads beobachten zu können:

- Der Benutzer soll die Möglichkeit haben, die Prioritäten der beiden Threads zu verändern.
- Die Ausführungsgeschwindigkeiten der Threads sollen als „Rennen“ zu beobachten sein.
- Vielleicht bauen Sie noch Bedienelement ein, um die Threads einzeln unterbrechen und reaktivieren zu können (per **wait()** und **notify()**).

Bei diesem Lösungsvorschlag


zeichnen beide Threads eine Laufschrift im unteren Bereich eines **JFrame**-Fensters. Zur Einstellung der Prioritäten sind Schieberegler vorhanden, realisiert mit **JSlider**-Komponenten (siehe API-Dokumentation).

# 16 Netzwerkprogrammierung

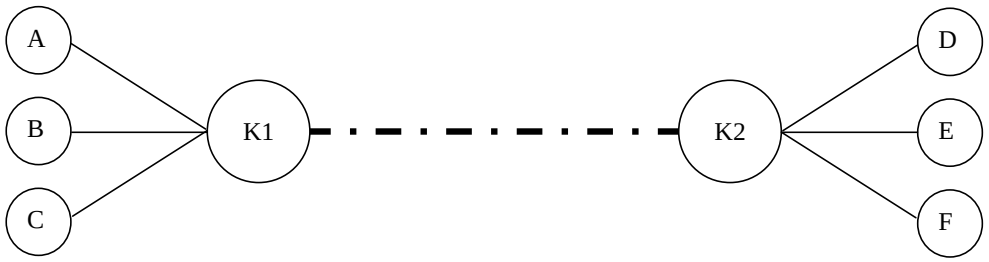
Konform zu ihrem bereits 1982 formulierten Leitsatz *The Network is the Computer* hat sich die Firma Sun Microsystems beim Java-Design erfolgreich darum bemüht, leistungsfähige und dabei möglichst einfach realisierbare Netzwerkanwendungen zu ermöglichen.

## *16.1 Wichtige Konzepte der Netzwerktechnologie*

Als **Netzwerk** bezeichnet man eine Anzahl von Systemen (z.B. Rechnern), die über ein **gemeinsames Medium** (z.B. Ethernet-Kabel, WLAN, Infrarotkanal) verbunden sind und über ein **gemeinsames Protokoll** (z.B. TCP/IP) Daten austauschen können.

Unter einem **Protokoll** ist eine Menge von **Regeln** zu verstehen, die für eine erfolgreiche Kommunikation von allen beteiligten Systemen eingehalten werden müssen.

Bei den meisten aktuellen Netzwerkprotokollen werden Daten **paketweise** übertragen. Zwischen zwei Kommunikationspartnern jeweils eine feste Leitung zu schalten und auch in „Funkpausen" aufrecht zu erhalten, wäre unökonomisch. Wenn über dieselbe Leitung, z.B. zwischen den Verbindungsknoten K1 und K2, Pakete zwischen verschiedenen Kommunikationspartnern, z.B. (A ↔ D), (B ↔ E), ausgetauscht werden, ist eine **Adressierung** der Pakete unabdingbar.




Von der Anwendungsebene (z.B. Versandt einer E-Mail über einen SMTP-Server (*Simple Mail Transfer Protocol*)) bis zur physikalischen Ebene (z.B. elektromagnetische Wellen auf einem Ethernet-Kabel) sind zahlreiche Übersetzungen vorzunehmen bzw. Aufgaben zu lösen, jeweils unter Beachtung der zugehörigen Regeln. Im nächsten Abschnitt werden die beteiligten **Ebenen** mit ihren jeweiligen Protokollen behandelt, wobei wir uns auf Themen mit Relevanz für die Anwendungsentwicklung konzentrieren.

### 16.1.1 Das OSI-Modell

Nach dem **OSI – Modell** (*Open System Interconnection*) der ISO (*International Standards Organisation*) werden sieben aufeinander aufbauende Schichten (engl.: *layers*) mit jeweiligen Aufgaben und zugehörigen Protokollen unterschieden. Bei der anschließenden Beschreibung dieser Schichten sollen wichtige Begriffe und vor allem die heute üblichen Internet-Protokolle (z.B. IP, TCP, UDP, ICMP) eingeordnet werden.

#### 1. Physikalische Ebene (Bit-Übertragung, z.B. über Kupferdrahtleitungen)

Hier wird festgelegt, wie von der Netzwerk-Hardware Bits zwischen zwei direkt verbundenen Stationen zu übertragen sind. Im einfachen Beispiel einer seriellen Verbindung über Kupferkabel wird z.B. festgelegt, dass zur Übertragung einer Null eine bestimmte Spannung während einer festgelegten Zeit angelegt wird, während eine Eins durch eine gleichlange Phase der Spannungsfreiheit ausgedrückt wird.

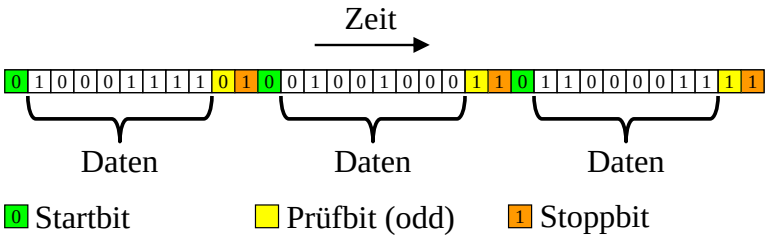
**2. Link-Ebene (gesicherte Frame-Übertragung, z.B. per Ethernet)**

Hier wird vereinbart, wie ein *Frame* zu übertragen ist, der aus einer Anzahl von Bits besteht und durch eine Prüfsumme gesichert ist. In der Regel gehören zum Protokoll dieser Ebene auch Start- und Endmarkierungen, damit sich die beteiligten Geräte rechtzeitig auf eine Informationsübertragung einstellen können.

Im Beispiel der seriellen Datenübertragung kann folgender Frame-Aufbau verwendet werden:

| | |
|---|---|
| Startbit: | 0 |
| 8 Datenbits | |
| Prüfbit: | odd (siehe unten) |
| Stoppbit: | 1 |

In folgender Abbildung sind drei Frames zu sehen, die nacheinander über eine serielle Leitung gesendet werden:




Das odd-Prüfbit wird so gesetzt, dass es die acht Datenbits zu einer ungeraden Summe ergänzt.

Bei einem Ethernet-Frame ist der Aufbau etwas komplizierter (siehe Spurgeon 2000, S. 40ff):

- Der Header enthält u.a. die MAC-Adressen (*Media Access Control*) von Sender und Empfänger. Diese Level-2 - Adressen sind nur für die subnetz-interne Kommunikation relevant.
- Es können zwischen Daten im Umfang von 46 bis 1500 Byte transportiert werden.

**3. Netzwerkebene (Paketübertragung, z.B. per IP)**

Die Frames der zweiten Ebene hängen von der verwendeten Netzwerktechnik ab, so dass auf der Strecke vom Absender bis zum Empfänger in der Regel *mehrere* Frame-Architekturen beteiligt sind (z.B. auf der Telefonstrecke zum Provider eine andere als auf dem weiteren Weg über Ethernet- oder ATM-Verbindungen). Auf der dritten Ebene kommen hingegen Informations-**Pakete** zum Einsatz, die auf der gesamten Strecke (im Intra- und/oder im Internet) unverändert bleiben und beim Wechsel der Netzwerktechnik in verschiedene Schicht 2 - Container umgeladen werden (siehe Abschnitt 16.1.2).

Durch die Protokolle der Schicht 3 sind u.a. folgende Aufgaben zu erfüllen:

- **Adressierung (über Subnetzgrenzen hinweg gültig)**
  Jedes Paket enthält eine Absender- und eine Zieladresse mit globaler Gültigkeit (über Subnetzgrenzen hinweg).
- **Routing**
  In komplexen (und ausfallsicheren) Netzen führen mehrere Wege vom Absender eines Paketes zum Ziel. Vermittlungsrechner (sog. Router) entscheiden darüber, welchen Weg ein Paket nehmen soll.

Bei aktuellen Netzwerken kommt auf der Ebene 3 überwiegend das **IP-Protokoll** zum Einsatz. Seine Pakete bezeichnet man auch als **IP-Datagramme**. In der heute noch üblichen IP-Version 4 (IPv4) besteht eine Adresse aus 32 Bits, üblicherweise durch vier per Punkt getrennte Dezimalzahlen (aus dem Bereich von 0 bis 255) dargestellt, z.B.:

192.168.178.12

Bei der kommenden IP-Version 6 (IPv6) besteht eine Adresse aus 128 Bits, welche durch acht per Doppelpunkt getrennte Hexadezimalzahlen (aus dem Bereich von 0 bis FFFF) dargestellt werden, z.B.:

2001:88c7:c79c:0000:0000:0000:88c7:c79c

Innerhalb eines Blocks dürfen führende Nullen weggelassen werden, z.B.:

2001:88c7:c79c:0:0:0:88c7:c79c

*Eine* Gruppe aufeinanderfolgender Blöcke mit dem Wert 0000 bzw. 0 darf durch zwei Doppelpunkte ersetzt werden, z.B.:

2001:88c7:c79c::88c7:c79c

Der OSI-Ebene 3 wird auch das **Internet Control Message Protocol** (ICMP) zugerechnet, das zur Übermittlung von Fehlermeldungen und verwandten Informationen dient. Wenn z.B. ein Router ein IP-Datagramm verwerfen muss, weil seine Maximalzahl von Weiterleitungen (*Time To Live*, TTL) erreicht wurde, dann schickt er in der Regel eine *Time Exceeded* – Meldung an den Absender. Auch die von **ping** - Anwendungen versandten *Echo Requests* und die zugehörigen Antworten zählen zu den ICMP - Nachrichten.

**4. Transportschicht (gesicherte Paketübertragung, z.B. per TCP)**

Zwar bemüht sich die Protokollebene 3 darum, Pakete auf möglichst schnellem Weg vom Absender zum Ziel zu befördern, sie kann jedoch nicht garantieren, dass *alle* Pakete in *korrekter Reihenfolge* ankommen. Dafür sind die Protokolle der Transportschicht zuständig, wobei momentan vor allem das **Transmission Control Protocol** (TCP) zum Einsatz kommt. Das TCP wiederholt z.B. die Übertragung von Paketen, wenn innerhalb einer festgelegten Zeit keine Bestätigung eingetroffen ist. Eine weitere Aufgabe der Protokollebene 3 besteht in der **Datenflusskontrolle** zur Vermeidung von Überlastungen.

**5. Sitzungsebene (Übertragung von Byte-Strömen zwischen Endpunkten, z.B. per TCP)**

Auf dieser Ebene sind Regeln angesiedelt, die den Datenaustausch zwischen zwei Anwendungen (meist auf verschiedenen Rechnern) ermöglichen. Auch solche Aufgaben werden in der heute üblichen Praxis vom Transmission Control Protocol (TCP) abgedeckt, das folglich für die OSI-Schichten 4 und 5 zuständig ist.

Damit eine spezielle Anwendung auf Rechner A mit einer speziellen Anwendung auf Rechner B kommunizieren kann, werden so genannte **Ports** verwendet. Hierbei handelt es sich um Zahlen zwischen 0 und 65535 ($2^{16}$ - 1), die eine kommunikationswillige bzw. -fähige Anwendung identifizieren. So wird es z.B. möglich, auf einem Rechner verschiedene Serverprogramme zu installieren, die trotzdem von Klienten aufgrund ihrer verschiedenen Ports (z.B. 21 für einen FTP-Server, 80 für einen WWW-Server) gezielt angesprochen werden können. Während die Ports von 0 bis 1023 ($2^{10}$ - 1) für Standarddienste fest definiert sind, werden die höheren Ports nach Bedarf vergeben, z.B. zur temporären Verwendung durch kommunikationswillige Klientenprogramme.

Eine TCP-Verbindung ist also bestimmt durch:

- Die IP-Adresse des Serverrechners und die Portnummer des Dienstes
- Die IP-Adresse des Klientenrechners und die dem Klientenprogramm für die Kommunikation zugeteilte Portnummer

Weitere Eigenschaften einer TCP-Verbindung:

- Das TCP-Protokoll stellt eine *virtuelle Verbindung* zwischen zwei Anwendungen her.
- Auf beiden Seiten steht eine als **Socket** bezeichnete Programmierschnittstelle zur Verfügung. Die beiden Sockets kommunizieren über **Datenströme** miteinander. Aus der Sicht des Anwendungsprogrammierers werden per TCP keine Pakete übertragen, sondern Ströme von Bytes.

Von den Internet-Protokollen ist auch das **User Datagram Protocol** (UDP) auf der Ebene 5 anzusiedeln. Es sorgt ebenfalls für eine Kooperation zwischen *Anwendungen* und nutzt dazu Ports wie das TCP. Allerdings sind die Ports praktisch die einzige Erweiterung gegenüber der IP-Ebene. Es handelt sich also um einen ungesicherten Paketversandt ohne Garantie für eine vollständige Auslieferung in korrekter Reihenfolge. Aufgrund der somit eingesparten Verwaltungskosten eignet sich das UDP zur Übertragung größerer Datenmengen, wenn dabei der Verlust einzelner Pakete zu verschmerzen ist (z.B. beim Multimedia - Streaming). Im Unterschied zu den Datenstrom-Sockets der TCP-Kommunikation spricht man beim UDP-Protokoll von *Datagramm-Sockets*. Java unterstützt auch das UDP-Protokoll. Wir werden uns jedoch in diesem Manuskript auf das wichtigere TCP-Protokoll beschränken.

### 6. Präsentation

Hier geht es z.B. um die Verschlüsselung oder Komprimierung von Daten. Die TCP/IP - Protokollfamilie kümmert sich nicht darum, sondern überlässt derlei Arbeiten den Anwendungen.

### 7. Anwendung (Protokolle für Endbenutzer-Dienstleistungen, z.B. per HTTP oder SMTP)

Hier wird für verschiedene Dienste festgelegt, wie Anforderungen zu formulieren und Antworten auszuliefern sind. Einem SMTP-Server - Programm (*Simple Mail Transfer Protocol*), der an Port 25 lauert, kann ein Klientenprogramm z.B. folgendermaßen eine Mail übergeben:

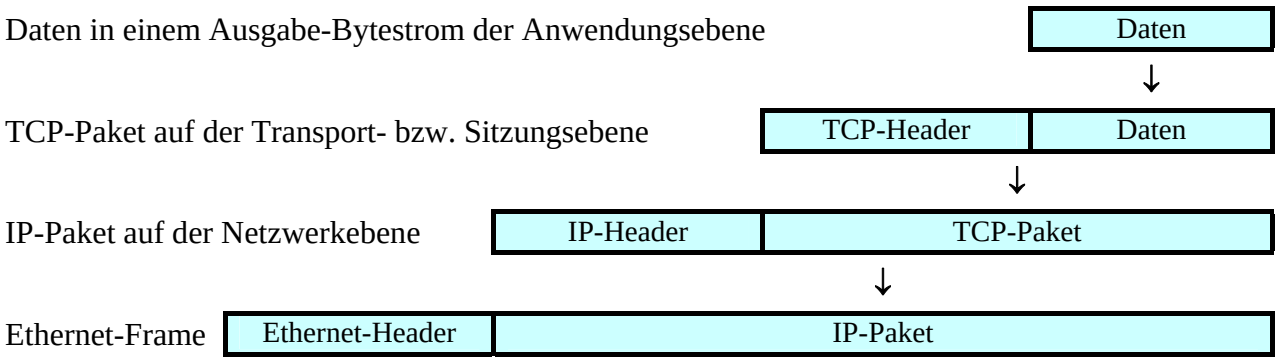
| **Klient** | **Serverantwort** |
|---|---|
| `telnet srv.srv-dom.de 25` | `220 srv.srv-dom.de ESMTP Postfix` |
| `HELO mainpc.client-dom.de` | `250 srv.srv-dom.de` |
| `MAIL FROM: otto@client-dom.de` | `250 Ok` |
| `RCPT TO: empf@srv-dom.de` | `250 Ok` |
| `DATA` | `354 End data with <CR><LF>.<CR><LF>` |
| `From: egal@weg.te` | |
| `To: ziel@soso.te` | |
| `Subject: Thema` | |
| `Dies ist der Inhalt!` | |
| `.` | `250 Ok: queued as 43A7D6D91AC` |
| `QUIT` | `221 Bye` |

Der Mailempfänger glaubt hoffentlich nicht an die angezeigten Adressen:


Noch häufiger als die Mail-Protokolle kommt im Internet auf Anwendungsebene das HTTP-Protokoll (*Hyper Text Transfer Protocol*) für den Austausch zwischen Web-Server und -Browser zum Einsatz.

### 16.1.2 Zur Funktionsweise von Protokollstapeln

Möchte eine Anwendung (genauer: eine aktive Anwendungsinstanz) auf dem Rechner A über ein TCP/IP – Netzwerk eine gemäß zugehörigem Protokoll (z.B. SMTP) zusammengestellte Sendung an eine korrespondierende Anwendung auf dem Rechner B schicken, dann übergibt sie eine Serie von Bytes an die TCP-Schicht von Rechner A, welche daraus TCP-Pakete erstellt. Wir beschränken uns auf den einfachen Fall, dass alle Daten in *ein* TCP-Paket passen, und machen die analoge Annahme auch für alle weiteren Neuverpackungen:




Wichtige Bestandteile des TCP-Headers sind:

- Die Portnummern der Quell- und Zielanwendung
- TCP-Flags
  Hierzu gehört z.B. das zur Gewährleistung der Auslieferung von TCP-Paketen benutzte ACK-Bit. Weil es bei allen Paketen einer Verbindung mit Ausnahme des initialen Pakets gesetzt ist, kann z.B. eine Firewall-Software an diesem Bit erkennen, ob ein von Außen eintreffendes Paket zur (unerwünschten) Verbindungsaufnahme dienen soll.

Das TCP-Paket wird weiter „nach unten“ durchgereicht zur IP-Schicht, die ihren eigenen Header ergänzt, der u.a. folgende Informationen enthält:

- Die IP-Adressen von Quell- und Zielrechner
- Typ des eingepackten Protokolls (z.B. TCP oder UDP)
- Time-To-Live (TTL)
  Beim Routing kann es zu Schleifen kommen. Damit ein Paket nicht ewig rotiert, startet es mit einer Time-To-Live - Angabe mit der maximalen Anzahl von erlaubten Router - Passagen, die von jedem Router dekrementiert wird. Muss ein Router den TTL-Wert auf null setzen, verwirft er das Paket und informiert den Absender eventuell per ICMP-Paket über den Vorfall.

Wenn der nächste Router auf dem Weg zum Zielrechner über ein lokales Netzwerk mit Ethernet-Technik erreicht wird, muss das IP-Paket in einen Ethernet-Frame verpackt werden, wobei der zusätzliche Header z.B. die MAC-Adresse des Routers aufnimmt.

Auf dem Zielrechner wird der umgekehrte Weg durchlaufen: Jede Schicht entfernt ihren eigenen Header und reicht den Inhalt an die nächst höhere Ebene weiter, bis die übertragenen Daten schließlich in einem Eingabestrom der zuständigen Anwendung (identifiziert über die Portnummer im TCP-Header) gelandet sind.

### 16.1.3 Optionen zur Netzwerkprogrammierung in Java

Java unterstützt sowohl die Socket-orientierte TCP- bzw. UDP-Kommunikation (auf der Ebene 5 des OSI-Modells) als auch die Nutzung wichtiger Protokolle auf der Anwendungsebene (z.B. HTTP, SMTP). Ein Zugriff auf tiefere Protokollschichten ist nur über externe, per JNI (*Java Native Interface*) angebundene Software möglich, was allerdings bei Netzwerkprogrammen selten erfor-

derlich ist. Die zur Netzwerkprogrammierung in Java erforderlichen API-Klassen befinden sich meist im Paket **java.net.**

Wir beschäftigen uns in diesem Manuskript u.a. mit folgenden Themen:

- Internet-Ressourcen (WWW-Seiten, Dateien) in Java-Programmen nutzen
- Client-Server - Programmierung auf Socket-Ebene
  Dabei beschränken wir uns auf das TCP-Protokoll, verzichten also auf das nur für wenige Anwendungen relevante UDP-Protokoll.
- Erstellung von Web-Applikationen mit Servlet- und JSP-Technik
  Dieses Thema wird in später im Kapitel zur Java Enterprise Edition (JEE) nachgeholt.

## *16.2 Internet-Ressourcen nutzen*

Zwar im Internet (speziell im World Wide Web) die Interaktivität im Vordergrund steht, sind vielfach automatisierte Routinezugriffe auf Webangebote per Programm von Nutzen (z.B. Abrufen von Wetterdaten, Abholen der monatlichen Provider-Rechnung, Download der aktuellen Signaturdatei einer Schutz-Software). In seiner Ausgabe vom 1. Februar 2010 berichtet das Computer-Magazin über „*Persönliche Webroboter*“ in Skriptsprachen wie Perl, Ruby oder PowerShell. Wer die Programmiersprache Java beherrscht, kann auf dem API und anderen Bibliotheken aufbauend sehr gute Lösungen erstellen.

Auf Internet-Ressourcen, die über einen so genannten **Uniform Resource Locator** (URL) ansprechbar sind, kann man in Java genau so einfach zugreifen wie auf lokale Dateien.

Ein URL wie z.B.

http://www.egal.de:81/cgi-bin/beispiel/cgi.pl?vorname=Kurt

ist folgendermaßen aufgebaut:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Syntax: | *Proto-koll* | :// | *User:Pass@* (optional) | *Rechner* | *:Port* (optional) | *Pfad* | *?URL-Parameter* (optional) |
| Beispiel: | http | :// | | www.egal.de | :81 | /cgi-bin/beispiel/cgi.pl | ?vorname=Kurt |

Bei vielen statischen Webseiten kann am Ende der Pfadangabe durch # eingeleitet noch ein seiteninternes Sprungziel genannt werden, z.B.:

http://www.cs.tut.fi/~jkorpela/forms/methods.html#fund

Die URL-Parameter dienen zur Anforderung von individuellen bzw. dynamisch erstellten Webseiten unter Verwendung der GET-Methode aus dem HTTP-Protokoll (siehe Abschnitt 16.2.5.3). Durch das Zeichen & getrennt dürfen auch mehrere Parameter (Name-Wert - Paare) angegeben werden, z.B.:

?vorname=Kurt&nachname=Schmidt

In der deutschen Sprachpraxis wird meist über *die URL* zu einer Webressource gesprochen, was auch der angesehene Übersetzungedienstleiter http://www.leo.org/ bestätigt:

| ENGLISCH | DEUTSCH |
|---|---|
| | **3 Treffer** |
| **Unmittelbare Treffer** | |
| **URL** - *acronym for 'uniform resource locator'* [comp.] | die (*seltener:* der) **URL** |
| uniform resource locator [abbr.: **URL**] [comp.] | der Uniform-Resource-Locator [Abk.: **URL**] |
| **Zusammengesetzte Einträge** | |
| uniform resource locator [abbr.: **URL**] [comp.] | die Internetadresse |

Im Manuskript wird regeltreu von *dem URL* gesprochen, nach Möglichkeit aber eine geschlechtsneutrale Formulierung gewählt (z.B. der Plural *die URLs*).

### 16.2.1 Die Klasse URL

In vielen Fällen kommt man in Java-Programmen beim Zugriff auf Internet-Ressourcen mit der einfachen Klasse **URL** aus dem Paket **java.net** aus. Das folgende Programm fordert per **URL**-Objekt die Homepage der Universität Trier an und listet die ersten sieben Zeilen des HTML-Codes auf:

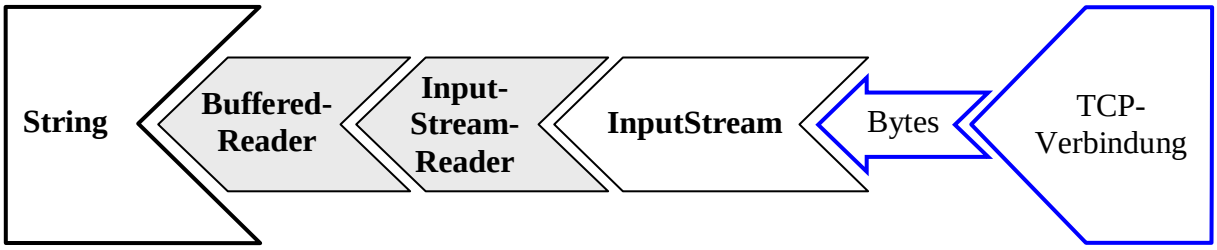
```
import java.net.*;
import java.io.*;

class URLDemo {
   public static void main(String[] args)   {
      BufferedReader br = null;
      try {
         URL url = new URL("http://www.uni-trier.de/");
         br = new BufferedReader(new InputStreamReader(url.openStream()));
         String s;
         for (int i = 0; i < 7; i++) {
            s = br.readLine();
            if (s == null)
               break;
            System.out.println(s);
         }
      } catch (MalformedURLException e) {
         e.printStackTrace();
      } catch (IOException e ) {
         e.printStackTrace();
      } finally {
         try {
            br.close();
         } catch (Exception ignored) { }
      }
   }
}
```

In Java wird fast jeder Datenaustausch via Netz (mit Ausnahme der UDP-Kommunikation) mit derselben Datenstromtechnik abgewickelt, die auch beim seriellen Datenaustausch mit dem lokalen Dateisystem zum Einsatz kommt. Wir werden daher bei den meisten Beispielprogrammen neben dem Paket **java.net** mit den Java-API-Klassen zur Netzwerkprogrammierung auch das Paket **java.io** mit den Datenstromklassen importieren.

Wird dem **URL**-Konstruktor ein **String**-Objekt mit irregulärer Syntax übergeben, ist eine **MalformedURLException** – Exception fällig, auf die sich ein Programm vorbereiten muss.

Die **URL**-Methode **openStream()** öffnet die Verbindung zur Ressource und gibt ein **InputStream**-Objekt für den Zugriff auf die vom angesprochenen Server gelieferten Bytes zurück. Bei Verbindungsproblemen wirft **openStream()** eine **IOException**, die bekanntlich vom Aufrufer in einer **catch**-Klausel behandelt oder im Methodenkopf angekündigt werden muss.

Im Beispiel wandelt ein **InputStreamReader**-Objekt die angelieferten Bytes in Unicode-Zeichen, wobei unter Windows das Kodierungsschema Windows Latin-1 (Cp1252) voreingestellt ist. Um zeilenweise mit der bequemen Methode **readLine()** lesen zu können, schaltet man in der Regel noch einen **BufferedReader** hinter den **InputStreamReader**, so dass sich folgende „Pipeline“ ergibt:




Ein Programmlauf bringt (am 18.02.2010) folgendes Ergebnis:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="de" lang="de">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
```

An Stelle der Klasse **BufferedReader** kann auch die Klasse **Scanner** eingesetzt werden (siehe Abschnitt 12.5).

Zur direkten Lektüre durch Benutzer ist der HTML-Code wenig geeignet. Im der Standardbibliothek von Java 6 wird die Fähigkeit zur Darstellung (engl. *rendern*) von HTML-Code durch die Swing-Komponente **JEditorPane** realisiert, die wir im Editor-Projekt von Kapitel 11 zur Darstellung von unformatiertem Text verwendet haben. Allerdings beschränkt sich die **JEditorPane**-Anzeigekompetenz auf die betagte HTML-Version 3.2, so dass z.B. die Homepage der Universität Trier sehr ungewohnt dargestellt wird:


Außerhalb der Standardbibliothek sind vermutlich leistungsfähigere Java-Lösungen zur HTML-Darstellung verfügbar. Andererseits muss man nicht unbedingt das Ziel verfolgen, eine Alternative zu den aktuellen Web-Browsern zu entwickeln.

In welchem Ausmaß durch das explizite Schließen des Eingabestroms Netzwerkressourcen eingespart werden, hängt vom beteiligten Anwendungsprotokoll und vom Verhalten des Servers ab. Nach einer HTTP-Anforderung werden die beteiligten Ressourcen (z.B. der lokale Port) in der Regel nach einiger Zeit automatisch frei gegeben, was sich unter Windows z.B. mit dem Werkzeug **TCPView**[73] beobachten lässt. Mit **close()** erreicht man die sofortige Freigabe, was sich bei einer großen Anzahl von Verbindungen bzw. Anforderungen lohnen kann.

[73] Das von Mark Russinovich entwickelte Programm ist auf der Microsoft-Webseite http://technet.microsoft.com/en-us/sysinternals/bb897437.aspx

Man sollte die Methode **close()** in der **finally**-Klausel aufrufen, damit sie auf jeden Fall ausgeführt wird (vgl. Abschnitt 9.2.1.2). Weil dabei eine Ausnahme auftreten kann, ist erneut eine **try-catch**-Konstruktion erforderlich, wobei man oft auf eine Ausnahmebehandlung verzichtet. Ein **close()**-Aufruf unmittelbar vor dem Ende des Programms bringt natürlich keinen praktischen Vorteil.

Mit den folgenden **URL**-Methoden lassen sich wichtige URL-Bestandteile ermitteln:

**getProtocol()**, **getHost**(), **getPort()**, **getPath()**, **getFile()**, **getQuery()**

Über weitere Möglichkeiten der Klasse **URL** informiert die API-Dokumentation.

### 16.2.2 Die Klasse URLConnection

Erhält ein Objekt der angenehm einfach verwendbaren Klasse **URL** den Auftrag **openStream()**, dann wird hinter den Kulissen ein Objekt der Klasse **URLConnection** über seine Methode **getInputStream()** gebeten, einen Netzwerkeingabestrom zu erstellen, der Daten vom Server beschaffen kann. Durch expliziten Einsatz der Klasse **URLConnection** gewinnt man flexiblere (teilweise auf das HTTP-Protokoll beschränkte) Möglichkeiten, Anforderungen zu formulieren und die Antworten eines Servers auszuwerten. Vor allem kann man ...

- über **Request-Header** eine Anforderung näher spezifizieren. Wer z.B. an einer Ressource nur bei entsprechend aktuellem Änderungsdatum interessiert ist, kann dies per **If-Modified-Since** – Feld ausdrücken.
- über **Response-Header** Meta-Informationen über den von einem WWW-Server gelieferten Inhalt erhalten. Im Feld **Content-Type** wird z.B. das Format einer Ressource beschrieben.

Die Klasse **URLConnection** hält Methoden bereit, um die Header-Felder zu besetzen bzw. auszuwerten (siehe unten). Eine Liste mit allen Header-Feldern der HTTP-Version 1.1 findet sich im RFC-Dokument (*Request For Comments*) 2616, das auf der folgenden Webseite des Internet-Normierungs-Gremiums zu finden ist:

http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

Zum Erzeugen einer **URLConnection** steht kein öffentlicher Konstruktor zur Verfügung. Man ruft stattdessen die **openConnection()** - Methode eines passenden **URL**-Objekts auf.

Im folgenden Programm wird über einen per Kommandozeile festgelegten URL (oder http://web.de/) eine Webseite angefordert, falls diese seit dem 11.02.2010 geändert worden ist:

```
import java.net.*;
import java.io.*;
import java.text.*;
import java.util.*;

class URLConnectionDemo {
   public static void main(String[] args) {
      BufferedReader br = null;
      try {
         String urlString;
         if (args.length == 0)
            urlString = "http://web.de/";
         else
            urlString = args[0];

         URL url = new URL(urlString);
         URLConnection urlConn = url.openConnection();
```

```
            DateFormat df = DateFormat.getDateTimeInstance();
            urlConn.setIfModifiedSince(df.parse("11.02.2008 00:00:00").getTime());

            urlConn.connect();

            System.out.println("\nResponse-Header:");
            System.out.println("  Content-Type:\t\t"+urlConn.getContentType());
            System.out.println("  Content-Length:\t"+urlConn.getContentLength());
            System.out.println("  Expiration:\t\t"+df.format(
              new Date(urlConn.getExpiration())));
            System.out.println("  Last Modified:\t"+df.format(
              new Date(urlConn.getLastModified()))+"\n");

            br=new BufferedReader(new InputStreamReader(urlConn.getInputStream()));
            String zeile;
            while ((zeile = br.readLine()) != null)
              System.out.println(zeile);
          } catch (MalformedURLException e) {
            e.printStackTrace();
          } catch (IOException e) {
            e.printStackTrace();
          } catch (ParseException e) {
            e.printStackTrace();
          } finally {
            try {
              br.close();
            } catch (Exception e) { }
          }
        }
      }
```

Die **URL**-Methode **openConnection()** baut noch keine Verbindung zum Server auf, sondern liefert ein **URLConnection**-Objekt und schafft so die Möglichkeit, die zum **URL**-Objekt gehörige Anforderung über Request-Header näher zu spezifizieren. Generell dient dazu die Methode

**public void setRequestProperty(String** *key***, String** *value***)**

Hier wird z.B. das **If-Modified-Since** – Feld gesetzt:

```
urlConn.setRequestProperty("If-Modified-Since","Mon, 11 Feb 2008 00:00:00 GMT");
```

Einige Felder können aber auch mit speziellen **URLConnection** - Methoden gesetzt werden, z.B. das Feld **If-Modified-Since** mit der Methode **setIfModifiedSince()**:

```
DateFormat df = DateFormat.getDateTimeInstance();
urlConn.setIfModifiedSince(df.parse("11.02.2008 00:00:00").getTime());
```

Im Beispiel wird eine Klartext - Datums/Zeit - Angabe mit Hilfe der Klassen **Date** und **DateFormat** in die von **setIfModifiedSince()** benötigte Anzahl von Millisekunden seit dem 1. Januar 1970 (GMT) umgewandelt.

Man kann sich übrigens nicht unbedingt darauf verlassen, dass sich ein angesprochener Server nach der **If-Modified-Since** - Angabe richtet. Das RFC-Dokument 2616 zum HTTP-Protokoll (URL: siehe oben) enthält zu dieser Frage eher eine Empfehlung als eine Vorschrift:

```
c) If the variant has not been modified since a valid If-
   Modified-Since date, the server SHOULD return a 304 (Not
   Modified) response.
```

Erst durch Aufruf der **URLConnection**-Methode **connect()** wird die TCP-Verbindung zur Gegenstelle tatsächlich geöffnet. Gelingt dies, können anschließend die Response-Header der Webserver-Antwort über passende **URLConnection**-Methoden abgefragt werden. Das Beispielprogramm hat am 18.02.2010 bei einem Start ohne Kommandozeilenparameter folgende Ausgaben geliefert:

```
Response-Header:
  Content-Type:    text/html; charset=iso-8859-1
  Content-Length:  101509
  Expiration:      18.02.2010 17:11:15
  Last Modified:   18.02.2010 17:09:25
```

Die **URLConnection**-Methoden **getExpiration()** und **getLastModified()** liefern Millisekunden seit dem 1. Januar 1970 (GMT), die im Beispiel mit Hilfe der Klassen **Date** und **DateFormat** in verständliche Ausgaben übersetzt werden.

Über die **URLConnection**-Methode **getInputStream()** erreicht man denselben Eingabestrom mit den angeforderten Daten, den auch die **URL**-Methode **openStream()** (siehe Abschnitt 16.2.1) liefert.

In welchem Ausmaß durch das explizite Schließen des Eingabestroms Netzwerkressourcen eingespart werden, wurde schon in Abschnitt 16.2.1 diskutiert.

Sind Verbindungsprobleme zu befürchten, sollte vor dem **connect()**-Aufruf mit der Methode

**public void setConnectTimeout(int** *timeout*)

eine maximale Wartezeit festgelegt werden. Ein Überschreiten dieser Zeit wird von **connect()** per **SocketTimeoutException** signalisiert.

### 16.2.3 Datei-Download

Mit den Klassen **URL** und **URLConnection** kann man nicht nur HTML- und sonstige Textdateien von einem Server beziehen, sondern auch binäre Dateien herunterladen, was in folgendem Beispiel mit der Archivdatei **PaswStat17.zip** geschieht:

```
import java.net.*;
import java.io.*;

class DateiDownload {
   public static void main(String[] args) {
      BufferedInputStream in = null;
      BufferedOutputStream out = null;
      try {
         String urlString =
            "http://www.uni-trier.de/fileadmin/urt/doku/spss/v17/PaswStat17.zip";
         URL url = new URL(urlString);
         System.out.println("Die Datei "+urlString+" wird herunter geladen ...");
         in = new BufferedInputStream(url.openStream());
         out = new BufferedOutputStream(
               new FileOutputStream((new File(url.getPath()).getName())));
         int i, n = 0;
         while ((i = in.read()) != -1) {
            out.write(i);
            n++;
         }
         System.out.println("Fertig! (Bytes geschrieben: "+n+")");
      } catch (Exception e) {
         e.printStackTrace();
      } finally {
         try {
            in.close();
            out.close();
         } catch (Exception e) { }
      }
   }
}
```

Wir kommen mit Byte-orientierten Strömen aus, kombinieren diese aber zur Transportbeschleunigung jeweils mit einem Puffer.

Die URL-Methode **getPath()** liefert im Beispiel die Zeichenfolge:

**/fileadmin/urt/doku/spss/v17/PaswStat17.zip**

Um eine Datei mit dem Namen **PaswStat17.zip** im aktuellen Verzeichnis anzulegen, wird aus der Zeichenfolge ein **File**-Objekt erzeugt und mit **getName()** befragt.

### 16.2.4 Die Klasse HttpURLConnection

Von den Klassen **URL** und **URLConnection** werden einige Spezifika des HTTP-Protokolls *nicht* unterstützt (z.B. Statuscode). Abhilfe schafft die aus **URLConnection** abgeleitete Klasse **HttpURLConnection**, die u.a. folgende Erweiterungen bietet:

- **getResponseCode()**
  liefert den Statuscode einer Anfrage
- **usingProxy()**
  informiert darüber, ob ein Proxy-Server involviert ist

Bei passendem Protokoll liefert die URL-Methode **openConnection()** ohnehin ein **HttpURL-Connection**-Objekt, so dass nur noch eine Typumwandlung erforderlich ist, damit die erweiterte Funktionalität verfügbar wird, z.B.:

```
URL url = new URL(urlString);
URLConnection urlConn = url.openConnection();
urlConn.connect();
if (urlConn instanceof HttpURLConnection) {
   HttpURLConnection huc = (HttpURLConnection) urlConn;
   . . .
}
```

Über ein **HttpURLConnection**-Objekt lässt sich nun z.B. der Statuscode einer Webserver-Antwort ermitteln:

```
System.out.println("\nRequest-Status:\n  Code:\t\t" + huc.getResponseCode() +
   "\n  Message:\t" + huc.getResponseMessage());
```

Hat alles geklappt, erhält man den Code 200:

```
Request-Status:
  Code:         200
  Message:      OK
```

Den folgenden Request-Status haben Sie vermutlich schon öfter gesehen:

```
Request-Status:
  Code:         404
  Message:      Not Found
```

### 16.2.5 Dynamisch erstellte Webinhalte anfordern

#### *16.2.5.1 Überblick*

WWW-Server halten in der Regel nicht nur statische HTML-Seiten und sonstige Dateien bereit, sondern bieten auch verschiedene Technologien, um HTML-Seiten dynamisch nach Kundenwunsch zu erzeugen und an Klientenprogramme (meist **WWW-Browser**) auszuliefern (z.B. mit den Ergebnissen eines Suchauftrags oder mit einer individuellen Produktkonfiguration). WWW-Nutzer äußern ihre Wünsche, indem sie per Browser (z.B. Mozilla-Firefox, MS Internet Explorer) eine Formularseite (mit Eingabeelementen wie Textfeldern, Kontrollkästchen usw.) ausfüllen und ihre Daten zum **WWW-Server** übertragen. Dieses Programm (z.B. Apache, MS Internet Information Server) analysiert und beantwortet Formulardaten aber nicht selbst, sondern überlässt diese Arbeit externen Anwendungen, die in unterschiedlichen Programmier- bzw. Skriptsprachen erstellt werden

können (z.B. Java, PHP, ASP.NET, Perl). Traditionell kooperieren WWW-Server und Ergänzungsprogramm über das so genannte **Common Gateway Interface** (**CGI**), wobei das Ergänzungsprogramm bei jeder Anforderung neu gestartet und nach dem Erstellen der HTML-Antwortseite wieder beendet wird. Mittlerweile werden jedoch Lösungen bevorzugt, die stärker mit dem Webserver verzahnt sind, permanent im Speicher verbleiben und so eine bessere Leistung bieten (z.B. PHP als Apache - Modul). So wird vermieden, dass bei jeder Anforderung ein Programm (z.B. der PHP-Interpreter) gestartet und eventuell auch noch eine Datenbankverbindung aufwändig hergestellt werden muss. Außerdem wird bei den genannten Lösungen die nicht sehr wartungsfreundliche Erstellung kompletter HTML-Antwortseiten über Ausgabeanweisungen der jeweiligen Programmiersprache vermieden. Stattdessen können in *einem* Dokument statische HTML-Abschnitte mit Bestandteilen der jeweiligen Programmiersprache zur dynamischen Produktion individueller Abschnitte kombiniert werden. Wir werden anschließend der Einfachheit halber alle Verfahren zur dynamischen Produktion individueller HTML-Seiten als *CGI - Lösungen* bezeichnen. Eine wichtige Gemeinsamkeit dieser Verfahren besteht darin, dass die Browser zur Formulierung ihrer Anforderungen (Requests) die Methoden **GET** und **POST** aus dem HTTP - Protokoll benutzen (siehe unten).

Bei den integrierten, performanten und wartungsfreundlichen Lösungen spielt auch Java eine herausragende Rolle, und insbesondere bei anspruchsvollen Server-Dienstleistungen (z.B. Online-Banking) kommen häufig *Java-Servlets* bzw. *Java Server Pages* zum Einsatz, die von einem Java - Application Server wie *Tomcat* oder *Glassfish* ausgeführt werden (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).

Im aktuellen Abschnitt wird die Anforderung von dynamisch erstellen Webinhalten durch klientenseitige Java-Programme behandelt. Damit lässt sich in vielen Fällen das Abrufen von Informationen (z.B. Börsenkurse, Wetterdaten) vereinfachen bzw. automatisieren. Außerdem werden uns die Erfahrungen in der Klientenrolle nützlich sein, wenn wir später die „Seite wechseln“ und die Realisation dynamischer Webangebote durch serverseitige Java-Lösungen behandeln.

Es besteht eine Verwandtschaft zu den neuerdings sehr populären **Webdiensten** (engl.: *Web Services*), die allerdings keine HTML-Seiten für Browser produzieren, sondern XML - Dateien. Von dieser interessanten Technik zur Erstellung von verteiltem Anwendungen kann in diesem Kurs weder die server- noch die klientenseitige Programmierung behandelt werden.

#### *16.2.5.2 Arbeitsablauf*

Beim CGI - Einsatz sind üblicherweise folgende Programme beteiligt:

- WWW-Browser
- WWW-Server
  In der Regel läuft dieses Programm auf einem fremden Rechner im Internet.
- CGI-Lösung
  Wir bezeichnen der Einfachheit halber jedes Verfahren zur dynamischen HTML-Produktion als *CGI - Lösung*.

Der Browser zeigt eine vom Server erhaltene HTML-Seite mit Formular an, über die Benutzer eine CGI-Anfrage konfigurieren und abschicken können. Wir betrachten ein einfaches Formular mit zugehörigem CGI-Skript, um einige technische Details zu erläutern.

In diesem Browser-Fenster


wird eine HTML-Datei angezeigt, die über den URL

http://urtkurs.uni-trier.de/prokur/netz/cgig.html

abrufbar ist und den folgenden HTML-Code mit Formular enthält:

```
<html>
<head>
<title>CGI-Demo</title>
</head>
<h1>Nenne Deinen Namen, und ich sage Dir, wie Du hei&szlig;t!</h1>
<form method="get" action="cgig.php">
<table border="0" cellpadding="0" cellspacing="4">
 <tr>
  <td align="right">Vorname:</td>
  <td><input name="vorname" type="text" size="30"></td>
 </tr><tr>
  <td align="right">Nachname:</td>
  <td><input name="nachname" type="text" size="30"></td>
 </tr>
 <tr> </tr>
 <tr>
  <td align="right"> <input type="submit" value=" Absenden "> </td>
  <td align="right">  <input type="reset" value=" Abbrechen"> </td>
  </td>
 </tr>
</table>
</form>
</html>
```

Klickt der Benutzer auf den Schalter **Absenden**, werden die Formularfelder mit der Syntax

*Name=Wert*

als Parameter für die CGI - Software zum WWW-Server übertragen. Zwei Felder werden jeweils durch ein &-Zeichen getrennt, so dass im obigen Beispiel mit den Feldern `vorname` und `nachname` (siehe HTML-Quelltext) folgende Sendung resultiert:

```
vorname=Kurt&nachname=M%FCller
```

Den Umlaut „ü“ kodiert der Browser automatisch durch ein einleitendes Prozentzeichen und seinen Hexadezimalwert im Zeichensatz der Webseite. Wenn im HTML-Code kein Zeichensatz angegeben ist, verwendet der Browser eine Voreinstellung (im Beispiel: ISO 8859-1). Analog werden andere Zeichen behandelt, die nicht zum Standard - ASCII-Code gehören. Weitere Regeln dieser so genannten **URL-Kodierung**:

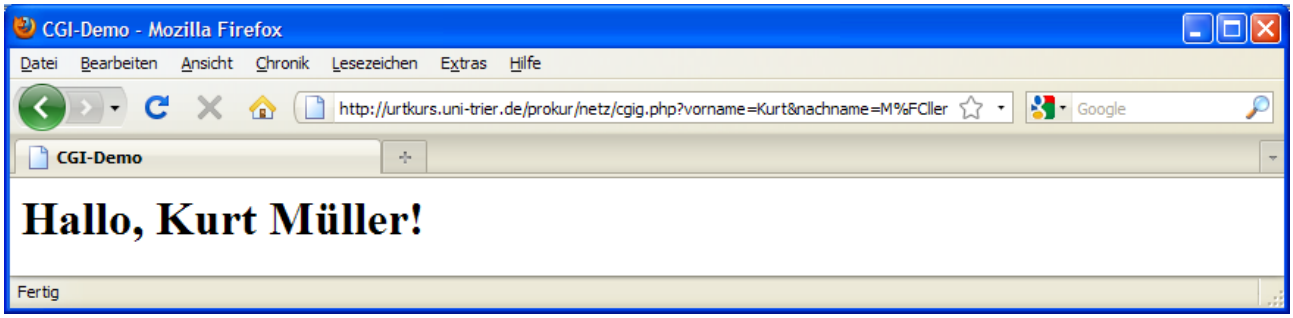
- Leerzeichen werden durch ein „+“ ersetzt.
- Die mit einer speziellen Bedeutung belasteten Zeichen (also &, +, = und %) werden durch ihren Hexadezimalwert im Zeichensatz dargestellt.

Auf gleich noch näher zu erläuternde Weise übergibt der WWW-Server die Formulardaten an das im **action**-Attribut der Formulardefinition angegebene externe Programm oder Skript. Im Beispiel handelt es sich um folgendes **PHP**-Skript, das wenig kreativ aus den übergebenen Namen einen Gruß formuliert:

```
<?php
 $vorname = $_GET["vorname"];
 $nachname = $_GET["nachname"];
 echo "<html>\n<head><title>CGI-Demo</title>\n</head>\n";
 echo "<body>\n<h1>Hallo, ".$vorname." ".$nachname."!</h1>\n</body>\n</html>";
?>
```

Im Skript wird die auszugebende HTML-Seite über **echo**-Kommandos an die Standardausgabe geschickt, und der WWW-Server befördert die PHP-Produktion über das HTTP-Protokoll an den Browser, der den empfangenen HTML-Quelltext


anzeigt:




### *16.2.5.3 GET*

Zum Versandt der Name-Wert - Paare eines Formulars an einen WWW-Server kennt das HTTP-Protokoll zwei Methoden (GET und POST), die nun vorgestellt und dabei auch gleich in Java-Klientenprogrammen realisiert werden sollen.

Bei der GET-Methode, die man im **form** - Tag einer HTML-Seite durch die Angabe

```
method="get"
```

wählt (siehe oben), schickt der Browser die Name-Wert - Paare als URL-Bestandteil hinter einem trennenden Fragezeichen an den WWW-Server. Weil nach Eintreffen der Antwortseite die zugrunde liegende Anforderung in der Adresszeile des Browsers erscheint, kann die GET-Syntax dort inspiziert werden (siehe oben).

Aus der Integration der HTTP-Parameter in die Anforderung an den WWW-Server ergibt sich eine Längenbeschränkung, wobei die konkreten Maximalwerte vom Server und vom Browser abhängen.

Man sollte vorsichtshalber eine Anforderungsgesamtlänge von 255 Zeichen einhalten und ggf. die POST-Technik verwenden, die keine praxisrelevante Längenbeschränkung kennt.

Der WWW-Server schreibt die Name-Wert - Paare in eine Umgebungsvariable namens **QUERY_STRING** und stellt auf analoge Weise der CGI-Software gleich noch weitere Informationen zur Verfügung, z.B.:

```
QUERY_STRING="vorname=Kurt&nachname=M%3Fller"
REMOTE_PORT="1211"
REQUEST_METHOD="GET"
```

In obigem PHP-Skript erfolgt der der Zugriff auf die Parameter in der Umgebungsvariablen **QUERY_STRING** über den superglobalen Array **$_GET**.

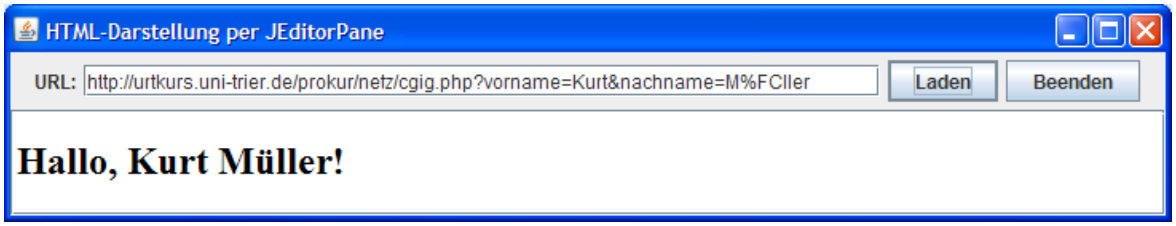
Um in Java eine CGI-Software anzusprechen, die per GET mit Parametern versorgt werden möchte, genügt ein Objekt der angenehm einfach aufgebauten Klasse **URL** (siehe Abschnitt 16.2.1). Für die URL-Kodierung der Parameterwerte (vgl. Abschnitt 16.2.5.2) sorgt eine statische Methode der Klasse **URLEncoder**, wobei durch die Angabe eines Zeichensatzes korrekte Hexadezimalwerte der Sonderzeichen sichergestellt werden sollten. Im folgenden Beispiel kommt der Windows-Zeichensatz Cp1252 zum Einsatz:

```
import java.io.*;
import java.net.*;

class GET {
   public static void main(String[] args) {
      BufferedReader br = null;
      try {
         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
         System.out.print("Vorname:  ");
         String vorname = in.readLine();
         System.out.print("Nachname: ");
         String nachname = in.readLine();
         System.out.println();
         URL url = new URL(
               "http://urtkurs.uni-trier.de/prokur/netz/cgig.php?vorname="+
               URLEncoder.encode(vorname,"Cp1252")+
               "&nachname="+URLEncoder.encode(nachname,"Cp1252"));
         br = new BufferedReader(new InputStreamReader(url.openStream()));
         String zeile;
         while ((zeile = br.readLine()) != null)
               System.out.println(zeile);
      } catch (Exception e) {
         e.printStackTrace();
      } finally {
         try {
            br.close();
         } catch (Exception e) { }
      }
   }
}
```

Weil das Programm die vom CGI-Skript gelieferte HTML-Seite nur als unformatierten Text darstellt, ist sein Auftritt nicht berauschend:

```
Vorname:  Kurt
Nachname: Müller

<html>
<head><title>CGI-Demo</title>
</head>
<body>
<h1>Hallo, Kurt Müller!</h1>
</body>
</html>
```

Mit der Swing-Komponente **JEditorPane** ist eine formatierte Anzeige dieser simplen HTML-Seite möglich:




Wie Sie aus Abschnitt 16.2.1 wissen, unterstützt **JEditorPane** allerdings nur die HTML-Version 3.2.

In welchem Ausmaß durch das explizite Schließen des Eingabestroms Netzwerkressourcen eingespart werden, wurde schon in Abschnitt 16.2.1 diskutiert.

#### *16.2.5.4 POST*

Bei der POST-Methode, die man im **form** - Tag einer HTML-Seite durch die Angabe

```
method="post"
```

wählt, werden CGI-Parameter (im selben Format wie bei der GET-Methode) mit Hilfe des WWW-Servers zur Standardeingabe der CGI-Software übertragen. Was genau gemäß HTTP-Protokoll zu tun ist, braucht Java-Programmierer kaum zu interessieren, weil die Klasse **URLConnection** einen Ausgabestrom zur Verfügung stellt, über den man die CGI-Standardeingabe mit Parametern versorgen kann.

In folgendem Beispielprogramm wird zunächst wie in Abschnitt 16.2.2 über die **URL**-Methode **openConnection()** ein Objekt der Klasse **URLConnection** (genauer: **HttpURLConnection**) erzeugt. Anschließend wird dieses Objekt mit dem Methodenaufruf **setDoOutput(true)** darauf vorbereitet, dass Daten zum Server übertragen zu übertragen sind. An den mit **getOutputStream()** angeforderten Ausgabestrom wird ein **PrintWriter** angekoppelt, um die URL-kodierten CGI-Parameter mit der bequemen **print()**-Methode „posten“ zu können:

```
import java.io.*;
import java.net.*;

class POST {
   public static void main(String[] args) {
      BufferedReader br = null;
      PrintWriter pw = null;
      try {
         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
         System.out.print("Vorname:  ");
         String vorname = in.readLine();
         System.out.print("Nachname: ");
         String nachname = in.readLine();
         System.out.println();
         URL url = new URL("http://urtkurs.uni-trier.de/prokur/netz/cgip.php");
         URLConnection urlConn = url.openConnection();
         urlConn.setDoOutput(true);
         pw = new PrintWriter(urlConn.getOutputStream());
         pw.print("vorname="+URLEncoder.encode(vorname,"Cp1252")+
            "&nachname="+URLEncoder.encode(nachname,"Cp1252"));
         pw.flush();
         br = new BufferedReader(new InputStreamReader(urlConn.getInputStream()));
         String zeile;
         while ((zeile = br.readLine()) != null)
              System.out.println(zeile);
      } catch (Exception e) {
         e.printStackTrace();
```

```
      } finally {
        try {
          br.close();
          pw.close();
        } catch (Exception e) { }
      }
    }
  }
```

Weil **PrintWriter** grundsätzlich puffern (vgl. Abschnitt 12.4.1.4), sorgt im Beispiel ein Aufruf der Methode **flush()** dafür, dass die Parameterdaten auf die Reise gehen.

In welchem Ausmaß durch das explizite Schließen der Netzwerkströme Ressourcen eingespart werden, wurde schon in Abschnitt 16.2.1 diskutiert.

Das angesprochene PHP-Skript unterscheidet sich kaum von der GET-Variante: Anstelle des superglobalen Arrays **$_GET** ist der analoge Array **$_POST** zu verwenden.

## *16.3 IP-Adressen bzw. Host-Namen ermitteln*

Jeder an das Internet angeschlossene Rechner verfügt über eine **IP-Adresse** (32-bittig in IPv4, 128-bittig in IPv6) sowie über einen **Host-Namen**, wobei die Zuordnung vom **Domain Name System** (DNS) geleistet wird.

In Java werden IP-Adressen durch Objekte der Klasse **InetAddress** repräsentiert. Zum Erzeugen neuer **InetAddress**-Objekte fehlt ein öffentlicher Konstruktor, doch stehen für diesen Zweck statische **InetAddress**-Methoden bereit, z.B.:

- **public static InetAddress getLocalHost()**
  liefert ein **InetAddress**-Objekt zum lokalen Rechner
- **public static InetAddress getByName(String** *host*)
  liefert ein **InetAddress**-Objekt zum Rechner mit dem angegebenen Host-Namen
- **public static InetAddress getByAddress(byte[]** *address*)
  liefert ein **InetAddress**-Objekt zum Rechner mit der angegebenen IP-Adresse

Das folgende Programm stellt für den lokalen Rechner IP-Adresse und Host-Namen fest:

```
import java.net.*;
class InetAddressDemo {
  public static void main(String[] args) {
    try {
      InetAddress lh = InetAddress.getLocalHost();
      System.out.println("IP-Adresse des lokalen Rechners:\t"+
        lh.getHostAddress());
      System.out.println("Host-Name des lokalen Rechners:\t\t"+
        lh.getHostName());
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Eine Beispielausgabe:

```
IP-Adresse des lokalen Rechners:   192.168.178.12
Host-Name des lokalen Rechners:    domino
```

Das nächste Beispielprogramm kann zwischen einer IPv4-Adresse und einem Host-Namen übersetzen und bietet dabei einen zeitgemäßen Bedienkomfort:


Aufgrund der Swing-Oberfläche ist der Quelltext deutlich länger als beim vorherigen Beispiel. Daher wird nur der für beide Schaltflächen zuständige **ActionEvent**-Handler wiedergegeben:

```
public void actionPerformed(ActionEvent ae) {
  try {
    if (ae.getSource() == cbGetIP) {
      InetAddress ia = InetAddress.getByName(tfHostName.getText());
      tfIP.setText(ia.getHostAddress());
    } else {
      byte[] ipAddr = new byte[4];
      StringTokenizer st = new StringTokenizer(tfIP.getText(), ".", false);
      for (int i = 0; i < 4; i++)
        ipAddr[i] = (byte) Integer.parseInt(st.nextToken());
      InetAddress ia = InetAddress.getByAddress(ipAddr);
      tfHostName.setText(ia.getHostName());
    }
  } catch(Exception e) {
    JOptionPane.showMessageDialog(null, e,
        "Exception", JOptionPane.INFORMATION_MESSAGE);
  }
}
```

Ein Objekt der Klasse **StringTokenizer** hilft dabei, aus einer Zeichenfolge mit einer IPv4-Adresse den im **getByAddress()**-Aufruf benötigten **byte**-Array zu gewinnen. Den vollständigen Quellcode finden Sie im Ordner

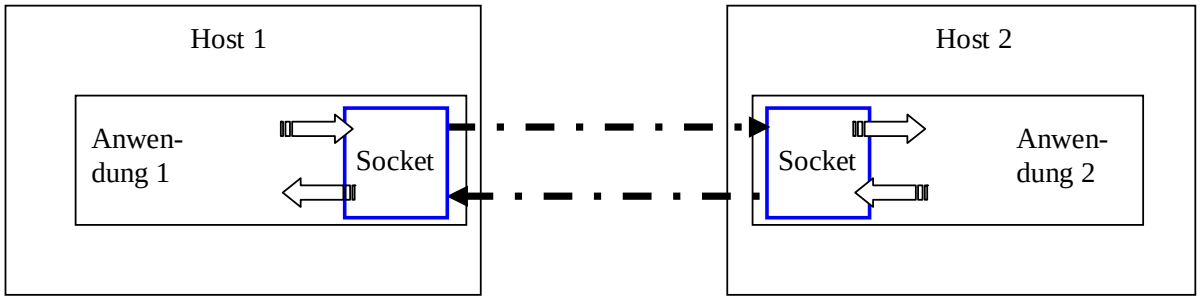
**…\BspUeb\Netzwerk\DNS**

## *16.4 Socket-Programmierung*

Unsere bisherigen Beispielprogramme in Kapitel 16 haben hauptsächlich WWW-Inhalte von Servern bezogen und dazu API-Klassen benutzt, die ein fest „verdrahtetes“ Anwendungsprotokoll (meist HTTP) realisieren. Im aktuellen Abschnitt gewinnen wir eine erweiterte Flexibilität durch den direkten Einsatz des TCP-Protokolls. Daraus ergibt sich z.B. die Möglichkeit, *eigene* Anwendungsprotokolle zu entwickeln. Das auf der Transport- bzw. Sitzungsebene des OSI-Modells (siehe Abschnitt 16.1.1) angesiedelte TCP-Protokoll schafft zwischen zwei (durch *Portnummern* identifizierten) Anwendungen, die sich meist auf verschiedenen (durch *IP-Adressen* identifizierten) Rechnern befinden, eine virtuelle, *Datenstrom-orientierte* und *gesicherte Verbindung*. An beiden Enden der Verbindung steht das von praktisch allen aktuellen Programmiersprachen unterstützte *Socket-API* zur Verfügung, das in Java durch die Klasse **Socket** realisiert wird.

TCP-Programmierer müssen sich nicht um IP-*Pakete* kümmern, weder die Zustellung noch die Integrität oder die korrekte Reihenfolge überwachen, sondern (nach den Regeln eines Protokolls der Anwendungsschicht) Bytes in einen Ausgabestrom einspeisen bzw. aus einen Eingabestrom entnehmen und interpretieren. Dabei ist der Ausgabestrom des Senders virtuell mit dem Eingabestrom des Empfängers verbunden.

Ein **Socket**-Objekt bietet einen Ausgabe- *und* einen Eingabestrom, sodass zwei Anwendungen mit Hilfe ihrer **Socket**-Objekte bidirektional miteinander kommunizieren können:




Wir beschäftigen uns in diesem Abschnitt mit der Erstellung von Klienten- *und* Serveranwendungen. Ein wesentlicher Unterschied zwischen beiden Rollen besteht darin, dass ein Serverprogramm fest an einen Port gebunden ist und auf dort eingehende Verbindungswünsche wartet, während ein Klientenprogramm nur bei Bedarf aktiv wird und dabei einen dynamisch zugewiesenen Port benutzt.

### 16.4.1 TCP-Klient

Wir erstellen einen TCP-Klienten, der die aktuelle Tageszeit bei einem Daytime-Server erfragt. Der Daytime-Dienst (vgl. RFC 867) eignet sich wegen des extrem einfachen Anwendungsprotokolls für unsere Zwecke, ist aber bei längeren Paketlaufzeiten für die Zwecke der Zeitsynchronisation zu ungenau und wurde daher durch das *Network Time Protocol* (NTP) ersetzt. Meist ist der Zugang zu einem Daytime-Server per Firewall auf Rechner im lokalen Netzwerk restringiert. Wer zum Üben keinen ansprechbaren Daytime-Server findet, sei auf den Abschnitt 16.4.2 vertröstet, wo wir einen eigenen Daytime-Server erstellen.

Für den Daytime-Klienten erzeugen wir ein Objekt aus der Klasse **Socket**, das mit einem Daytime-Server Verbindung aufnehmen soll. Im Konstruktor muss neben dem Host-Namen bzw. der IP-Adresse des Servers auch die Portnummer des Zeitansagers auftauchen. Daytime-Dienste lauschen am TCP-Port 13, z.B.:

```
Socket time = new Socket("uhr.uni-trier.de", 13);
```

Mit **getInputStream()** besorgen wir uns beim **Socket**-Objekt seinen Byte-orientierten Eingabestrom, transformieren diesen per **InputStreamReader** in einen Zeichenstrom und greifen über die bequemen Methoden eines **BufferedReaders** zu:

```
import java.io.*;
import java.net.*;
class DaytimeClient {
   public static void main(String[] args) {
      BufferedReader br = null;
      try {
         Socket time = new Socket("uhr.uni-trier.de", 13);
         System.out.println("Verbindung hergestellt (lokaler Port: "+
              time.getLocalPort()+")");
         time.setSoTimeout(1000);
         br = new BufferedReader(new InputStreamReader(time.getInputStream()));
         System.out.print("Aktuelle Zeit: "+br.readLine());
      } catch (Exception e) {
         e.printStackTrace();
      } finally {
         try {
            br.close();
         } catch (Exception e) { }
      }
   }
}
```

Sofern Netz und Server mitspielen, kann die gewünschte Uhrzeit per **readLine()** ermittelt werden, z.B.:

```
Verbindung hergestellt (lokaler Port: 2690)
Aktuelle Zeit: Thu Feb 18 20:51:25 2010
```

Oft ist es empfehlenswert, für ein **Socket**-Objekt per **setSoTimeout()** eine maximale Wartezeit für das Lesen aus seinem **InputStream** festzulegen, damit das Programm bei Verbindungsproblemen nicht zu lange blockiert wird, z.B.:

```
time.setSoTimeout(1000);
```

Beim Überschreiten der vereinbarten Wartezeit wird eine **SocketTimeoutException** geworfen, auf die das Programm geeignet reagieren kann.

Die Methode **setSoTimeout()** ist relevant bei Lese- und Schreiboperationen, hat keinen Einfluss auf die maximale Wartezeit bei der Verbindungsaufnahme, die im Beispielprogramm per **Socket()**-Konstruktor angefordert wird. Hier ist eine vermutlich plattformabhängige Timeout-Zeit im Spiel, die unter Windows XP ca. 20 Sekunden beträgt. Mit der folgenden Konstruktion lässt sich für die Verbindungsaufnahme eine kürzere Timeout-Zeit unter der Kontrolle des Programmierers erzwingen:

```
Socket time = new Socket();
time.connect(new InetSocketAddress("uhr.uni-trier.de", 13), 1000);
```

Neben einem verbindungslosen **Socket**-Konstruktor und der **Socket**-Methode **connect()** kommt ein Objekt der Klasse **InetSocketAddress** zum Einsatz.

Mit dem **close()**-Aufruf erreicht man das Schließen der Datenstrom-Pipeline und die sofortige Freigabe des lokalen Ports. Das Beispiel soll an die prinzipiell empfehlenswerte Praxis erinnern, wobei ein **close()**-Aufruf für einen Eingabestrom unmittelbar vor dem Programmende eigentlich überflüssig ist.

### 16.4.2 TCP-Server

Als Gegenstück zum eben präsentierten Zeitklienten erstellen wir nun einen Zeitserver, der am TCP-Port 13 lauscht und anfragenden Klienten die aktuelle Tageszeit mitteilt. Ein solcher Server kann in der Entwicklungsphase unter Verwendung der so genannten **loopback-Adresse** 127.0.0.1 durchaus auf dem eigenen Rechner aufgesetzt und von ebenfalls lokal ausgeführten Klientenprogrammen genutzt werden. Unter Windows XP ist der Server allerdings trotz lokaler Adresse nur dann ansprechbar, wenn der Rechner über eine *aktive* Netzwerkverbindung (z.B. per Ethernet oder WLAN) verfügt.

#### *16.4.2.1 Firewall-Ausnahme für einen TCP-Server unter Windows*

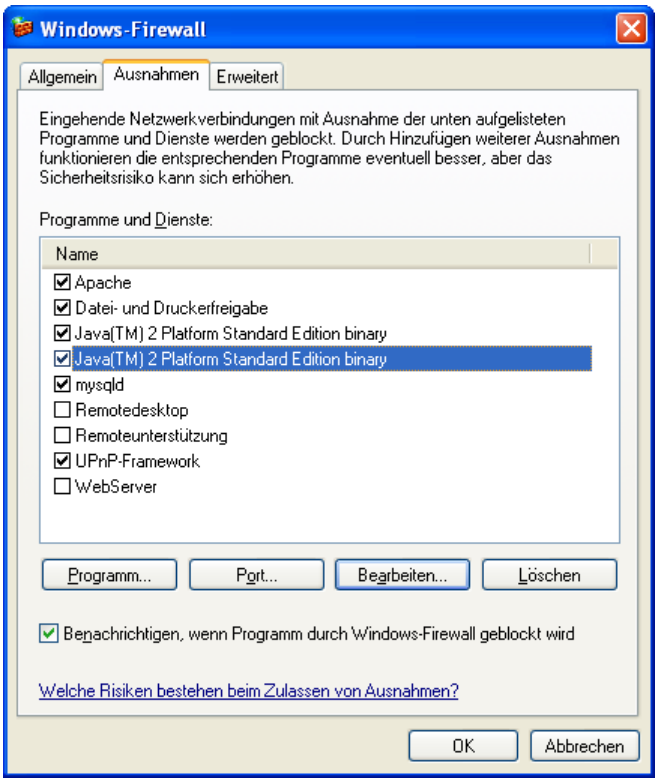
Damit unter Windows ein per **java.exe** oder **javaw.exe** gestartetes Serverprogramm an einen Port tätig werden darf, muss seit der Version XP mit Service Pack 2 eine Firewall-Ausnahme vereinbart werden.

**Windows XP**

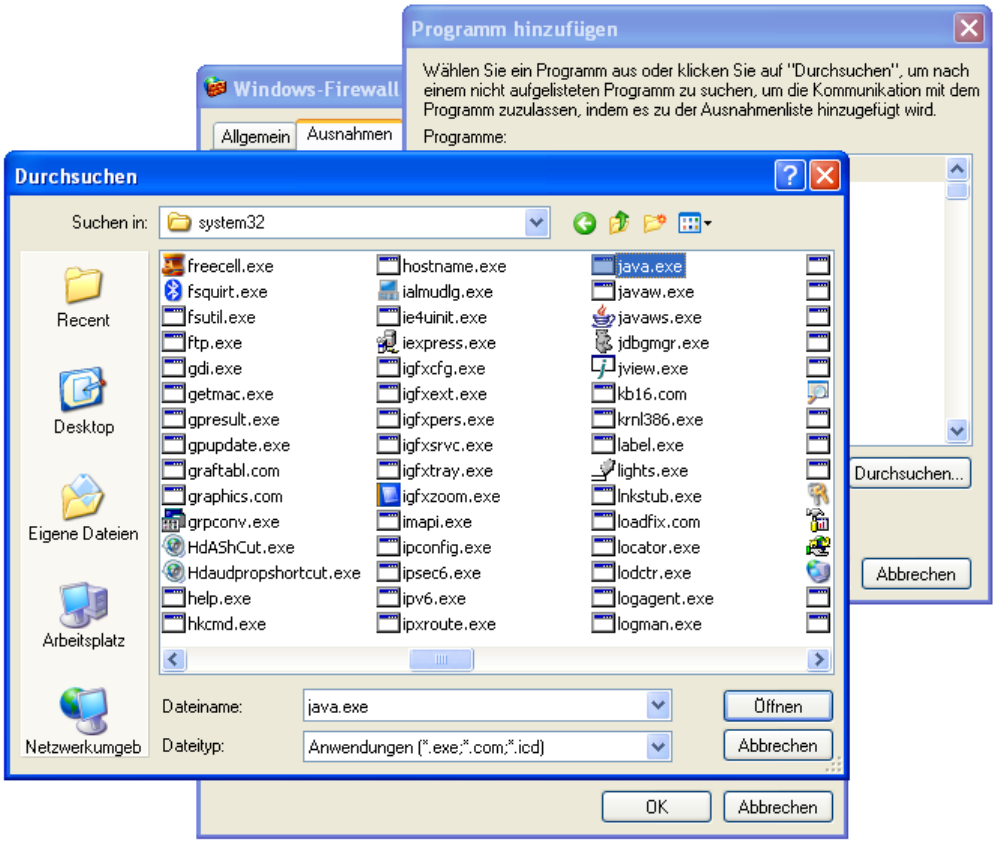
Unter Windows XP öffnet man über

**Systemsteuerung > Firewall**

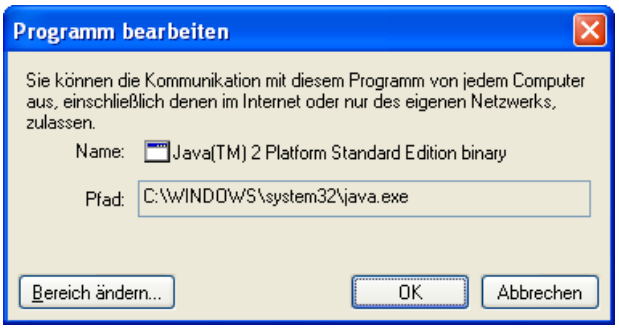
die folgende Dialogbox



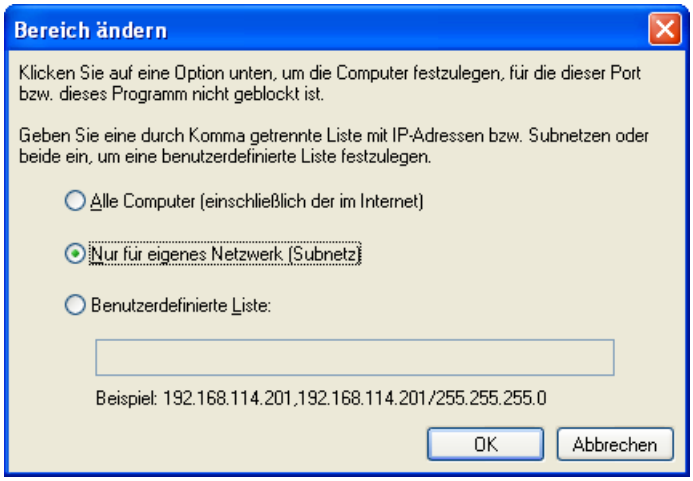

Nach einem Mausklick auf den Schalter **Programm** wählt man einen JRE-Starter (**java.exe** oder **javaw.exe**):




und nimmt ihn in die Ausnahmeliste auf. Im markierten Zustand lässt sich eine Ausnahme **bearbeiten**:

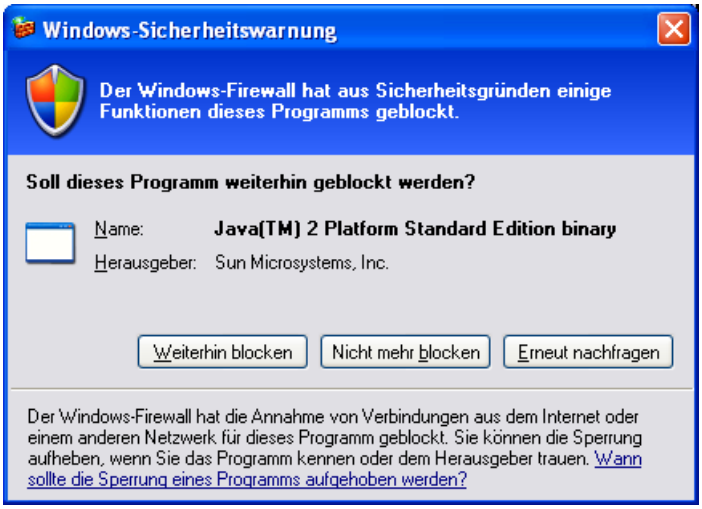



Man sollte den **Bereich** so **ändern**, dass nur vertrauenswürdige Rechner den Serverprozess (also eine beliebige, von der JRE ausgeführte Java-Anwendung!) erreichen können, z.B.:




Eventuell ist es sicherer, einen einzelnen Port freizugeben statt beliebigen Java-Anwendungen (gestartet von **java.exe** bzw. **javaw.exe**) den Dienst an beliebigen Ports zu erlauben.

Statt vorausschauend eine Ausnahme für die Windows-Firewall einzutragen, kann man auf die besorgte Nachfrage des Betriebssystems



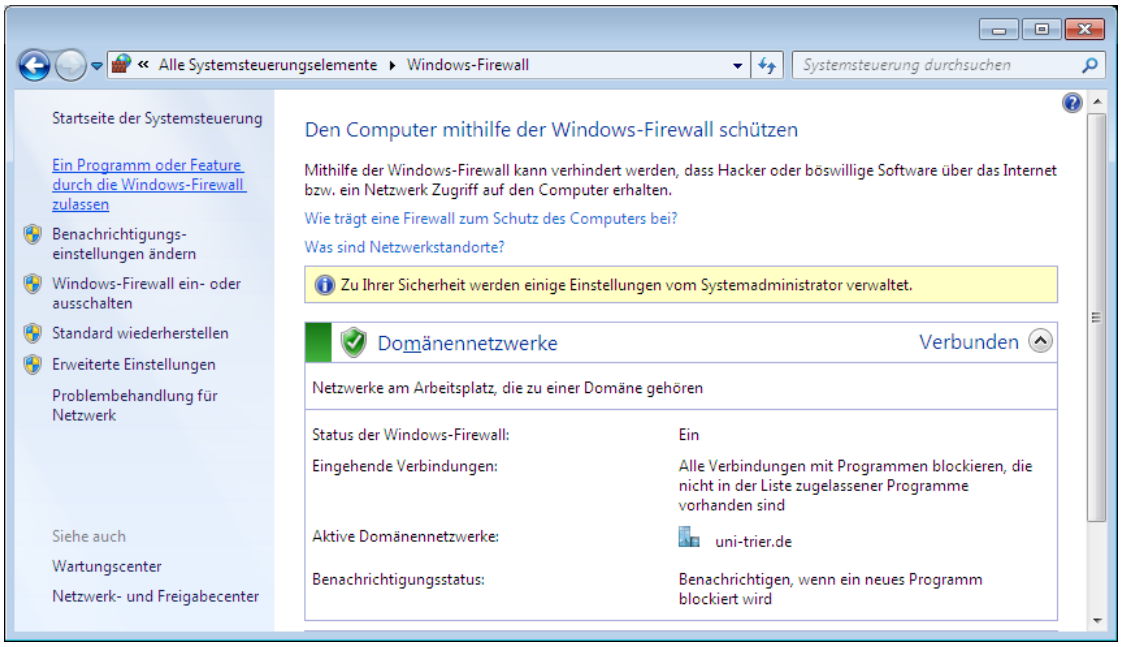

beim Start eines Servers warten und dann mit dem Schalter **Nicht mehr blockieren** reagieren. Eine so aufgenommene Ausnahme kann anschließend gemäß obiger Beschreibung bearbeitet werden.

**Windows 7**

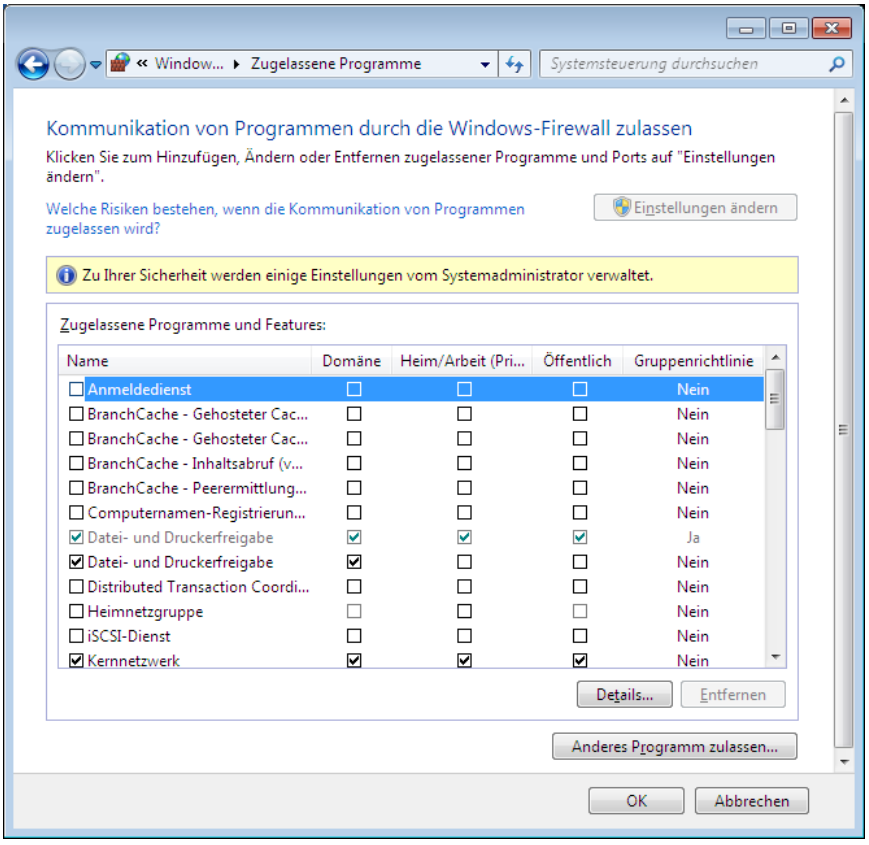
Unter Windows 7 (getestet mit der 64 Bit-Version) öffnet man über

**Systemsteuerung > Windows-Firewall**

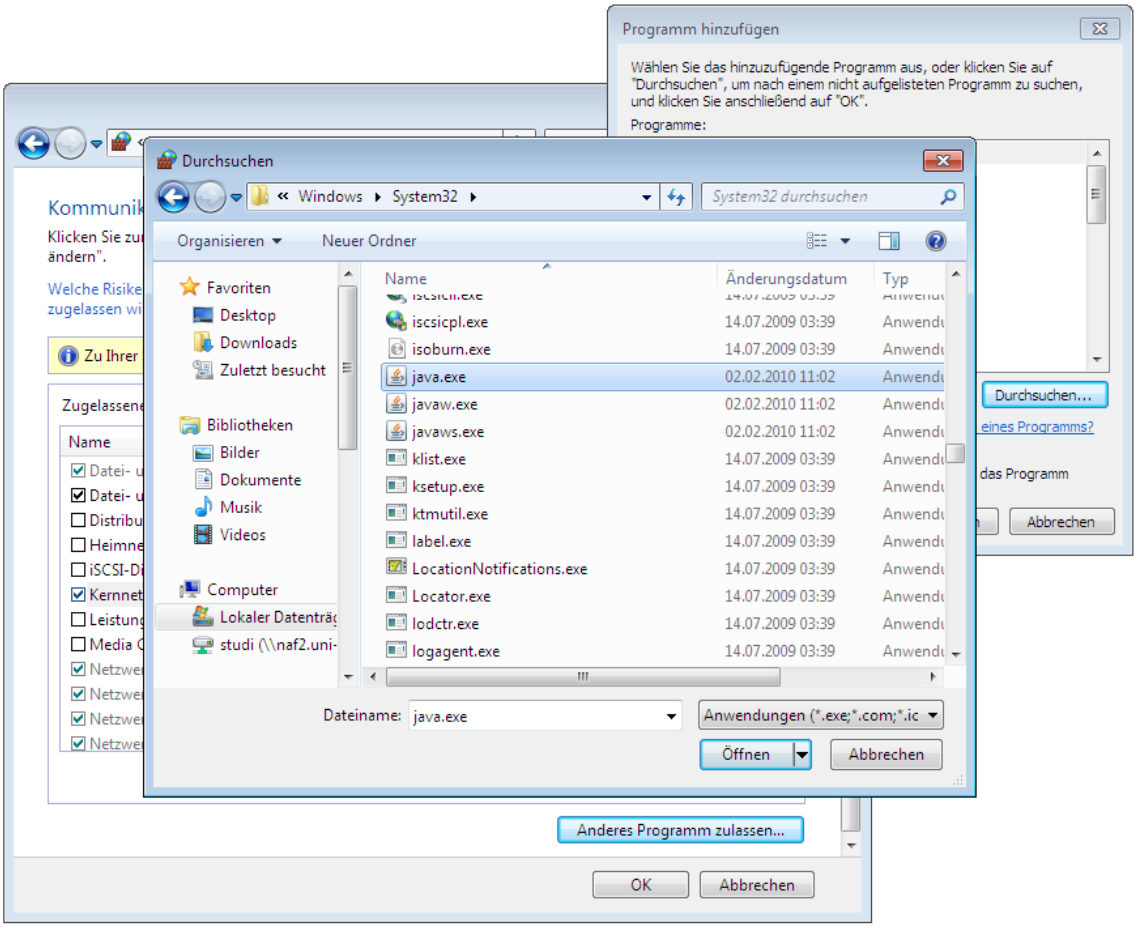
die folgende Dialogbox



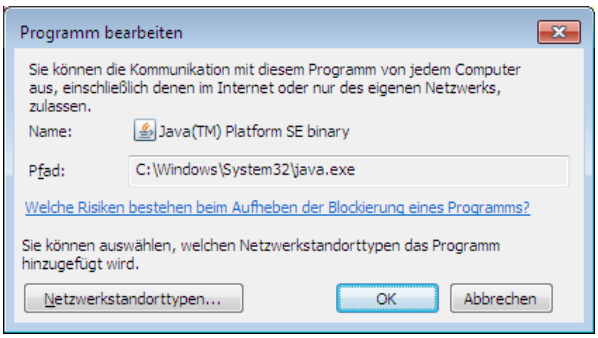

und klickt auf den Link **Ein Programm oder Feature durch die Windows-Firewall zulassen**. Es erscheint die folgende Dialogbox, wo ein Mausklick auf den Schalter **Anderes Programm zulassen** zu setzen ist:



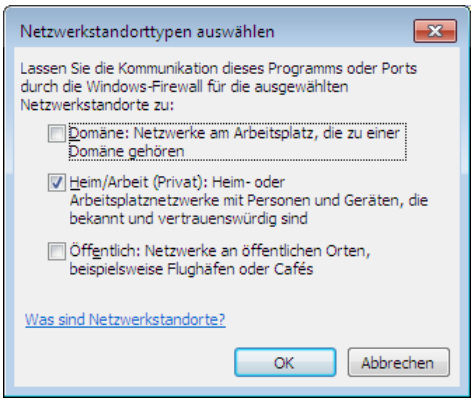

In der Dialogbox mit der Titelzeile **Programm hinzufügen** setzt man einen Mausklick auf den Schalter **Durchsuchen** und wählt schließlich einen JRE-Starter (**java.exe** oder **javaw.exe**):



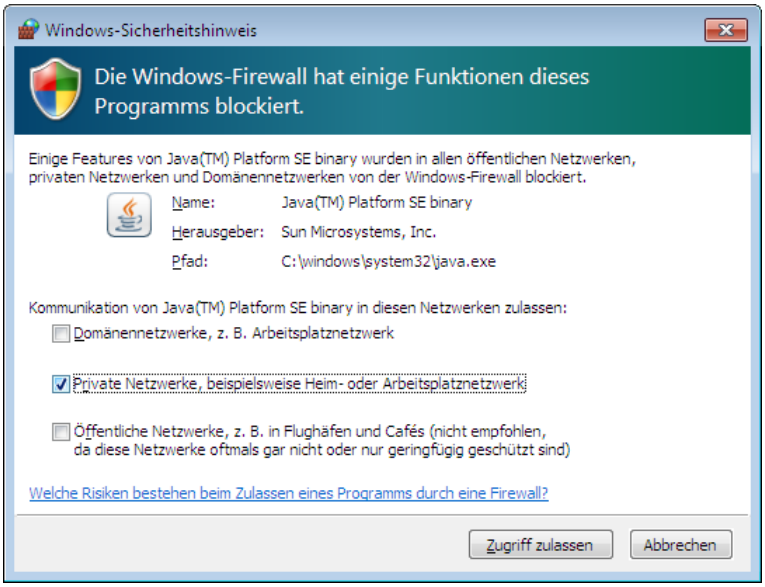

Über die Schalter-Sequenz **Öffnen > Hinzufügen** wird der JRE-Starter in die Liste der zugelassenen Programme aufgenommen. Im markierten Zustand lassen sich **Details** regeln:




Man sollte die **Netzwerkstandorttypen** so ändern, dass nur vertrauenswürdige Rechner den Serverprozess (also eine beliebige, von der JRE ausgeführte Java-Anwendung!) erreichen können, z.B.:



Statt vorausschauend eine Ausnahme für die Windows-Firewall einzutragen, kann man auf die besorgte Nachfrage des Betriebssystems




beim Start eines Servers warten, die zulässigen Kommunikationspartner festlegen und den **Zugriff zulassen**. Eine so aufgenommene Genehmigung kann man später gemäß obiger Beschreibung konfigurieren oder auch entfernen.

### *16.4.2.2 Singlethreading-Server*

Nach diesen Vorbereitungen widmen wir uns wieder dem geplanten Zeitserver. Dabei kommt ein Objekt aus der Klasse **ServerSocket** zum Einsatz, das an den im Konstruktor angegebenen Port 13 gebunden wird.

```
ServerSocket timeServer = new ServerSocket(13);
```

Für jede Klientenverbindung wird ein eigenes Objekt aus der schon bekannten **Socket**-Klasse benötigt. Mit der Methode **accept()** beauftragen wir das **ServerSocket**-Objekt, auf eingehende Verbindungswünsche zu warten und ggf. zu anfragenden Klienten ein **Socket**-Objekt zu liefern:

```
Socket client;
  . . .
client = timeServer.accept();
```

In den Ausgabestrom eines solchen Objekts wird dann die (wie in Abschnitt 16.2.2) mit Hilfe der Klassen **Date** und **DateFormat** erstellte Zeitangabe geschrieben:

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.text.*;

class DaytimeServer {
  public static void main(String[] args) {
    DateFormat df = DateFormat.getDateTimeInstance();
    String zeit;
    Socket client;
    try {
      ServerSocket timeServer = new ServerSocket(13);
      System.out.println("Zeitserver gestartet ("+
        df.format(new Date())+")");
```

```
        while(true) {
          client = timeServer.accept();
          PrintWriter pw = new PrintWriter(client.getOutputStream(), true);
          zeit = df.format(new Date());
          System.out.println("\n"+zeit+ " Anfrage von\n  IP-Nummer: "
            +client.getInetAddress().getHostAddress()
            +" (Port: "+client.getPort()+")");
          pw.println(zeit);
          client.close();
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
}
```

Während unser Zeitklient (siehe Abschnitt 16.4.1) nach einer Anfrage beendet ist, lauscht der Server permanent an Port 13 und bedient (nacheinander) beliebig viele Klienten:

```
Zeitserver gestartet (18.02.2010 23:21:05)

18.02.2010 23:21:18 Anfrage von
  IP-Nummer: 192.168.0.2 (Port: 1142)

18.02.2010 23:21:25 Anfrage von
  IP-Nummer: 192.168.0.7 (Port: 1051)

18.02.2010 23:22:23 Anfrage von
  IP-Nummer: 192.168.0.5 (Port: 1036)
```

Ist ein Klient versorgt, wird das zugehörige **Socket**-Objekt geschlossen:

```
client.close();
```

Es taugt nicht mehr für Netzwerkzwecke und wird abgeschrieben. Die **Socket**-Methode **close()** sorgt auch für das Schließen des Ein- und des Ausgabestroms. Folglich muss ein verbundener **PrintWriter** seinen Puffer rechtzeitig entleeren, was im Beispiel durch die aktivierte **Autoflush**-Option

```
PrintWriter pw = new PrintWriter(client.getOutputStream(), true);
```

und das Verwenden der Methode **println()** sichergestellt wird.

#### *16.4.2.3 Multithreading-Server*

Bei einer ernsthaften Server-Programmierung kommt man an einer Multithreading-Lösung nicht vorbei, damit mehrere Klienten simultan bedient werden können. Dabei nicht für jede Anfrage zeitaufwändig ein neuer Thread gestartet werden muss, sollte ein Threadpool zum Einsatz kommen (siehe Abschnitt 15.6), z.B.:

```
ExecutorService es = Executors.newCachedThreadPool();
```

Wir erstellen nun einen Multithreading - Echo-Server am Port 9999, der alle Sendungen eines Klienten zurückspiegelt, auf die Botschaft `quit` aber mit dem Abbau der Verbindung reagiert. Im Hauptprogramm lauert ein **ServerSocket**-Objekt endlos auf Verbindungswünsche. Wie im letzten Beispiel erzeugt seine **accept()**-Methode für jeden Klientenkontakt ein neues **Socket**-Objekt. Zur Versorgung des Klienten wird außerdem ein Objekt der Klasse `EchoServerThread` generiert und in einem Pool-Thread startet:

```
import java.net.*;
import java.util.*;
import java.util.concurrent.*;
import java.text.*;

class EchoServer {
 private static int nconn, nummer;
 private static DateFormat df = DateFormat.getDateTimeInstance();

 public static void main(String[] args) {
  ExecutorService es = Executors.newCachedThreadPool();
  try {
   ServerSocket echoServer = new ServerSocket(9999);
   System.out.println("Echoserver gestartet ("+df.format(new Date())+")");
   while(true) {
    Socket client = echoServer.accept();
    incr();
    nummer++;
    prot("Verbindung Nummer "+nummer+":\n  IP-Nummer: "
    +client.getInetAddress().getHostAddress()
    +" (Port: "+client.getPort()+")");
    es.execute(new EchoServerThread(client, nummer));
   }
  } catch (Exception e) {
   prot("Fehler: \n  "+e.toString());
  }
 }

 static synchronized void incr() {nconn++;}

 static synchronized void decr(int nummer) {
  nconn--;
  System.out.println("\n"+df.format(new Date())+
   " Verbindung Nummer "+nummer+" beendet");
 }

 static synchronized int getNumActive() {return nconn;}

 static void prot(String s) {
  System.out.println("\n"+df.format(new Date())+" "+s);
 }
}
```

In der Instanzvariablen `nconn` wird die Anzahl der aktiven Verbindungen aufbewahrt. Weil die auf `nconn` schreibend oder lesend zugreifenden `EchoServer`-Methoden in verschiednen Threads ablaufen können, werden sie durch Synchronisieren Thread-sicher gemacht. Bei der Methode `prot()` für Kontrollausgaben ist *keine* Synchronisation erforderlich, weil die **PrintStream**-Methode **println()** für Thread-Sicherheit sorgt:[74]

```
public void println(String x) {
  synchronized (this) {
    print(x);
    newLine();
  }
}
```

In der **run()**-Methode der Klasse `EchoServerThread` findet die Kommunikation über den Eingabe- und Ausgabestrom des **Socket**-Objekts statt. Im Konstruktor des am Ausgang angedockten

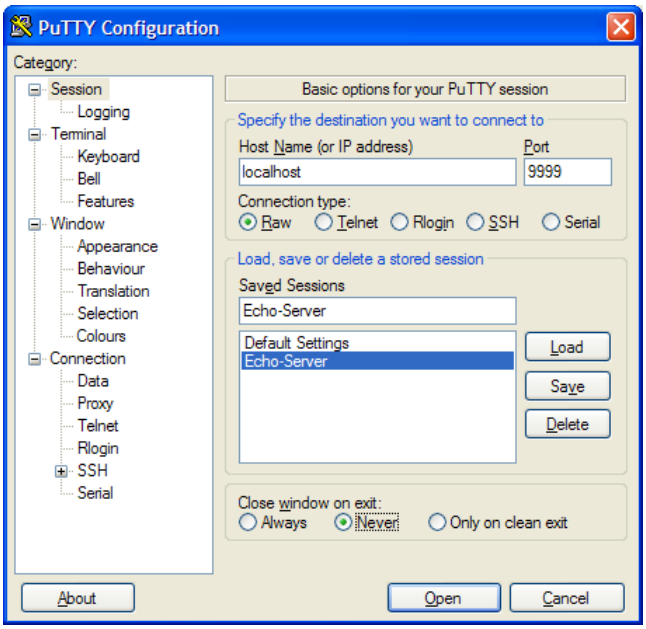
[74] Sie finden diese Definition in der Datei **PrintStream.java**, die wiederum im Archiv **src.zip** mit den API-Quelltexten steckt. Das Quelltextarchiv kann bei der JDK-Installation auf die Festplatte Ihres PCs befördert werden.

**PrintWriter**s wird die **AutoFlush**-Eigenschaft auf **true** gesetzt, so dass er jede abgeschlossene Zeile automatisch aus dem Puffer abschickt.

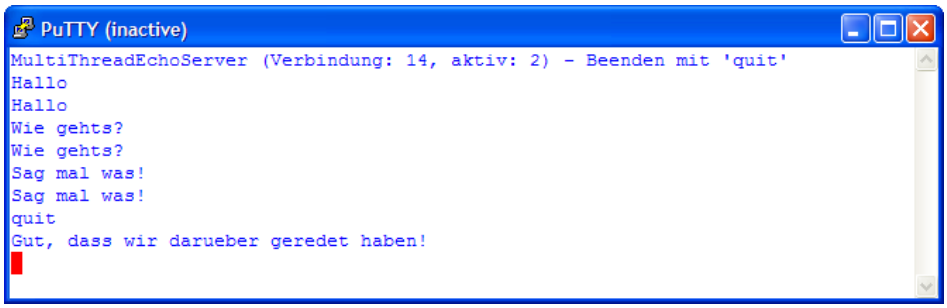
```
import java.io.*;
import java.net.*;

class EchoServerThread extends Thread {
 private Socket client;
 private int nummer;

 EchoServerThread(Socket cl, int nr) {
  client = cl;
  nummer  = nr;
 }
 public void run() {
  String zeile;
  try {
   BufferedReader br = new BufferedReader(
    new InputStreamReader(client.getInputStream()));
   PrintWriter pw = new PrintWriter(client.getOutputStream(), true);
   pw.println("MultiThreadEchoServer (Verbindung: "+nummer+
    ", aktiv: "+EchoServer.getNumActive()
    +") - Beenden mit 'quit'");
   while ((zeile = br.readLine()) != null) {
    if (zeile.trim().equals("quit")) {
    pw.println("Gut, dass wir darueber geredet haben!");
    break;
    }
    pw.println(zeile);
   }
  } catch (Exception e) {
   EchoServer.prot("Fehler: \n  "+e.toString()+" (Verbindung "+nummer+")");
  } finally {
   EchoServer.decr(nummer);
   try {client.close();} catch (Exception ignored) {}
  }
 }
}
```

Man benötigt zum Testen des Multithreading - Echo-Servers keine spezielle Klienten-Software, sondern kann ihn unter Windows z.B. mit dem universellen (u.a. für die Protokolle SSH und Telnet geeigneten) Terminal-Klienten **PuTTY** ansprechen.[75] Lauscht der Echo-Server auf dem lokalen Rechner am Port 999, taugt z.B. die folgende PuTTY-Konfiguration zur Kontaktaufnahme:

[75] Das Windows-Programm ist über die Webseite http://www.putty.org/ kostenlos zu beziehen.




Danach steht ein geduldig wiederholender Gesprächspartner bereit:




Wie die folgende Abbildung zeigt, kann der Server tatsächlich mehrere Klienten simultan versorgen kann:

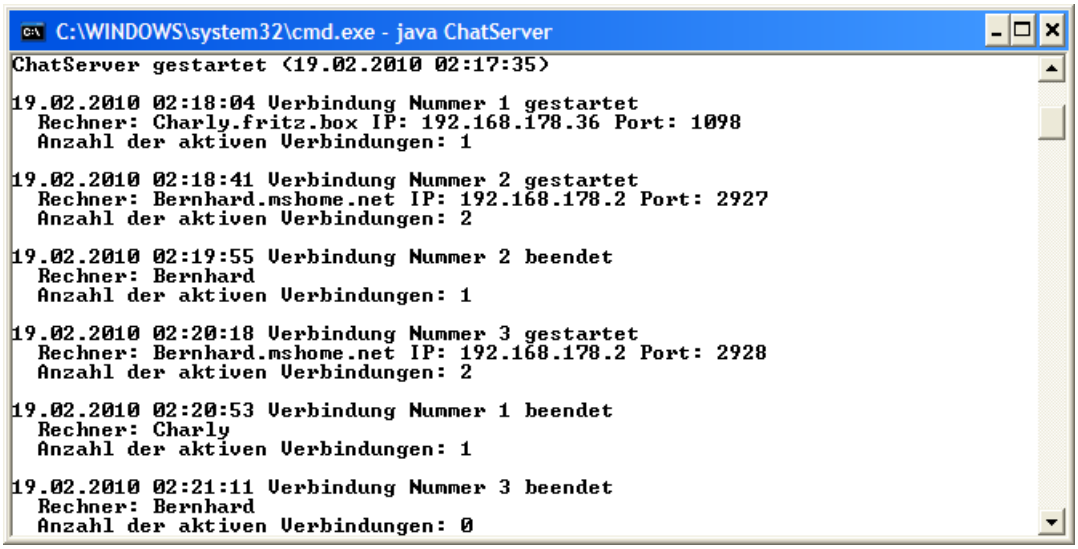

Ein PuTTY-Klient bewährt sich auch in anderen Situationen als Werkzeug für Tests und Fehleranalysen.

## *16.5 Übungsaufgaben zu Kapitel 16*

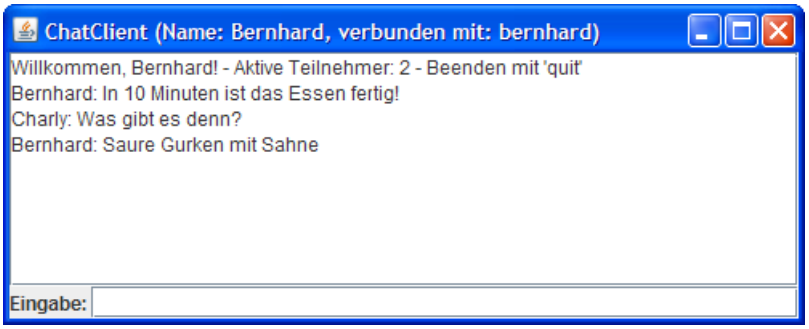
1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Einem TCP-Server wird seine Port-Nummer beim Start zugewiesen.
2. Über ein Objekt der Klasse **InetAddress** lässt sich der Host-Name zu einer IP-Adresse ermitteln.
3. Im OSI-Modell ist das HTTP-Protokoll auf der Ebene 5 einzuordnen.
4. Die **Socket**-Methode **setSoTimeout()** hat keinen Einfluss auf die maximale Wartezeit bei der Verbindungsaufnahme.

2) Erstellen Sie einen Chat-Server, der an Port 9999 lauert, mehrere Klienten simultan bedient und dabei die einkommenden Beiträge an alle Teilnehmer weiterleitet. Einige Protokollausgaben zum Geschehen können auch nicht schaden, z.B.:




Erstellen Sie einen passenden Klienten, z.B.:




Hier erlaubt ein **JTextField**-Steuerelement das Verfassen eigener Beiträge, und die Gesprächsbeiträge erscheinen in einem **JTextArea**-Steuerelement.

# Anhang

## *A. Operatorentabelle*

In der folgenden Tabelle sind alle im Kurs behandelten Operatoren in absteigender Priorität (von oben nach unten) aufgelistet. Gruppen von Operatoren mit gleicher Priorität sind durch fette horizontale Linien begrenzt.

| **Operator** | **Bedeutung** |
|---|---|
| [] | Array-Index |
| () | Methodenaufruf |
| . | Komponentenzugriff |
| ! | Negation |
| ++, -- | Prä- oder Postinkrement bzw. -dekrement |
| - | Vorzeichenumkehr |
| (*Typ*) | Typumwandlung |
| new | Objekterzeugung |
| *, / | Punktrechnung |
| % | Modulo |
| +, - | Strichrechnung |
| + | Stringverkettung |
| <<, >> | Links- bzw. Rechts-Shift |
| >, <, >=, <= | Vergleichsoperatoren |
| instanceof | Typprüfung |
| ==, != | Gleichheit, Ungleichheit |
| & | Bitweises UND |
| & | Logisches UND (mit unbedingter Auswertung) |
| ^ | Exklusives logisches ODER |
| \| | Bitweises ODER |

| Operator | Bedeutung |
|---|---|
| \| | 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. **Falsch**
   Eine Klasse kann Bauplan *und* Akteur sein.
2. **Richtig**
3. **Falsch**
   Jedes Java-Programm muss eine Startklasse enthalten, und eine Startklasse benötigt eine Methode namens **main()**.
4. **Falsch**
   Der vom Java-Compiler erstellte Bytecode muss vom Java-Interpreter in den Maschinencode der aktuellen CPU übersetzt werden.
5. **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**

## Kapitel 3 (Elementare Sprachelemente)

### *Abschnitt 3.1 (Einstieg)*

### Aufgabe 1

| | |
|---|---|
| Der Aufruf klappt: | `public static void main(String[] irrelevant) { … }`<br>Der *Name* des **main()**-Parameters ist beliebig. |
| Der Aufruf scheitert: | `public void main(String[] argz) { … }`<br>Der Modifikator **static** fehlt. |
| Der Aufruf scheitert: | `public static void main() { … }`<br>Falsche Parameterliste |
| Der Aufruf klappt: | `static public void main(String[] argz) { … }`<br>Die Modifikatoren **static** und **public** müssen vor dem Rückgabetyp stehen, wobei ihre Reihenfolge irrelevant ist. |
| Der Aufruf scheitert: | `public static void Main(String[] argz) { … }`<br>Falscher Anfangsbuchstabe im Methodennamen |

### Aufgabe 2

Unzulässig sind:

- `4you`
  Bezeichner müssen mit einem Buchstaben beginnen.
- `else`
  Schlüsselwörter wie **else** sind als Bezeichner verboten.

### Aufgabe 3

Das Paket **java.lang** der Standardbibliothek wird automatisch in jede Quellcodedatei importiert (vgl. Abschnitt 3.1.6).

### *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 als Zeichenfolge behandelt, 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 {`<br>`  public static void main(String[] args) {`<br>`  System.out.println("3.3 + 2 = " + (3.3 + 2));`<br>`  }`<br>`}` | `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-Escapesequenz für das Zeichen *c* zugewiesen:

```
char z = '\u0063';
```

Der dezimalen Zahl 99 entspricht die hexadezimale Zahl 0x63 (= 6 · 16 + 3).

**Aufgabe 3**

Durch das Suffix `l` im Literal `7l` wird der Datentyp **long** verlangt, und ein **long**-Wert kann wegen des größeren Wertebereichs nicht implizit in einen **int**-Wert gewandelt werden.

**Aufgabe 4**

Die Variable `i` ist nur im innersten Block gültig.

**Aufgabe 5**

```
class Prog {
  public static void main(String[] args) {
    System.out.println("Dies ist ein Java-Zeichenkettenliteral:\n   \"Hallo\"");
  }
}
```

**Aufgabe 6**

```
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 Konsolenanwendungen)*

**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("c = "+ c);
    System.out.print("\nNun bitte eine gebrochene Zahl (mit Dezimalkomma!): ");
    double d = Simput.gdouble();
    System.out.printf("d = %f", d);
  }
}
```

### *Abschnitt 3.5 (Operatoren und Ausdrücke)*

**Aufgabe 1**

| Ausdruck | Typ | Wert | Anmerkungen |
|---|---|---|---|
| `6/4*2.0` | **double** | `2.0` | Abarbeitung mit Zwischenergebnissen:<br>**`6/4`**`*2.0`<br>**`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.<br>Abarbeitung mit Zwischenergebnissen:<br>**`(int)6`**`/4.0*3`<br>**`6/4.0`**`*3`<br>**`1.5*3`** |
| `(int)(6/4.0*3)` | **int** | `4` | Abarbeitung mit Zwischenergebnissen:<br>`(int)(`**`6/4.0`**`*3)`<br>`(int)(`**`1.5*3`**`)`<br>**`(int)4.5`** |
| `3*5+8/3%4*5` | **int** | `25` | Abarbeitung mit Zwischenergebnissen:<br>**`3*5`**`+8/3%4*5`<br>`15+`**`8/3`**`%4*5`<br>`15+`**`2%4`**`*5`<br>`15+`**`2*5`**<br>**`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 $2 \neq 3$ ist.
- `8 % 3` ergibt 2.

`erg2` erhält den Wert Null, 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 (>, ==) stehen in der Priorität über den logischen Operatoren, 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 wohl 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.

**Aufgabe 2**

| Ausdruck | Typ | Wert | Anmerkungen |
|---|---|---|---|
| `1/(1/0.0)+5` | **double** | `5.0` | Abarbeitung mit Zwischenergebnissen:<br>`1/(`**`1/0.0`**`)+5`<br>**`1/Infinity`**`+5`<br>`0.0+5` |
| `1/(1/0)+5.0` | | | Die Ganzzahldivision durch Null führt zu einem Laufzeitfehler. |
| `1/(0/0.0)+5.0` | **double** | `NaN` | `(0/0.0)` ergibt keine Zahl (NaN). |

***Abschnitt 3.7 (Anweisungen)***

**Aufgabe 1**

Weil die **else**-Klausel der *zweiten* **if**-Anweisung zugeordnet wird, ergibt sich folgender Gewinnplan:

| `losNr` | Gewinn |
|---|---|
| durch 13 teilbar | Nichts |
| nicht durch 13, aber durch 7 teilbar | 1 € |
| weder durch 13, noch durch 7 teilbar | 100 € |

**Aufgabe 2**

Im logischen Ausdruck der **if**-Anweisung findet an Stelle eines Vergleichs eine *Zuweisung* statt.

**Aufgabe 3**

In der **switch**-Anweisung wird es versäumt, per **break** den „Durchfall“ zu verhindern.

**Aufgabe 4**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Elementare Sprachelemente\PrimitivOB**

**Aufgabe 5**

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. Die restlichen Zeilen werden als selbständiger Anweisungsblock gedeutet, aber nie ausgeführt.

**Aufgabe 6**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Elementare Sprachelemente\DM2EuroS**

**Aufgabe 7**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Elementare Sprachelemente\FloP**

Bei $i = 52$ erhält man letztmals das mathematisch korrekte Vergleichsergebnis. Dies ist gerade die Anzahl der Bits in der **double**-Mantisse.

Die Zahl 1,0 hat nach der Norm IEE 754 die folgende normalisierte **double**-Darstellung (vgl. Abschnitt 3.3.4.1)

$$(-1)^0 \cdot 2^{(1023-1023)} \cdot (1+0{,}0)$$

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 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

| 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:

### Aufgabe 8

Lösungsvorschläge mit den beiden Algorithmus-Varianten befinden sich in den Ordnern:

**...\BspUeb\Elementare Sprachelemente\GGT.Diff**
**...\BspUeb\Elementare Sprachelemente\GGT.Mod**

### Aufgabe 9

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 ist darauf zu achten, dass es nicht zum Ganzzahlüberlauf kommt (vgl. Abschnitt 3.6.1):

| Quellcode | Ausgabe |
|---|---|
| `public class Prog {`<br>`  public static void main(String[] args) {`<br>`  long start = System.currentTimeMillis();`<br>`  long mrd = 1000000000;`<br>`  System.out.println("Summe = " +`<br>`  (mrd*(mrd+1)/2) + "\nZeit  = " +`<br>`  (System.currentTimeMillis() - start));`<br>`  }`<br>`}` | `Summe = 500000000500000000`<br>`Zeit  = 0` |

## Abschnitt 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 Auslesen des Werts
- *Verzicht* auf eine **public**-Methode zum Verändern des Werts

**Aufgabe 3**

1. **Falsch**
   Es kann durchaus sinnvoll sein, private Methoden für den ausschließlich klasseninternen Gebrauch zu definieren.
2. **Richtig**
3. **Falsch**
   Der Rückgabetyp spielt bei der Signatur keine Rolle.
4. **Falsch**
   Typkompatibilität genügt, d.h. die Aktualparametertypen müssen sich erweiternd (vgl. Abschnitt 3.5.7) in die Formalparametertypen konvertieren lassen.
5. **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 Null verschiedenen Aktualparameterwert gegen diese Regel.

**Aufgabe 5**

Zum Informationstransport hat eine Methode folgende Möglichkeiten:

- Rückgabewert
  Weil auch Klassen als Rückgabetypen zugelassen sind, können auf diesem Weg auch komplexe Informationspakete transportiert werden.
- Parameter mit Referenztyp
  Ein referenziertes Objekt besteht mit den bei seinen Instanzvariablen vorgenommenen Änderungen nach dem nach dem Ende der Methode weiter.
- Änderung von Klassenvariablen

Eine Änderung der Umgebung kann so erfolgen:

- Objekt erstellen und Referenz übergeben
  Die Referenz auf ein methodenintern erstelltes Objekt kann per Rückgabewert oder Referenzparameter übergeben werden.
- Änderung eines per Referenzparameter ansprechbaren Objekts
  Ein referenziertes Objekt besteht mitsamt den dort vorgenommenen Änderungen nach dem nach dem Ende der Methode weiter.
- Änderung von Klassenvariablen

**Aufgabe 6**

Die beiden Methoden können *nicht* in einer Klasse koexistieren, weil ihre Signaturen identisch sind:

- gleiche Länge
- an beiden Position identische Parametertypen

Dass an jeder Position die beiden typgleichen Formalparameter verschiedene Namen haben, ist irrelevant.

**Aufgabe 7**

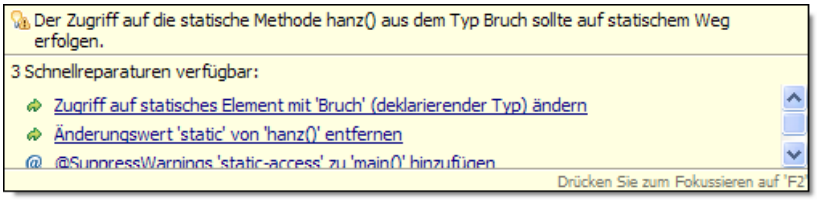
1. **Falsch**
   Der Standardkonstruktor hat dieselbe Schutzstufe wie die Klasse.
2. **Richtig**
3. **Falsch**
   Eine Methode kann eine Referenz auf ein von ihr erzeugtes Objekt an die Außenwelt übergeben (z.B. per Rückgabewert). Ein Objekt wird erst dann überflüssig und ein potentielles Garbage Collector - Opfer, wenn im gesamten Programm keine Referenz mehr auf das Objekt vorhanden ist.
4. **Leider falsch**
   Bedauerlicherweise gelingt in Methoden berechtigter Klassen der Aufruf 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 ersetzt werden (mit der lokalen `Bruch`-Referenzvariablen `b1`):

   ```
   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:

   In C# ist der Aufruf von Klassenmethoden über eine Instanzreferenz sinnvollerweise verboten.

**Aufgabe 8**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Klassen und Objekte\TimeDuration**

**Aufgabe 9**

| 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 10**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Klassen und Objekte\FakulRek**

Das Programm eignet sich übrigens recht gut dazu, durch Wahl eines hinreichend großen Arguments einen Stapelüberlauf (**StackOverflowError**) zu provozieren.

**Aufgabe 11**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Klassen und Objekte\R2Vec**

**Aufgabe 12**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Swing\VE\PrimDiagVE**

## Abschnitt 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 Zeichenketten)*

**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 eignet sich die Klasse **StringBuffer**.

**Aufgabe 2**

Der interne **String**-Pool wird erweitert durch die Anweisungen mit den Kommentarnummern (1) und (5).

**Aufgabe 3**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Elementare Klassen\Zeichenketten\PerZuf**

**Aufgabe 4**

Die Klasse **StringBuffer** 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 {`<br>`  public static void main(String[] args) {`<br>`  System.out.println("Min. byte-Zahl:\n " + Byte.MIN_VALUE);`<br>`  }`<br>`}` | `Min. byte-Zahl:`<br>` -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**

**Aufgabe 3**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Elementare Klassen\Wrapper\Mint**

**Aufgabe 4**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Elementare Klassen\Zeichenketten\StringUtil**

***Abschnitt 5.4 (Aufzählungstypen)***

**Aufgabe 1**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Elementare Klassen\Enumerationen\Wochentage**

**Kapitel 6 (*Typgenerizität und Kollektionen*)**

**Aufgabe 1**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Pakete\Generische Typen und Methoden\GenMax**

**Aufgabe 2**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Pakete\Kollektionsklassen\DataMat**

**Aufgabe 3**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Pakete\Kollektionsklassen\Mengenlehre**

**Kapitel 7 (*Pakete*)**

**Aufgabe 1**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Pakete\DemoPack**

**Aufgabe 2**

Die beteiligten Klassen befinden sich im selben Verzeichnis und sind keinem Paket explizit zugeordnet, so dass sie sich im (anonymen) Standardpaket befinden. Klassen eines Paketes haben per Voreinstellung wechselseitig volle Rechte.

**Aufgabe 3**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Pakete\PrimDiag**

Zum Erstellen des Archivs mit dem JDK-Werkzeug **jar.exe** eignet sich das Kommando:

```
>jar cmf0 PDManifest.txt PrimDiag.jar *.class
```

Mit Hilfe der fertigen Archivdatei lässt sich die Klasse `PrimDiag` folgendermaßen starten:

```
>javaw -jar PrimDiag.jar
```

Ist auf einem Rechner die aktuelle JRE der Firma Sun Microsystems installiert, sollte auch der Start per Doppelklick klappen.

### Abschnitt 8 (*Vererbung und Polymorphie*)

#### Aufgabe 1

In der Klasse `General` fehlt ein parameterfreier Konstruktor.

#### Aufgabe 2

In der Klasse `Figur` haben `xpos` und `ypos` den voreingestellten Zugriffsschutz (**package**). Weil `Kreis` nicht zum selben Paket gehört, hat diese 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

Die erste Antwort ist richtig. 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 ohne die Namenserweiterung **.class** an, z.B.:

```
>javap -c Kreis
```

### Abschnitt 9 (*Ausnahmebehandlung*)

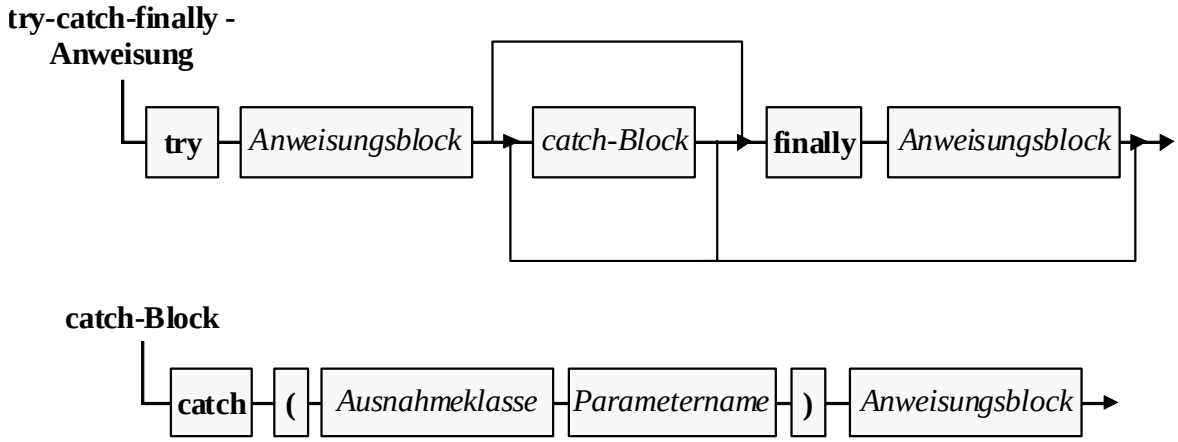
#### Aufgabe 1

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 JRE 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 weiter geht.

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

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Ausnahmebehandlung\DoWhileBad**

### Aufgabe 4

Lösungsvorschlag:

```
class Prog {
  public static void main(String[] args) {
    Object o = null;
    System.out.println(o.toString());
  }
}
```

### Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Ausnahmebehandlung\DuaLog\IllegalArgumentException**

## Abschnitt 10 (*Interfaces*)

### Aufgabe 1

1. **Falsch**
2. **Richtig**
3. **Falsch**
   Das in Abschnitt 10.1 vorgeführt API-Interface **Serializable** enthält keine Methoden.
4. **Richtig**

5. **Falsch**
   Interface-Methoden sind grundsätzlich verpflichtend. In der Dokumentation zum API-Interface **java.util.Collection<E>** ist bei etlichen Methoden der Zusatz *optional operation* zu finden. Hier darf man keinesfalls *optional implementation* lesen (siehe Abschnitt 10.1).

### Aufgabe 2

Ein Methodendefinitionskopf kann trotz mehrfacher Verpflichtungen innerhalb einer Klasse nur einmal implementiert werden, so dass Unklarheiten beim Aufruf ausgeschlossen sind.

### Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Interfaces\Bruch implements Comparable**

### Aufgabe 4

Bisher sind uns im Kurs u.a. die folgenden generischen Interfaces begegnet:

- In Abschnitt 6.4 (*Java Collection Framework*)
  **Collection<E>**, **List<E>**, **Set<E>**, **Comparator<E>**, **Map<K,V>**
- In Kapitel 10 (*Interfaces*)
  **Comparable<T>**

### Aufgabe 5

Eine das Interface **Comparator<T>** implementierende Klasse muss die beiden folgenden Instanzmethoden besitzen:

- **public int compare(T o1, T o2)**
  Diese Methode vergleicht zwei Objekte vom Typ **T**, wobei die Rückgabe analog zur **Comparable<T>** - Methode **compareTo()** folgende Bedeutung haben muss:
  ein negativer Wert

  | | |
  |---|---|
  | negativ | das erste Objekt ist kleiner |
  | 0 | die beiden Objekte sind gleich |
  | positiv | das erste Objekt ist größer |

- **public boolean equals(Object o)**
  Mit dieser Methode meldet ein Comparator, ob er funktional äquivalent zum Parameter-Comparator ist. Eine taugliche Methode ist schon in der Urahnklasse **Object** vorhanden.

### Aufgabe 6

Beim Aufruf der folgenden Methode

```
static <T extends Basis & SayOne & SayTo> void moin(T x) {
  for (int i = 0; i < x.ib; i++) {
    x.sayOne();
    x.sayTo();
  }
}
```

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**

## Abschnitt 11 (*GUI-Programmierung mit Swing*)

### Aufgabe 1

1. **Falsch**
   Siehe Abschnitt 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**

### 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 Interessent 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**<br>(Genau genommen geht es um die Inhaltsschicht eines **JFrame**-Objekts.) | **java.awt.BorderLaout** |
| **javax.swing.JPanel** | **java.awt.FlowLaout** |
| **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\EuroKonverter**

**Aufgabe 11**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Swing\Zwischenablage**

**Abschnitt 12 (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**

**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 **Fehler! Verweisquelle konnte nicht gefunden werden.**). 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 Kombi-Rückgabewert. Bei unerwarteten Problemen wirft **read()** hingegen eine **IOException**.

**Aufgabe 3**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Swing\Editor\E5 (mit Sichern)**

**Aufgabe 4**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\EA\BufferedOutputStream\Konsole**

**Aufgabe 5**

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 vom **PrintWriter** automatisch vorgenommene Pufferung wird außer Kraft gesetzt. Für das Programm ist folgender Konstruktoraufruf besser geeignet:

```
PrintWriter pw = null;
   . . .
pw = new PrintWriter(fos);
```

**Aufgabe 6**

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 (0x0D, 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 12.5 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 7**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\EA\ASCII-Text**

**Aufgabe 8**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\EA\DataMatrix**

**Abschnitt 13 (Applets)**

**Aufgabe 1**

1. **Richtig**
2. **Richtig**
3. **Richtig**
4. **Falsch**

**Aufgabe 2**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Applets\PrimApplet**

**Aufgabe 3**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Applets\EuroKonverter**

**Abschnitt 14 (Multimedia)**

**Aufgabe 1**

1. **Richtig**
2. **Falsch**
   Bei schnell ablaufenden Veränderungen ist die direkte Ausgabe sinnvoll, wobei das per **paint()** bzw. **paintComponent()** ausgegebene Modell in der Regel ebenfalls aktuell gehalten werden muss.
3. **Falsch**
   Das Java2D-API unterscheidet logische und geräteabhängige Koordinaten mit einer übersetzenden Transformation, verwendet aber per Voreinstellung die identische (unwirksame) Transformation.
4. **Falsch**
   **Font**-Objekt können generell überhaupt nicht geändert werden. Für die Farbe ist nicht die Klasse **Font** zuständig, sondern der Grafikkontext.

**Aufgabe 2**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Multimedia\Grafik\Wuerfel**

**Aufgabe 3**

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Multimedia\Sound\Hintergrundmusik**

### Abschnitt 15 (Multithreading)

#### Aufgabe 1

1. **Falsch**
2. **Falsch**
   Ein Programm endet, wenn alle Benutzer-Threads beendet sind. Die Daemon-Threads (z.B. für den Garbage Collector) werden dann automatisch beendet.
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.

#### Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Multithreading\Thread-Prioritäten**

Vorsicht: Lässt man beide Threads mit hoher Priorität laufen, reagiert die graphische Benutzeroberfläche eventuell sehr zäh.

### Abschnitt 16 (Netzwerkprogrammierung)

#### Aufgabe 1

1. **Falsch**
   Server benötigen eine feste Port-Nummer, die den Klienten schon vor der Verbindungsaufnahme bekannt sein muss.
2. **Richtig**
3. **Falsch**
   Das HTTP-Protokoll gehört zur Schicht 7 (Anwendung).

4. **Richtig**
   Es ist aber ein **Socket**-Konstruktor mit Timeout-Parameter für die Verbindungsaufnahme vorhanden.

### Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

**...\BspUeb\Netzwerk\TCP-Programmierung\Chat**

## Literatur


Ball, J., Carson,D. B., Evans, I., Haase, K & Jendrock, E. (2006). *The Java EE 5 Tutorial*. Santa Clara, CA: Sun Microsystems.

Baltes-Götz, B. (2009). *Einführung in das Programmieren mit C# 3.0*. Online-Dokument: http://www.uni-trier.de/index.php?id=22777

Balzert, H. (1999). *Lehrbuch der Objektmodellierung: Analyse und Entwurf*. Heidelberg: Spektrum.

Bracha, G. (2004). *Generics in the Java Programming Language*. Online-Dokument: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

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.

Fowler, A. (2003). *Painting in AWT and Swing*. Online-Dokument: http://java.sun.com/products/jfc/tsc/articles/painting/index.html

Goll, J., Weiß, C. & Rothländer, P. (2000). *Java als erste Programmiersprache*. Stuttgart: Teubner.

Gosling, J., Joy, B., Steele, G. L. & Bracha, G. (2005). The Java Language Specification (3rd ed.). Online-Dokument: http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf

Hennebrüder, S. (2007). *Hibernate. Das Praxisbuch für Entwickler*. Bonn: Galileo.

Horstmann, C. S. & Cornell, G. (2002). *Core Java. Volume II – Advanved Features*. Palo Alto, CA: Sun Microsystems Press.

Kröckertskothen, T. (2001). *Java 2. Grundlagen und Einführung*. Hannover: RRZN.

Krüger, G. & Stark, T. (2007). *Handbuch der Java-Programmierung* (Version 5.0.2). Online-Dokument: http://www.javabuch.de/download.html

Künneth, T. (2009). *Einstieg in Eclipse 3.5*. Bonn: Galileo.

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. (2006). *Praxisbuch Objektorientierung. Professionelle Entwurfsverfahren*. Bonn: Galileo

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#. Eine 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

Münz, S. (2007). *SELFHTML 8.1.2*. Online-Dokument: http://aktuell.de.selfhtml.org/extras/download.shtml

MySQL AB (2008). *MySQL 5.0 Reference Manual*. Online-Dokument http://dev.mysql.com/doc/

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.

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). *The Java Tutorial*. Online-Dokument: http://java.sun.com/docs/books/tutorial/

Ullenboom, C. (2009). *Java ist auch eine Insel* (8. Aufl.). Bonn: Galileo. OpenBook: http://www.galileocomputing.de/openbook/javainsel8/

# Index

**T**