Universitäts-
Rechenzentrum

Universität Trier

Bernhard Baltes-Götz

Einführung in das objektorientierte Programmieren mit Java 5.0

2007.12.06

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
Leiter: Dr. Peter Leinen
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) für Studierende aus allen Fachbereichen anbietet, ist aber auch zum Selbststudium geeignet.

Java

Die von der Firma Sun 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.
Neben der Java 2 Standard Edition (J2SE) zur Entwicklung von lokal installierter Software für Arbeitsplatzrechner, auf die viele weltweit populäre Softwarepakete setzen (z.B. Matlab, SPSS), gibt es sehr erfolgreiche Java-Varianten für unternehmensweite oder serverorientierte Lösungen (Java 2 Enterprise Edition, J2EE) sowie für Kommunikationsgeräte mit beschränkter Leistung (Java 2 Micro Edition, J2ME).
Wir beschäftigen uns überwiegend mit der Java-Basistechnologie (J2SE), jedoch werden im Abschnitt über serverseitige Weblösungen auch wichtige Bestandteile der J2EE vorgestellt.

Lernziele

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 und –Servlets für den Einsatz im Internet behandelt.

Voraussetzungen bei den Leser(innen)

- EDV-Allgemeinbildung
  Dass die Leser(innen)[1] wenigstens durchschnittliche Erfahrungen bei der *Anwendung* von Computerprogrammen haben sollten, versteht sich von selbst.
- Programmierkenntnisse
  Programmierkenntnisse werden *nicht* vorausgesetzt. Leser *mit* Programmiererfahrung werden sich bei den ersten Abschnitten eventuell etwas langweilen.
- Motivation
  Generell ist mit einem erheblichen Zeitaufwand bei der Lektüre und bei der aktiven Auseinandersetzung mit dem Stoff (z.B. durch das Lösen von Übungsaufgaben) zu rechnen.
- EDV-Plattform
  Weil im Kurs ausschließlich Java-Software zum Einsatz kommt, ist die Plattformunabhängigkeit garantiert, obwohl in den Bildschirmphotos meist PCs unter MS-Windows zu sehen sind.

Software zum Üben

Für die unverzichtbaren Übungen sollte ein Rechner zur Verfügung stehen, auf dem das **J2SE Development Kit** der Firma Sun in einer Version ab 5.0 mitsamt der zugehörigen Dokumentation installiert ist. Als integrierte Entwicklungsumgebung zum komfortablen Erstellen von Java-Software wird **Eclipse** in einer Version ab 3.0 empfohlen. Jedoch wird auch die für ältere Rechner besser geeignete Umgebung **JCreator LE** in einer Version ab 3.0 durch Übungsaufgaben und Bedienungshinweise unterstützt.
Die genannte Software ist kostenlos verfügbar, mit Ausnahme des Windows-Programms JCreator LE für alle signifikanten Plattformen (Windows, Linux, MacOS, UNIX). Nähere Hinweise zum Bezug, zur Installation und zur Verwendung der Software folgen in Abschnitt 0.

---

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

Manuskript und Dateien zum Kurs

Die aktuelle Version dieses Manuskripts ist als PDF-Datei zusammen mit den behandelten Beispielen und Lösungsvorschlägen zu den 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

Alle Beispiel- und Übungsprojekte sind für die Entwicklungsumgebungen Eclipse 3.x (Ordner **BspUeb**) und JCreator LE 3.x (Ordner **BspUebJC**) vorhanden.

Leider fehlt dem Kurs noch ein Abschnitt über die mit Java 5 hinzugekommene *Generizität*, und einige neu ins Manuskript aufgenommenen Abschnitte (*Netzwerk*, *Datenbankanbindung*, *serverseitige Weblösungen*) haben wohl noch kein zufrieden stellendes Niveau erreicht. Fehlermeldungen und Verbesserungsvorschläge an die E-Mail-Adresse baltes@uni-trier.de werden dankbar entgegen genommen.

Trier, im Mai 2006 Bernhard Baltes-Götz

Inhaltsverzeichnis

1 Einleitung

1.1 Beispiel für die objektorientierte Softwareentwicklung mit Java

In diesem Abschnitt soll eine Vorstellung davon vermittelt werden, was ein Computerprogramm (in Java) ist, und wie man es erstellt. 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. Dazu muss das Programm ...

- den betroffenen Gegenstandsbereich modellieren,
- Algorithmen realisieren, die in endlich vielen Schritten und unter Verwendung von endlich vielen Ressourcen (z.B. Speicher) bestimmte Ausgangszustände in akzeptable Zielzustände überführen.

Wir wollen präzisere und komplettere Definitionen zum komplexen Begriff eines Computerprogramms den Lehrbüchern überlassen (siehe z.B. Echtle & Goedicke 2000, Goll et al. 2000) und stattdessen ein Beispiel 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 eigentlich nichts vorhanden.
- Repräsentative Java-Programme sind wegen ihrer Länge und Komplexität (aus der Sicht des Anfängers) als Einstiegsbeispiel nicht gut geeignet.

Im folgenden Beispielprogramm wird trotz angestrebter Einfachheit *nicht* auf objektorientiertes Programmieren (OOP) verzichtet. Seine Aufgabe besteht darin, für Brüche mit ganzzahligem Zähler und Nenner elementare Operationen 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. Dabei versucht man, alle beteiligten **Objekt-Sorten** zu identifizieren und definiert für sie jeweils eine **Klasse**. Diese ist gekennzeichnet durch

- **Eigenschaften** (**Variablen**)
  Technisch handelt es sich bei einer Variablen um einen benannten Speicherplatz, der Werte bestimmten Typs (z.B. Zahlen, Zeichen) aufnehmen kann. 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 bereits erzeugten Brüche).
- **Handlungskompetenzen** (**Methoden**)
  In den Methoden sind die oben angesprochenen Algorithmen realisiert.

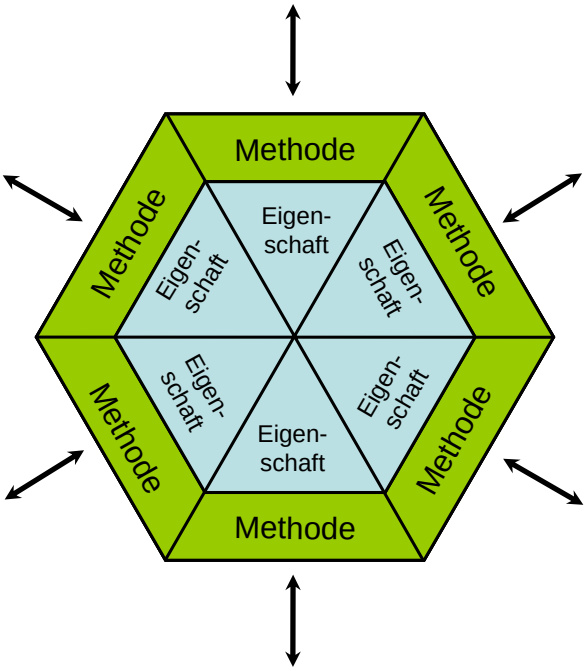
Dass jedes Objekt gleich in eine Klasse („Schublade“) gesteckt wird, mögen die Anhänger einer ausgeprägt individualistischen Weltanschauung bedauern. Auf einem geeigneten Abstraktionsniveau betrachtet lassen sich jedoch die meisten Objekte der realen Welt ohne großen Informationsverlust in Klassen einteilen.
In unserem einfachen Beispiel 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 Variablen (Java-Datentyp **int**) repräsentiert:

- `zaehler`
- `nenner`

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 erläutenden Gründen die Produktivität der Softwareentwicklung durch Datenkapselung gefördert.

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



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

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 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 kon-

sequente 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:

- **Sicherheit**
  Die Eigenschaften sind vor unsinnigen und gefährlichen Zugriffen geschützt, wenn Veränderungen nur über die vom Klassendesigner entworfenen Methoden möglich sind.
- **Produktivität durch perfekte Modularisierung**
  Bei der Entwicklung großer Softwaresysteme unter Beteiligung zahlreicher Programmierer ermöglicht die Datenkapselung eine perfekte Modularisierung. Der Klassendesigner trägt die volle Verantwortung dafür, dass die von ihm entworfenen Methoden korrekt arbeiten. Andere Programmierer müssen beim Verwenden der Klasse lediglich die Methoden kennen. Das Innenleben einer Klasse kann vom Designer nach Bedarf geändert werden, ohne dass andere Programmbestandteile angepasst werden müssen.

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

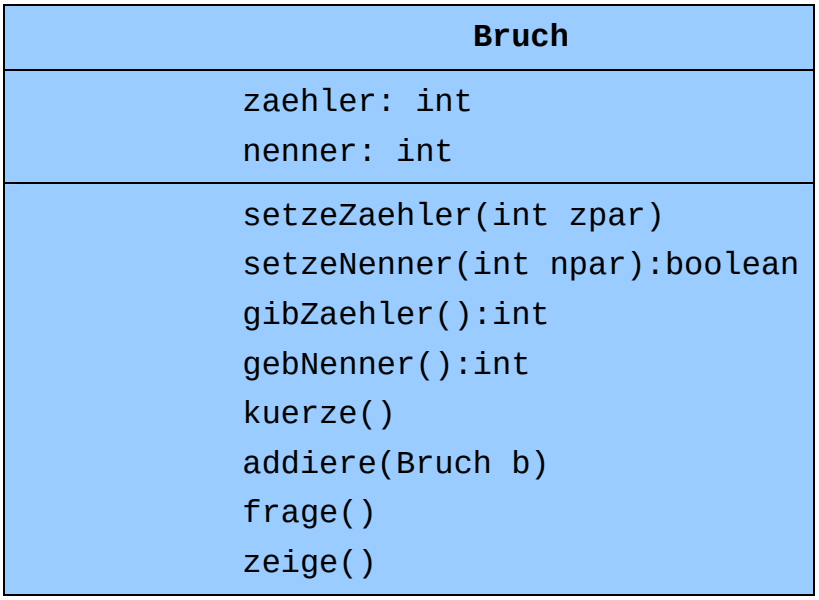
- `setzeZaehler(int zpar)`, `setzeNenner(int npar)`
  Das Objekt wird beauftragt, seinen `zaehler` bzw. `nenner` auf einen bestimmten Wert zusetzen. Ein direkter Zugriff auf die Eigenschaften soll fremden Klassen nicht erlaubt sein (Datenkapselung). Bei dieser Vorgehensweise kann das Objekt z.B. verhindern, dass sein Nenner auf Null gesetzt wird.
- `gibZaehler()`, `gebNenner()`
  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.
- `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) steuern kann.
- `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.

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 (Variablen).
  Hinter dem Namen einer Eigenschaft gibt man ihren Datentyp (siehe unten) an.

- Unten stehen die Methoden (Handlungskompetenzen).
  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.

1.1.2 Objektorientierte Programmierung

In unserem einfachen Beispielprojekt soll nun die `Bruch`-Klasse in der Programmiersprache Java kodiert werden, wobei die Eigenschaften (Variablen) 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 dieser Klassendefinition selbstverständlich jetzt noch fremd, doch sind die Variablendeklarationen und Methodenimplementationen als zentrale Bestandteile leicht zu erkennen:

```
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;
      }
    }

    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.status);
      do {
        System.out.print("Nenner : ");
        n = Simput.gint();
        if (n == 0 && Simput.status)
          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);
    }
  }
```

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).

1.1.3 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 im Vergleich zur Auseinandersetzung mit Java als Sprache und Klassenbibliothek.

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

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

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

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

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

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

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

Folglich ist t auch ein Teiler von v:

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

Weil die Paare (u, v) und (u, u - v) dieselbe Menge gemeinsamer Teiler besitzen, sind auch die größten gemeinsamen Teiler identisch.

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 3.6 werden Sie eine erheblich effizientere Variante implementieren.

1.1.4 Startklasse und main()-Methode

Bislang wurde im Anwendungsbeispiel aufgrund einer objektorientierten Analyse des Aufgabenbereichs die Klasse `Bruch` entworfen und in Java realisiert. Daraus lassen sich im Programmablauf selbständige, eigenverantwortliche Objekte erstellen, die etliche Methoden beherrschen und nach Eintreffen entsprechender Nachrichten ausführen.
Um auf möglichst einfache Weise mit Hilfe der `Bruch`-Klasse ein ablauffähiges Programm zu erstellen, benötigt man eine Art Skript, das `Bruch`-Objekte erzeugt und diesen Objekten Befehle zustellt, die (zusammen mit dem Verhalten des Anwenders) den Programmablauf voranbringen.
In klassischer Terminologie könnte man hier vom *Hauptprogramm* reden. Aus objektorientierter Perspektive wird die Hauptrolle jedoch von der `Bruch`-Klasse gespielt, weil große Teile der im Aufgabenbereich erforderlichen Handlungskompetenz (z.B. Bruch erfragen oder kürzen) in die Klassendefinition eingegangen sind.
Das angesprochene Skript zur Ablaufsteuerung kann in Java auf unterschiedliche Weise realisiert werden. Eine typische Lösung besteht darin, eine eigene Klasse zu definieren, die aber nicht als Bauplan für Objekte dient, sondern eine so genannte *Klassenmethode* namens **main()** zur Ablaufsteuerung enthält. Wie Sie bald im Detail erfahren, wird ein Java-Programm grundsätzlich über die **main()**-Klassenmethode einer beteiligten Klasse in Gang gesetzt.
In unserem Beispiel wäre es durchaus möglich, die `Bruch`-Klasse um eine solche Methode zu erweitern. Indem wir eine *andere* Klasse zum Starten verwenden, wird u.a. gleich demonstriert, wie

leicht das Hauptergebnis unserer Arbeit (die `Bruch`-Klasse) für thematisch verwandte Projekte genutzt werden kann.

Im Beispiel soll die *Startklasse* den Namen `BruchAddition` erhalten. In ihrer **main()**-Methode werden 2 Objekte (Instanzen) aus der Klasse `Bruch` erzeugt und mit dem Ausführen verschiedener Methoden beauftragt:

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 zwei Klassen mit folgender Aufgabenverteilung 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) besitzen **und** alle erforderlichen Instanzvariablen (Attribute) mitbringen.
  - Beim Umgang mit den `Brüchen` sind wenige Probleme zu erwarten, weil nur klasseneigene Methoden Zugang zu kritischen Attributen 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 aufgerufen wird und einen einfachen Algorithmus zur Addition von zwei Brüchen unter Verwendung von `Bruch`-Objekten realisiert. Man kann sich vorstellen, dass die Klasse `BruchAddition` beim Programmstart die Regie übernimmt, als wichtige Helfer zwei Objekte der Klasse `Bruch` erzeugt und diesen Objekten Aufträge zur Bewältigung der einzelnen Teilaufgaben erteilt.
  Mit einer Wiederverwendbarkeit des `BruchAddition`-Quellcodes in anderen Projekten ist kaum zu rechnen.

In der Regel bringt man den Quellcode jeder Klasse in einer eigenen Textdatei 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. Es ist erlaubt, aber nicht empfehlenswert, den Quellcode der Klass `BruchAddition` ebenfalls in der Datei **Bruch.java** unterzubringen.

Wie aus den vorgestellten Klassen ein ausführbares Programm entsteht, erfahren Sie in Abschnitt 0. Wer sich schon jetzt von der Nützlichkeit unseres Bruchrechnungsprogramms überzeugen möchte, findet eine ausführbare Version an der im Vorwort angegebenen Stelle im Ordner

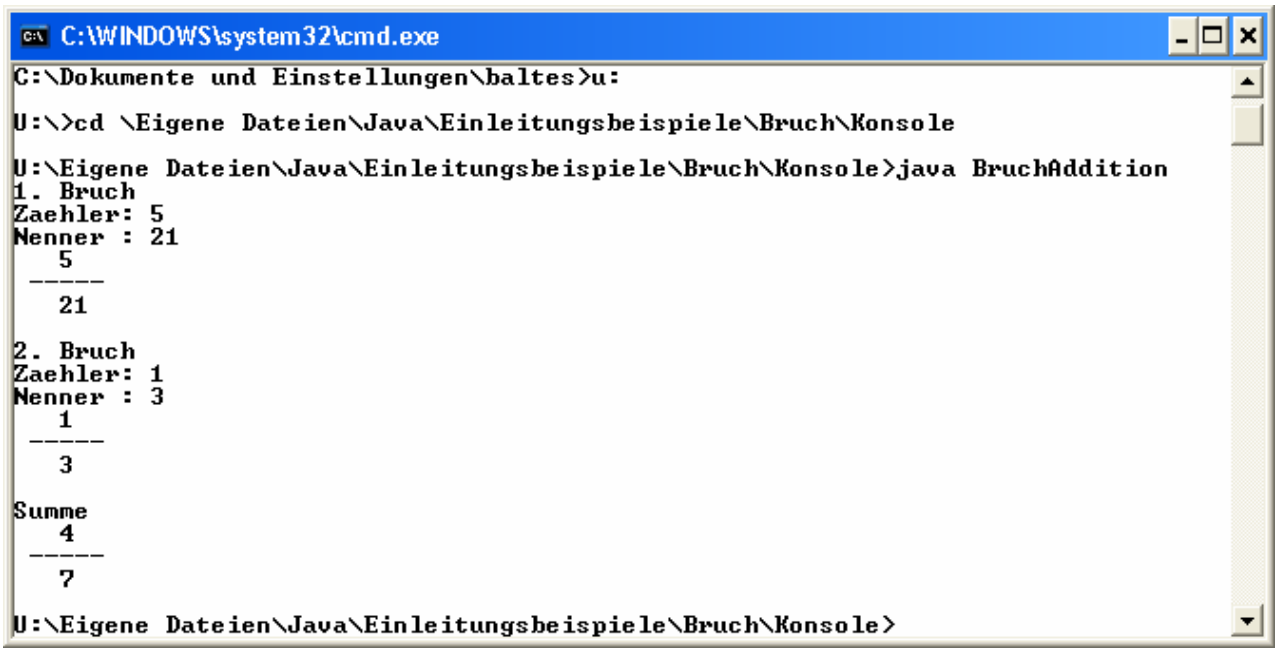
...\BspUeb\Einleitungsbeispiele\Bruch\Konsole

und kann bei Bedarf die *beiden* **class**-Dateien auf einen eigenen Datenträger kopieren.
Weil die Klasse `Bruch` wie viele weitere Beispielklassen mit konsolenorientierter Benutzerinteraktion die (nicht zum Java-API gehörige) Klasse `Simput` benötigt, 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 **BruchAddition.class** zu starten:

- Öffnen Sie ein Konsolenfenster, z.B. mit
  Start > Programme > Zubehör > Eingabeaufforderung
- Wechseln Sie zum Ordner mit den **class**-Dateien, z.B.:
  ```
  u:
  cd \Eigene Dateien\Java\Einleitungsbeispiele\Bruch\Konsole
  ```
- Starten Sie das Programm unter Beachtung der Groß/Kleinschreibung beim Klassennamen mit
  ```
  java BruchAddition
  ```

Ab jetzt sind lästige Bruchadditionen kein Problem mehr:



1.1.5 Ausblick auf Anwendungen mit graphischer Benutzerschnittstelle

Das obige Beispielprogramm arbeitet der Einfachheit halber mit einer konsolen-orientierten Ein- und Ausgabe. Nachdem wir in dieser übersichtlichen Umgebung grundlegende Sprachelemente erarbeitet haben, werden wir uns 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 unsere Entwicklungsumgebung Eclipse die Erstellung graphischer Benutzeroberflächen durch einen visuellen Editor erleichtert.

Zum Ausprobieren startet man aus dem Ordner

...\BspUeb\Einleitungsbeispiele\Bruch\GUI

die Klasse **BruchAdditionGui.class**, die sich auf etliche andere Klassen stützt, die folglich im selben Ordner anwesend sein müssen, z.B.:

```
java BruchAdditionGui
```

In diesem Abschnitt 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.

1.2 Quellcode, Bytecode und Maschinencode

Eben haben Sie Java als eine Programmiersprache kennen gelernt, die Ausdrucksmittel zur Modellierung von Anwendungsbereichen bzw. zur Formulierung von Algorithmen bereitstellt. Unter einem *Programm* wurde dabei der vom Entwickler zu formulierende *Quellcode* verstanden.
Während Sie derartige Programme bald mit Leichtigkeit lesen und begreifen werden, kann die CPU eines Rechners nur einen *speziellen* Satz von Befehlen verstehen, die als Folge von Nullen und Einsen (= *Maschinencode*) formuliert werden müssen. Die ebenfalls CPU-spezifische Assemblersprache stellt eine für Menschen lesbare Form des Maschinencodes dar. Mit dem Assembler- bzw. Maschinenbefehl

```
mov eax, 4
```

wird z.B. der Wert 4 in das EAX-Register (ein Speicherort im Prozessor) geschrieben.
Die CPU holt sich einen Maschinenbefehl nach dem anderen aus dem Hauptspeicher und führt ihn aus, heutzutage immerhin mehrere hundert Millionen Befehle pro Sekunde.
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 auf eine besondere Weise:

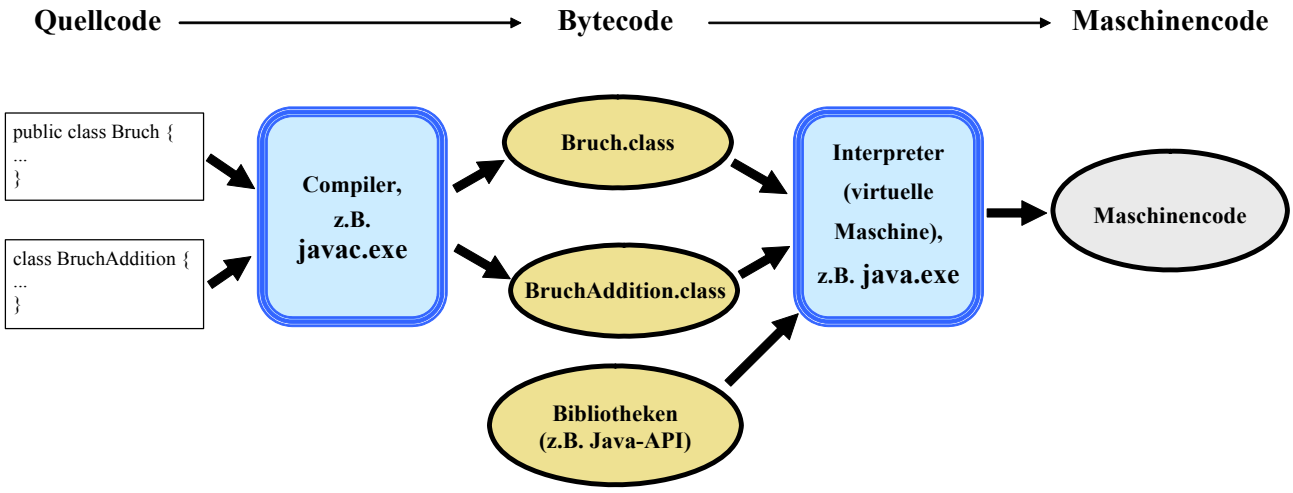
Kompilieren: Quellcode → Bytecode

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 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.
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 (z.B. das Werkzeug **java** aus der J2SE). 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.
Mittlerweile kommen bei der Ausführung von Java-Programmen leistungssteigernde Techniken (**J**ust-**i**n-**T**ime – Compiler, **HotSpot** – Compiler mit Analyse des Laufzeitverhaltens) zum Einsatz, welche die Bezeichnung *Interpreter* fraglich erscheinen lassen. Allerdings ändert sich nichts an der Aufgabe, aus dem plattformunabhängigen Bytecode den zur aktuellen Hardware passenden Maschinencode zu erzeugen. So wird wohl keine Verwirrung gestiftet, wenn in diesem Manuskript weiterhin vom *Interpreter* die Rede ist.

In der folgenden Abbildung sind die Dateinamen zum `Bruch`-Beispiel und außerdem die Namen des Compilers **javac.exe** und des Interpreters **java.exe** aus dem Java 2 Software Development Kit (SDK) eingetragen:



1.3 Java als Programmiersprache und als Klassenbibliothek

Damit die Programmierer nicht das Rad (und ähnliche Dinge) ständig neu erfinden müssen, bietet Java in seinem **API** (**A**pplication **P**rogram **I**nterface) eine große Bibliothek mit fertigen Klassen für nahezu alle Routineaufgaben. Im Abschnitt über Pakete werden die wichtigsten API-Bestandteile grob skizziert. An eine systematische Behandlung ist des enormen Umfangs wegen nicht zu denken.

An dieser Stelle soll geklärt werden, 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 wurden in der Regel durch eine Erweiterung der Java-Klassenbibliothek realisiert. Hier gab es erhebliche Änderungen, so dass die Firma Sun seit der Bibliotheksversion 1.2 offiziell von der **Java 2** – Plattform spricht.
Einige Klassen sind mittlerweile schon als **deprecated** (überholt, 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 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 ersetzen. So arbeiten wir also derzeit mit der Java 2 Plattform, Version 5.0, alias 1.5.

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

1.4 Herkunft und zentrale Merkmale der Java-Technologie

Die Programmiersprache Java wurde ab 1990 von einem Team der Firma Sun unter Leitung von James Gosling entwickelt (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 Installation im Fenster des lokalen 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 drei Editionen für spezielle Einsatzfelder definiert:

- **Java 2 Standard Edition (J2SE)** zur Entwicklung von Anwendersoftware
  Darauf wird sich unser Kurs im Wintersemester 2005/2006 beschränken.
- **Java 2 Enterprise Edition (J2EE)** für unternehmensweite oder serverorientierte Lösungen
  Nach derzeitiger Planung findet im Sommersemester 2006 ein Fortsetzungskurs statt, der sich auch mit der J2EE beschäftigt.
- **Java 2 Micro Edition (J2ME)** 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 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 weiteren 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* (internen, gekapselten Datenelementen) 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, mit dem Anwender und mit anderen Systembestandteilen 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. In den Abschnitten 0 und 0 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 Portabilität von Java resultiert vor dem Hintergrund der in Abschnitt 1.2 beschriebenen Übersetzungsprozedur daraus, dass sich Bytecode-Interpreter relativ gut für aktuelle Rechner-Plattformen implementieren lassen.
Man mag einwenden, dass sich der Quellcode vieler Programmiersprachen (z.B. C++) auf verschiedenen Rechnerplattformen kompilieren lässt. Diese Quellcode-Portabilität aufgrund kompatibler Sprachdefinitionen 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 **Standardlaufzeitumgebung** mit ihren insgesamt ca. 3000 Klassen weit reichende Fähigkeiten für die Gestaltung graphischer Benutzerschnittstellen, für Datenbank- und Netzwerkzugriffe usw., so dass plattformunabhängige Anwendungen mit modernem Funktionsumfang und Design realisiert werden können. Einer Java-Anwendung stehen zudem durch Verwendung Unicode-Technologie auf allen Plattformen dieselben Zeichensätze zur Verfügung.
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 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)
- erratisches Programmverhalten beim Zugriff auf voreilig entsorgte Objekte

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

---

[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.

Mittlerweile gibt es etliche Java-Entwicklungsumgebungen, die auch eine graphische Gestaltung von Benutzeroberflächen analog zu Visual Basic oder Delphi erlauben, 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 exzellentes Werkzeug zur Gestaltung visueller Klassen (mit Auftritt auf dem Bildschirm). Einen ähnlichen Funktionsumfang bieten auch andere Entwicklungsumgebungen wie Suns's *Netbeans* oder der *JBuilder* von Borland.

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.

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, CGI-Aufrufe per URL, **RMI** (**R**emote **M**ethod **I**nvocation) oder **SOAP** (**S**imple **O**bject **A**ccess **P**rotocol) erfolgen. Diese Begriffe müssen Sie übrigens jetzt nicht verstanden haben. Wenn sie später im Kurs relevant werden, folgt eine Erklärung.

Performanz

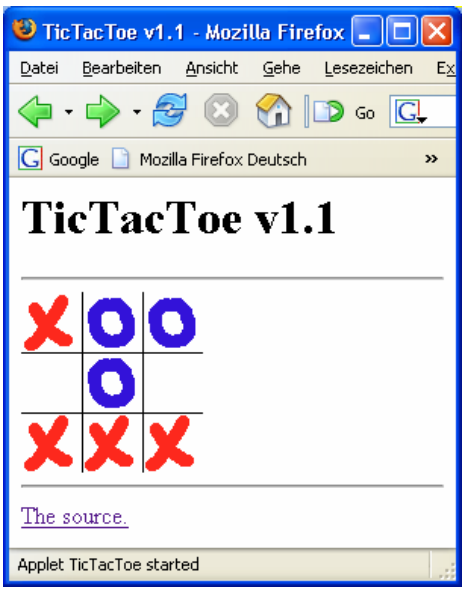
Der durch Sicherheit (Bytecode-Verifikation), Stabilität (z.B. Garbage Collector) und Portabilität verursachte Performanznachteil von Java-Programmen (z.B. gegenüber C++) ist durch die Entwicklung leistungsfähiger virtueller Java-Maschinen mittlerweile weitgehend irrelevant geworden, wenn es nicht gerade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

1.5 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 den ersten Abschnitten dieses Kurses eine Durststrecke überstehen. Da hilft vielleicht ein Blick auf die im JDK 5.0 enthaltenen Applet-Demos. Wurde das JDK in den Ordner **C:\Programme\Java\jdk1.5.0** installiert, sind die Demos im Ordner

C:\Programme\Java\jdk1.5.0\demo\applets

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



Auch die Java-Quellcodedateien sind verfügbar, so dass dem Lernen durch Nachahmen und Experimentieren nichts im Wege steht.
Außerdem sind die Applet-Demos auch auf der Webseite

http://java.sun.com/applets/jdk/1.4/index.html

zu finden.

Wer neben Java auch noch Wahrscheinlichkeitstheorie und Statistik lernen möchte, wird auf folgender 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.:

Dieses Applet startet in einem eigenen Fenster.

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) noch sehr viele Arbeitserleichterungen und Profiwerkzeuge bietet (z.B. einen visuellen Designer zur Gestaltung der Benutzeroberfläche).

Erfüllt ihr Rechner nicht die unten angegebenen Voraussetzungen für die recht voluminöse Eclipse-IDE (*Integrated Development Environment*), stellt die sehr schlanke und ebenfalls kostenlos verfügbare Java-Entwicklungsumgebung **JCreator LE** eine brauchbare Alternative dar. Neben einem guten Editor (z.B. mit Syntaxhervorhebung) werden einige Arbeitserleichterungen geboten (z.B. ein Klasseninspektor). Dass kein visueller GUI-Designer vorhanden ist, stellt in unserem Einführungskurs, der über weite Strecken konsolenorientiert arbeiten wird, kein großes Problem dar. Im Anhang finden Sie Informationen zum Bezug und zur Bedienung des Programms.

2.1 JDK und Eclipse installieren

Auf der Begleit-CD zum Kurs finden Sie etliche Programmpakete, die auf Ihrem Arbeitsplatzrechner unter Windows zu installieren sind, damit Sie aktiv am Kurs teilnehmen und Spaß am Programmieren haben können. Installieren Sie die Pakete in der Reihenfolge ihres Auftretens in der folgenden Tabelle:

JDK 1.5.0 (alias 5.0) Update5	Das JDK der Firma Sun enthält u.a. die zum Ausführen von Bytecode erforderliche JRE (Java Runtime Environment, JVM), den Java-Compiler **javac**, zahlreiche Werkzeuge (z.B. den Dokumentationsgenerator **javadoc** und den Archivgenerator **jar**), etliche Demos sowie den Quellcode der Klasse im Kern-API. Voraussetzungen: • Windows 98 (SE), Windows ME, Windows 2000 (SP3+), Windows XP (SP1+) • ca. 250 MB Festplattenspeicher Pfad auf der Begleit-CD: \Software\Java\J2SE\JDK SE 5.0 Update 5 URL zum Herunterladen: http://java.sun.com/j2se/1.5.0/download.jsp Installation: • Doppelklick auf **jdk-1_5_0_05-windows-i586-p.exe** • Lizenz akzeptieren und alle Vorschläge übernehmen

FilZip 3.0.4	Bei Bedarf können Sie dieses Hilfsprogramm zum Erstellen und Auspacken von Archiven installieren und anschließend für die ZIP-Dateien verwenden. Nach meiner Erfahrung arbeitet die in Windows XP integrierte ZIP-Funktionalität sehr langsam. Pfad auf der Begleit-CD: \Software\Sonstiges\FilZip 3.0 URL zum Herunterladen: http://www.filzip.com/ Installation: • Doppelklick auf **fz304.exe** • Lizenz akzeptieren und alle Vorschläge übernehmen
Dokumentation zum JDK 1.5.0 (alias 5.0)	Diese systematische Dokumentation zu allen Klassen im Java-API ist bei der Erstellung von Java-Software unverzichtbar. Voraussetzungen: • ca. 250 MB Festplattenspeicher Pfad auf der Begleit-CD: \Software\Java\J2SE\JDK SE 5.0 Update 5\Dokumentation URL zum Herunterladen: http://java.sun.com/j2se/1.5.0/download.jsp Installation: Archiv **jdk-1_5_0-doc.zip** im JDK-Ordner extrahieren, so dass z.B. der Ordner **C:\Programme\Java\jdk1.5.0_05\docs** entsteht Vorgehen mit FilZip: o Kontextmenü zur ZIP-Datei öffnen o **FillZip > Extrahieren nach** o **Ziel:** C:\Programme\Java\jdk1.5.0_05
Eclipse SDK 3.1.1	Wir installieren zunächst das Basispaket zu unserer Entwicklungsumgebung Eclipse, dem später noch mehrere Plugins folgen. Voraussetzungen: • Windows (98, ME, 2000, XP), Linux, Mac OS X, UNIX • JRE ab 1.4.2 Weil wir bereits das JDK 5.0 (alias 1.5) installiert haben, ist auch eine passende JRE vorhanden. • Pentium III CPU mit 600 MHz, 1 GHz ist empfehlenswert • 256 MB RAM, 512 MB sind empfehlenswert • ca. 200 MB Festplattenspeicher (inkl. der gleich behandelten Plugins und Language Packs) Pfad auf der Begleit-CD: \Software\Java\Eclipse\SDK 3.1.1 URL zum Herunterladen: http://www.eclipse.org/downloads/index.php Installation: Archiv **eclipse-SDK-3.1.1-win32.zip** in **C:\Programme\eclipse** extrahieren Vorgehen mit FilZip: o Kontextmenü zur Zip-Datei öffnen o **FillZip > Extrahieren nach** o **Ziel:** C:\Programme

Eclipse-Plugins • **Eclipse Modeling Framework (EMF) 2.1.0** • **Graphical Editor Framework (GEF) 3.1** • **Visual Editor (VE) 1.1.0.1** • **UML 2 1.1.1**	Halten Sie bei der Installation der Plugins die nebenstehende Reihenfolge ein und extrahieren Sie nacheinander die Zip-Dateien • emf-**sdo-runtime-2.1.0.zip** • **GEF-runtime-3.1.zip** • **VE-runtime-1.1.0.1.zip** • **uml2-1.1.1.zip** in den Eclipse-Programmordner **C:\Programme\eclipse**. Vorgehen mit FilZip: • Kontextmenü zu den Zip-Dateien öffnen • **FilZip > Extrahieren nach** • **Ziel:** C:\Programme Der Pfad ab **eclipse** ist in den Zip-Dateien enthalten Hinweis: Bei einigen Info- und Lizenzdateien (z.B. **epl-v10.html** und **notice.html**) wird vor dem Überschreiben gewarnt. Behalten Sie jeweils die neueste Version. Vermutlich ist die Entscheidung aber irrelevant.
Language Packs zu Eclipse	Nach den bisher beschriebenen Installationen sollten Sie über eine englischsprachige Eclipse-Installation verfügen. Durch das Auspacken einiger weiterer Archivdateien, die Sie jeweils im **Language Pack** - Unterordner finden, erhalten Sie eine perfekt lokalisierte deutsche Eclipse-Oberfläche. Beim SDK und bei den Plugins sind jeweils zwei Dateien auszupacken, wobei das schon mehrfach für FilZip beschriebene Vorgehen verwendet werden kann: • **SDK 3.1.1** o **NLpack1-eclipse-SDK-3.1.1a-win32.zip** o **NLpack1_FeatureOverlay-eclipse-SDK-3.1.1.zip** • **EMF 2.0.1** o **NLpack1-emf-sdo-runtime-2.1.1.zip** o **NLpack1_FeatureOverlay-emf-sdo-runtime-2.1.1.zip** • **GEF 3.1** o **NLpack1-GEF-runtime-3.1.1.zip** o **NLpack1_FeatureOverlay-GEF-runtime-3.1.1.zip** • **VE 1.1.0.1** o **NLpack1-VE-runtime-1.1.0.1.zip** o **NLpack1_FeatureOverlay-VE-runtime-1.1.0.1.zip** • **UML 2 1.1.11** o **NLpack1-uml2-1.1.1.zip** o **NLpack1_FeatureOverlay-uml2-1.1.1.zip**

2.2 Java-Entwicklung mit Texteditor und JDK

Für die Betriebsysteme Windows, Linux und Solaris kann das aktuelle JDK samt Installationsanleitung und Dokumentation über folgende Webseite bezogen werden:

http://java.sun.com/j2se/

2.2.1 Editieren

Wir verfassen den Quellcode mit einem beliebigen Texteditor, unter Windows z.B. mit **Notepad** (**Editor**). 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 {` `  public static void main(String[] args) {` `  System.out.println("Hallo Allerseits!");` `  }` `}`	`Hallo Allerseits!`

Im Unterschied zu *hybriden* Programmiersprachen wie C++ und Delphi, die neben der objektorientierten auch die rein prozedurale Programmiertechnik unterstützen, verlangt Java auch für solche Trivialprogramme eine **Klassendefinition**.
Die mit dem Starten der Anwendung beauftragte Klasse `Hallo` erzeugt 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.

Das POO-Programm ist zwar nicht „vorbildlich“, eignet sich aber aufgrund seiner Kürze zum Erläutern wichtiger Regeln, an die Sie sich so langsam gewöhnen müssen. 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 dieses Manuskript habe ich mich am Verhalten der Java-Urheber orientiert: Gosling et at. (2005) lassen bei ausführbaren Klassen 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 dazu, Objekte zu erzeugen.
- Die Definition der Methode **main()** wird von drei Schlüsselwörtern eingeleitet, deren Bedeutung für Neugierige hier schon erwähnt wird:
  - **public**
    Wie eben erwähnt, wird die Methode **main()** beim Programmstart vom Interpreter gesucht und ausgeführt. Weil es sich beim Interpreter aus Sicht des Programms um einen *externen* Akteur handelt, muss (zumindest ab SDK 1.4.x) die Methode **main()** explizit über den Modifikator **public** für die Öffentlichkeit frei gegeben werden.
  - **static**
    Mit diesem Modifikator wird **main()** als **Klassenmethode** gekennzeichnet. Im Unterschied zu den *Instanz*methoden der *Objekte* gehören die Klassenmethoden, oft

auch als *statische Methoden* bezeichnet, zur *Klasse* und können ohne vorherige Objektkreation ausgeführt werden.
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 7).

- **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 Stringelementen 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 der **main()**-Definition ist die im Beispiel verwendete Parameterliste obligatorisch, weil der Interpreter ansonsten die Methode beim Programmstart nicht erkennt und sich ungefähr so äußert:
    ```
    Exception in thread "main" java.lang.NoSuchMethodError: main
    ```
    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.
- In der **main()**-Methode unserer `Hallo`-Klasse wird die **println()**-Methode des vordefinierten Objektes **System.out** dazu benutzt, einen Text an die Standardausgabe zu senden. Zwischen dem Objekt- und dem Methodennamen steht ein Punkt. Bei dem 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.

Speichen Sie Ihr Quellprogramm unter dem Namen **Hallo.java** in einem geeigneten Verzeichnis, z.B. in

U:\Eigene Dateien\Java\Kurs\Hallo

Beachten Sie bitte:

- Der Dateinamensstamm (vor dem Punkt) sollte unbedingt mit dem Klassennamen übereinstimmen. Ansonsten resultiert eine Namensabweichung zwischen Quellcode- und Bytecodedatei, denn die vom Compiler erzeugte Bytecodedatei übernimmt den Namen der Klasse. Bei einer Klasse mit dem Zugriffsmodifikator **public** (siehe unten) *muss* der Dateinamensstamm mit dem Klassennamen übereinstimmen.
- Die Dateinamenserweiterung muss unbedingt **.java** lauten.
- Unter Windows ist beim Dateinamen die Groß-/Kleinschreibung zwar irrelevant, doch sollte auch hier aus ästhetischen Gründen und im Hinblick auf eine mögliche Weitergabe der Quellcodedatei auf Konsistenz 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:

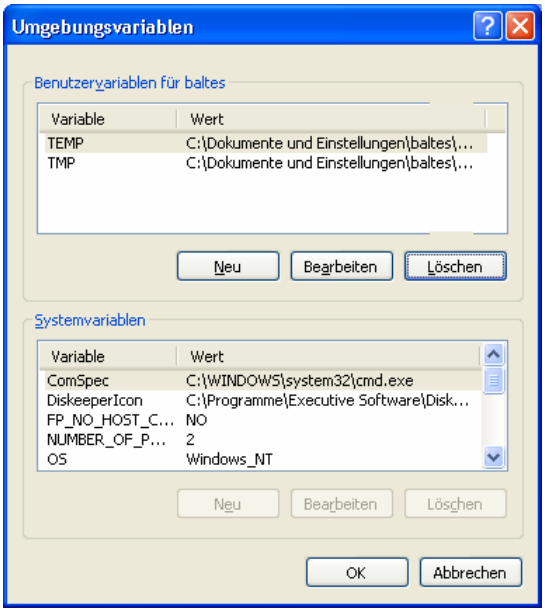
```
javac Hallo.java
```

Damit dieser Aufruf von jedem Verzeichnis aus klappt, muss nach der JDK-Installation das **bin**-Unterverzeichnis (mit dem Compiler **javac**) in die Definition der Umgebungsvariablen PATH- aufgenommen werden. Auf den Pool-PCs der Universität Trier kann dies im Rahmen einer Initialisierungsprozedur mit diversen Vorbereitungen für die Programmentwicklung mit Java geschehen, die folgendermaßen zu starten ist:

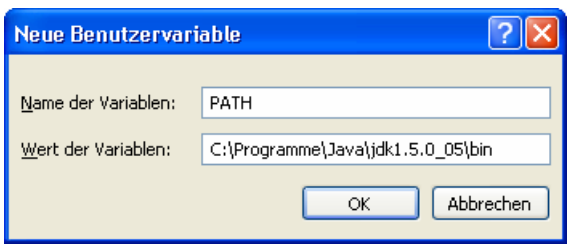
Start > Programme > Programmentwicklung > Java >
Java 2 SE Development Kit 1.5.0 > Initialisierung der Benutzerumgebung

Nach Ausführung der Prozedur müssen Sie sich allerdings bei Windows ab- und wieder anmelden, um die neue PATH-Definition in Kraft zu setzen.
Dies ist *nicht* erforderlich, nachdem Sie unter Windows XP auf dem üblichen Weg für die PATH-Erweiterung gesorgt haben:

- 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 eine neue **Benutzervariable** PATH anlegen oder die vorhandene **Systemvariable** gleichen Namens erweitern:



- Wer unter Windows XP Professional mit normalen Benutzerrechten arbeitet, legt eine neue Benutzervariable an, z.B.:



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

Beim Kompilieren einer Quellcodedatei werden auch alle darin benutzten fremden Klassen neu übersetzt, falls deren Bytecodedatei 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`).

2.2.3 Ausführen

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

```
java Hallo
```

Hier ist zu beachten:

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

  ```
  U:\Eigene Dateien\Java\Kurs\Hallo>java Hallo.class
  Exception in thread "main" java.lang.NoClassDefFoundError: Hallo/class
  ```

- 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!).
- Die Version des installierten Interpreters kann mit folgenden Kommando ermittelt werden:

  ```
  U:\Eigene Dateien\Java\Kurs\Hallo>java -version
  java version "1.5.0_05"
  Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_05-b05)
  Java HotSpot(TM) Client VM (build 1.5.0_05-b05, mixed mode)
  ```

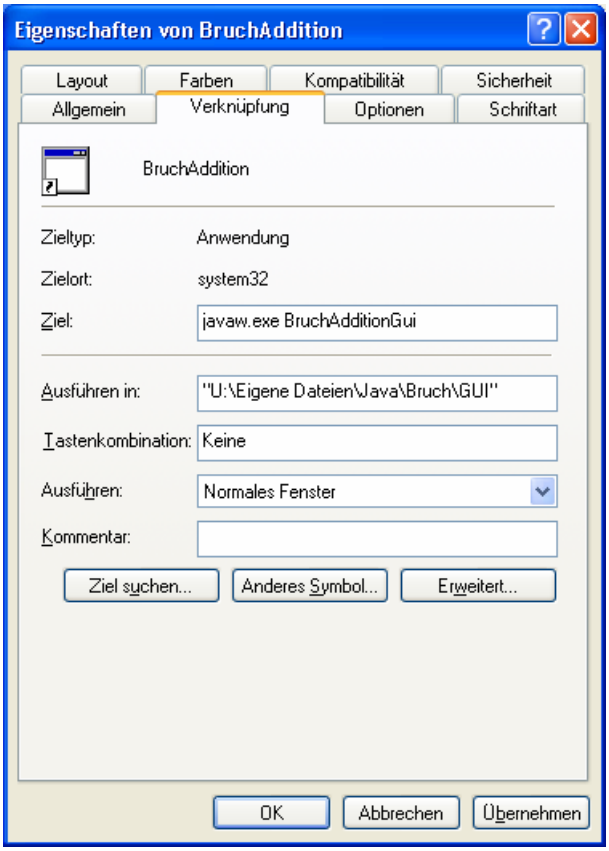
  Wir erfahren, dass Suns **HotSpot - Client VM** im Einsatz ist. Sie stellt sich beim Übersetzen des Java-Bytecodes in Maschinencode besonders geschickt an (Analyse des Laufzeitverhaltens) und ist älteren JVMs, die mit einem **JIT**-Compiler (**J**ust-**i**n-**T**ime) arbeiten, deutlich überlegen.

Zum *Ausführen* eines Java-Programms wird *nicht* das vollständige JDK (mit Entwicklerwerkzeugen, Dokumentation etc.) benötigt, sondern lediglich die virtuelle Java-Maschine (Java VM) mit dem Interpreter **java.exe** und der Klassenbibliothek. Dieses - auch als **Java Runtime Environment** (JRE) bezeichnete - Produkt ist bei Sun separat über folgenden URL erhältlich:

http://java.sun.com/j2se/1.5.0/download.jsp

Man darf die JRE zusammen mit eigenen Programmen weiter gegeben, um diese auf beliebigen Rechnern lauffähig zu machen. Das JDK enthält natürlich auch die Laufzeitumgebung (im Verzeichnis **…\jre**), so dass diese nicht zusätzlich installiert werden muss.

Das beim Einsatz von **java.exe** unvermeidliche Konsolenfenster kann beim Starten einer Java-Anwendung mit graphischer Benutzeroberfläche störend wirken. Unter MS-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 Bytecodedateien 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 6.4) oder ZIP-Archiven spezifizieren, die nach **class**-Dateien durchsucht werden sollen, z.B.:

```
C:\User>echo %classpath%
.;C:\Dokumente und Einstellungen\baltes\Eigene Dateien\Java\lib
```

Befinden sich alle benötigten Klassen entweder in der Java-API-Bibliothek (siehe Abschnitt 6.5) oder im aktuellen Verzeichnis, dann wird ab Java 1.2 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 oft zu Problemen bei CLASSPATH-Variablen, die das aktuelle Verzeichnis nicht enthalten, z.B.:

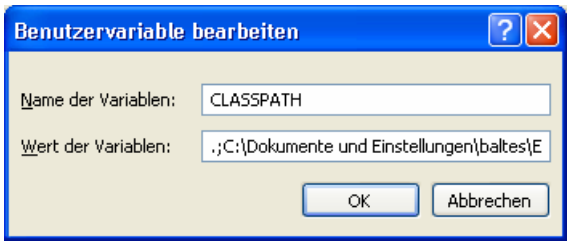
```
U:\Eigene Dateien\Java\Hallo>java Hallo
Exception in thread "main" java.lang.NoClassDefFoundError: Hallo
```

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 **\BspUeb\Simput**) zum Einsatz kommt, sollte die Umgebungsvariable CLASSPATH so gesetzt werden, dass der Compiler und der Interpreter die Klasse **Simput.class** finden. Auf den Pool-PCs der Universität Trier kann dies im Rahmen einer Initialisierungsprozedur mit diversen Vorbereitungen für die Programmentwicklung mit Java geschehen, die folgendermaßen zu starten ist:

Start > Programme > Programmentwicklung > Java >
Java 2 SE Development Kit 1.5.0 > Initialisierung der Benutzerumgebung

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.

Leider ignoriert unsere Entwicklungsumgebung Eclipse die CLASSPATH-Umgebungsvariable, bietet aber eine alternative Möglichkeit zur Definition eines Klassenpfads (s. u.).

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 -classpath .;..\Konsole BruchAdditionGui.java
javaw.exe -cp ".;U:\Eigene Dateien\Java\Bruch\Konsole" BruchAdditionGui
```

Auch hier muss der aktuelle Pfad ausdrücklich aufgelistet werden, wenn er 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 Kommandozeilenoption wird eine eventuell vorhandene CLASSPATH-Umgebungsvariable für den gestarteten Compiler- oder Interpreterlauf deaktiviert.

2.2.5 Programmfehler beheben

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

- **Syntaxfehler**
  Diese verstoßen gegen eine Syntaxregel der verwendeten Programmiersprache, werden vom Compiler gemeldet und sind daher relativ leicht zu beseitigen.
- **Semantikfehler**
  Hier liegt kein Syntaxfehler vor, aber das Programm verhält sich anders als erwartet.

Die Java-Entwickler haben dafür gesorgt (z.B. durch strenge Typisierung, Beschränkung der impliziten Typanpassung, Zwang zur Behandlung von Ausnahmen), dass möglichst viele Fehler(quellen) 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, führt ein Übersetzungsversuch zu folgender Reaktion:

```
U:\Eigene Dateien\Java\Kurs\Hallo>javac Hallo.java
Hallo.java:3: package system does not exist
                system.out.print("Hallo Allerseits!");
                      ^
1 error
```

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 erfahren wir den Namen der betroffenen Quellcodedatei und die Zeilennummer.
- Anschließend protokolliert der Compiler die betroffene Zeile und markiert, an welcher Stelle er damit Probleme hatte.

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.
Auch die Fehler*beschreibung* fällt im Beispiel brauchbar aus, obwohl der Compiler falsch vermutet, dass mit dem verunglückten Bezeichner ein Paket (siehe unten) gemeint sei.

Zum Abschluss wollen wir noch mit einem Fehler der zweiten Art experimentieren und im Kopf der **main()**-Methodendefinition den Modifikator **static** streichen, der Klassenmethoden kennzeichnet (vgl. Abschnitt 2.2.1). Während der *Compiler* am veränderten Programm nichts zu bemängeln hat, klagt der *Interpreter*:

```
U:\Eigene Dateien\Java\Kurs\Hallo>java Hallo
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Ein Verständnis für das Fehlerobjekt `NoSuchMethodError`[1] setzt schon etwas mehr Java - Know How voraus: Es existiert zwar eine Methode namens **main()**, doch gehört diese mangels **static**-Modifikator zu den (prinzipiell herstellbaren) *Objekten* der Klasse `Hallo`, während der Interpreter eine Klassenmethode mit diesem Namen sucht.
Ansonsten ist der Fehler relativ harmlos, weil er die Ausführung des Programms verhindert. Übler sind Fehler, die zu Abstürzen oder gar zu falschen Ergebnissen führen.

2.3 Java-Entwicklung mit Eclipse

Zu Eclipse sind umfangreiche Bücher entstanden mit einer Beschreibung der zahlreichen Profiwerkzeuge zur Softwareentwicklung. Wir wollen zunächst das objektorientierte Programmieren mit Java lernen und beschränken uns daher auf sehr elementare Komponenten und Funktionen von Eclipse.
Später werden wir mit großem Vergnügen den als Plugin installierten Visual Editor zu Gestaltung von graphischen Benutzeroberflächen einsetzen.

Eclipse setzt nur die JRE voraus und bringt einen eigenen Compiler mit, der kompatibel mit der Sprachdefinition von Java 5.0 ist. Es ist aber sinnvoll, das JDK als Basis für Eclipse zu nutzen, damit auch die Dokumentation und der Quellcode zu den API-Klassen verfügbar sind.

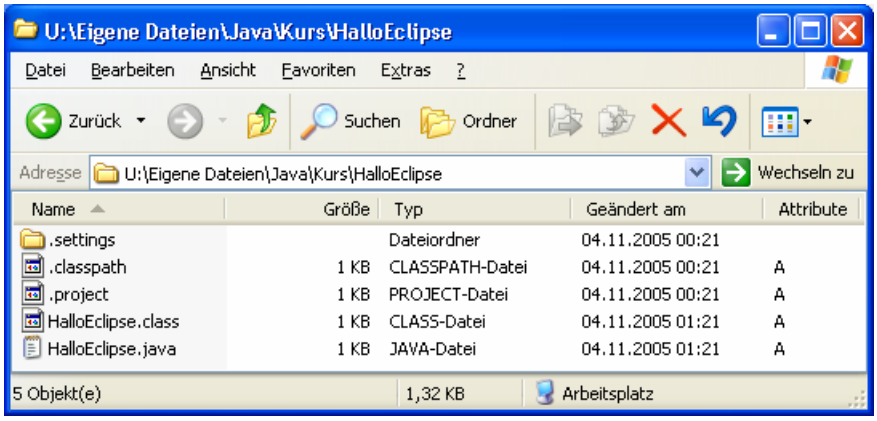
---

[1] Tatsächlich handelt es sich auch hier um ein Objekt im oben beschriebene Sinn (mit Eigenschaften und Methoden), das allerdings *nicht* zur Modellierung des Aufgabenbereichs dient.

Wir beschränken uns anschließend auf elementare Informationen für Einsteiger, die später nach Bedarf ergänzt werden. Weiterführende Informationen bieten das Hilfesystem und das Eclipse-Handbuch des Regionalen Rechenzentrums für Niedersachsen (RRZN 2005), das in der URT-Skriptenstelle zu erwerben ist.

2.3.1 Arbeitsbereich und Projekt

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-Instanz 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

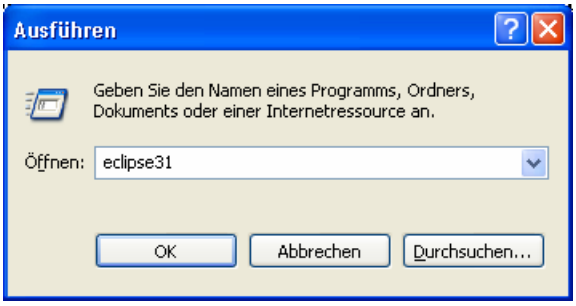
Um Eclipse 3.1 auf den Pool-PCs an der Universität Trier zu starten, müssen Sie über

Start > Ausführen

den Befehl

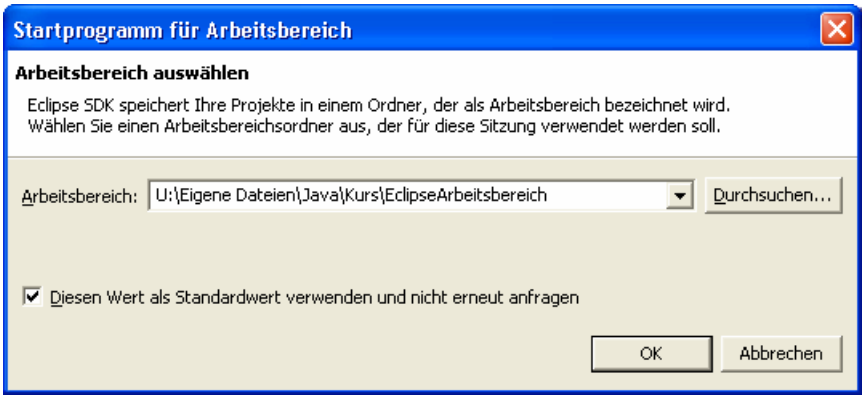
eclipse31

ausführen:



Auf Ihrem eigenen Rechner starten Sie Eclipse über die Datei **eclipse.exe** im Installationsordner (meist **c:\Programme\eclipse**).

Eclipse erkundigt sich beim 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 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

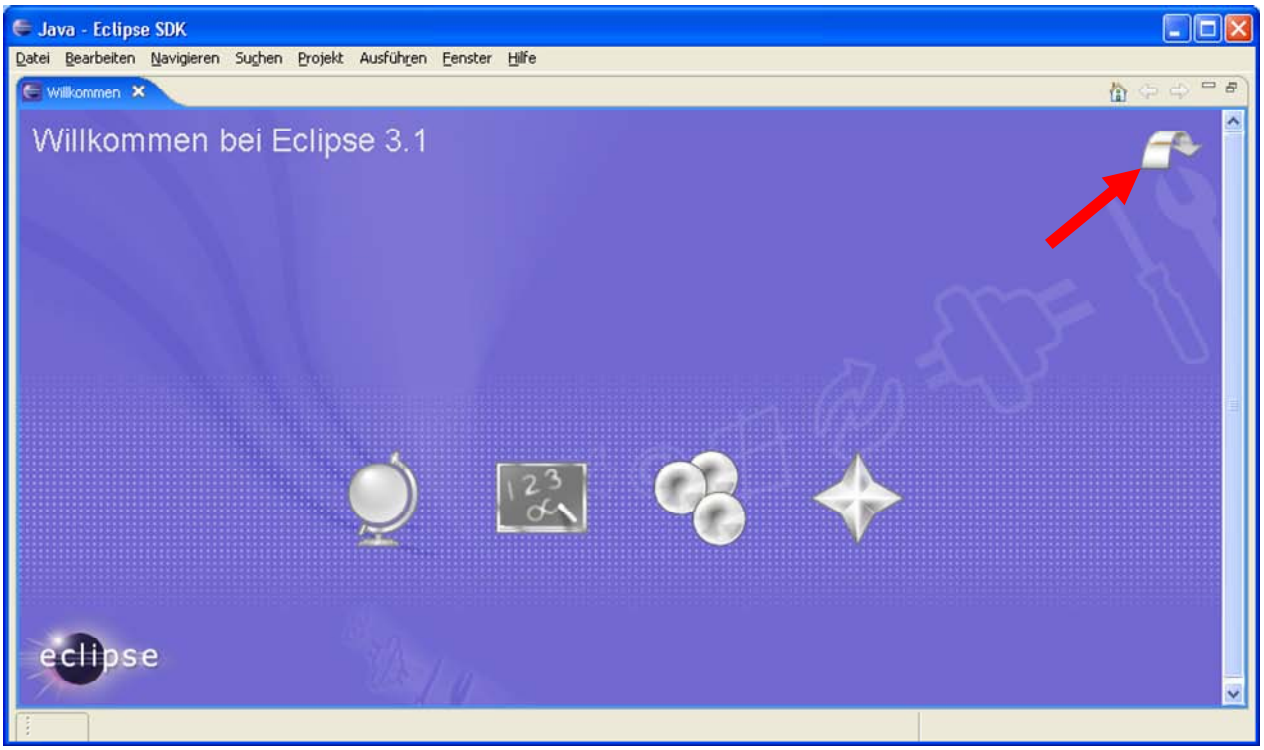
  Datei > Arbeitsbereich wechseln

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

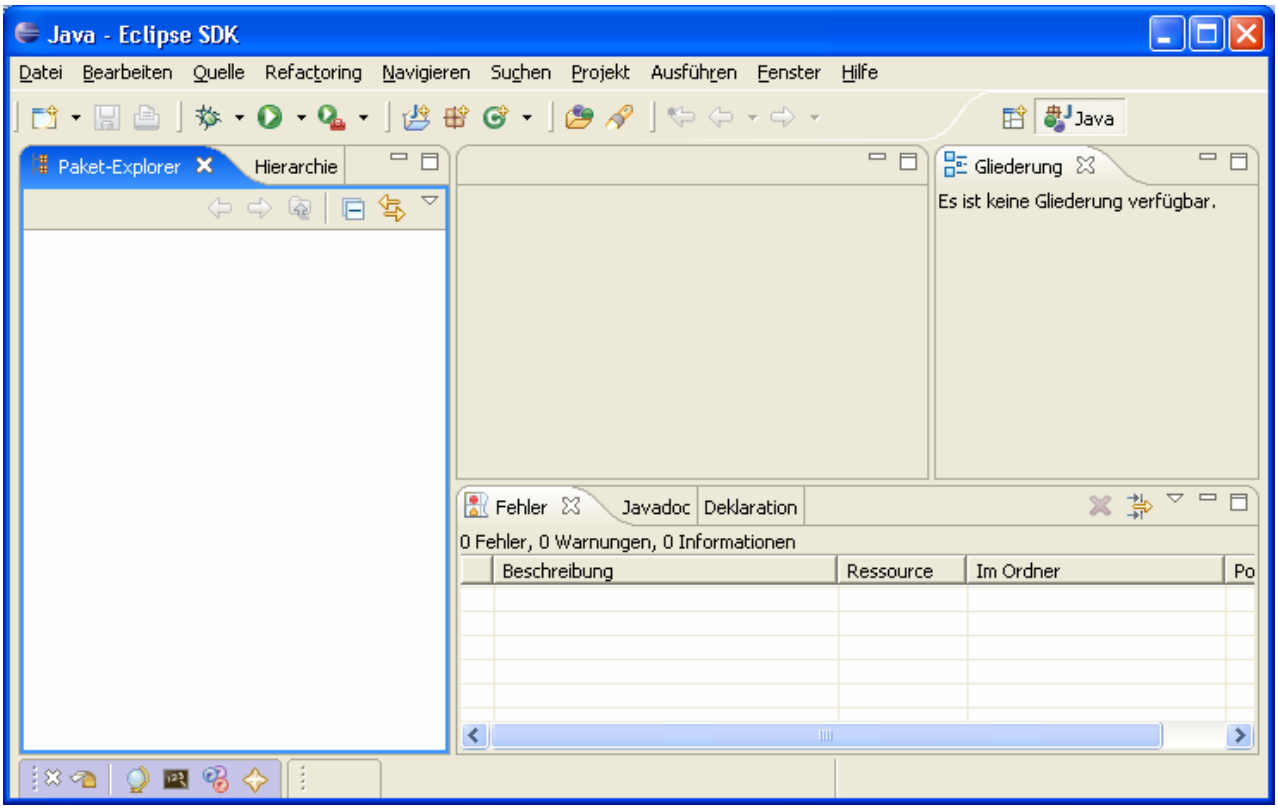
  Fenster > Benutzervorgaben > Allgemein > Start und Beendigung > Arbeitsbereich bei Start anfordern

  reaktivieren Sie die routinemäßige Arbeitsbereichsanfrage beim Start.

Beim ersten Eclipse-Start werden Sie recht eindrucksvoll begrüßt:



Mit einem Mausklick auf den Pfeil in der rechten oberen Ecke gelangen Sie zur Werkbank (Workbench) mit zahlreichen Werkzeugen für eine erfolgreiche und komfortable Softwareentwicklung:

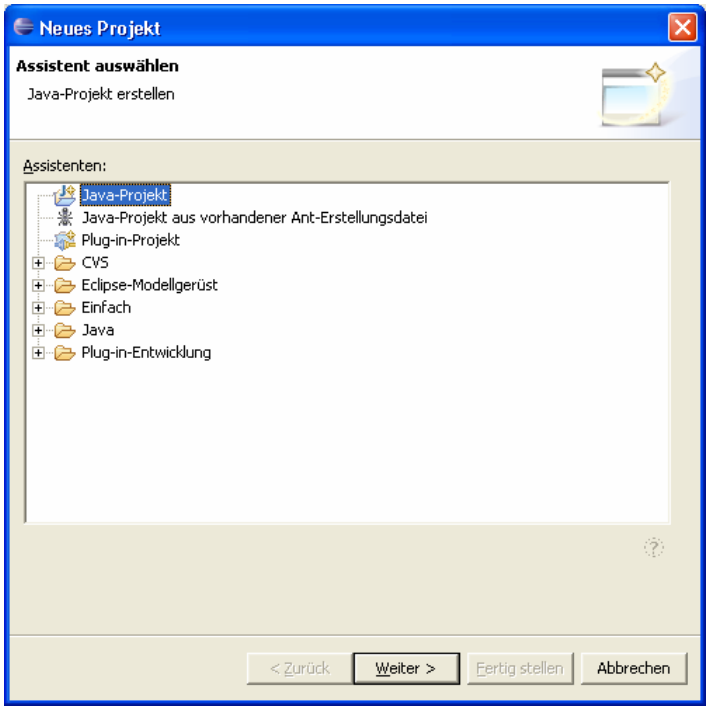


2.3.3 Projekt anlegen

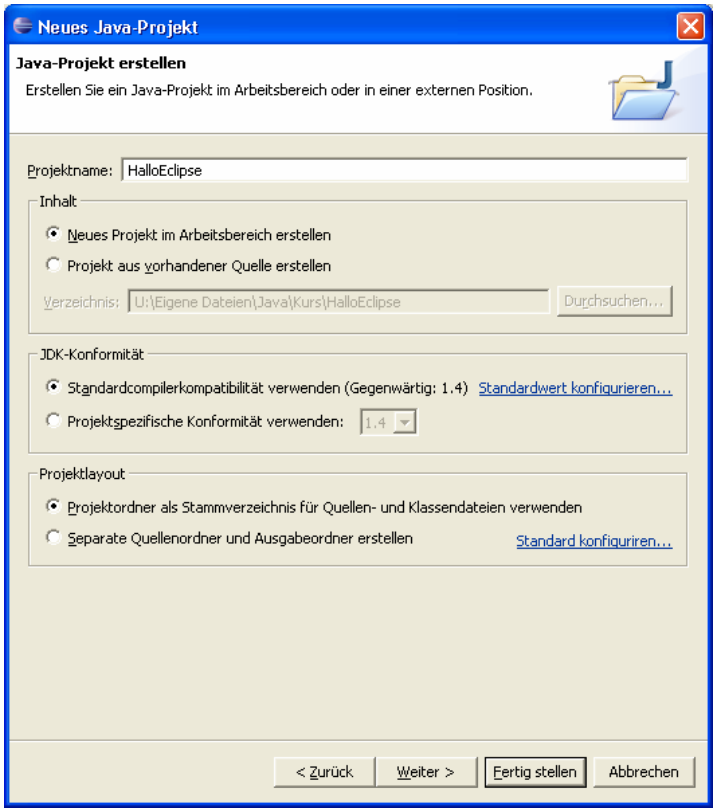
Wir starten mit

Datei > Neu > Projekt

den Assistenten für neue Projekte und wählen mit **Weiter** den voreingestellten Typ **Java-Projekt**:



Im nächsten Dialog wählt man u.a. den Projektnamen und das Java-Kompatibilitätsniveau, z.B.:



In der Regel kann man jetzt auf **Fertigstellen** klicken und damit alle weiteren Dialoge des Assistenten ignorieren. Viele Einstellungen eines Projektes sind später über den Menübefehl

Projekt > Eigenschaften

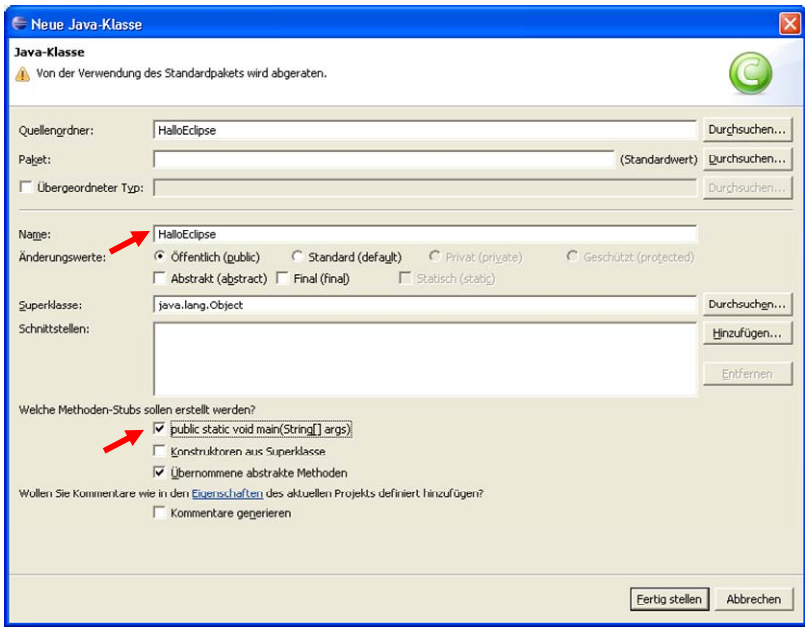
zu ändern.

2.3.4 Klasse hinzufügen

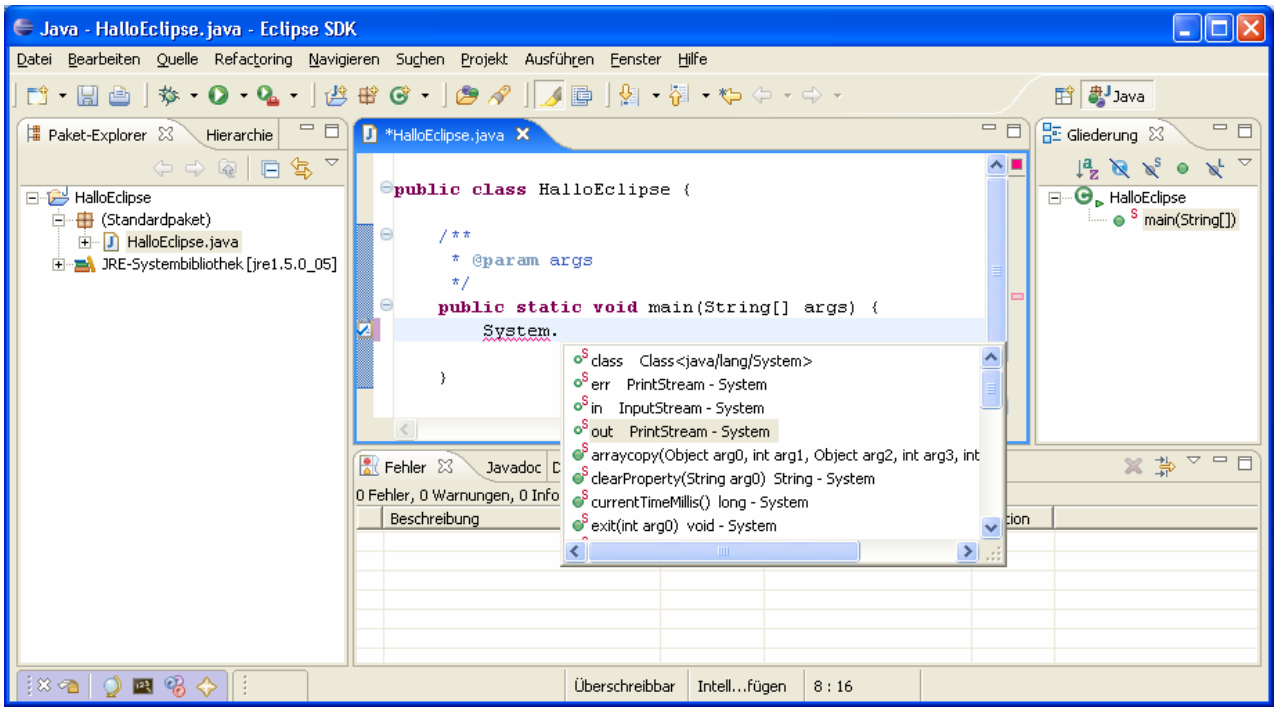
Wir starten mit

Datei > Neu > Klasse

die Definition einer Klasse, legen im folgenden Dialog deren Namen fest und lassen eine **main()**-Methode anlegen:



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

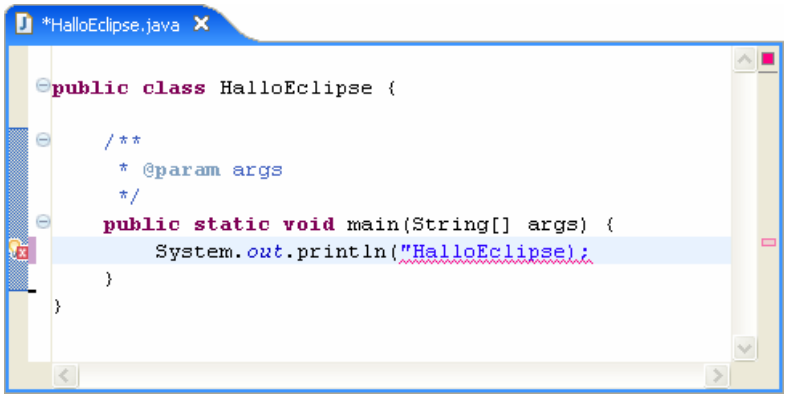


Sobald wir einen Punkt hinter den Klassennamen System setzen, erscheint eine Liste mit allen zulässigen Fortsetzungen, wobei wir uns im Beispiel für die Klassenvariable **out** entscheiden, die auf ein Objekt der Klasse **PrintStream** zeigt. Nach dem nächsten Punkt werden u.a. die Methoden der Klasse **PrintStream** aufgelistet, und wir komplettieren die Nachricht an das Objekt **out** folgendermaßen:

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

2.3.5 Übersetzen und Ausführen

Analog zu einem Textverarbeitungsprogramm mit Rechtschreibungskontrolle während der Eingabe informiert Eclipse ohne expliziten Übersetzungsauftrag über Fehler, z.B.:

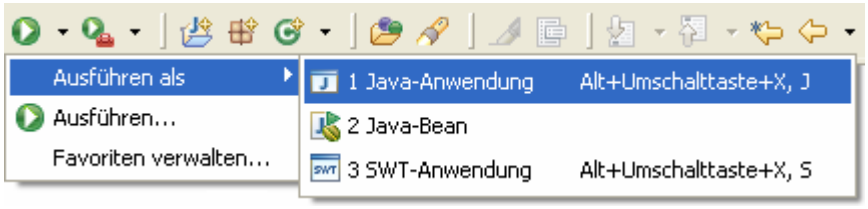


Bei jeder Speicherung wird eine Quelle automatisch neu übersetzt.

Den ersten Start unserer Anwendung veranlassen wir mit

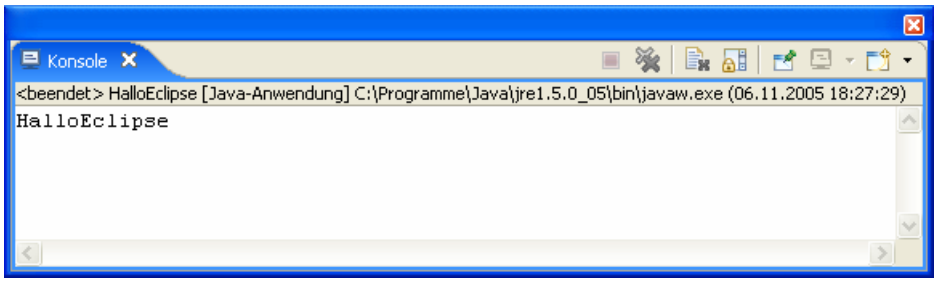
Ausführen > Ausführen als > Java-Anwendung

oder äquivalent mit dem Klappmenü zur Schaltfläche  :



Weil sich Eclipse diese Wahl merkt, genügt bei späteren Starts ein Mausklick auf den Schalter oder die Tastenkombination **Strg+F11**.

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



2.3.6 Einstellungen ändern

2.3.6.1 Java-Kompatibilitätsniveau

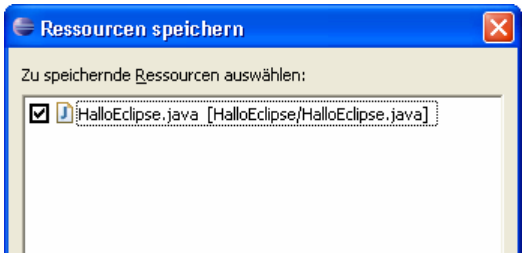
Wenn Sie die Neuerungen von Java 5.0 regelmäßig nutzen wollen, sollten Sie nach

Fenster > Benutzervorgaben > Java > Compiler

die Konformitätsstufe des Compilers entsprechend voreinstellen:

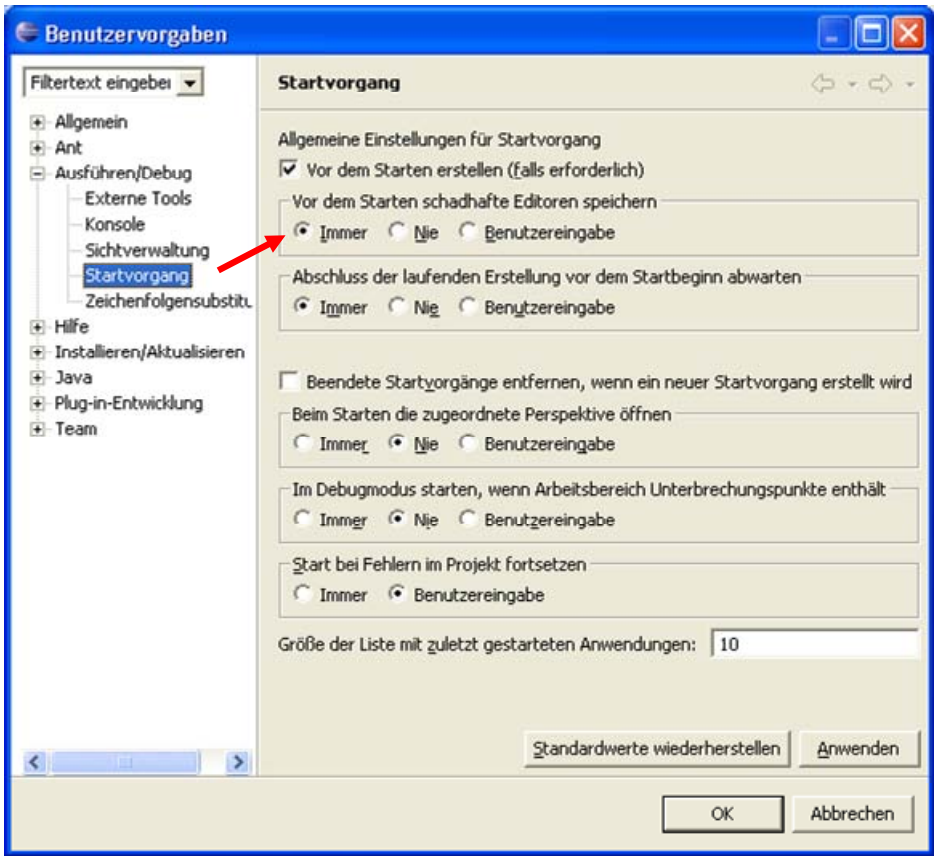
2.3.6.2 Automatische Quellcodesicherung beim Ausführen

Wenn Sie einen geänderten Quellcode ausführen lassen, schlägt Eclipse die vorherige Sicherung vor, z.B.:



Auf folgende Weise veranlasst man Eclipse, vor der Ausführung geänderte Quellen automatisch zu speichern:

- **Fenster > Benutzervorgaben > Ausführen/Debug > Startvorgang**
- Wählen Sie im Optionsfeld **Vor dem Starten schadhafte Editoren speichern**[1] den Wert **Immer**:



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 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.

---

[1] Das im EDV-Bereich häufig für geänderte Daten benutzte englische Wort *dirty* ist etwas unglücklich durch *schadhaft* übersetzt worden.

Wir üben den Import am Beispiel des Bruchadditionsprojekts mit graphischer Benutzeroberfläche, das sich im folgenden Ordner befindet:

…\BspUeb\Einleitungsbeispiele\Bruch\Gui

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

U:\Eigene Dateien\Java\Kurs\Einleitungsbeispiele\Bruch\Gui

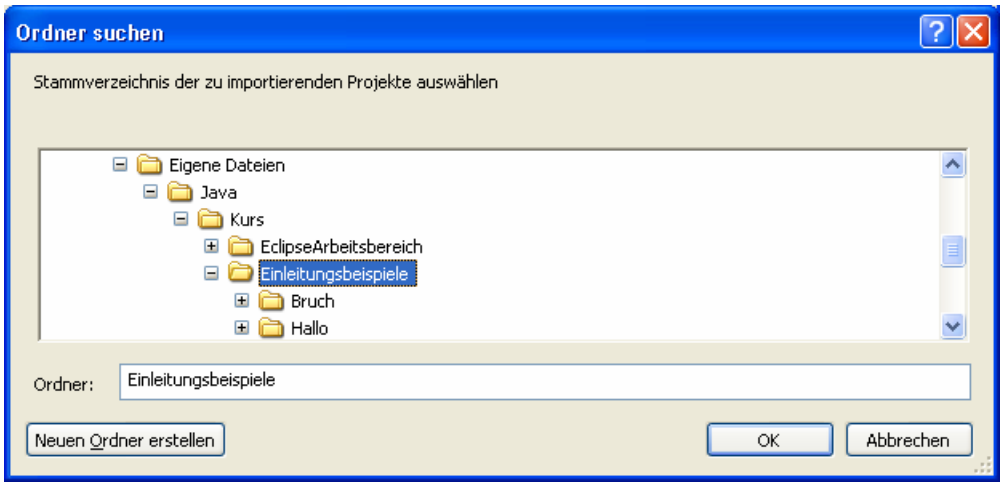
Starten Sie Eclipse mit Ihrem persönlichen Arbeitsbereich, z.B. in

U:\Eigene Dateien\Java\Kurs\EclipseArbeitsbereich

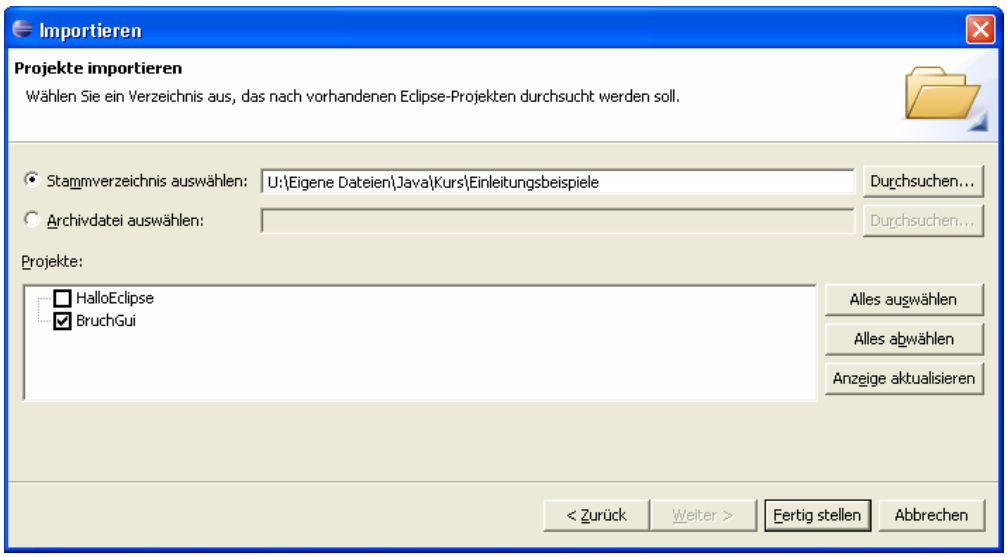
Initiieren Sie den Import mit

Datei > Importieren > Vorhandene Projekte in den Arbeitsbereich > Weiter

Klicken Sie in der Dialogbox **Importieren** auf den Schalter **Durchsuchen**, und wählen Sie im Pfad einen Knoten oberhalb des zu importierenden Projektordners, z.B.:



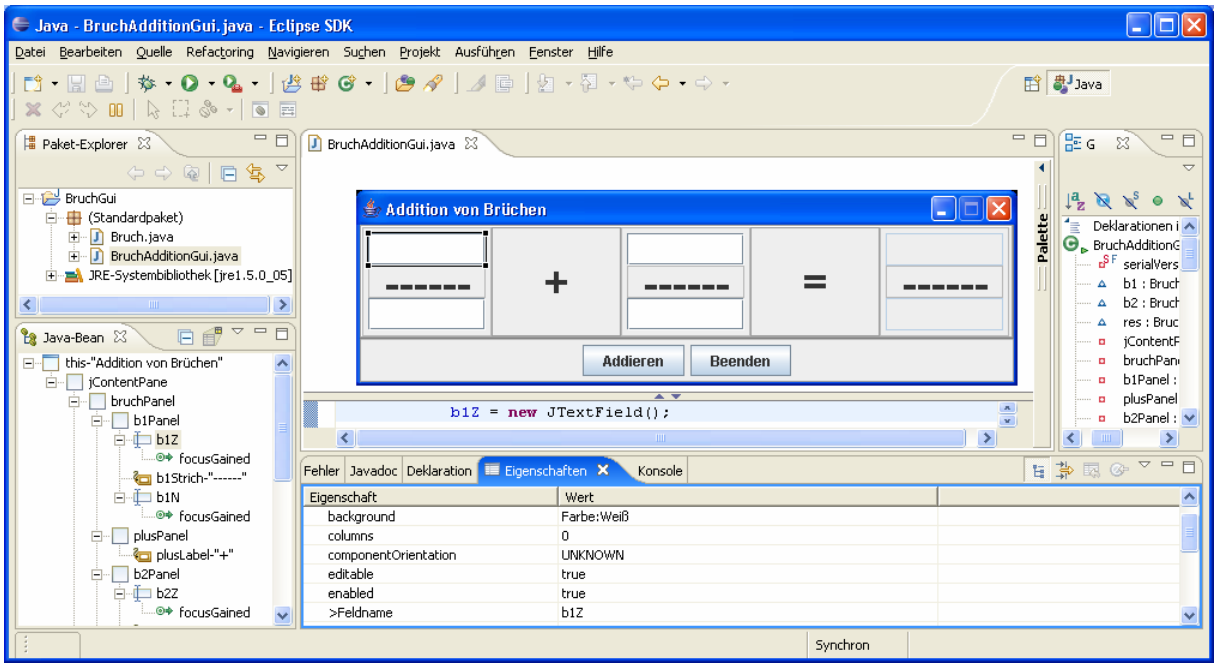
In der Dialogbox **Importieren** müssen Sie eventuell aus mehreren importfähigen Projekten eine Teilmenge bestimmen und mit **Fertigstellen** Ihre Wahl quittieren:



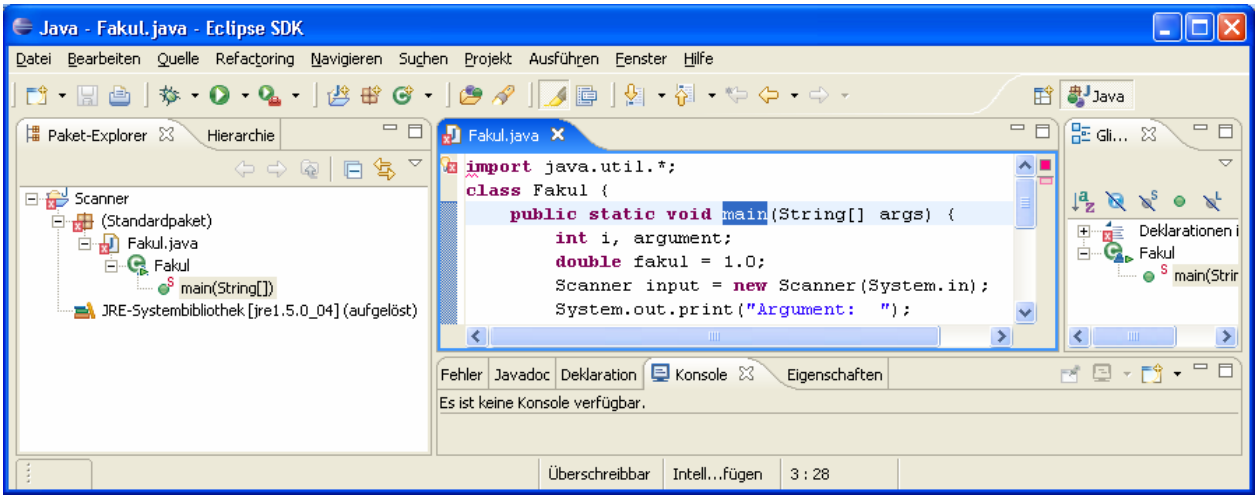
Anschließend sollte das importierte Projekt auf Ihrer Eclipse-Werkbank auftauchen, so dass Sie z.B. über die Kontextmenüoption

Öffnen mit > Visual Editor

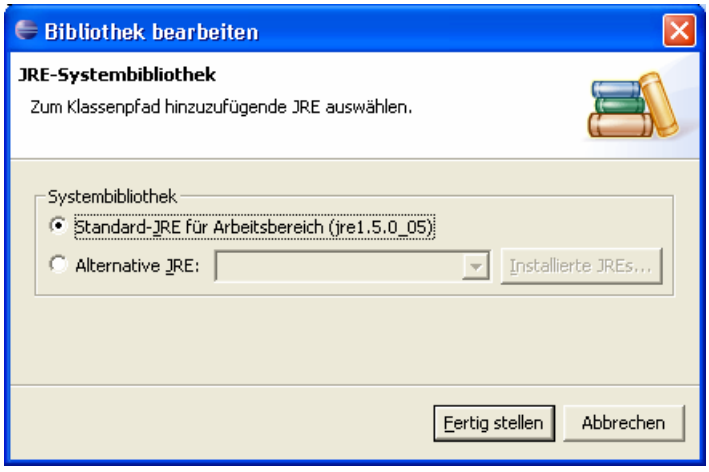
zur Datei **BruchAdditionGui.java** das Anwendungsfenster im visuellen Editor öffnen können:



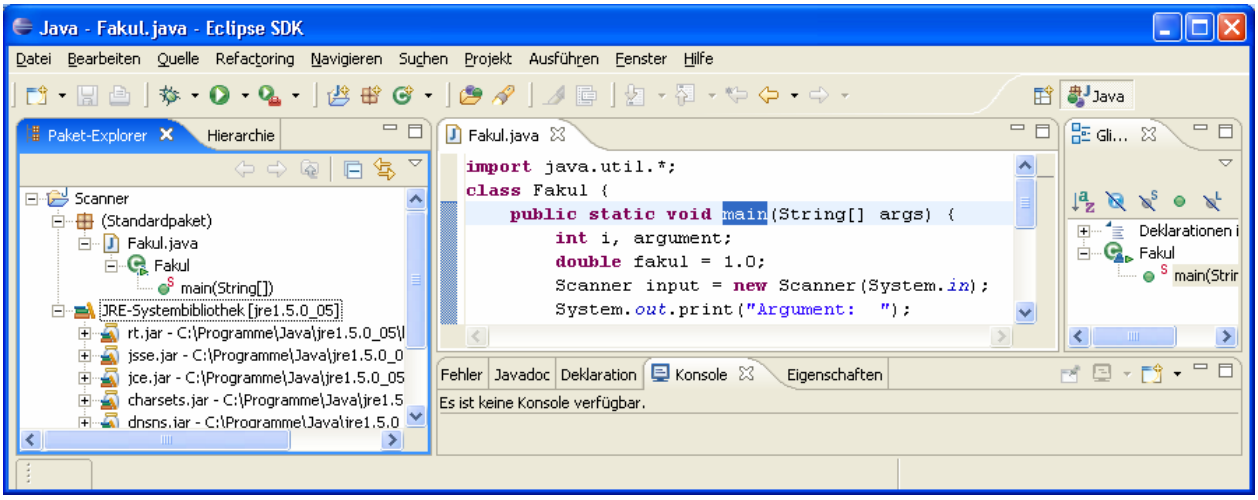
Wenn Eclipse nach dem Import die Klassen aus der Java-Standardbibliothek nicht kennt, enthält das importierte Projekt einen inkompatiblen Pfad für die JRE, z.B.:



Um Abhilfe zu schaffen, öffnet man im Paket-Explorer das Kontextmenü zum JRE-Eintrag des Projektes und wählt die Option **Konfigurieren**. In der Dialogbox **Bibliothek bearbeiten** kann die Voreinstellung übernommen werden:

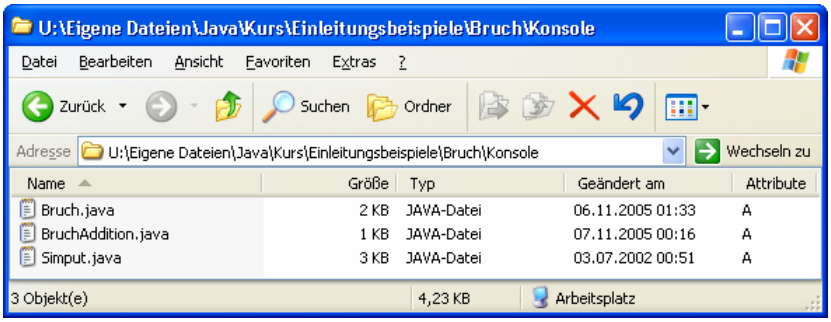


Anschließend sind die Orientierungsprobleme behoben:

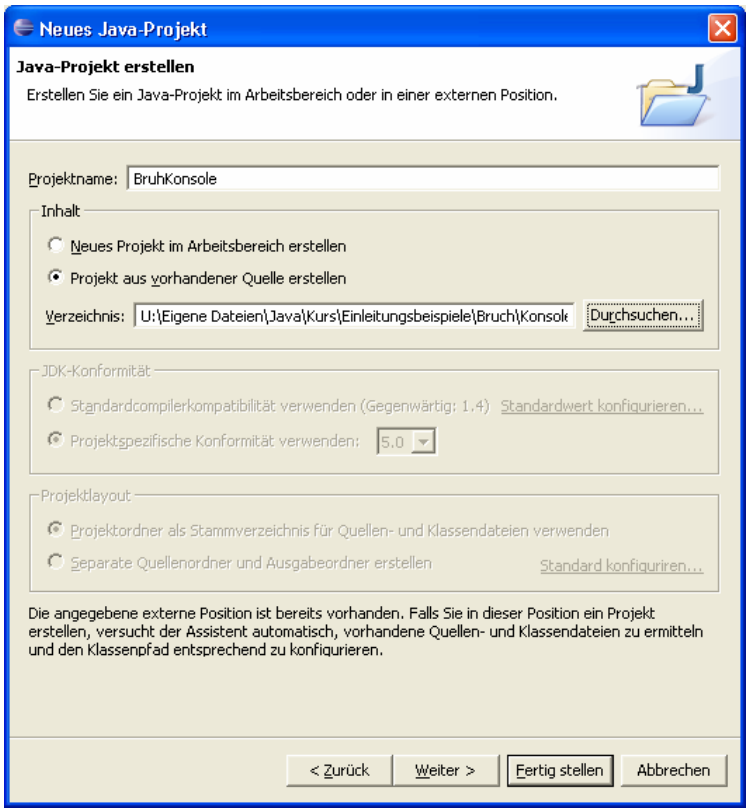


2.3.8 Projekt aus vorhandenen Quellen erstellen

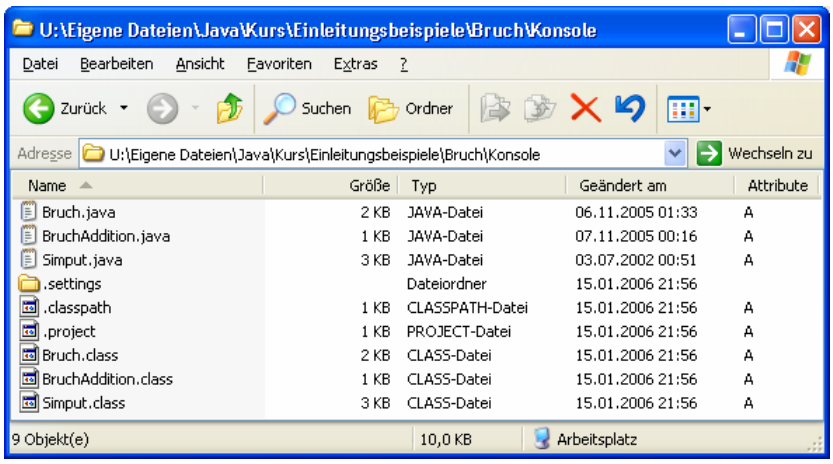
Gelegentlich soll ein neues Projekt unter Verwendung bereits existierender Quelldateien erstellt werden, z.B.:



In diesem Fall wählt man in der Dialogbox **Neues Java-Projekt** die Option **Projekt aus vorhandener Quelle erstellen**:



Eclipse übernimmt den Ordner und ergänzt dort seine Projektdateien, z.B.:



2.4 Übungsaufgaben zu Abschnitt 0

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 Starten eines Java-Programms muss man den Namen der auszuführenden Klasse samt Extension angeben (**.class**).
2. Beim Übersetzen einer Java-Klasse muss man den Dateinamen samt Extension angeben (**.java**).
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 Java-Compiler die API-Klassen findet, muss deren Pfad in die Umgebungsvariable CLASSPATH eingetragen werden.

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** in den Klassenpfad aufnehmen.

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 als bei traditionellen, *nicht* objektorientierten Sprachen (z.B. C) aus.

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 **Klassendefinitionen**.
  Meist verwendet man für den Quellcode einer Klasse jeweils eine eigene Datei. Der Compiler erzeugt auf jeden Fall für jede Klasse eine eigene Bytecodedatei.
  Unser Bruchrechnungsbeispiel in Abschnitt 1.1 besteht aus den beiden Klassen `Bruch` und `BruchAddition`.
- 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)** und den Modifikatoren **public** und **static**. Diese wird beim Starten der Klasse vom Interpreter ausgeführt. Daher muss sie der Öffentlichkeit zugänglich (Modifikator **public**) und als Klassenmethode unabhängig von der Existenz konkreter Objekte ausführbar sein (Modifikator **static**). Auch die Startklasse selbst muss für den Interpreter sichtbar sein, was aber auch ohne explizite Verwendung des Modifikators **public** der Fall ist.
  Beim Bruchrechnungs-Beispiel in Abschnitt 1.1 ist die Klasse `BruchAddition` startfähig. In den POO-Beispielen von Abschnitt 0 existiert jeweils nur eine Klasse, die infolgedessen startfähig sein muss.
- Eine Klassendefinition enthält …
  - die Deklarationen ihrer **Variablen** (Eigenschaften)
  - und die Definitionen ihrer **Methoden** (Handlungskompetenzen).
- Sowohl die Klassen- als die Methodendefinitionen bestehen aus:
  - **Kopf**
    In Abschnitt 2.2.1 finden sich im Zusammenhang mit dem `Hallo`-Beispiel nähere Erläuterungen.
  - **Rumpf**
    Dieser besteht aus einem Block mit beliebig vielen **Anweisungen**, mit denen z.B. Variablen deklariert oder verändert werden. Ein Klassen- oder Methodenrumpf wird durch geschweifte Klammern begrenzt.
- Eine **Anweisung** ist die kleinste ausführbare Einheit eines Programms. In Java sind bis auf wenige Ausnahmen alle Anweisungen mit einem **Semikolon** abzuschließen.

Während der Beschäftigung mit elementaren Java-Sprachelementen werden wir der Einfachheit halber mit einer relativ untypischen, jedenfalls nicht wirklich objektorientierten Programmstruktur arbeiten, die Sie schon aus dem `Hallo`-Beispiel kennen. Es ist zwar eine Klassendefinition vor-

---

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

handen, doch diese dient nur als Hülle für eine „Hauptfunktion“ (im Sinne der altehrwürdigen Programmiersprache C), die in der Klassenmethode **main()** untergebracht wird. Es werden keine weiteren Methoden definiert und auch keine Objekte erzeugt, allerdings durchaus vordefinierte Objekte benutzt (z.B. **System.out** in **Hallo.java**).
Bei den Demonstrations- bzw. Übungsprogrammen in Abschnitt 0 werden wir also folgende Programmstruktur benutzen:

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

Manche Autoren sind der Meinung, dass derartige Programme, die wir in Abschnitt 1.4 als *pseudo-objektorientiert* bezeichnet haben, keinen optimalen Einstieg in die objektorientierte Programmierung darstellen. Ein erlaubtes Hilfsmittel sind sie m. E. aber doch, weil die spätere Behandlung von Klassen und Objekten auf einem soliden begrifflichen und motivationalen Fundament aufbauen wird. Damit die POO-Beispiele Ihrem Programmierstil nicht prägen, wurde an den Beginn des Kurses ein Beispiel gestellt (Bruchrechnung), das bereits etliche OOP-Prinzipien realisiert.

3.1.2 Syntaxdiagramme

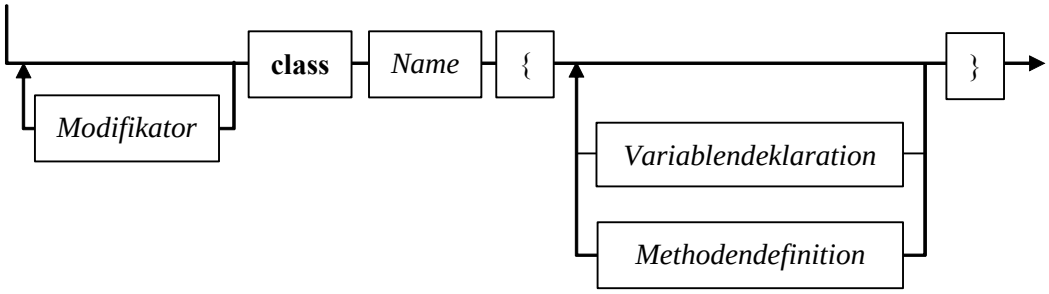
Um für Java-Sprachbestandteile (z.B. Definitionen oder Anweisungen) die Bildungsvorschriften kompakt und genau zu beschreiben, werden wir im Kurs u.a. so genannte **Syntaxdiagramme** einsetzen, für die folgende Vereinbarungen gelten:

- Jede Diagrammpassage in Pfeilrichtung ergibt zulässige Syntax.
- Für **terminale Sprachbestandteile**, die aus einem Syntaxdiagramm exakt in der angegebenen Form in konkreten Quellcode zu übernehmen sind, wird **fette** Schrift verwendet.
- *Platzhalter* sind durch *kursive* Schrift gekennzeichnet. Oft stehen Platzhalter für komplexe Sprachbestandteile, die an anderer Stelle erklärt sind (z.B. *Methodendefinition*).

Als Beispiele betrachten wir die Syntaxdiagramme zur Definition von Klassen und Methoden. Aus didaktischen Gründen zeigen die Diagramme nur solche Sprachbestandteile, die bisher in einem Beispiel verwendet oder im Text beschrieben wurden, so dass sie langfristig keinesfalls als Referenz taugen. Trotz der Vereinfachung sind die Syntaxdiagramme aber für die meisten Leser vermutlich nicht voll verständlich, weil etliche Bestandteile noch nicht systematisch beschrieben wurden (z.B. *Modifikator*, *Variablendeklaration*).
In diesem Abschnitt geht es aber nicht nur darum, Syntaxdiagramme als metasprachliche Hilfsmittel einzuführen, sondern die vorgestellten Beispiele tragen hoffentlich auch zur Klärung der wichtigen Begriffe *Klasse* und *Methode* bei.

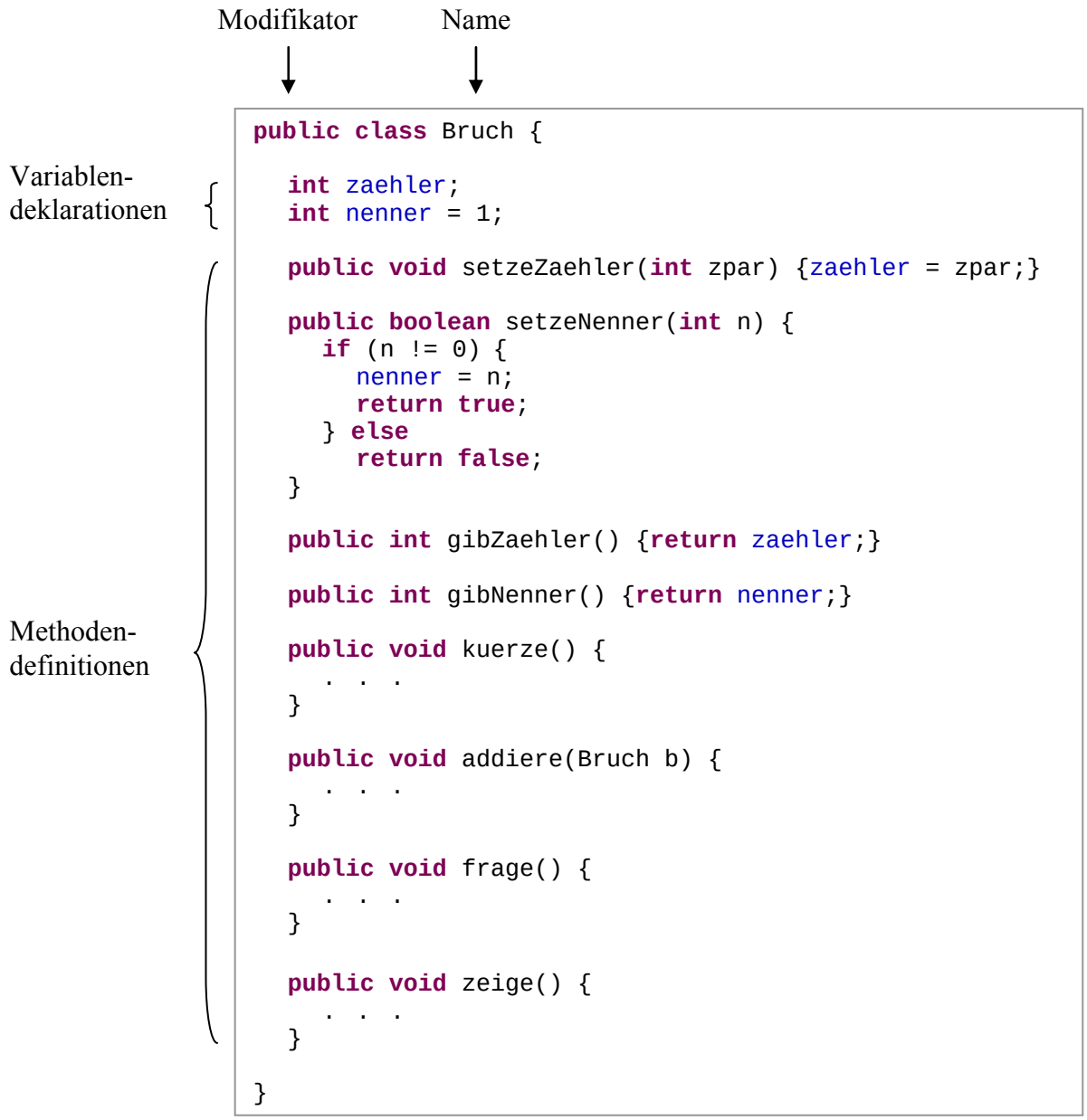
Wir arbeiten vorerst mit dem folgenden, leicht vereinfachten Klassenbegriff:

Klassendefinition



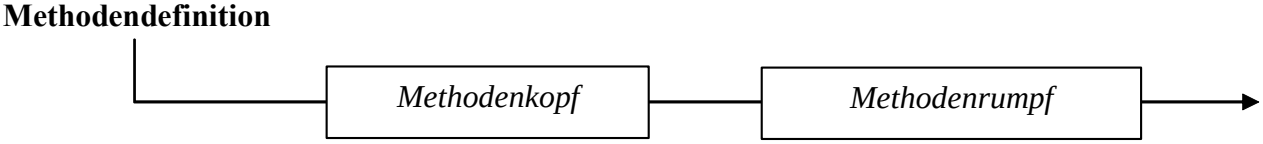
Solange man sich auf zulässigen Pfaden bewegt (immer in Pfeilrichtung, eventuell auch in Schleifen), an den Stationen (Rechtecken) entweder den konstanten Sprachbestandteil exakt übernimmt

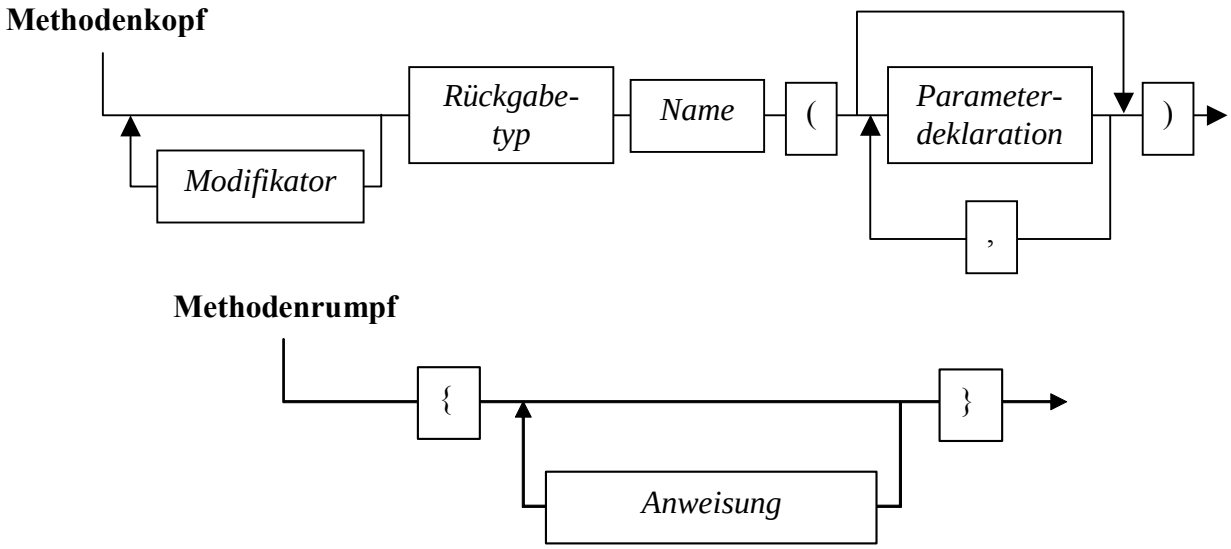
oder den Platzhalter auf zulässige (eventuell an anderer Stelle erläuterte) Weise ersetzt, sollte eine syntaktisch korrekte Klassendefinition entstehen.
Als Beispiel betrachten wir die Klasse `Bruch`:



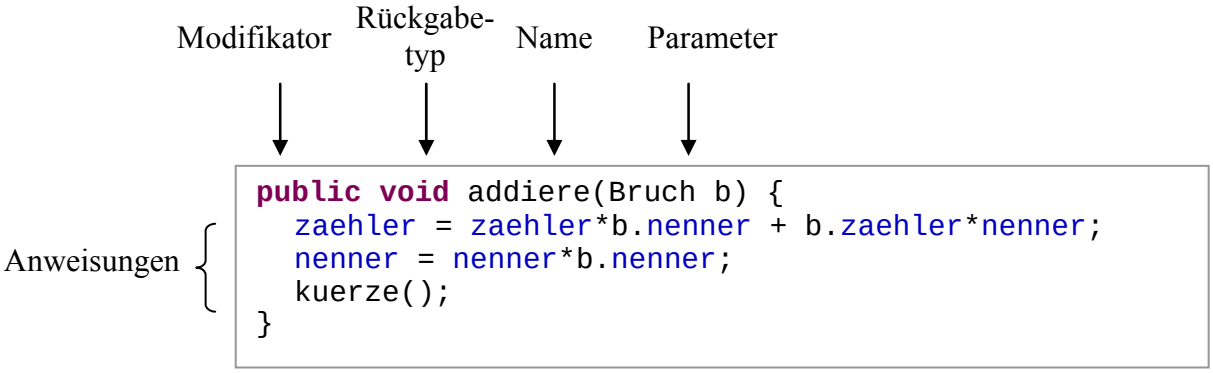

```
public class Bruch {

   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() {
      . . .
   }

}
```

Weil *ein* Syntaxdiagramm für die komplette *Methodendefinition* recht unübersichtlich ist, werden die Begriffe *Methodenkopf* und *Methodenrumpf* separat beschrieben:





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



3.1.3 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 farblichen Gestaltung vieler Beispiele. Es handelt sich um die Syntaxhervorhebungen von Eclipse, die via Zwischenablage in den Text übernommen wurden. Natürlich sind diese optischen Arbeitshilfen ausschließlich für den Programmierer gedacht, und der Compiler ist komplett farbenblind. Manuskript-Beispiele, die ohne den Weg über die Eclipse-Entwicklungsumgebung direkt in der Text gelangt sind, erkennt man am durchgehend schwarzen Text.

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");
   }
}
```

3.1.4 Kommentare

Java bietet drei Möglichkeiten, den Quelltext zu kommentieren:

- 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.

- Zwischen einer Einleitung durch `/*` und einer Terminierung durch `*/` kann sich ein ausführlicher Kommentar auch über mehrere Zeilen erstrecken, z.B.:

  ```
  /*
     Hier könnte ein Kommentar zur anschließend
     definierten Klasse stehen.
  */
  public class Beispiel {
     .     .     .
  }
  ```

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

- Vor der Definition bzw. Deklaration von Klassen, Interfaces (s. u.), Methoden oder Variablen darf ein **Dokumentationskommentar** stehen, eingeleitet mit `/**` und beendet mit `*/`, z.B.:

  ```
  /**
     Hier könnte ein Dokumentationskommentar zur anschließend
     definierten Klasse stehen.
  */
  public class Beispiel {
     .     .     .
  }
  ```

  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-Dokumentation auf folgendem Weg

  Tool Specifications > Javadoc Tool > Javadoc Tool Reference Page

3.1.5 Namen

Für Klassen, Methoden, Variablen, Parameter und sonstige Elemente eines Java-Programms benötigen wir eindeutige Namen, für die folgende Regeln gelten:

- 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 den meisten 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[6]	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 Kontextes (s. u.) eindeutig sein.

3.1.6 Einfache Ausgabe bei Konsolenanwendungen

Wie Sie bereits an einigen Beispielen beobachten konnten, lässt sich eine Konsolenausgabe in Java recht bequem über die Methodenaufrufe **System.out.print()** oder **System.out.println()** erzeugen.[7] 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, …

- wenn eine Benutzereingabe unmittelbar hinter einer Ausgabe in derselben Zeile ermöglichst 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:

- Zeichenfolgen, in doppelte Anführungszeichen eingeschlossen
  Beispiel:
  ```
  System.out.print("Hallo Allerseits!");
  ```

---

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

[7] 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 Variablen sind **static**, können also verwendet werden, ohne ein Objekt aus der Klasse **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.

- Sonstige Ausdrücke (siehe Abschnitt 3.4), deren Werte automatisch in Zeichenfolgen umgewandelt werden.

  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 „+“-Operator 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.

Eine einfache Möglichkeit zur Tastatur*ein*gabe wird später bei passender Gelegenheit vorgestellt.

3.1.7 Übungsaufgaben zu Abschnitt 3.1

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

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

2) Welche von den folgenden Bezeichnern sind unzulässig?
4you maiLink else Lösung b_____

3) Wie ist das fehlerhafte „Rechenergebnis“ in folgendem Programm zu erklären?

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.println("3.3 + 2 = " + 3.3 + 2);` `  }` `}`	`3.3 + 2 = 3.32`

Sorgen Sie mit einem Paar runder Klammern dafür, dass die gewünschte Ausgabe erscheint.

3.2 Variablen und primitive 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 Objektes werden gespeichert, solange das Objekt existiert.
- Die zur Ausführung einer Methode benötigten Daten werden bis zum Ende des Methodenaufrufs aufbewahrt.

Zum Speichern eines Wertes (z.B. einer Zahl) wird eine so genannte **Variable** verwendet, worunter Sie sich einen **benannten und typisierten Speicherplatz** vorstellen können.
Eine Variable erlaubt über ihren Namen den lesenden oder schreibenden Zugriff auf den zugeordneten Platz 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 Programmiersprache arbeiten.
Allerdings verlangt Java beim Umgang mit Variablen im Vergleich zu anderen Programmier- oder Skriptsprachen einige Sorgfalt, letztlich mit dem Ziel, Fehler zu vermeiden:

- Variablen müssen **explizit deklariert** werden.
  Wenn Sie versuchen, eine nicht deklarierte Variable zu verwenden, beschwert sich der Compiler, z.B.:

  ```
  U:\Eigene Dateien\Java\Prog\Prog.java:3: cannot resolve symbol
  symbol  : variable i
  location: class Test
                  i = 5;
                  ^
  ```

  Durch den Deklarationszwang werden z.B. Programmfehler wegen falsch geschriebener Variablennamen verhindert.
- Java ist **streng typisiert**.
  Jede Variable besitzt einen **Datentyp**, und dieser legt fest, welche Informationen (z.B. ganze Zahlen, reelle Zahlen, Zeichen) gespeichert werden können.

In obigem Beispiel wird die Variable `ivar` vom Typ **int** deklariert, der ganze Zahlen im Bereich von –2147483648 bis 2147483647 als Werte aufnehmen kann.

Die Variable `ivar` erhält auch gleich den **Initialisierungswert** 4711. Auf diese oder andere Weise müssen Sie jeder lokal (innerhalb einer Methode) definierten Variablen einen Wert zuweisen, bevor Sie zum ersten Mal lesend darauf zugreifen (vgl. Abschnitt 3.2.4).

Als wichtige Eigenschaften einer Java-Variablen halten wir fest:

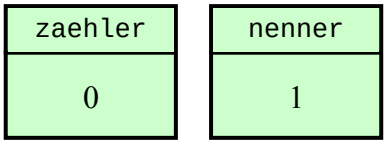
- **Name**
  Es sind beliebige Bezeichner gemäß Abschnitt 3.1.5 erlaubt. Eine Beachtung der folgenden Konventionen verbessert die Lesbarkeit des Quellcodes, insbesondere auch für andere Programmierer:
  - Variablennamen beginnen mit einem Kleinbuchstaben.
  - Besteht ein Name aus mehreren Wörtern (z.B. `isBusy`), schreibt man ab dem zweiten Wort die Anfangsbuchstaben groß.
- **Datentyp**
  Damit sind die möglichen Werte und der Speicherplatzbedarf festgelegt.
- **Aktueller Wert**
- **Ort im Hauptspeicher**
  Im Unterschied zu anderen Programmiersprachen (z.B. C++) spielt in Java die Verwaltung von Speicheradressen keine Rolle. Wir werden jedoch (nicht nur zur Förderung unserer EDV-Allgemeinbildung) zwei wichtige Speicherregionen unterscheiden (*Stack* und *Heap*), in denen Java-Programme ihre Variablen bzw. Objekte ablegen. Dieses Hintergrundwissen hilft z.B., wenn ein *stack overflow* als Fehler gemeldet wird.

3.2.1 Klassifikation von Datentypen und Variablen

Bei den von Java unterstützten Datentypen ist die die folgende Unterscheidung außerordentlich wichtig:

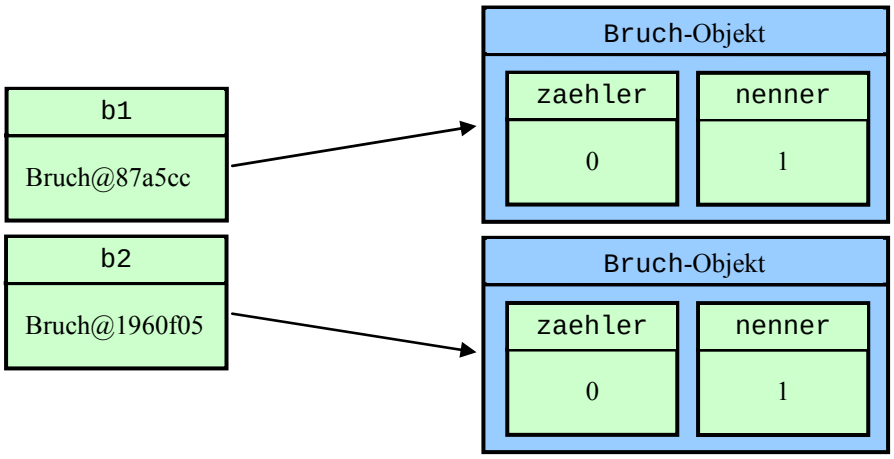
- **Primitive Typen**
  Besitzt eine Variable einen primitiven Typ, dann enthält ihr Speicherplatz unmittelbar den Wert, der über den Variablennamen angesprochen wird.
  Beispiele (aus der `Bruch`-Klassendefinition):

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



- **Referenztypen**
  Besitzt eine Variable einen Referenztyp, dann kann ihr Speicherplatz die Adresse eines Objektes aus einer bestimmten Klasse aufnehmen. Sobald ein solches Objekt erzeugt und seine Adresse der Referenzvariablen zugewiesen worden ist, kann das Objekt über die Referenzvariablen angesprochen werden.
  Man kann jede Klasse (aus dem Java-API übernommen oder selbst definiert) als Datentyp verwenden, also Referenzvariablen dieses Typs deklarieren.
  In der **main()**-Methode der Klasse `BruchAddition` 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. 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 werden uns nicht mit solchen Adressen plagen, sondern die dort abgelegten Objekte über Referenzvariablen ansprechen.
Jedes `Bruch`-Objekt enthält die Instanzvariablen `zaehler` und `nenner` vom primitiven Typ **int**.

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 3.2.5).
- **Instanzvariablen** (Elementvariablen, Eigenschaften)
  Jedes Objekt (synonym: jede *Instanz*) einer Klasse verfügt über einen vollständigen Satz der Instanzvariablen (Eigenschaften) der Klasse. So besitzt z.B. jedes Objekt der Klasse `Bruch` einen `zaehler` und einen `nenner`. Auf Instanzvariablen kann in allen Methoden der eigenen Klasse zugegriffen werden. Wenn entsprechende Rechte eingeräumt wurden, ist dies auch in Methoden fremder Klassen möglich.
- **Klassenvariablen**
  Diese Variablen beziehen sich auf eine gesamte Klasse, nicht auf einzelne Instanzen. Z.B. hält man oft in einer Klassenvariablen fest, wie viele Objekte einer Klasse bereits erzeugt worden sind. Auch für Klassenvariablen gilt, dass neben den klasseneigenen Methoden bei entsprechender Rechtevergabe auch Methoden fremder Klassen zugreifen dürfen.

In Abschnitt 0 werden wir ausschließlich mit *lokalen* Variablen arbeiten. Im Zusammenhang mit der systematischen Behandlung der objektorientierten Programmierung werden die Instanzvariablen (Eigenschaften) und die 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.

In diesem Abschnitt wurden zwei Einteilungskriterien für Java-Variablen vorgestellt, die gemeinsam folgendes Klassifikations-Schema ergeben:

		Einteilung nach Zuordnung		
		Lokale Variable	Instanzvariable	Klassenvariable
Einteilung nach Datentyp (Inhalt)	Primitiver Datentyp			
	Referenzvariable			

Die Erläuterungen zur Klassifikation von Datentypen und Variablen sind etwas kompliziert geraten. Bis jetzt müssen Sie nur die Ausführungen verstanden haben, die für *lokale* Variablen von *primitivem* Datentyp relevant sind.

3.2.2 Primitive Datentypen

Als *primitiv* bezeichnet man in Java die auch in älteren Programmiersprachen bekannten *Standardtypen* zur Aufnahme von einzelnen Zahlen, Zeichen oder Wahrheitswerten:

Typ	Beschreibung	Werte	Speicher-bedarf in Bit
byte	Diese Variablentypen speichern ganze Zahlen. Beispiel: `int alter = 31;`	–128 … 127	8
short		–32768 … 32767	16
int		–2147483648 ... 2147483647	32
long		–9223372036854775808 … 9223372036854775807	64
float	Variablen vom Typ **float** können reelle Zahlen mit einer Genauigkeit von mindestens 7 Dezimalstellen speichern. Beispiel: `float pi = 3.141593f;` **float**-Literale (s. u.) benötigen den Suffix **f** (oder **F**).	Minimum: $-3.4028235 \cdot 10^{38}$ Maximum: $3.4028235 \cdot 10^{38}$ Kleinster Betrag: $1.4 \cdot 10^{-45}$ (IEEE-754)	32 (1 für das Vorzeichen, 8 für den Exponenten, 23 für die Mantisse)
double	Variablen vom Typ **double** können reelle Zahlen mit einer Genauigkeit von mindestens 15 Dezimalstellen speichern. Beispiel: `double pi = 3.1415926535898;`	Minimum: $-1.7976931348623157 \cdot 10^{308}$ Maximum: $1.7976931348623157 \cdot 10^{308}$ Kleinster Betrag: $4.9 \cdot 10^{-324}$ (IEEE-754)	64 (1 für das Vorzeichen, 11 für den Exponenten, 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 Zeichensatz. Daher zählt **char** zu den integralen (ganzzahligen) Datentypen. Beispiel: `char zeichen = 'j';` **char**-Literale (s. u.) sind mit *einfachen* Anführungszeichen einzurahmen.	Unicode-Zeichen	16
boolean	Variablen vom Typ **boolean** können Wahrheitswerte aufnehmen. Beispiel: `boolean cond = false;`	**true**, **false**	1

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 C/C++ fällt noch 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.

3.2.3 Vertiefung: Darstellung reeller Zahlen

Bei den Datentypen **float** und **double** werden auch „relativ glatte“ Zahlen nicht unbedingt genau gespeichert, wie das folgende Programm demonstriert:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.println(1.3 - 1.3f);` `  System.out.println(1.25 - 1.25f);` `  }` `}`	`4.768371586472142E-8` `0.0`

Während die **float**-Zahl 1,3 tatsächlich nur mit 7 Stellen Genauigkeit abgespeichert wird, treten bei der **float**-Zahl 1,25 *keine* Darstellungsfehler auf. Diese Ergebnisse sind durch das zugrunde liegende **IEEE-754** – Format für die Darstellung von reellen Zahlen zu erklären.
Es handelt sich um ein **Gleitkommaformat**, wobei jede Zahl als Produkt aus 3 getrennt zu speichernden Faktoren dargestellt wird:

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

Als Basis dient beim IEEE-Standard 754 die 2.
Ferner werden alle Zahlen *normalisiert*, d.h. auf eine Mantisse im Intervall [1,0; 2) gebracht, z.B.:

$$25{,}28 = 1{,}58 \cdot 2^4$$

Im ersten Bit einer binären Gleitkommavariablen wird das Vorzeichen gespeichert (0: positiv, 1: negativ).
Für den Exponenten stehen 8 (**float**) bzw. 11 (**double**) Bit zur Verfügung. Allerdings sind im Exponenten die Werte 0 und 255 (**float**) bzw. 0 und 2047 (**double**) für Spezialfälle (z.B. +/-Unendlich) reserviert. Um Beträge kleiner 1 darstellen zu können, wird der Exponent mit einer Verschiebung um den Wert 127 (**float**) bzw. 1023 (**double**) abgespeichert, so dass z.B. beim Exponenten 0 im Speicher der Wert

$$01111111_2 = 127$$

landet.
Die führende 1 der normalisierten Mantisse wird *nicht* abgespeichert, und für den Rest stehen 23 (**float**) bzw. 52 (**double**) Bit zur Verfügung, so dass die binäre Genauigkeit 24 bzw. 53 Bit beträgt.

Eine **float**- bzw. **double**-Variable mit dem Vorzeichen *v* (0 oder 1), dem Exponenten *e* und der Restmantisse *m* (jeweils dezimal interpretiert) speichert den Wert:

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

In der folgenden Tabelle finden Sie die **float**-Darstellungen einiger Zahlen:

Wert	**float-Darstellung**		
	Vorz.	**Exponent**	**Mantisse**
1,0	0	01111111	00000000000000000000000
1,25	0	01111111	01000000000000000000000
2,0	0	10000000	00000000000000000000000
2,75	0	10000000	01100000000000000000000
-3,5	1	10000000	11000000000000000000000

Nun kommen wir endlich zur Erklärung der eingangs dargestellten Genauigkeitsunterschiede beim Speichern der Zahlen 1,25 bzw. 1,3. Das *i*-te Bit in der Restmantisse einer **float**- oder **double**-Variablen hat die Wertigkeit 2^{-i}, so dass sich ihr dezimaler Restmantissenwert folgendermaßen ergibt:

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

Während die Restmantisse

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

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

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

Sehr aufmerksame Leser werden sich darüber wundern, wieso die Tabelle mit den elementaren Datentypen in Abschnitt 3.2.2 z.B.

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

als betragsmäßig kleinsten **float**-Wert nennt, obwohl der minimale Exponent nach obigen Überlegungen -127 beträgt, was zum dezimalen Exponentialfaktor

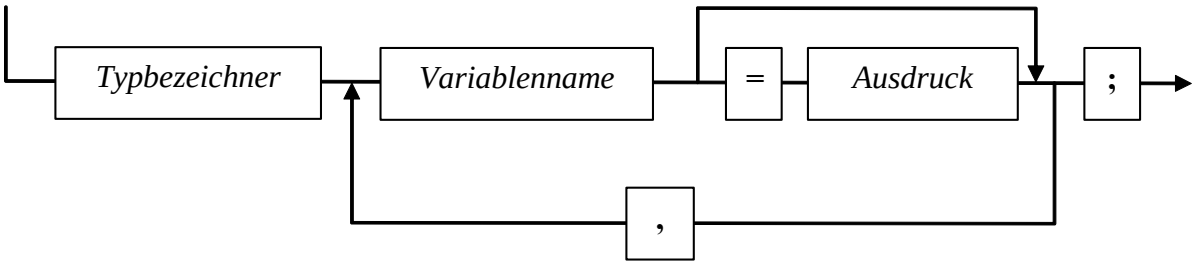
$$10^{-38}$$

führt. Nun hat man allerdings die bisher beschriebene normalisierte Darrstellung um eine *denormalisierte* ergänzt, um eine glattere Annäherung an die 0 zu ermöglichen.

3.2.4 Variablendeklaration, Initialisierung und Wertzuweisung

In Java-Programmen muss jede Variable vor ihrer ersten Verwendung deklariert[8] werden. Dabei sind auf jeden Fall der Name und der Datentyp anzugeben, wie das Syntaxdiagramm zur **Variablendeklarations-Anweisung** zeigt:

Variablendeklarationsanweisung



Neben den primitiven Datentypen können auch Klassen (aus dem Java-API oder selbst definiert) in einer Variablendeklaration verwendet werden, z.B.:

```
int zaehler; // Instanzvariable in der Klasse Bruch
Bruch b1 = new Bruch(); // lokale Variable in main() von BruchAddition
```

Im zweiten Beispiel wird per **new**-Operator ein `Bruch`-Objekt erzeugt und dessen Adresse in die 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.

---

[8] 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.

Wir betrachten vorläufig nur *lokale* Variablen, die innerhalb einer Methode existieren. Diese können an beliebiger Stelle im Methoden-Quellcode deklariert werden, aus nahe liegenden Gründen jedoch vor ihrer ersten Verwendung.

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.

Neu deklarierte Variablen kann man optional auch gleich **initialisieren**, also auf einen gewünschten Wert setzen.
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);
  }
}
```

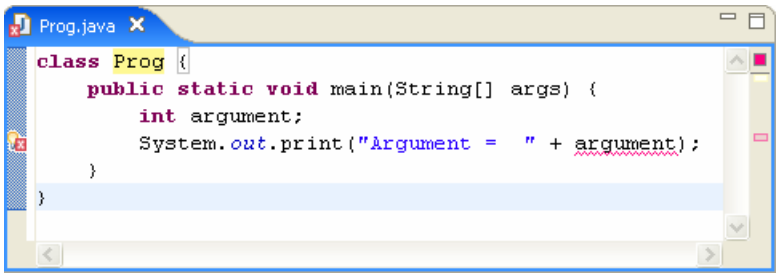
Der JDK-Compiler meint dazu:

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

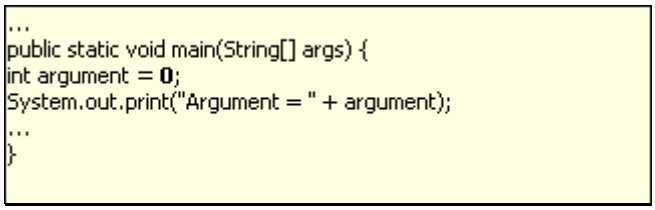
Ähnlich äußert sich auch der Eclipse-Compiler:

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

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

Unsere Entwicklungsumgebung gibt nach einem Mausklick auf das Fehlersymbol neben der betroffenen Zeile

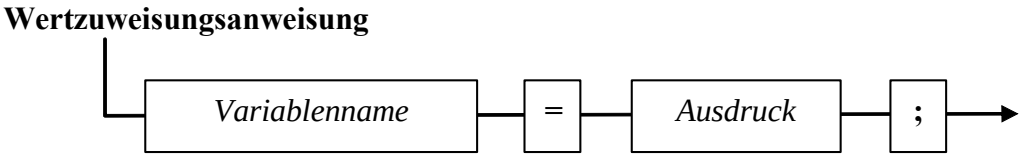


sogar konkrete Hinweise zur Verbesserung des Quellcodes:



Weil Instanz- und Klassenvariablen automatisch mit dem 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:



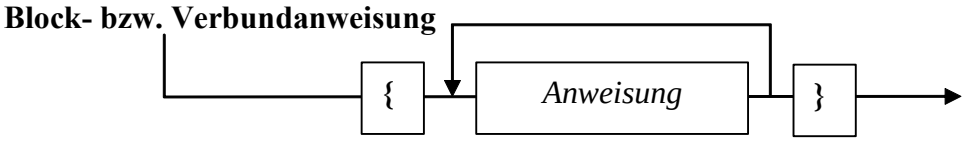
Beispiel: `ggt = az;`

Durch diese Wertzuweisungsanweisung aus der `kuerze()`-Methode unserer `Bruch`-Klasse wird die **int**-Variable `ggt` auf den Wert der **int**-Variablen `az` gesetzt.

U.a. haben Sie mittlerweile auch schon zwei Sorten von Java-Anweisungen kennen gelernt: Variablendeklaration und Wertzuweisung.

3.2.5 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 eines Blockes dürfen sich selbstverständlich auch wiederum Blockanweisungen befinden. Einfacher ausgedrückt: Blöcke dürfen geschachtelt werden. Oft treten Blöcke zusammen mit Bedingungen oder Wiederholungen (s. u.) auf, z.B.:

Quellcode	Ausgabe
`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);` `  }` `}`	`Wert1 = 1` `Wert2 = 2`

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 Blockes. Nur in diesem **Gültigkeitsbereich** (engl. *scope*) kann sie über ihren Namen angesprochen werden.
Beim Verlassen des Blockes wird der belegte Speicher frei gegeben, so dass die im obigen Beispiel auskommentierte Zeile folgende Fehlermeldung des Eclipse-Compilers produzieren würde:

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

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

Bei hierarchisch geschachtelten Blöcken ist es in Java *nicht* erlaubt, auf mehreren Stufen Variablen mit identischem Namen zu deklarieren. Mit dem Verzicht auf diese kaum sinnvolle C++ - Option haben die Java-Designer eine Möglichkeit beseitigt, schwer erkennbare Programmierfehler zu begehen.

Bei der übersichtlichen Gestaltung von Java-Programmen ist das Einrücken von Anweisungsblöcken sehr zu empfehlen. In Bezug auf die räumliche Anordnung (vor allem der einleitenden Blockklammer) haben sich unterschiedliche Stile entwickelt, z.B.:

```
if (wert1 == 1)
{
  int wert2 = 2;
  System.out.println("Wert2 = "+wert2);
}
```

```
if (wert1 == 1) {
  int wert2 = 2;
  System.out.println("Wert2 = "+wert2);
}
```

Wählen bzw. entwerfen Sie sich eine Regel, und halten Sie diese möglichst durch (zumindest innerhalb eines Programms).

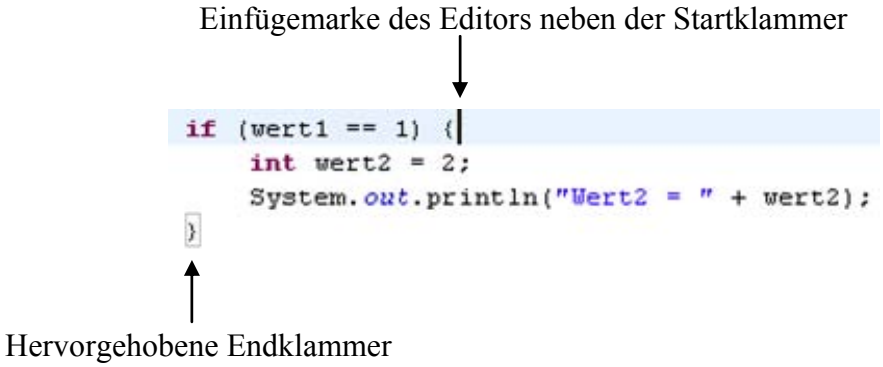
Bei Eclipse kann ein markierter Block aus mehreren Zeilen mit

<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:



3.2.6 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 Quelltext über ihren Variablennamen angesprochen werden, denn:

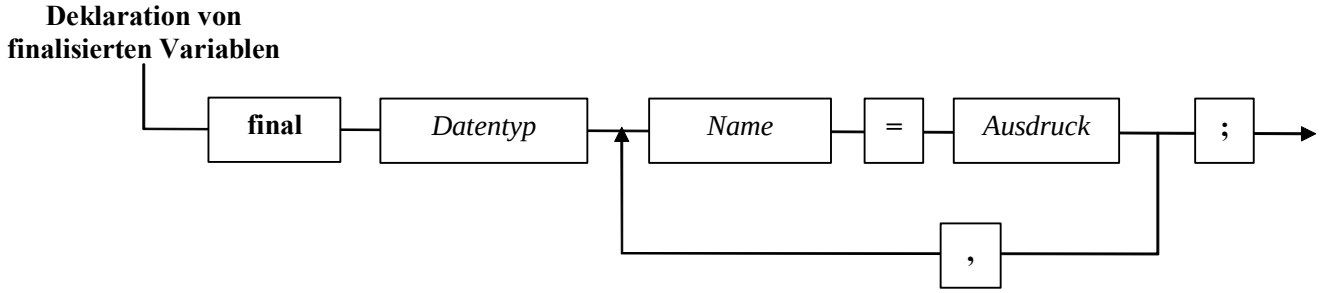
- Bei einer späteren Änderung des Wertes ist nur die Quellcode-Zeile mit der Variablendeklaration und -initialisierung betroffen.
- Der Quellcode ist leichter zu lesen.

Beispiel:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  final double MWST = 1.19;`	`Brutto: 119.0`

```
    double netto = 100.0, brutto;
    brutto = netto * MWST;
    System.out.println("Brutto: " + brutto);
  }
}
```

Variablen, die nach ihrer Initialisierung im gesamten Programmverlauf garantiert denselben Wert besitzen sollen, deklariert man als **final**:



Die Unterschiede im Vergleich zur gewöhnlichen Variablendeklaration:

- Am Anfang steht der Modifikator **final**.
- Eine finalisierte Variable *muss* natürlich initialisiert werden. Dabei ist ein Ausdruck zu verwenden, den der *Compiler* berechnen kann.

Man schreibt üblicherweise die Namen von finalisierten Variablen (Konstanten) komplett in Großbuchstaben, um sie im Quelltext gut von anderen Sprachelementen unterscheiden zu können. Besteht ein Name aus mehreren Wörtern, trennt man sie der Lesbarkeit halber durch einen Unterstrich (z.B.: `MAXIMALE_QUOTE`).

3.2.7 Literale

Die im Quellcode auftauchenden expliziten Werte bezeichnet man als *Literale*. Wie Sie aus dem Abschnitt 3.2.6 wissen, ist es oft sinnvoll, Literale *innerhalb von Konstanten-Deklarationen* zu verwenden, z.B.:

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

Auch die Literale besitzen in Java stets einen Datentyp, wobei in Bezug auf die primitiven Datentypen einige Regeln zu beachten sind:

3.2.7.1 Ganzzahlliterale

Ganzzahlige Werte haben in Java den Datentyp **int**, wenn nicht durch den Suffix **L** oder **l** der Datentyp **long** erzwungen wird, z.B.:

```
final long BETRAG = 2147483648L;
```

Mit dem Präfix „0x“ (oder auch „0X“) werden ganzzahlige Werte im *Hexadezimalsystem* mit der Basis 16 ausgedrückt. Die Anweisungen:

```
int i = 11, j = 0x11;
System.out.println("i = " + i + ", j = " + j);
```

liefern die Ausgabe:

```
i = 11, j = 17
```

Den dezimalen Wert 17 für die Variable `j` erhält man aufgrund der Wertigkeiten im Hexadezimalsystem folgendermaßen:

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

Etwas tückisch ist der Präfix für die (selten benötigten) Zahlen im *Oktalsystem* mit der Basis 8. Die harmlos wirkenden Anweisungen:

```
int i = 11, j = 011;
System.out.println("i = " + i + ", j = " + j);
```

liefern ein auf den ersten Blick verblüffendes Ergebnis:

```
i = 11, j = 9
```

Durch ein anderes Beispiel:

```
int i = 8, j = 08;
```

kommen wir dem Rätsel auf die Spur. Hier meldet der Compiler:

```
Das Literal Octal 08 (digit 8) des Typs int liegt außerhalb des gültigen Bereichs
at Prog.main(Prog.java:3)
```

Die führende Null ist keinesfalls irrelevant, sondern fungiert als Präfix für Werte im Oktalsystem, so dass wir den Wert der Variablen `j` nun nachvollziehen können:

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

Weil im Oktalsystem nur die Ziffern 0 bis 7 erlaubt sind, beschwert sich der Compiler zu Recht über die Ziffernfolge „08“.

Wer bislang mit den nichtdezimalen Zahlensystemen wenig vertraut ist, muss *nicht* befürchten, dass diese im weiteren Kursverlauf eine nennenswerte Rolle spielen.

3.2.7.2 Reellwertige Literale[9]

Zahlen mit Dezimalpunkt oder Exponent sind in Java vom Typ **double**, wenn nicht durch den Suffix **F** oder **f** der Datentyp **float** erzwungen wird, z.B.:

```
9.78f
```

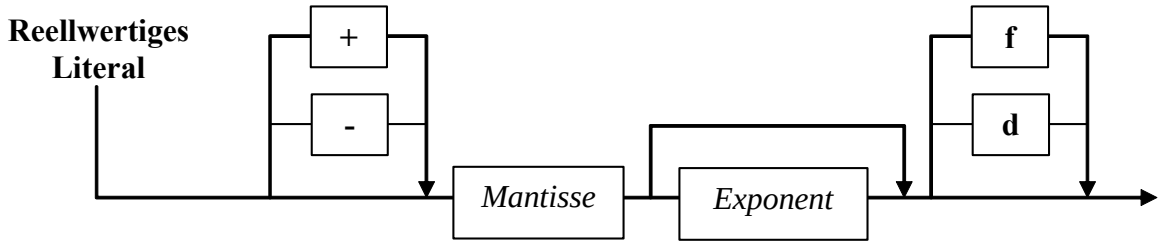
Die Java-Compiler achten bei Wertzuweisungen streng auf die Typkompatibilität. Z.B. führt die folgende Zeile:

```
final float PI = 3.141593;
```

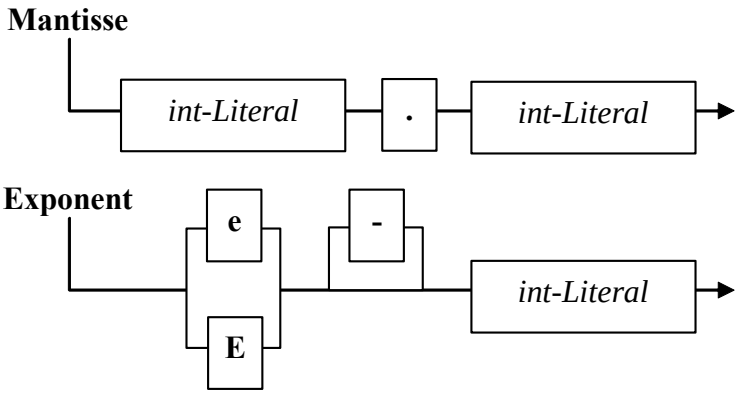
zu der folgenden Fehlermeldung des Eclipse-Compilers:

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

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



[9] Hier wird der üblichere Ausdruck *Gleitkommaliteral* vermieden, weil in praktisch allen Programmiersprachen ein Dezimal*punkt* zu verwenden ist.



Bei den in reellwertigen Literalen auftretenden **int**-Literalen sind die oben beschriebenen Präfixe (0, 0x, 0X) und Suffixe (L, l) *nicht* erlaubt.
Beispiele:

```
-9.78f
0.45875e-20
```

3.2.7.3 char-Literale

char-Literale werden in Java mit *einfachen* Hochkommata eingerahmt. Es sind erlaubt:

- **Einfache Zeichen**
  Beispiel:
  ```
  char a = '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`
  - ein Einfaches oder doppeltes Hochkomma sowie der Rückwärts-Schrägstrich:
    `\'`
    `\"`
    `\\`

  Beispiel:
  ```
  char rs = '\\';
  ```
- **Unicode-Escape-Sequenzen** mit einer hexadezimalen Unicode-Zeichennummer nach der Einleitung durch **\u**
  So lassen sich Zeichen nutzen, die per Tastatur nicht einzugeben sind.
  Beispiel:
  ```
  char alpha = '\u03b1';
  ```
  Im Konsolenfenster werden die Unicode-Zeichen oberhalb von \u00ff in der Regel als Fragezeichen dargestellt. Auf einer graphischen Benutzeroberfläche 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 ein Ton (ASCII-Nummer 7) erzeugt werden:

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

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

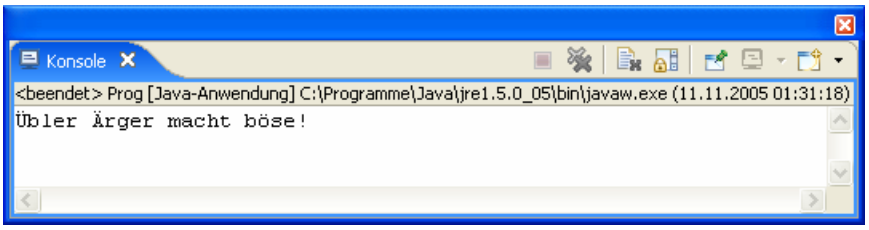
Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.println("Übler Ärger macht böse!");` `  }` `}`	`▄bler ─rger macht b÷se!`

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 (Cp1252).

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:



3.2.7.4 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.

3.2.7.5 boolean-Literale

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

```
boolean cond = true;
```

3.2.8 Übungsaufgaben zu Abschnitt 3.2

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

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

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

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

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

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

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

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

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

5) 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);
  }
}
```

3.3 Einfache Eingabe bei Konsolenanwendungen

3.3.1 Beschreibung der Klasse `Simput`

Für die Übernahme von Tastatureingaben ist in Java eine ziemlich wasserdichte Methodologie vorgeschrieben, wie das folgende Beispielprogramm zur Berechnung der Fakultät zeigt, das sich auf die Standardbibliothek der Java-Version 1.4 beschränkt:

```
import java.io.*;
class Fakul {
  public static void main(String[] args) {
    int i, argument;
    double fakul = 1.0;
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    System.out.print("Argument:  ");
    try {
      argument = Integer.parseInt(in.readLine());
      for (i = 1; i <= argument; i++)
        fakul = fakul * i;
      System.out.println("Fakultaet: " + fakul);
    } catch(Exception e) {
      System.out.println("Falsche Eingabe");
    }
  }
}
```

Seit der Version 5.0 (alias 1.5) übernimmt die Klasse **Scanner** aus dem Paket **java.util** einen Teil der Arbeit:[10]

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

Bei beiden Varianten kommen aufwändige Programmiertechniken zum Einsatz, mit denen wir uns momentan noch nicht beschäftigen wollen. Damit wir bis zum Erlernen dieser Techniken in Demonstrations- bzw. Übungsprogrammen nicht auf Tastatureingaben verzichten müssen, habe ich eine Klasse namens `Simput` mit Methoden zur bequemen Übernahme von Tastatureingaben 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

---

[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.

Mit Hilfe der Klassenmethode `Simput.gint()` lässt sich das Fakultätsprogramm etwas einfacher realisieren. Die nach wie vor benötigte **for**-Wiederholungsanweisung wird in Abschnitt 3.6.3 behandelt.

```
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);
  }
}
```

`Simput.gint()` erwartet vom Benutzer eine per **Enter**-Taste quittierte Eingabe und versucht, diese als **int**-Literal 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.
Der Erfolg bzw. Misserfolg eines `gint()`-Aufrufs wird in der booleschen Klassenvariablen `Simput.status` durch die Werte **true** bzw. **false** dokumentiert, und die Klassenvariable `Simput.errdes` enthält im Fehlerfall darüber hinaus eine nähere Erläuterung.
In obigem Beispielprogramm ignoriert die aufrufende Methode `main()` allerdings die diagnostischen Informationen. Wir werden ebenfalls in vielen Beispielen der folgenden Abschnitte den `gint()`-Rückgabewert der Kürze halber ohne Statuskontrolle 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 3.6.2 behandelt.

Quellcode	Ein- und Ausgabe
`class Fakul {` `  public static void main(String args[]) {` `  int i, argument;` `  double fakul = 1.0;` `  System.out.print("Argument:  ");` `  argument = Simput.gint();` `  if (Simput.status == true) {` `  for (i = 1; i <= argument; i += 1)` `  fakul = fakul * i;` `  System.out.println("Fakultaet: " + fakul);` `  }` `  else` `  System.out.println(Simput.errdes);` `  }` `}`	`Argument:  ff` `Falsche Eingabe!` `Eingabe konnte nicht konvertiert werden.`

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

- `static public char gchar()`
  Liest ein Zeichen von der Konsole
- `static public double gdouble()`
  Liest eine reelle Zahl (mit Dezimal*punkt*) von der Konsole

Außerdem sind Methoden mit einer moderneren Fehlerbehandlung über den Exception-Mechanismus vorhanden, auf die wir aber aus didaktischen Gründen vorläufig noch verzichten.

3.3.2 Simput-Installation für die JDK-Werkzeuge und Eclipse

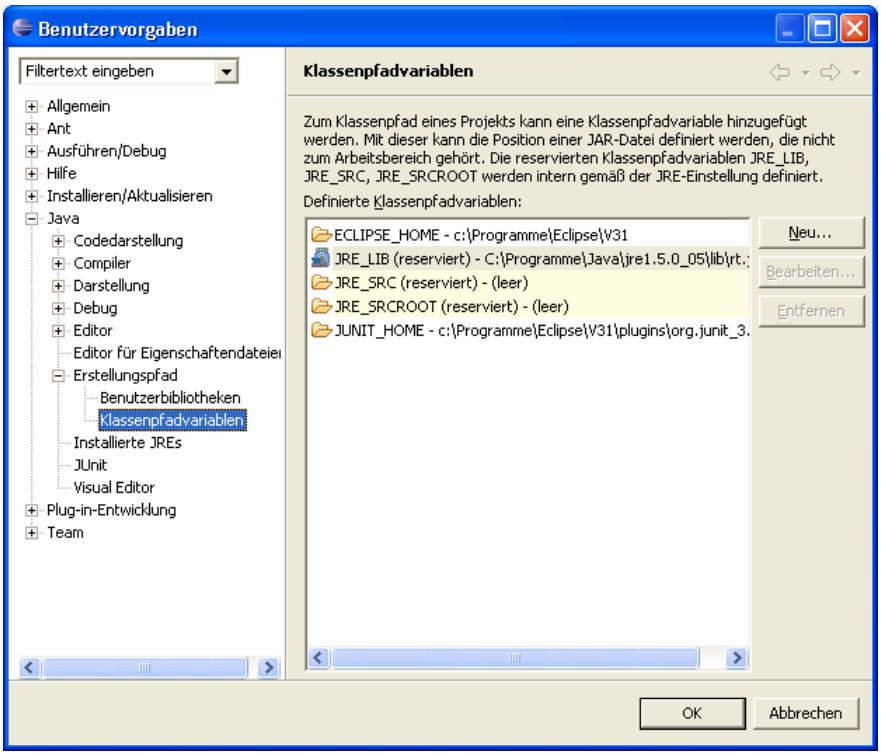
Damit beim Übersetzen per **javac.exe** und/oder beim Ausführen per **java.exe** die `Simput`-Klasse mit ihren Methoden verfügbar ist, muss die Datei **Simput.class** entweder im aktuellen Verzeichnis liegen oder über den CLASSPATH erreichbar sein, wenn sie nicht bei jedem Compiler- oder Interpreteraufruf 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, die man auch als **Paket** (engl.: *package*) oder als **Klassenbibliothek** bezeichnen kann (siehe Abbildung in Abschnitt 1.2). Im selben Ordner wie die Bytecode-Datei **Simput.class** finden Sie daher auch die Archivdatei **Simput.jar**. Wir werden uns in Abschnitt 6 mit dem Aufbau und der Erstellung von Paketen ausführlich beschäftigen.

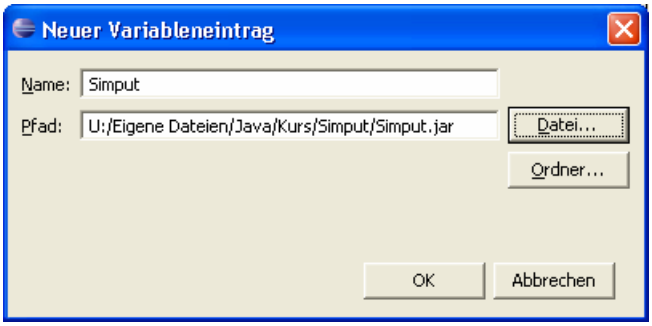
Namen und Pfad dieser Archivdatei hinterlegt man am besten in einer **Klassenpfadvariablen** auf Arbeitsbereichsebene, damit das Archiv in einzelnen Projekten ohne Pfadangaben angesprochen werden kann. Nach

Fenster > Benutzervorgaben > Java > Erstellungspfad > Klassenpfadvariablen

kann man in der folgenden Dialogbox



über den Schalter **Neu** die Definition einleiten

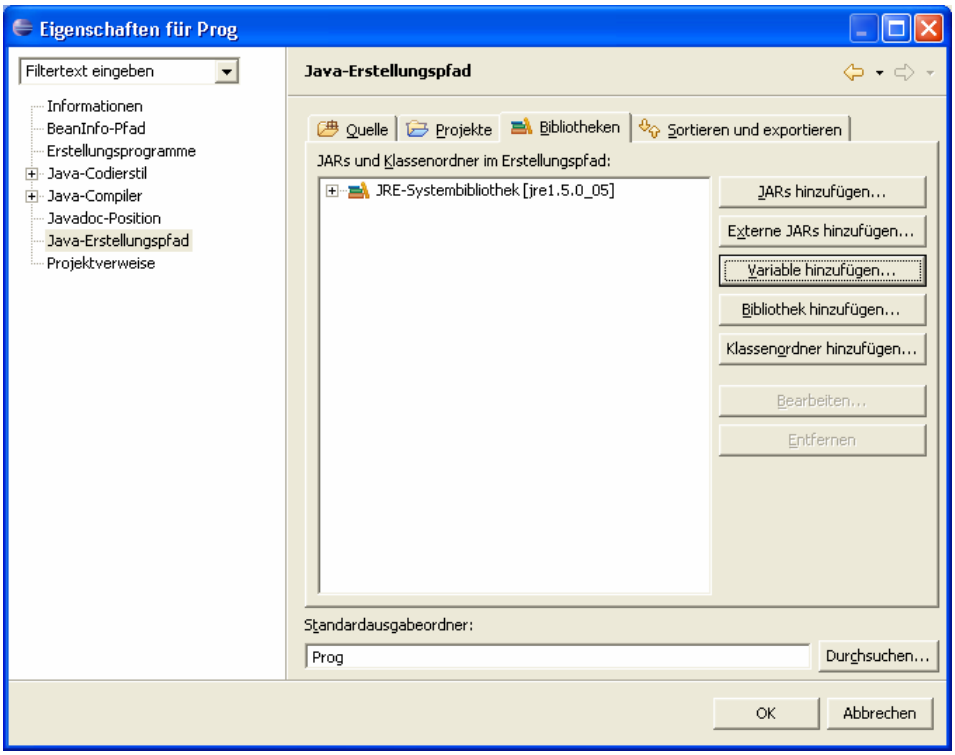


und über den Schalter **Datei** das Archiv wählen.

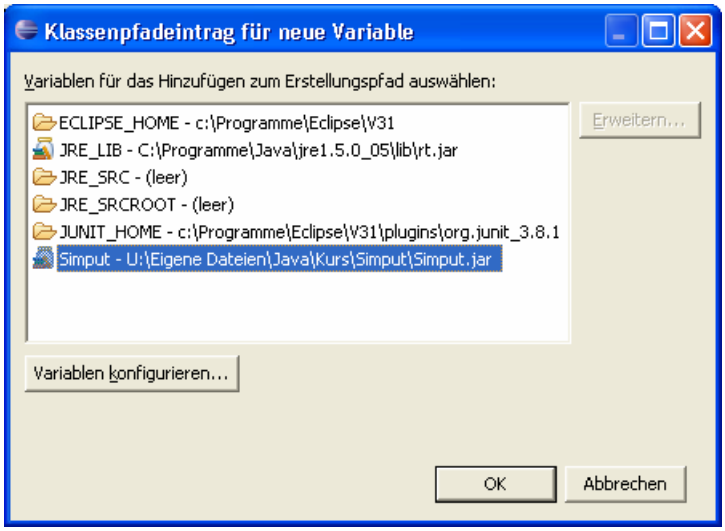
Soll ein konkretes Projekt die Klassenpfadvariable nutzen, muss sie in seinem Eigenschaftsdialog (erreichbar via Kontextmenü zum Projekteintrag im Paket-Explorer) über

Java-Erstellungspfad > Bibliotheken

hinzugefügt werden.



In der folgenden Dialogbox wählt man die gewünschte Variable:



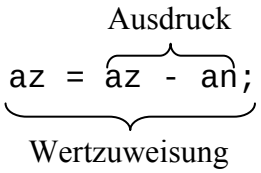
3.3.3 Übungsaufgabe zu Abschnitt 3.3

Führen Sie die in Abschnitt 3.3.2 beschriebene Eclipse-Konfiguration aus, und lassen Sie das in Abschnitt 3.3.1 beschriebene Fakultätsprogramm mit `Simput.gint()` – Aufruf laufen.
Testen Sie auch die `Simput`-Methoden `gdouble()` und `gchar()`.

3.4 Operatoren und Ausdrücke

Im Zusammenhang mit der Variablendeklaration und der Wertzuweisung haben wir das Sprachelement *Ausdruck* ohne Erklärung benutzt; diese soll nun nachgeliefert werden. Im aktuellen Abschnitt 3.4 werden wir Ausdrücke als wichtige Bestandteile von Java-Anweisungen recht detailliert untersuchen. Große Begeisterungsstürme werden dabei wohl nicht aufbrausen, doch wir benötigen diese handwerklichen Grundlagen, um die Methoden und damit die Handlungskompetenzen unserer Klassen bzw. Objekte korrekt zu implementieren. Schließlich besteht ein erheblicher Teil der von einem Computer-Programm verrichteten Arbeit darin, Ausdrücke auszuwerten.

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



Durch diese Anweisung aus der `kuerze()`-Methode unserer `Bruch`-Klasse wird der lokalen **int**-Variablen `az` der Wert das 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 die Ausdrücke auch als *temporäre Variablen* auffassen, die folglich einen **Datentyp** und einen **Wert** besitzen.[11]

Schon bei einem Literal, einer Variablen oder einem Methodenaufruf handelt es sich um einen Ausdruck.

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 komplexe Ausdrücke, wobei Typ und Wert von den Argumenten und den Operatoren abhängen.

Beispiel: `2 + 1.5`
Hier resultiert der **double**-Wert 3,5.

In der Regel beschränken sich Java-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` auf den neuen Wert 13 gesetzt.

---

[11] Besteht ein Ausdruck allerdings aus einem Methodenaufruf mit dem Pseudorückgabetyp **void** (siehe unten), dann liegt *kein* Wert vor.

Die meisten Java-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: `-a`
Die Vorzeichenumkehr erwartet *ein* numerisches Argument.

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

Weil als Argumente einer Operation auch *Ausdrücke* von passendem Ergebnistyp erlaubt sind, können beliebig komplexe Ausdrücke aufgebaut werden. Unübersichtliche Exemplare sollten jedoch als potentielle Fehlerquellen vermieden werden.

3.4.1 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 daher gelegentlich als **numerische Ausdrücke** bezeichnet.

Es hängt von den Datentypen der Operanden ab, ob bei den Berechnungen die **Ganzzahl**- oder die **Gleitkommaarithmetik** zum Einsatz kommt. Besonders auffällig sind die Unterschiede im Verhalten des Divisionsoperators, z.B.:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  int i = 2, j = 3;` `  double a = 2.0;` `  System.out.println(i/j);` `  System.out.println(a/j);` `  }` `}`	`0` `0.6666666666666666`

Bei der Ganzzahldivision werden die Stellen nach dem Dezimaltrennzeichen abgeschnitten, was gelegentlich durchaus erwünscht ist.
Im Zusammenhang mit dem Über- bzw. Unterlauf (siehe Abschnitt 3.5) 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 3.4.6). 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** oder **int**.	**Ganzzahl-arithmetik**	**int**
Beide Operanden haben einen integralen Typ, und mindestens 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 numerische Ausdrücke stehen, und *Var* eine numerische Variable vertritt:

Operator	Bedeutung	Beispiel	
		Programmfragment	**Ausgabe**
-Num	**Vorzeichenumkehr**	`int i = -2;` `System.out.println(-i);`	`2`
Num1 + Num2	**Addition**	`int i = 2, j = 3;` `System.out.println(i+j);`	`5`
Num1 – Num2	**Subtraktion**	`double a = 2.6, b = 1.1;` `System.out.println(a-b);`	`1.5`
*Num1 * Num2*	**Multiplikation**	`int i = 4, j = 5;` `System.out.println(i*j);`	`20`
Num1 / Num2	**Division**	`System.out.println(8.0/5);` `System.out.println(8/5);`	`1.6` `1`
Num1 % Num2	**Modulo (Divisionsrest)** Dieser Operator wird meist auf integrale Argumente angewandt, ist jedoch auch bei **float** und **double** erlaubt.	`int i = 19, j = 5;` `System.out.println(i%j);`	`4`
++Var *--Var*	**Präinkrement bzw. -dekrement** Als Argumente sind hier nur Variablen erlaubt. *++Var* liefert *Var* + 1 erhöht *Var* um 1 *--Var* liefert *Var* - 1 verringert *Var* um 1	`int i = 4;` `double a = 0.2;` `System.out.println(++i +"\n"+` `--a);`	`5` `-0.8`
Var++ *Var--*	**Postinkrement bzw. -dekrement** Als Argumente sind hier nur Variablen erlaubt. *Var++* liefert *Var* erhöht *Var* um 1 *Var--* liefert *Var* verringert *Var* um 1	`int i = 4;` `System.out.println(i++ +"\n"+` `i);`	`4` `5`

Bei den Prä- und Postinkrement- bzw. dekrementoperatoren ist zu beachten, dass sie *zwei* Effekte haben:

- Das Argument wird ausgelesen, um den Wert des Ausdrucks zu ermitteln.
- Der Wert des Argumentes wird verändert.
  Wegen dieses „Nebeneffektes“ sind Prä- und Postinkrement- bzw. dekrementausdrücke im Unterschied zu sonstigen arithmetischen Ausdrücken bereits vollständige Anweisungen (vgl. Abschnitt 3.6.1):

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

3.4.2 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**[12] zur Verfügung. In folgendem Programm wird die Methode **pow()** zur Potenzberechnung genutzt:

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

Alle Methoden der Klasse **Math** sind **static**, können also verwendet werden, ohne ein Objekt aus der Klasse **Math** zu erzeugen. Später werden wir uns ausführlich mit der Verwendung von Methoden aus Java-Standardpaketen befassen.

In syntaktischer Hinsicht halten wir fest, dass ein Methodenaufruf einen Ausdruck darstellt und auch als Argument komplexerer Ausdrücke verwendet werden darf, sofern die Methode einen passenden Rückgabewert liefert.

3.4.3 Vergleichsoperatoren

Bislang haben wir *numerische* Ausdrücke betrachtet, die mit Hilfe von arithmetischen Operatoren aus einfachen Elementen aufgebaut werden. Durch Verwendung von *Vergleichsoperatoren* und *logischen Operatoren* entstehen Ausdrücke mit dem Ergebnistyp **boolean**, die als **logische Ausdrücke** bezeichnet werden sollen. Sie können die booleschen Werte **true** (wahr) und **false** (falsch) annehmen und eignen sich dazu, *Bedingungen* zu formulieren, z.B.:

```
if (arg > 0)
  System.out.println(Math.log(arg));
```

Ein **Vergleich** ist ein besonders einfach aufgebauter logischer Ausdruck, bestehend aus zwei Ausdrücken und einem Vergleichsoperator.
In der folgenden Tabelle mit den von Java unterstützten Vergleichsoperatoren stehen

- *Expr1* und *Expr2* für beliebige, miteinander vergleichbare Ausdrücke
- *Num1* und *Num2* für numerische Ausdrücke (vom Datentyp **byte**, **short**, **int**, **long**, **char**, **float** oder **double**)

[12] 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.
Das Paket **java.lang** wird im Unterschied zu allen anderen API-Paketen automatisch importiert.

Operator	Bedeutung	Beispiel	
		Programmfragment	Ausgabe
Expr1 == Expr2	**Gleichheit**	`int i = 2, j = 3;` `System.out.println(i==j);`	`false`
Expr1 != Expr2	**Ungleichheit**	`int i = 2, j = 3;` `boolean erg = i!=j;` `System.out.println(erg);`	`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. Einer der häufigsten 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.
Im ersten **println()**-Aufruf des folgenden Beispielprogramms wird das Ergebnis eines Vergleichs auf die Konsole geschrieben[13]:

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

Nach dem Entfernen eines Gleichheitszeichens wird aus dem logischen Ausdruck ein *Wertzuweisungsausdruck* (siehe Abschnitt 3.4.7) mit dem Datentyp **int** und dem Wert 2:

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

Der versehentlich gebildete Ausdruck sorgt nicht nur für eine unerwartete Ausgabe, sondern verändert auch den Wert den Variablen `i`, was im weiteren Verlauf eines Programms recht unangenehm werden kann.

Mit Zusammenhang mit den Gleitkommatypen, insbesondere bei **float** und **double**, gilt es beim Identitätsoperator (auch bei korrekter Schreibweise) eine weitere Problematik zu beachten: Aufgrund der inhärenten Genauigkeitsprobleme bei den genannten Datentypen muss man in der Regel an Stelle des Identitätsvergleichs eine Technik wählen, die technisch bedingte Abweichungen von der reinen Mathematik toleriert, z.B.:

[13] Wir wissen schon aus Abschnitt 3.1.6, dass **println()** einen beliebigen Ausdruck verarbeiten kann, wobei automatisch eine Zeichenfolgenr-Repräsentation erstellt wird.

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  final double DRF = 1.0e-14;` `  double d1 = 10.0 - 9.9;` `  double d2 = 0.1;` `  System.out.println(d1 == d2);` `  System.out.println(Math.abs((d1 - d2)/d1) < DRF);` `  }` `}`	`false` `true`

Nach diesem Vorschlag werden zwei **double**-Werte d_1 und d_2 dann als „numerisch“ identisch betrachtet, wenn der relative Abweichungsbetrag[14] kleiner ist als das Zweifache des maximalen Rundungsfehlers in einer normalisierten **double**-Mantisse (vgl. Abschnitt 3.2.3) mit 14 signifikanten *Nachkommastellen*:

$$\left|\frac{d_1 - d_2}{d_1}\right| < 1{,}0 \cdot 10^{-14}$$

Für die Betrachtung der *relativen* Abweichung spricht, dass Mantissen-Rundungsfehler (verursacht durch die beschränkte Speichergenauigkeit) in Abhängigkeit vom Exponenten zu sehr unterschiedlichen Gesamtfehlern führen.

Bei konkreten Algorithmen kann durchaus eine gröbere Indifferenzschwelle angemessen sein, wenn mit einer Fehlerkumulation zu rechnen ist.

In einer späteren Übungsaufgabe (siehe Abschnitt 3.6.4) sollen Sie sich erneut mit dem Problem der begrenzten Maschinengenauigkeit befassen.

3.4.4 Logische Operatoren

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

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

[14] Für den vorgeschlagenen Begriff der numerischen Identität ist die Wahl der Bezugsgröße (d_1 oder d_2) beliebig.

Argument 1 *La1*	Argument 2 *La2*	**Logisches UND** *La1* **&&** *La2* *La1* **&** *La2*	**Log. ODER** *La1* **||** *La2* *La1* **|** *La2*	**Exkl. ODER** *La1* **^** *La2*
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

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

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

Der Unterschied zwischen den beiden logischen UND-Operatoren **&&** und **&** bzw. zwischen den beiden logischen ODER-Operatoren **||** und **|** ist für Einsteiger vielleicht etwas unklar, weil man

spontan den nicht ausgewerteten logischen Ausdrücken keine Bedeutung beimisst. Allerdings ist es in Java erlaubt, „Nebeneffekte“ in einen logischen Ausdruck einzubauen, wie das Beispiel

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

zeigt. Der Postinkrementoperator erhöht beim Auswerten des rechten Ausdrucks den Wert der Variablen `i`. Eine solche Auswertung wird jedoch in der folgenden Variante des Beispiels unterdrückt, wenn bereits nach Auswertung des linken Ausdrucks das Gesamtergebnis **false** fest steht:

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

Wie der Tabelle auf Seite 77 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 | noch für *bitorientierte* Operatoren verwendet, die allerdings zwei *integrale* Argumente (z.B. **int**) erwarten (siehe Abschnitt 3.4.5).

3.4.5 Bitorientierte Operatoren

Über unseren momentanen Bedarf hinaus gehend bietet Java einige Operatoren zur bitweisen Manipulation von Variableninhalten. Statt einer systematischen Darstellung der verschiedenen Operatoren (siehe z.B. Java-Tutorial, SUN Inc. 2005) beschränken wir uns auf ein Beispielprogramm, das zudem generell nützliche Einblicke in die Speicherung von **char**-Daten 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 **CBit** liefert die Unicode-Kodierung zu einem vom Benutzer erfragten Zeichen Bit für Bit. Dabei kommt die Methode `gchar()` aus unserer Bequemlichkeitsklasse `Simput` zum Einsatz, die das vom Benutzer eingetippte und mit **Enter** quittierte Zeichen abliefert (vgl. Abschnitt 3.3.1). Außerdem wird mit der **for**-Schleife eine Wiederholungsanweisung verwendet, die erst einige Abschnitte später offiziell vorgestellt wird:

Quellcode	Ausgabe
`class CBit {` `  public static void main(String[] args) {` `  char cbit;` `  System.out.print("Zeichen: ");` `  cbit = Simput.gchar();` `  System.out.print("Unicode: ");` `  for(int i = 15; i >= 0; i--) {` `  if ((1 << i & cbit) != 0)` `  System.out.print("1");` `  else` `  System.out.print("0");` `  }` `  System.out.println();` `  }` `}`	`Zeichen: x` `Unicode: 0000000001111000`

Der **Links-Shift-Operator** << in:

```
1 << i
```

verschiebt die Bits in der binären Repräsentation der Ganzzahl 1 um `i` Stellen nach links.
Von den 32 Bit, die ein **int**-Wert insgesamt belegt (s. o.), interessieren im Augenblick nur die rechten 16. Bei der 1 erhalten wir:

```
0000000000000001
```

Im 10. Schleifendurchgang (`i` = 6) geht dieses Muster z.B. über in:

```
0000000001000000
```

Nach dem Links-Shift- kommt der **bitweise UND-Operator** zum Einsatz:

```
1 << i & cbit
```

Das Operatorzeichen **&** wird leider in doppelter Bedeutung verwendet: Wenn beide Argumente vom Typ **boolean** sind, wird **&** als *logischer* Operator interpretiert (siehe Abschnitt 3.4.4). 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 1 enthält, wenn *beide* Argumentmuster an dieser Stelle eine 1 besitzen und anderenfalls eine 0.
Bei `cbit = 'x'` ist das Unicode-Bitmuster

```
0000000001111000
```

zu bearbeiten, und `1 << i & cbit` liefert z.B. bei `i` = 6 das Muster:

```
0000000001000000
```

Das von `1 << i & cbit` erzeugte Bitmuster hat den Typ **int** und kann daher mit der 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 1 hat.

Wir hätten dasselbe Bitmuster auch bei einer Untersuchung des **int**-Wertes 120 gefunden, weil das kleine ‚x' im Unicode-Zeichensatz diese Nummer besitzt. Durch die spielerischen Übungen mit den bitorientierten Operatoren soll nicht der falsche Eindruck entstehen, dass man beim Umgang mit dem Unicode-Zeichensatz einzelne Bits zu beachten hat.

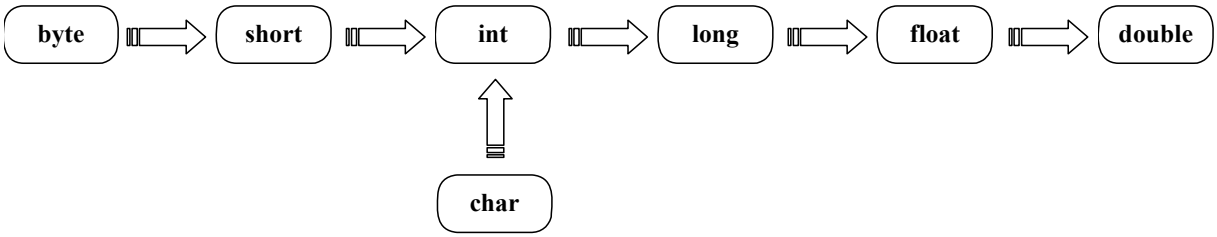
3.4.6 Typumwandlung (Casting) bei primitiven Datentypen

Beim Auswerten des Ausdrucks

```
2.0/7
```

trifft der Divisionsoperator auf ein **double**- und auf ein **int**-Argument, so dass nach der Tabelle in Abschnitt 3.4.1 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 Konvertierung 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 {` `  public static void main(String[] args) {` `  long i = 9223372036854775313L;` `  double d = i;` `  System.out.println(i);` `  System.out.printf("%.2f\n", d);` `  }` `}`	`9223372036854775313` `9223372036854776000,00`

Bei der zweiten Ausgabe kommt die seit Java 5 (alias 1.5) verfügbare Methode **printf()** zum Einsatz, die als ersten Parameter eine Formatierungszeichenfolge kennt. Im Beispiel wird die Ausgabe der Variablen `d` im Gleitkommaformat mit zwei Dezimalstellen veranlasst.

Das folgende Programm demonstriert u.a. die auf den ersten Blick recht verblüffende automatische Konversion von **char** nach **int**:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  int i = 2, j = 3;` `  System.out.println('x'/2);` `  System.out.println("Wg. i=" + i + " und j=" + j);` `  System.out.println("ist i>j " + (i>j));` `  }` `}`	`60` `Wg. i=2 und j=3` `ist i>j false`

Gelegentlich gibt es gute Gründe, über den **Casting-Operator** eine *explizite* Typumwandlung zu erzwingen. Im folgenden Programm 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:

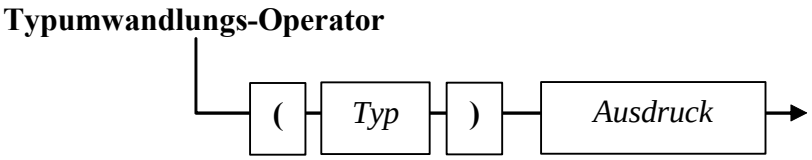
Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  double a = 3.14159;` `  System.out.println((int) 'x');` `  System.out.println((int) a);` `  System.out.println((int) 1.45367e50);` `  }` `}`	`120` `3` `2147483647`

In der zweiten Ausgabeanweisung des Beispielprogramms wird per Casting-Operation der ganzzahlige Anteil eines **double**-Wertes ermittelt.
Wie die dritte Ausgabe zeigt, sind bei einer explizit angeforderten **einschränkenden Konvertierung** schwere Programmierfehler möglich, wenn die Wertebereiche der beteiligten Variablen bzw. Datentypen nicht beachtet werden.

Während sich Java-Compiler strikt 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:

```
byte a = 16;
```

Die Java-Syntax zur expliziten Typumwandlung wurde unverändert von der Programmiersprache C übernommen:



3.4.7 Zuweisungsoperatoren

Bei den ersten Erläuterungen zu Wertzuweisungen blieb aus didaktischen Gründen unerwähnt, dass in Java eine Wertzuweisung als *Ausdruck* aufgefasst wird, dass wir es also mit einem 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.

Analog zum Verhalten des Inkrement- bzw. Dekrementoperators sind auch beim Zuweisungsoperator *zwei* Effekte zu unterscheiden:

- Die als linkes Argument fungierende Variable wird verändert.
- Es wird ein Wert für den Ausdruck produziert.

In folgendem Beispiel fungiert ein Zuweisungsausdruck als Parameter für einen **println()**-Methodenaufruf:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  int ivar = 13;` `  System.out.println(ivar = 4711);` `  System.out.println(ivar);` `  }` `}`	`4711` `4711`

Beim Auswerten des Ausdrucks entsteht der an **println()** zu übergebende Wert *und* die Variable `ivar` wird verändert.

Selbstverständlich kann eine Zuweisung auch als Operand in einen übergeordneten Ausdruck integriert werden, z.B.:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  int i = 2, j = 4;` `  i = j = j * i;` `  System.out.println(i + "\n" + j);` `  }` `}`	`8` `8`

Weil bei mehrfachem Auftreten des Zuweisungsoperators eine Abarbeitung von **rechts nach links** erfolgt (vgl. Tabelle in Abschnitt 3.4.8), passiert im letzten Beispiel folgendes:

- `j * i` liefert das Ergebnis 8, das der Variablen `j` zugewiesen wird und gleichzeitig den Wert des Ausdrucks `j = j * i` darstellt.
- In der zweiten Zuweisung (bei Betrachtung von rechts nach links) wird der Wert des Ausdrucks `j = j * i` an die Variable `i` übergeben.

Wie wir spätestens seit Abschnitt 3.2.4 wissen, stellt ein Zuweisungsausdruck (z.B. `a = 3`) bereits eine vollständige **Anweisung** dar, sobald man ein Semikolon dahinter setzt. Dies gilt auch für die die Prä- und Postinkrementausdrücke (vgl. Abschnitt 3.4.1) sowie für Methodenaufrufe, jedoch **nicht** für die anderen Ausdrücke, die in Abschnitt 3.4 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;` `i += 3;`	`5`
Var **-=** *Expr*	*Var* erhält den neuen Wert *Var* - *Expr*.	`int i = 10, j = 3;` `i -= j*j;`	`1`
Var ***=** *Expr*, *Var* **/=** *Expr*	*Var* erhält den neuen Wert *Var* * *Expr* bzw. *Var* / *Expr*.	`int i = 10;` `i /= 5;`	`2`
Var **%=** *Expr*	*Var* erhält den neuen Wert *Var* % *Expr*.	`int i = 10;` `i %= 5;`	`0`

Konditionaloperator
Wiederum eher für Schreibfaule als für Anfänger geeignet ist der Konditionaloperator, mit dessen Hilfe sich die bedingungsabhängige Entscheidung zwischen zwei Ausdrücken sehr kompakt formulieren lässt. Eine Besonderheit des Konditionaloperators besteht darin, dass er *drei* Argumente verarbeitet, welche durch die Zeichen **?** und **:** sortiert werden. In der folgenden Tabelle steht *La* für einen logischen Ausdruck, *Expr1* und *Expr2* stehen für Ausdrücke mit einem zuweisungskompatiblem Typ:

Syntax	Erläuterung	Beispiel	
		Programmfragment	Ausgabe
La **?** *Expr1* **:** *Expr2*	Ist der logische Ausdruck *La* wahr, liefert der Konditional-Operator den Wert von *Expr1*, anderenfalls den Wert von *Expr2*.	`int i = 12, j = 3, k;` `k = (i > j)?i:j;` `System.out.println(k);`	`12`

3.4.8 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 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 numerischen Ausdrücken an die mathematische Regel *Punktrechnung geht vor Strichrechnung*.

2) Assoziativität (Auswertungsrichtung)
Steht ein Argument zwischen zwei Operatoren mit gleicher Priorität, 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 Bezug auf die Nebeneffekte mancher Operatoren ist zu beachten, dass Java einen Klammerausdruck erst dann auswertet, wenn sein Wert benötigt wird, im Fall der bedingten logischen Operatoren also eventuell nie, z.B.:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  int a = 1, b= 2, c = 3;` `  boolean kond = a++ == 2 && (b++ == 3 && c++ == 4);` `  System.out.println(kond);` `  System.out.println(a+" "+b+" "+c);` `  }` `}`	`false` `2 2 3`

In der folgenden Tabelle sind die bisher behandelten Operatoren in absteigender Priorität aufgelistet. Gruppen von Java-Operatoren mit gleicher Priorität sind durch fette horizontale Linien begrenzt. 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**
V	Variable
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.

3.4.9 Übungsaufgaben zu Abschnitt 3.4

1) Welche Werte und Datentypen besitzen die folgenden Ausdrücke?

```
6/4*2.0
(int)6/4.0*3
3*5+8/3%4*5
```

2) Welche Werte haben die Variable `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) % 2;
    erg2 = (++i == j ? 7 : 8) % 2;
    System.out.println("erg1 = "+erg1+"\nerg2 = "+erg2);
  }
}
```

3) 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 recht 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*

4) 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(1.0) = 2.7182818284590455`

Hinweise:

- Suchen Sie mit Hilfe der JDK-Dokumentation zur Klasse **Math** im API-Paket **java.lang** eine passende Methode.
- Zum Einlesen des Argumentes können Sie die Methode `gdouble()` aus unserer Bequemlichkeitsklasse `Simput` verwenden, die eine vom Benutzer (mit oder ohne Dezimalpunkt) eingetippte und mit **Enter** quittierte Zahl als **double**-Wert abliefert.

5) 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-Betrages können Sie die Methode `gdouble()` aus unserer Bequemlichkeitsklasse `Simput` verwenden.

3.5 Über- und Unterlauf bei numerischen Variablen

Wie Sie inzwischen wissen, haben die numerischen Datentypen jeweils einen bestimmten Wertebereich. Liegt das Ergebnis eines Ausdrucks außerhalb des Zieltyp-Wertebereichs, ist ein korrektes Speichern nicht möglich. Bei der Auswertung eines komplexen Ausdrucks kann ein Wertebereichs-Problem auch schon „unterwegs“ auftreten. Im betroffenen Programm ist mit einem mehr oder weniger gravierenden Fehlverhalten zu rechnen, so dass ein solcher Über- oder Unterlauf unbedingt vermieden bzw. rechtzeitig diagnostiziert werden muss.

3.5.1 Überlauf bei Ganzzahltypen

Ohne besondere Vorkehrungen stellt ein Java-Programm im Falle eines Ganzzahl-Überlaufs 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
```

Oft kann ein Überlauf durch Wahl eines geeigneten Datentyps verhindert werden. Im letzten Beispiel sollte für die kritische Variable an Stelle von **int** der Typ **long** verwendet werden:

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

Dann erhält man das korrekte Ergebnis:

```
2147483647 + 5 = 2147483652
```

3.5.2 Unendliche und undefinierte Werte bei den Typen float und double

Auch bei den Gleitkommatypen **float** und **double** kann ein Überlauf auftreten, obwohl der unterstützte Wertebereich hier weit größer ist. Dabei kommt es weder zu einem sinnlosen Zufallswert noch zu einem Ausnahmefehler, sondern zum speziellen Gleitkommawert +/- Unendlich, mit dem 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("Unendl. + 10 =\t\t"+ (bigd + 10));
    System.out.println("13.0/0.0 =\t\t" + (13.0 / 0.0)+"");
  }
}
```

liefert die Ausgabe:

```
Double.MAX_VALUE =      1.7976931348623157E308
Double.MAX_VALUE * 10 = Infinity
Unendl. + 10 =          Infinity
13.0/0.0 =              Infinity
```

Mit Hilfe der Unendlich-Werte „gelingt“ offenbar sogar die Division durch 0.

Java weigert sich aber zu Recht, „∞ - ∞“ zu berechnen. Hier resultiert wie bei der Division von 0 durch 0 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("Unendl. – Unendl. =\t"+(bigd-bigd));
    System.out.println("0.0 / 0.0 =\t\t" + (0.0/0.0));
  }
}
```

Es liefert die Ausgabe:

```
Unendl. – Unendl. =   NaN
0.0 / 0.0 =    NaN
```

Zu den letzten Beispielprogrammen ist noch anzumerken, dass man über die öffentliche Eigenschaft **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 {` `  public static void main(String[] args) {` `  double d = 0.0/0.0;` `  if (Double.isNaN(d))` `  System.out.println("0/0 ist NaNsense");` `  }` `}`	`0/0 ist NaNsense`

Die **if**-Anweisung (siehe Abschnitt 3.6.2) sorgt dafür, dass die Ausgabe nur unter einer bestimmten Bedingung (Rückgabewert **true** von **isNaN()**) erfolgt.

3.5.3 Unterlauf bei den Typen float und double

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

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  double smalld = Double.MIN_VALUE;` `  System.out.println(smalld);` `  smalld /= 2.0;` `  System.out.println(smalld);` `  }` `}`	`4.9E-324` `0.0`

Die öffentliche Eigenschaft **Double.MIN_VALUE** enthält den betragsmäßig kleinsten Wert, der in einer **double**-Variablen gespeichert werden kann.

3.6 Anweisungen (zur Ablaufsteuerung)

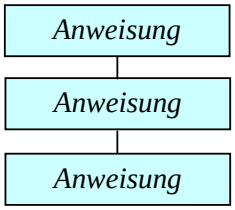
Wir haben uns im Abschnitt 0 über elementare Sprachelemente zunächst mit (lokalen) **Variablen** und primitiven **Datentypen** vertraut gemacht. Dann haben wir gelernt, aus Variablen, Literalen und Methodenaufrufen mit Hilfe von **Operatoren** komplexe **Ausdrücke** zu bilden. Diese wurden entweder mit der Methode **System.out.println()** ausgegeben oder in Wertzuweisungen verwendet. Unsere Beispielprogramme bestanden im Wesentlichen aus einer **Sequenz** von **Anweisungen**, die wir nach bedarfsgerecht eingeführten Regeln gebildet haben. Nun werden wir uns systematisch mit dem allgemeinen Begriff einer Java-Anweisung und vor allem mit wichtigen Spezialfällen (bedingte Anweisungen, Schleifen) befassen.

3.6.1 Überblick

Ausführbare Programmteile, die in Java stets als Methoden von Klassen zu realisieren sind, bestehen aus Anweisungen (engl. *statements*), die nacheinander ausgeführt werden.

Am Ende von Abschnitt 3.6 werden Sie die folgenden Typen von Anweisungen kennen:

- **Variablendeklarationsanweisung**
  Die Variablendeklarationsanweisung wurde schon in Abschnitt 3.2.4 eingeführt.
  Beispiel: `int i = 1, j = 2, k;`
- **Ausdrucksanweisungen**
  Folgende Ausdrücke werden zu Anweisungen, sobald man ein Semikolon dahinter setzt:
  - Wertzuweisungen (vgl. Abschnitte 3.2.4 und 3.4.7)
    Beispiel: `k = i + j;`
  - Prä- bzw. Postinkrement- oder -dekrementoperationen
    Beispiel: `i++;`
    Hier ist nur der „Nebeneffekt“ des Ausdrucks `i++` von Bedeutung. Sein Wert bleibt ungenutzt.
  - Methodenaufrufe
    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 kann dann verwendet werden, wenn die Syntax eine Anweisung verlangt, aber nichts geschehen soll.
- **Anweisungen zur Ablaufsteuerung**
  Die Methoden der bisherigen Beispielprogramme in Abschnitt 0 bestanden meist aus einer *Sequenz* von Anweisungen, die bei jedem Programmlauf alle nacheinander ausgeführt wurden:



Die meisten Algorithmen erfordern jedoch **Fallunterscheidungen** und/oder **Iterationen**. Im weiteren Verlauf von Abschnitt 3.6 werden Java-Anweisungen voreingestellt, die eine entsprechende Ablaufsteuerung erlauben.

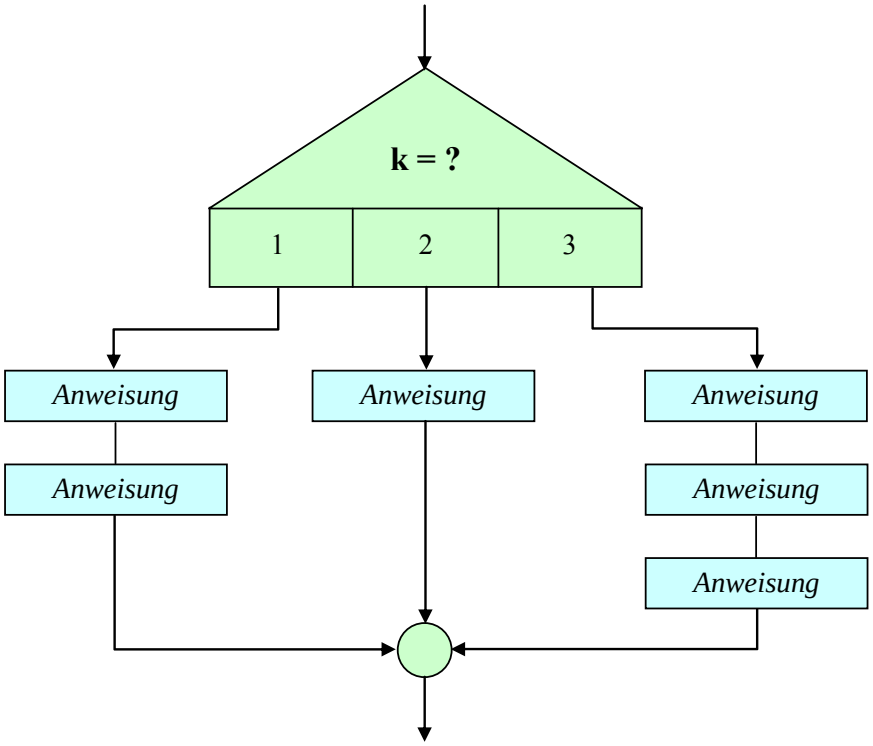
- **Block- bzw. Verbundanweisung**
  Bereits in Abschnitt 3.2.5 haben wir uns im Zusammenhang mit dem Gültigkeitsbereich für lokale Variablen mit der Blockanweisung beschäftigt. Wie gleich näher erläutert wird, fasst man z.B. *dann* mehrere Abweisungen zu einem Block zusammen, wenn diese Anweisungen unter einer gemeinsamen Bedingung ausgeführt werden sollen. Es wäre ja sehr unpraktisch, dieselbe Bedingung für jede betroffene Anweisung wiederholen zu müssen.

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.

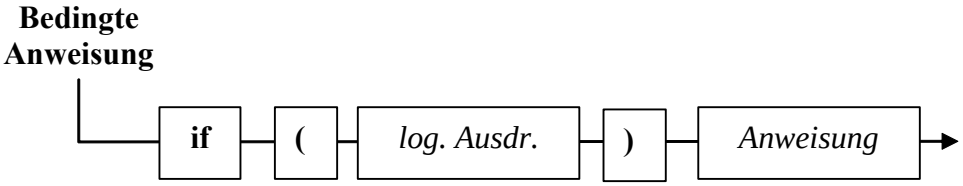
3.6.2 Bedingte Anweisung und Verzweigung

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, z.B.:



3.6.2.1 if-Anweisung

Mit der folgenden Syntax sorgt man dafür, dass eine Anweisung nur dann ausgeführt wird, wenn ein logischer Ausdruck den Wert **true** annimmt:



In diesem Beispiel wird eine Meldung ausgegeben, wenn die Variable `anz` den Wert 0 besitzt:

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

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

Soll auch etwas passieren, wenn der 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.

Im folgenden **if-else** - Beispiel wird der natürliche Logarithmus zu einer Zahl geliefert, falls diese positiv ist. Anderenfalls erscheint eine Fehlermeldung mit Alarmton (Unicode-Escape-Sequenz **\u0007**). Das Argument wird vom Benutzer über die `Simput`-Methode `gdouble()` erfragt (vgl. Abschnitt 3.3).

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

Als bedingt auszuführende Anweisung ist durchaus wiederum eine **if**- bzw. **if-else** - Anweisung erlaubt, so dass sich komplexere Fallunterscheidungen formulieren lassen:

if (*logischer Ausdruck 1***)**
  Anweisung 1
else if (*logischer Ausdruck 2***)**
  Anweisung 2
  . . .
  . . .
else if (*logischer Ausdruck k***)**
  Anweisung k
else
  Default-Anweisung

Enthält die letzte **if**-Anweisung eine **else**-Klausel, so wird die zugehörige Anweisung ausgeführt, wenn alle logischen Ausdrücke den Wert **false** haben. Die Wahl der Bezeichnung *Default-Anweisung* in der Syntaxdarstellung orientierte sich an der im Anschluss vorzustellenden **switch**-Anweisung.

Beim Schachteln von bedingten Anweisungen kann es zum genannten **dangling-else** - Problem kommen, wobei ein Missverständnis zwischen Compiler und Programmierer hinsichtlich der Zuordnung einer **else**-Klausel besteht. Im folgenden Codefragment lässt die Einrückung vermuten, dass der Programmierer die **else**-Klausel der *ersten* **if**-Anweisung zuordnen wollte:

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

Der Compiler ordnet sie 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.

Mit Hilfe von Blockklammern (nötigenfalls auch für eine einzelne Anweisung) kann die gewünschte Zuordnung erzwungen werden:

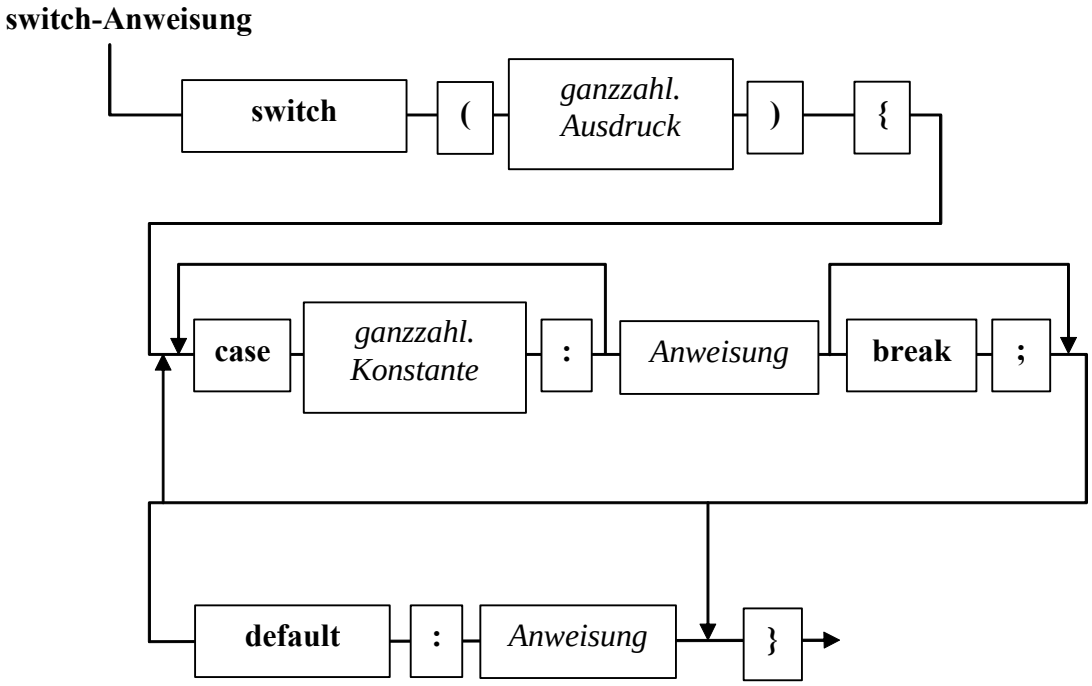
```
if (i > 0)
  {if (j > i)
    k = j;}
else
  k = i;
```

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 = i;
```

3.6.2.2 switch-Anweisung

Wenn eine Fallunterscheidung mit mehr als zwei Alternativen in Abhängigkeit von einem ganzzahligen Ausdruck vom Typ **byte**, **short**, **char** oder **int** (nicht **long**!) vorgenommen werden soll, dann ist die Mehrfachauswahl per **switch**-Anweisung weitaus handlicher als eine verschachtelte **if-else** -Konstruktion.
Der Genauigkeit halber wird die **switch**-Anweisung mit einem Syntaxdiagramm beschrieben. Wer die Syntaxbeschreibung im Quellcode-Layout bevorzugt, kann ersatzweise einen Blick auf das gleich folgende Beispiel werfen.



Stimmt der Wert des Ausdrucks mit der ganzzahligen Konstanten (finalisierte Variable oder Literal) zu 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 Auswahlwerte dieselbe Anweisung ausgeführt werden, setzt man die zugehörigen **case**-Marken unmittelbar hintereinander und lässt die Anweisung auf die letzte Marke folgen. Leider gibt es keine Möglichkeit, eine *Serie* von Fällen durch Angabe der Randwerte 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 = Character.getNumericValue(args[1].charAt(0));

      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 Lieblingsfarbe und ihre Lieblingszahl (aus dem Wertebereich von 0 bis 9) über Programmargumente (Kommandozeilenparameter) angeben, wobei die Farbe folgendermaßen durch einen Buchstaben zu kodieren ist:

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

Wer die Farbe Blau und die Zahl 7 bevorzugt, muss das Programm also folgendermaßen starten:

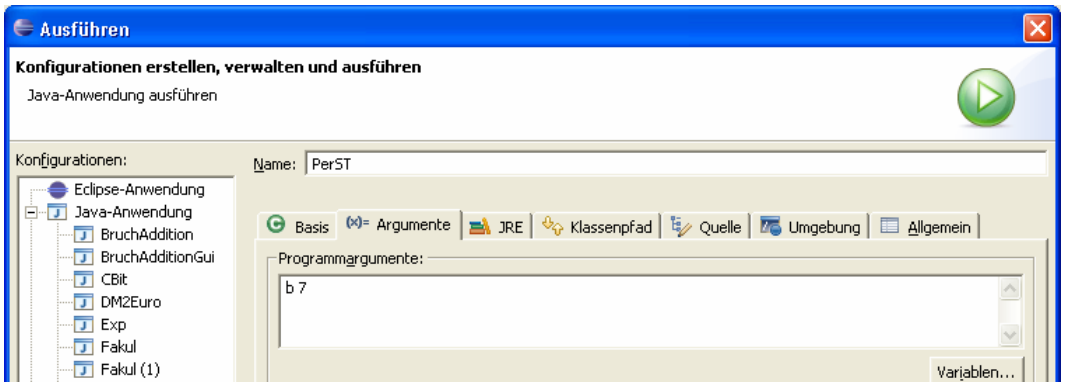
```
java PerST b 7
```

Im Programm wird jeweils nur eine Anweisung benötigt, um die Programmargumente in eine **char**- bzw. **int**-Variable zu befördern. Die zugehörigen Erklärungen werden Sie in Kürze mit Leichtigkeit verstehen:

- Das erste Element des Zeichenketten-Feldes `args` (mit der Nummer 0) wird (als Objekt der Klasse **String**) aufgefordert, die Methode **charAt()** mit dem Parameterwert 0 auszuführen. Die Methode liefert als Rückgabe das erste Zeichen (mit der Nummer 0), welches der **char**-Variablen `farbe` zugewiesen wird.
- Analog wird auch zum zweiten Element des Zeichenketten-Feldes `args` (Nummer 1) das erste Zeichen ermittelt. Dieses wird mit Hilfe der statischen Methode **getNumericValue()** aus der Klasse **Character** in eine Zahl vom Datentyp **int** gewandelt, die schließlich in der Variablen `zahl` landet.

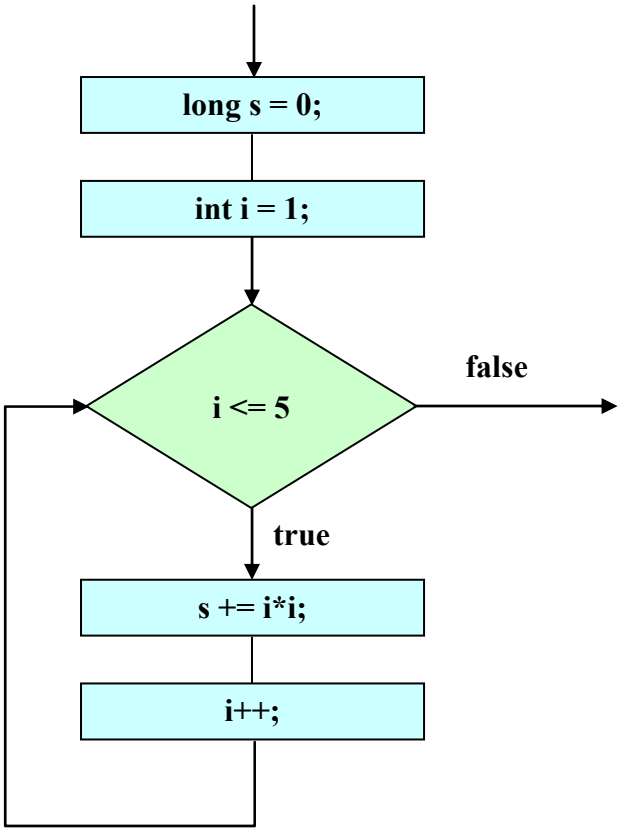
Soll das Programm in der Eclipse-Entwicklungsumgebung gestartet werden, können die Programmargumente folgendermaßen vereinbart werden:

- Menübefehl **Ausführen > Ausführen**
- In der Dialogbox **Ausführen** markiert man die zu startende **Java-Anwendung**, wählt die Registerkarte **Argumente** und trägt die gewünschten **Programmargumente** ein, z.B.:



3.6.3 Wiederholungsanweisung

Eine Wiederholungsanweisung (oder schlicht: *Schleife*) kommt zum Einsatz, wenn eine Anweisung *mehrfach* ausgeführt werden soll, wobei sich in der Regel schon der Gedanke daran verbietet, die Anweisung entsprechend oft in den Quelltext zu schreiben.
In der folgenden Abbildung 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. Zur Ablaufsteuerung wird am Anfang eine Zählvariable initialisiert und dann bei jedem Durchlauf aktualisiert (z.B. inkrementiert). Der zur Schleife gehörige Anweisungsblock wird ausgeführt, solange die Zählvariable eine festgelegte Grenze nicht überschreitet.
- **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 ein weiteres mal ausführen
- **false**: Schleife beenden

Bei der *kopf*gesteuerten **while**-Schleife wird die Bedingung *vor Beginn* eines Durchgangs geprüft, bei der *fuß*gesteuerten **do**-Schleife hingegen *am Ende*.

Die gesamte Konstruktion aus Schleifensteuerung und (Verbund-)anweisung stellt in syntaktischer Hinsicht *eine* zusammengesetzte Anweisung dar.

3.6.3.1 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 Initialisierung der Indexvariablen, die Fortsetzungsbedingung und die Aktualisierungsvorschrift untergebracht werden können. Am Ende steht die wiederholt auszuführende (Block-)Anweisung:

for (*Initialisierung*; *Bedingung*; *Aktualisierung***)**
    Anweisung

Zu den drei Bestandteilen der Schleifensteuerung sind einige Erläuterungen erforderlich, wobei hier etliche weniger typische bzw. sinnvolle Möglichkeiten weggelassen werden:

- ***Initialisierung***
  In der Regel wird man sich auf *eine* Indexvariable beschränken und dabei einen Ganzzahltyp wählen. Diese Variable wird auf einen Startwert gesetzt und nötigenfalls dabei auch deklariert, z.B.:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  long s = 0;` `  for (int i = 1; i <= 5; i++)` `  s += i*i;` `  System.out.println("Quadratsumme = " + s);` `  }` `}`	`Quadratsumme = 55`

  Deklarationen und Initialisierungen werden vor dem ersten Durchlauf ausgeführt.
  Eine im Initialisierungsteil 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 *zu Beginn* eines Schleifendurchgangs geprüft. Resultiert der Wert **true**, so findet eine weitere Wiederholung des Anweisungsteils statt, anderen falls wird die **for**-Anweisung verlassen. Folglich kann es auch passieren, dass überhaupt kein Durchlauf zustande kommt.

- ***Aktualisierung***
  Die Aktualisierung wird *am Ende* eines Schleifendurchgangs (nach Ausführung der Anweisung) vorgenommen.

Zu den (zumindest stilistisch) bedenklichen Konstruktionen, die der Compiler klaglos umsetzt, gehören **for**-Schleifenköpfe ohne Initialisierung oder ohne Aktualisierung, wobei die trennenden Strichpunkte trotzdem zu setzen sind. In solchen Fällen ist die Umlaufzahl einer **for**-Schleife natür-

lich nicht mehr aus der Schleifensteuerung abzulesen. Dies gelingt auch dann nicht, wenn eine Indexvariable in der Schleifenanweisung modifiziert wird.

Obwohl wir uns bisher nur anhand von Beispielen mit Arrays (Feldern mit einer festen Anzahl von Elementen desselben Datentyps) und mit anderen Kollektionen noch gar nicht beschäftigt haben, sollen die mit Java 5.0 (alias 1.5) eingeführten Erweiterungen der **for**-Schleife (siehe z.B. Gosling et al. 2005) 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 3.6.2.2 demonstriert, wie man über den **String[]** - Parameter der Methode **main()** auf die Zeichenfolgen zugreifen kann, welche der Benutzer beim Programmstart als Argumente angegeben hat. In folgendem Programm wird durch eine **for**-Schleife neuer Bauart jedes Element im **String**-Array `args` ausgegeben:

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

Die allgemeine Struktur der neuen **for**-Variante:

for (*ElementTyp ElementName* : *Kollektion***)**
    Anweisung

Als Kollektion akzeptiert der Compiler:

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

3.6.3.2 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 Java - Wiederholungsanweisung man zur Lösung eines konkreten Iterationsproblems benutzt.
Unter der aktuellen Abschnittsüberschrift diskutiert man traditionsgemäß die **while**- und die **do**-Schleife.

3.6.3.2.1 while-Schleife

Die **while**-Anweisung kann als vereinfachte **for**-Anweisung beschreiben kann: Wer im Kopf einer **for**-Schleife auf Initialisierung 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 *zu Beginn* eines Schleifendurchgangs geprüft. Resultiert der Wert **true**, so wird der Anweisungsteil ausgeführt, anderenfalls wird die **while**-Anweisung verlassen, eventuell noch vor dem ersten Durchgang.

Im obigen Beispielprogramm kann man nach minimalen Änderungen auch eine **while**-Schleife verwenden:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  int i = 1;` `  long s = 0;` `  while (i <= 5) {` `  s += i*i;` `  i++;` `  }` `  System.out.println("Quadratsumme = " + s);` `  }` `}`	`Quadratsumme = 55`

3.6.3.2.2 do-Schleife

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

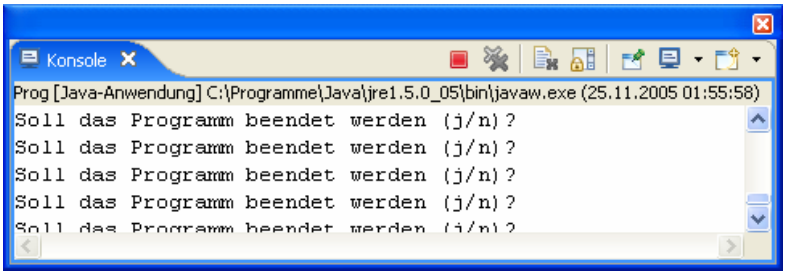
do
Anweisung
while (*Bedingung***);**

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, 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 derselben Zeile), um sie optisch von einer selbständigen **while**-Anweisung abzuheben.

Bei jeder Wiederholungsanweisung (**for**, **while** oder **do**) kann es in Abhängigkeit von der verwendeten Bedingung passieren, dass der Anweisungsteil unendlich oft ausgeführt wird. 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:



3.6.3.3 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 Auswahlanweisung zum Einsatz, z.B. in folgendem Programm zur (relativ primitiven) Primzahlendiagnose:

```
class Primitiv {
   public static void main(String[] args) {
      boolean tg;
      int i, mpk, 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.status == false || (zahl <= 2  &&  zahl != 0)) {
            System.out.println("Keine Zahl oder illegaler Wert!\n");
            continue;
         }
         if (zahl == 0) break;
         tg = false;
         mpk = (int) Math.sqrt(zahl);  //maximaler Primzahlenkandidat
         for (i = 2; i <= mpk; 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!");
   }
}
```

Man hätte die **continue**- bzw. **break**-Anweisungen zwar vermeiden können, doch werden bei dem vorgeschlagenen Verfahren die lästigen Spezialfälle (falsche Eingaben, 0 als Terminierungssignal) auf besonders übersichtliche Weise abgehakt, bevor der Kernalgorithmus startet.

Zum Kernalgorithmus der obigen Primzahlendiagnose sollte vielleicht noch erläutert werden, warum die Suche nach einem Teiler des Primzahlkandidaten bei dessen Wurzel enden kann:
Sei d ein Teiler der positiven, ganzen Zahl z, d.h. es gibt eine Zahl k (≥ 2) mit

$$z = k \cdot d$$

Dann ist auch k ein Teiler von z, und es gilt:

$$d \leq \sqrt{z} \quad \text{oder} \quad k \leq \sqrt{z}$$

Anderenfalls wäre das Produkt $k \cdot d$ ja größer als z. Wir haben also folgendes Ergebnis: Wenn eine Zahl z keinen Teiler kleiner oder gleich $\sqrt{z}$ hat, dann ist es eine Primzahl.

Zur Berechnung der Wurzel verwendet das Beispielprogramm die statische Methode **sqrt()** aus der Klasse **Math**, über die man sich bei Bedarf in der JDK-Dokumentation informieren kann.

3.6.4 Übungsaufgaben zu Abschnitt 3.6

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 {` `  public static void main(String[] args) {` `  boolean b = false;` `  if (b = false)` `  System.out.println("b ist false");` `  else` `  System.out.println("b ist true");` `  System.out.println("\nKontr.ausg. von b: "+b);` `  }` `}`	`b ist true` `Kontr.ausg. von b: false`

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

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

Warum liefert es zum Buchstaben *a* die Nummer 3, obwohl für diesen Fall die Anweisung

```
nr = 1
```

vorhanden ist?

4) Erstellen Sie eine Variante des Primzahlen-Testprogramms aus Abschnitt 3.6.3.3, das ohne **break**- bzw. **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 im Übungsaufgaben-Abschnitt 3.4.9 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 3.2.2). Folglich kann ein Rechner die **double**-Werte 1,0 und $1{,}0 + 0{,}5^i$ ab einem bestimmten Exponenten *i* nicht mehr voneinander unterscheiden. Bestimmen Sie mit einem Testprogramm den größten ganzzahligen Index *i*, für den man noch erhält:

$$1{,}0 + 0{,}5^i > 1{,}0$$

8) In dieser Aufgabe sollen Sie verschiedene Varianten von Euklids Algorithmus zur Bestimmung des größten gemeinsamen Teilers 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 1.1.2). 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 diese Problem vermieden, weil an Stelle der Subtraktion die Modulo-Operation zum Einsatz kommt, basierend auf folgendem Theorem: Für zwei natürliche Zahlen *u* und *v* ist der ggT gleich dem ggT von *u* und *u* % *v* (*u* modulo *v*). Der alternative Euklidische Algorithmus läuft folgendermaßen ab:

> Es wird geprüft, ob *u* durch *v* teilbar ist. Trifft dies zu, ist der *v* der ggT. Anderenfalls ersetzt man:
> *u* durch *v*
> *v* durch *u* % *v*
> und startet das Verfahren neu.

Um den Zeitaufwand für beide Varianten zu messen, eignet sich die statische Methode

> **public static long currentTimeMillis()**

aus der Klasse **System** im Paket **java.lang** (siehe API-Dokumentation). Sie liefert die aktuelle Zeit in Millisekunden (seit dem 1. Januar 1970).

Für die Beispielwerte *u* = 999000999 und *v* = 36 liefern beide Euklid-Varianten sehr verschiedene Ergebnisse:

```
ggT-Bestimmung mit Eu-
klid (Differenz)

Erste Zahl:   999000999

Zweite Zahl:  36

ggT:          9

Benoet. Zeit: 63
Millisek.
```

```
ggT-Bestimmung mit Eu-
klid (Modulo)

Erste Zahl:   999000999

Zweite Zahl:  36

ggT:          9

Benoet. Zeit: 0
Millisek.
```

4 Klassen und Objekte

Klassen sind die wesentlichen Bestandteile von Java-Programmen. Primär handelt es sich hier um Baupläne für Objekte, die mit Eigenschaften (Instanzvariablen) und Handlungskompetenzen (Methoden) ausgestattet werden.
In einem Programm werden natürlich nicht nur Klassen definiert, sondern auch **Objekte** aus diversen (selbst definierten und vorgegebenen) Klassen erzeugt. Diese führen auf eine Botschaft hin die zugehörige Methode aus und liefern entsprechende Ergebnisse zurück. Im Idealfall entwickelt sich der Programmablauf als „kompetente und freundliche" Interaktion zwischen Objekten, wobei sich natürlich auch die Benutzer beteiligen dürfen.

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. Auf die in Abschnitt 1 gewürdigte objektorientierte *Analyse* kann dabei aus Zeitgründen kaum eingegangen werden. Insbesondere bei größeren Projekten ist der Einsatz von objektorientierten Analyse- und Entwurfstechniken sehr empfehlenswert (siehe z.B. Balzert 1999).

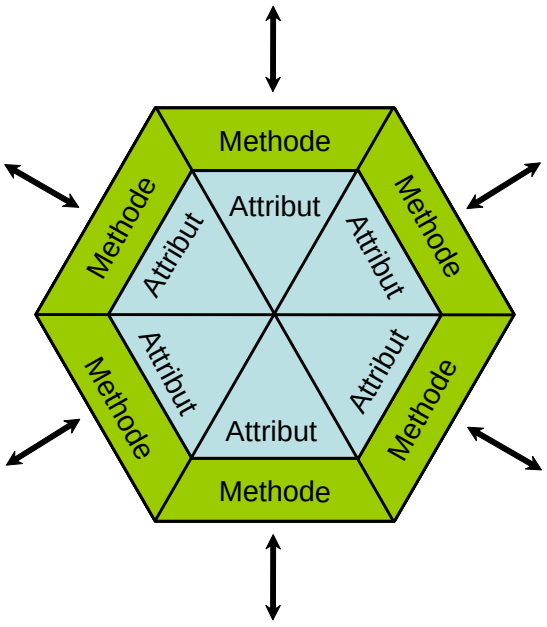
4.1 Überblick, historische Wurzeln, Beispiel

4.1.1 Einige Kernideen und Vorzüge der OOP

In diesem Abschnitt werden einige Kernideen bzw. Vorzüge ausgespart, die mit den bisherigen Beispielen noch nicht plausibel vermittelt werden können (z.B. Polymorphie).

Datenkapselung

In der OOP wird die traditionelle Trennung von Daten und Operationen aufgegeben. Hier besteht ein Programm aus **Klassen**, die durch Eigenschaften (Attribute) *und* zugehörige Methoden definiert sind.
Eine Klasse wird in der Regel ihre Eigenschaften gegenüber anderen Klassen verbergen (**Datenkapselung**, **information hiding**) und so vor ungeschickten Zugriffen schützen. Ihre Dienstleistungen sind dann ausschließlich über Methoden verfügbar, welche ihre *Schnittstelle* zur Außenwelt bilden. Dies kommt in der folgenden, an Goll et al. (2000) angelehnten, Abbildung zum Ausdruck:



Auf diese Weise ergeben sich gravierende Vorteile:

- **Erhöhte Sicherheit durch Schutz der Eigenschaften**
  Direkte Zugriffe auf die 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 als Klassen-Designer 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()` verändern, die aber den Wert null nicht akzeptiert.
  Treten in einem Programm trotz Datenkapselung Fehler wegen pathologischer Variablenausprägungen auf, sind diese relativ leicht zu lokalisieren, weil nur wenige Methoden verantwortlich sein können.
- **Komfort**
  Wer als Programmierer eine Klasse *verwendet*, braucht sich um den inneren Aufbau derselben nicht zu kümmern, so dass neben dem Fehlerrisiko auch der Einarbeitungsaufwand reduziert ist.
- **Flexibilität, Anpassungsfähigkeit**
  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.
- **Produktive Teamarbeit durch abgeschottete Verantwortungsbereiche**
  In großen Projekten können mehrere Programmierer nach der gemeinsamen Entwicklung von Schnittstellen relativ unabhängig an verschiedenen Klassen arbeiten.

Vererbung

Zu den Vorzügen der Klassen als statische Gebilde gesellt sich in der OOP ein höchst kreatives **Vererbungsverfahren**, das beste Voraussetzungen für die Erweiterung und Wiederverwendung von Software schafft: Bei der Definition einer neuen Klasse können alle Eigenschaften (Variablen) und Handlungskompetenzen (Methoden) einer Basisklasse übernommen werden. 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. Eine Klasse kann direkt genutzt werden (durch Erzeugen von Objekten) oder als Basisklasse dienen.
Wird bei einer Klassendefinition keine Basisklasse explizit angegeben, erbt die neue Klasse implizit von der Urahnklasse **Object** an der Wurzel der streng hierarchisch organisierten Klassenbibliothek im Java-API. Weil sich im Handlungsrepertoire der Urahnklasse auch die Methode **getClass()** befindet, kann man Instanzen beliebiger Klassen nach ihrem Datentyp befragen. Im folgenden Programm `BruchRechnung` (vgl. Abschnitt 1.1.2) wird ein `Bruch`-Objekt nach seiner Klassenzugehörigkeit befragt:

Quellcode	Ausgabe
`class BruchRechnung {` `  public static void main(String[] args) {` `  Bruch b1 = new Bruch();` `  System.out.println(b1.getClass().getName());` `  }` `}`	`Bruch`

Als Rückgabewert liefert **getClass()** ein Objekt der Klasse **Class**, welches nach dem Methodenaufruf **getName()** endlich eine Zeichenfolge mit der gewünschten Information liefert.

Realitätsnahe Modellierung

Klassen sind aber nicht nur ideale Bausteine für die rationelle Konstruktion von Softwaresystemen, sondern erlauben auch eine gute **Abbildung des Anwendungsbereichs**. In der zentralen Projektphase der objektorientierten Analyse und Modellierung sprechen Softwareentwickler und Auftraggeber dieselbe Sprache, so dass Missverständnisse und sonstige Kommunikationsprobleme weitgehend vermieden werden.
Neben den Klassen zur Modellierung von Akteuren oder Ereignissen des realen Anwendungsbereichs sind bei einer typischen Java-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) werden zur Strukturierung von Programmen zwei Techniken angeboten, die in überarbeiteter Form auch bei der OOP zum Einsatz kommen:

- **Unterprogrammtechnik**
  Bei diesem Grundprinzip des Softwareentwurfs geht es darum, ein Gesamtproblem in unabhängige Teilprobleme aufzuteilen, die jeweils in einem eigenen *Unterprogramm* gelöst werden. Wird die von einem Unterprogramm erbrachte Leitung wiederholt (an verschiedenen Stellen eines Programms) benötigt, muss jeweils nur ein Aufruf mit dem Namen des Unterprogramms und steuernden Parametern eingefügt werden. Durch diese Strukturierung ergeben sich kompaktere und übersichtlichere Programme, die leichter 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 den *Strukturen* der Programmiersprache C oder den *Records* der Programmiersprache Pascal lassen sich problemadäquate Datentypen konstruieren, die Elemente eines beliebigen, bereits bekannten Typs, enthalten 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 Strukturen bzw. Records der älteren Programmiersprachen werden in der OOP durch *Klassen* ersetzt, wobei diese Datentypen nicht nur durch eine Anzahl von *Eigenschaften* (Instanzvariablen 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.:

- optimierte Modularisierung mit Zugriffsschutz
  Die Daten sind übersichtlich und 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 Verarbeitungssequenz 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 deutlich einfacher zu verwenden als klassische Funktionsbibliotheken.

4.1.3 Auf-`Bruch` zu echter Klasse

In den Beispielprogrammen von Abschnitt 0 wurde mit der Klassendefinition lediglich eine in Java unausweichliche formale Anforderung an ausführbare Programmeinheiten erfüllt. Das in Abschnitt 1.1 vorgestellte Bruchrechnungsprogramm realisiert hingegen wichtige Prinzipien der objektorientierten Programmierung. Es wird nun wieder aufgegriffen und in verschiedenen Varianten bzw. Ausbaustufen als Beispiel verwendet.
Das Programm soll 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, Lernepisode, Testepisode, Fehler, Schüler) zu ergänzen.

Die folgende `Bruch`-Klassendefinition ist im Vergleich zur Variante in Abschnitt 1.1.2 geringfügig verbessert worden:

- 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.
- 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()` kommt die performante Modulo-Variante von Euklids Algorithmus zum Einsatz (vgl. Übungsaufgabe in Abschnitt 3.6.4).

Bei den unveränderten Methoden ist der Quellcode verkürzt 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;
      }
    }

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

    public void frage() {.  .  .}

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

Für die bei diversen Demonstrationen in den folgenden Abschnitten verwendeten „Hauptprogrammsklassen“ 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 Instanzvariablen (Eigenschaften) relativ wenig Mühe, weil wir viele Details schon von den *lokalen* Variablen kennen. Bei den Methoden (Handlungskompetenzen) gibt es mehr Neues zu lernen, so dass wir uns in Abschnitt 4.3 auf elementare Themen beschränken und später noch wichtige Ergänzungen behandeln.

Zunächst zwei Bemerkungen zum Kopf der Klassendefinition:

- Im Beispiel ist die Klasse `Bruch` ist als **public** definiert, damit sie uneingeschränkt von anderen Klassen aus beliebigen Paketen genutzt werden kann. Weil bei der startfähigen `BruchRechnung` eine solche Nutzung nicht in Frage kommt, wird hier auf einen Zugriffsmodifikator 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. `BruchAddition`), schreibt man der besseren Lesbarkeit wegen die Anfangsbuchstaben aller Wörter groß.

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.
- Für den Quellcode der Hauptprogrammsklasse sollte analog eine Datei namens **BruchRechnung.java** verwendet werden.
- Dateien mit Java-Quellcode benötigen auf jeden Fall die Namenserweiterung **.java**.

4.2 Instanzvariablen (Eigenschaften)

Die Instanzvariablen einer Klasse besitzen viele Gemeinsamkeiten mit den *lokalen* Variablen, die wir im Abschnitt über elementare Sprachelemente ausschließlich verwendet haben, doch gibt es auch wichtige Unterschiede, die im Mittelpunkt des aktuellen Abschnitts stehen.
Unsere Demo-Klasse `Bruch` besitzt nach der Erweiterung um ein beschreibendes Etikett folgende Instanzvariablen (Eigenschaften):

- `zaehler` (Datentyp **int**)
- `nenner` (Datentyp **int**)
- `etikett` (Datentyp **String**)

Jedes nach dem `Bruch`-Bauplan geschaffene Objekt enthält einen vollständigen, individuellen Satz dieser Variablen. Es kennt also z.B. seinen eigenen Zähler- und Nennerwert, nicht jedoch die korrespondierenden Eigenschaften anderer Brüche.

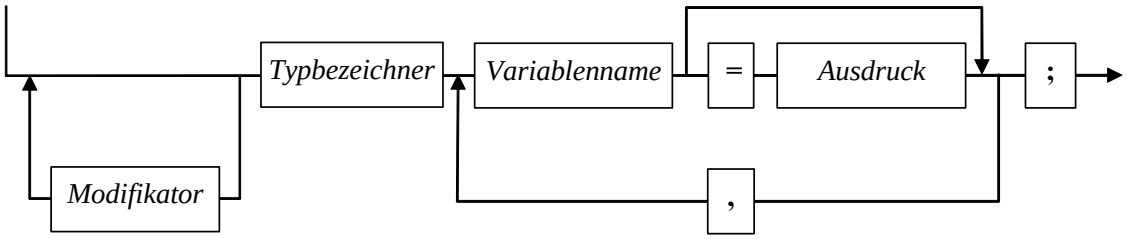
4.2.1 Deklaration mit Wahl der Schutzstufe

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 jeden anderen Ort (außerhalb der Methodendefinitionen) akzeptiert.

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

Deklaration von Instanzvariablen



Im Beispiel wird für alle Instanzvariablen mit dem Modifikator **private** die maximale Stufe der Datenkapselung angeordnet:

```
private int  zaehler;
private int  nenner = 1;
private String etikett = "";
```

Um fremden Klassen trotzdem einen (allerdings kontrollierten!) Zugang zu den `Bruch`-Instanzvariablen zu ermöglichen, wurden etliche Zugriffsmethoden in die Klassendefinition aufgenommen (z.B. `setzeNenner()`).

Gibt man bei der Deklaration einer Instanzvariablen keine Schutzstufe an, haben alle andere 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.
- In den Zugriffsmethoden für `zaehler` und `nenner` kann man neben der Null-Überwachung für den Nenner noch weitere Wertrestriktionen einbauen (z.B. auf positive Zahlen). Es sind sogar situationsabhängige Filtereffekte denkbar.
- 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`-Eigenschaften `zaehler` und `nenner` ist der Spielraum für Schnittstellen-kompatible Typanpassungen aber eher gering.

Trotz der überzeugenden Vorteile soll die Datenkapselung nicht zum Dogma erhoben werden. Sie ist überflüssig, wenn bei einer Eigenschaft Lese- und Schreibzugriff erlaubt sind, 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;
```

In Bezug auf die Namen gibt es keine Unterschiede zwischen den Instanzvariablen und den lokalen Variablen (vgl. Abschnitt 3.2). Insbesondere sollten folgende Konventionen eingehalten werden:

- Variablennamen beginnen mit einem Kleinbuchstaben.
- Besteht ein Name aus mehreren Wörtern (z.B. `isBusy`), schreibt man ab dem zweiten Wort die Anfangsbuchstaben groß.

4.2.2 Gültigkeitsbereich, Existenz und Ablage im Hauptspeicher

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

	lokale Variable	Instanzvariable
Gültigkeitsbereich Sichtbarkeit	Eine lokale Variable ist nur in ihrer eigenen Methode gültig (sichtbar): von der Deklarationsanweisung bis zum Ende des lokalsten Blocks. Nur in diesen Anweisungen kann sie angesprochen werden.	Eine Instanzvariable ist in *allen* Methoden der eigenen Klasse sichtbar, weshalb Mössenböck (2005, S. 80) sie auch als *global* bezeichnet. Je nach Schutzstufe ist sie auch für fremde Klassen sichtbar. Instanzvariablen werden allerdings durch gleichnamige lokale Variablen überlagert, sind also in manchen Methoden eventuell *nicht* sichtbar, gleichwohl weiterhin existent.
Existenz, 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.
Ablage im Speicher	Sie wird auf dem so genannten **Stack** (deutsch: *Stapel*) gespeichert. Innerhalb des für ein Programm verfügbaren Speichers dient dieses Segment zur Verwaltung von Methodenaufrufen.	Die Objekte landen mit ihren Instanzvariablen in einem Bereich des Programmspeichers, der als **Heap** (deutsch: *Haufen*) bezeichnet wird.

4.2.3 Initialisierung

Während bei lokalen Variablen *keine* Initialisierung stattfindet, erhalten die Instanzvariablen eines neuen Objektes automatisch folgende Startwerte:

Datentyp	**Initialisierung**
byte, **short**, **int**, **long**	0
float, **double**	0.0
char	'\u0000'
boolean	**false**
Referenz auf Objekt	**null**

Im Beispiel wird nur die automatische `zaehler`-Initialisierung unverändert übernommen:

- Beim `nenner` eines Bruches wäre die Initialisierung auf 0 bedenklich.
- 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 die **String**-Variable den Wert **null**, zeigt also auf nichts. Weil dies im Beispielprogramm zu Problemen führt, wird ein **String**-Objekt mit einer leeren Zeichenfolge erstellt. Das Erzeugen des **String**-Objektes erfolgt *implizit*, indem der Stringvariablen `etikett` ein Zeichenfolgen-Literal zugewiesen wird.

4.2.4 Zugriff durch eigene und fremde Methoden

In den Methoden einer Klasse können die Instanzvariablen des aktuellen (die Methode ausführenden) Objektes 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, dem Instanzvariablennamen über das Schlüsselwort **this** eine Referenz auf das aktuelle Objekt voranzustellen (siehe Abschnitt 4.6.3), z.B.:

```
System.out.println(" " + luecke  + "   " + this.zaehler + "\n" +
        " " + this.etikett + " -----\n" +
        " " + luecke  + "   " + this.nenner + "\n");
```

Beim Zugriff auf eine Instanzvariablen eines *anderen* Objektes derselben Klasse muss dem Variablennamen eine Referenz auf das Objekt vorangestellt werden, wobei die Bezeichner durch den **Punktoperator** getrennt werden. In folgendem Beispiel überträgt ein `Bruch`-Objekt den eigenen `nenner`-Wert in die korrespondierende Instanzvariable eines anderen `Bruch`-Objektes, das über die Referenzvariable `bc` angesprochen wird:

```
bc.nenner = this.nenner;
```

Direkte Zugriffe auf die Instanzvariablen eines Objektes durch 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**, also ohne Datenkapselung, deklariert, dann könnte z.B. der Nenner eines Bruches in der **main**()-Methode der (fremden!) Klasse `BruchRechnung` 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 vom Compiler verhindert, z.B.:

```
BruchRechnung.java:8: nenner has private access in Bruch
                b1.nenner = 0;
                  ^
1 error
```

4.3 Methoden

In einer Klassendefinition entwerfen wir Objekte mit zahlreichen Handlungskompetenzen (Methoden), die von anderen Programmbestandteilen über festgelegte Kommunikationsregeln 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 dabei andere Programmierer nichts an. Die Instanzvariablen (Eigenschaften) sind bei konsequenter Datenkapselung der Außenwelt nicht bekannt. Jeder Austausch findet über die Methoden statt, welche die *Schnittstelle* des Objektes bilden. Soll eine Eigenschaft trotz Datenkapselung zugänglich sein, sind entsprechende Methoden zum Lesen bzw. Verändern erforderlich. Unsere `Bruch`-Klasse besitzt Lese- und Schreibmethoden für alle Eigenschaften

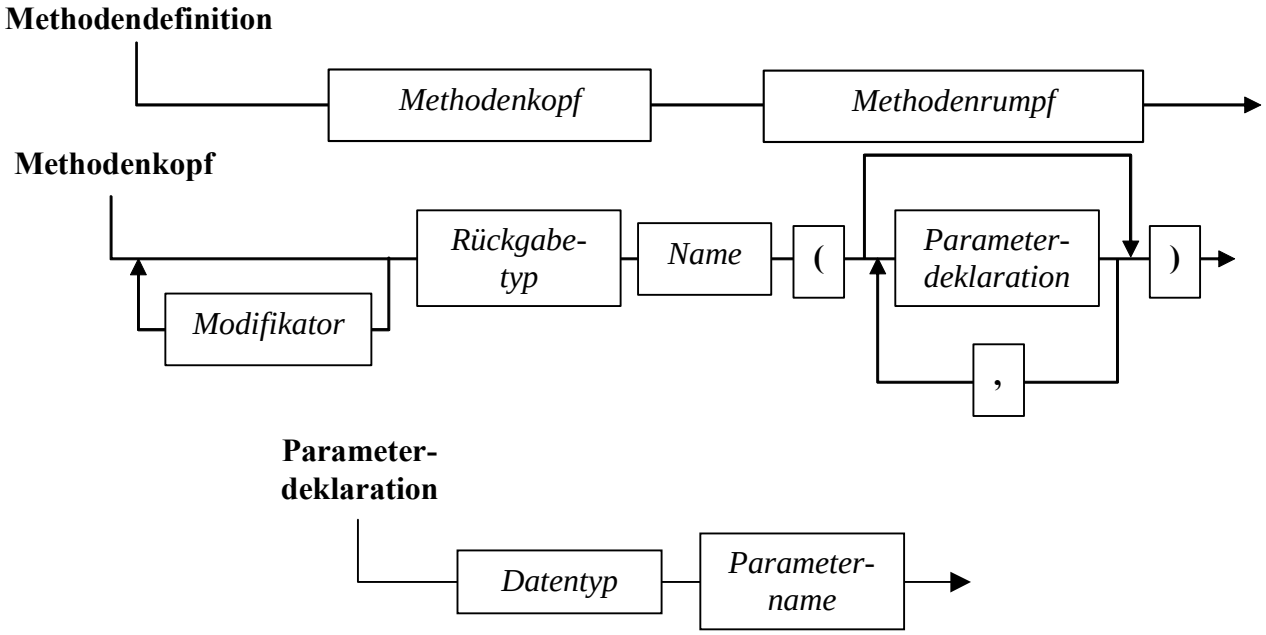
(`setzeZaehler()`, `setzeNenner()`, `setzeEtikett()`, `gibZaehler()`, `gibNenner()`, `gibEtikett()`), dies ist jedoch nicht bei jeder Klasse erforderlich bzw. wünschenswert.

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.

Während jedes Objekt einer Klasse seine eigenen Instanzvariablen auf dem Heap besitzt, sind die Methoden einer Klasse nur einmal vorhanden. Sie befinden sich in einem Bereich des programmeigenen Speichers, der als **Method Area** bezeichnet wird.

4.3.1 Methodendefinition

Das schon in Abschnitt 3.1.2 begonnene Serie von Syntaxdiagrammen zur Methodendefinition soll nun durch eine Explikation des Begriffs *Parameterdeklaration* vervollständigt werden:



In den folgenden Abschnitten werden wichtige Bestandteile dieser Syntaxdiagramme erläutert.

Vorweg soll eine weitgehend akzeptierte Konvention zu den **Namen von Methoden** angesprochen werden:

- Diese beginnen mit einem Kleinbuchstaben.
- Besteht ein Name aus mehreren Wörtern (z.B. `setzeNenner()`), schreibt man ab dem zweiten Wort die Anfangsbuchstaben groß.

4.3.1.1 Parameter

Im Kopf der Methodendefinition wird über so genannte **Formalparameter** festgelegt, welche Argumente eine Methode zur Spezifikation ihrer Arbeitsweise kennt. Beim späteren Aufruf einer Methode ist eine korrespondierende Liste von so genannten **Aktualparametern** anzugeben. In den Anweisungen des Methodenrumpfs sind die Formalparameter wie lokale Variablen zu verwenden, die mit den Werten der Aktualparameter initialisiert wurden.

Für jeden Formalparameter sind folgende Angaben zu machen:

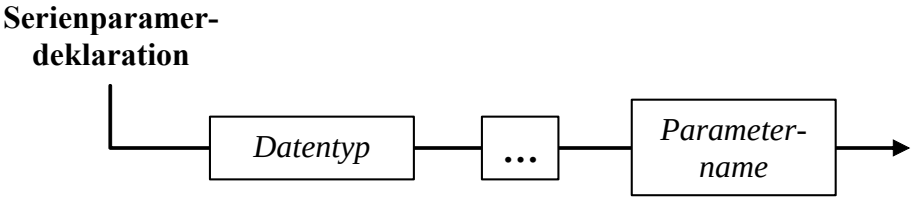
- **Datentyp**
  Es sind beliebige Typen erlaubt (primitive, Klassen).
  Der Datentyp eines Formalparameters muss auch dann explizit angegeben werden, wenn er mit dem Typ des Vorgängers übereinstimmt.
- **Name**
  Damit sich die Parameternamen von Variablennamen unterscheiden, hängen manche Programmierer ein Suffix an, z.B. *par* oder einen Unterstrich. Weil Formalparameter wie lokale Variablen zu behandeln sind, …
  - entstehen Namenskonflikte nur mit anderen lokalen Variablen derselben Methode,
  - werden namensgleiche Instanz- bzw. Klassenvariablen überlagert.
- **Position**
  Die beim späteren Aufruf der Methode übergebenen Aktualparameter werden gemäß ihrer Reihenfolge den Formalparametern zugeordnet.

Als Beispiel betrachten wir die folgende Variante der `Bruch`-Methode `addiere()` Das beauftragte Objekt soll den via Parameterliste (`zpar`, `npar`) übergebenen Bruch zu seinem eigenen Wert addieren und optional (Parameter `autokurz`) das Resultat gleich kürzen:

```
public void addiere(int zpar, int npar, boolean autokurz) {
  zaehler = zaehler*npar + zpar*nenner;
  nenner = nenner*npar;
  if (autokurz)
    kuerze();
}
```

Serienparameter

Seit der Version 5.0 (bzw. 1.5) bietet die J2SE auch Parameterlisten variabler Länge, wobei an Stelle des *letzten* Formalparameters eine *Serie* von Elementen desselben Typs über folgende Syntax deklariert werden kann:



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 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 variablen Parameterlisten in Java 5.0 eingeführt wurde (siehe Abschnitt 3.6.3.1). Im Beispiel wird diese Schleifenkonstruktion benutzt, um jedes Element im Array `bar` mit `Bruch`-Objekten durch Aufruf der ursprünglichen `addiere()`-Methode zum handelnden Bruch zu addieren. Mit den `Bruch`-Objekten `b1` bis `b4` sind z.B. folgende Aufrufe erlaubt:

```
b1.addiere(b2);
b1.addiere(b2, b3);
b1.addiere(b2, b3, b4);
```

Wer sich darüber wundert, dass verschiedene `addiere()`-Varianten in *einer* Klassendefinition existieren dürfen, sei auf Abschnitt 4.3.3 vertröstet.

4.3.1.2 Modifikatoren

Auch 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 Klasse startfähig sein, muss sie über eine Methode namens **main()** mit Schutzstufe **public** verfügen, damit sie vom Java-Interpreter ausgeführt werden kann.

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

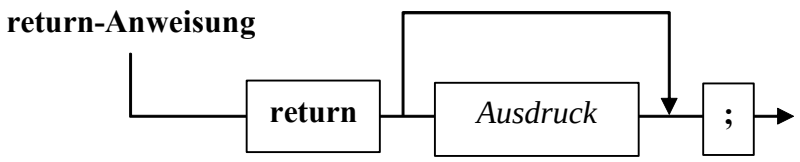
Nicht alle Modifikatoren im Methodenkopf dienen dem Zugriffsschutz: Die **main()**-Methode einer startfähigen Klasse muss nicht nur als **public**, sondern auch als **static** deklariert werden, damit sie als *Klassenmethode* (siehe unten) ausgeführt werden kann, bevor ein Objekt der Klasse existiert.

4.3.1.3 Rückgabewerte und return-Anweisung

Bei der Definition einer Methode muss festgelegt werden, von welchem Datentyp ihre Antwort an die aufrufende Methode ist.
Erfolgt *keine* Rückgabe, ist der Ersatztyp **void** anzugeben. Ein zugehöriger Methodenaufruf stellt dann *keinen* Ausdruck dar, wird aber durch ein nachgestelltes Semikolon zur Anweisung.
Als Beispiel betrachten wir die `Bruch`-Methode `setzeNenner()`, die den einzigen Parameterwert (Datentyp **int**) als neuen Nenner einträgt, falls dieser von null verschieden ist. Der Aufrufer erfährt durch einen Rückgabewert vom Datentyp **boolean**, ob sein Auftrag ausgeführt werden konnte (**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 mit einer **return**-Anweisung endet, die einen Rückgabewert passenden Typs liefert:



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

Soll eine Methode mehr als nur einen Wert von primitivem Datentyp zurückliefern, dann muss eine Klasse als Rückgabetyp benutzt werden (siehe Abschnitt 4.6).

4.3.1.4 Methodenrumpf

Über die Verbundanweisung, die den Rumpf einer Methode bildet, haben Sie bereits gelernt:

- 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 von Werten an den Aufrufer und/oder zum Beenden einer Methodenausführung.

Ansonsten können beliebige Anweisungen unter Verwendung von elementaren und objektorientierten Sprachelementen eingesetzt werden, um den Zweck einer Methode zu implementieren.
Die eben als Beispiel betrachtete `Bruch`-Methode `setzeNenner()` beschränkt sich auf die (bedingungsabhängige) Änderung einer Instanzvariablen. Im Allgemeinen wird eine Methode etwas mehr Aktivität entfalten und sich dabei auch nicht unbedingt auf die Veränderung von lokalen Variablen und objekteigenen Instanzvariablen beschränken. Über eine als Parameter übergebene Referenzvariable (siehe Abschnitt 4.6.1) sind z.B. auch Einwirkungen auf andere Objekte möglich. Dabei kann es sich um andere Objekte der eigenen Klasse oder um Objekte beliebiger anderer Klassen handeln.

4.3.2 Methodenaufruf

Die *Verwendung* eines Objektes sollte nach der reinen Lehre ausschließlich im Aufrufen seiner Methoden bestehen, z.B.:

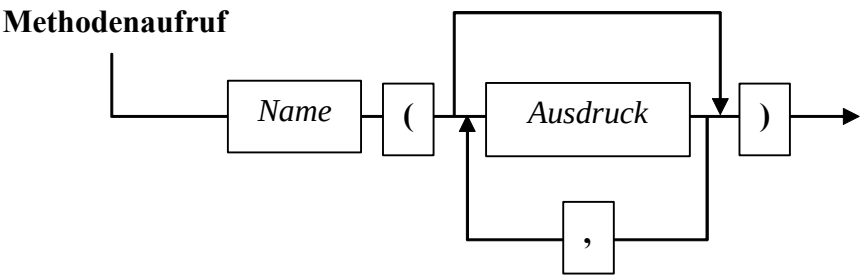
```
b1.zeige();
```

Nach objektorientierter Denkweise wird hier eine *Botschaft* an ein Objekt geschickt:

„b1, zeige dich!“.

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

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



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

Als Beispiel betrachten wir einen Aufruf der in Abschnitt 4.3.1.1 vorgestellten Variante der `Bruch`-Methode `addiere()`:

```
b1.addiere(1, 2, true);
```

Als Aktualparameter sind also keinesfalls *Variablen* erforderlich, sondern *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 auch als Argument in komplexeren Ausdrücken auftreten, z.B.:

```
if (!b1.setzeNenner(pn))
  System.out.println("Mit dem Nenner gab es ein Problem.");
```

Durch ein angehängtes Semikolon wird jeder Methodenaufruf zur vollständigen Anweisung (siehe erstes Beispiel). Auch eine Methode *mit* Rückgabewert kann als selbständige Anweisung eingesetzt werden, wobei der Rückgabewert ignoriert wird, z.B.:

```
b1.setzeNenner(pn);
```

Es bedarf eigentlich keiner Erwähnung, dass sich jede Anweisung mit Methodenaufruf in einer ausführbaren Programmeinheit, also in einer *Methode* befindet. Folglich kommt es bei der Programmausführung sehr häufig zu geschachtelten Methodenaufrufen, wobei die maximal erlaubte Schachtelungstiefe von der Kapazität des Stack-Speichers abhängt. Ein Überlauf dieses Speichers (*Stack Overflow*) ist am ehesten bei den so genannten *rekursiven* Methoden zu befürchten, die sich selbst aufrufen (siehe Abschnitt 4.8).

Soll in einer Methodenimplementierung vom aktuell handelnden Objekt eine andere Methode derselben Klasse ausgeführt werden, dann muss beim Aufruf *keine* Objektbezeichnung angegeben werden. In den verschiedenen Varianten der `Bruch`-Methode `addiere()` soll das beauftragte Objekt den via Parameterliste übergebenen Bruch zu seinem eigenen Wert addieren und das Resultat eventuell gleich kürzen. Zum Kürzen kommt natürlich die bereits vorhandene `Bruch`-Methode zum Einsatz. Weil sie vom gerade agierenden Objekt auszuführen ist, wird keine Objektbezeichnung benötigt:

```
public void addiere(int zpar, int npar, boolean autokurz) {
  zaehler = zaehler*npar + zpar*nenner;
  nenner = nenner*npar;
  if (autokurz)
    kuerze();
}
```

Wer gerne auch solche Methodenaufrufe mit der Syntax

Empfänger.Botschaft

realisieren möchte, kann mit dem Schlüsselwort **this** das aktuelle Objekt ansprechen, z.B.:

```
if (autokurz)
  this.kuerze();
```

4.3.3 Methoden überladen

Die in Abschnitt 4.3.1.1 vorgestellten `addiere()`-Varianten können problemlos in der `Bruch`-Klassendefinition mit der dort bereits vorhandenen `addiere()`-Variante koexistieren, weil die drei Methoden unterschiedliche Parametertypen besitzen. Generell ist eine solche *Überladung* erlaubt, wenn sich die *Signaturen* der beteiligten Methoden unterscheiden.

Zwei Methoden besitzen genau dann dieselbe Signatur, wenn

- ihre Namen identisch sind,
- ihre Parameterlisten gleich lang und die Typen korrespondierender Parameter identisch sind.

Der Compiler versucht, für alle Methodenaufrufe die angeforderte Überladung eindeutig zu bestimmen, und meldet ggf. einen Fehler.

Von einer Methode mehrere Überladungen in eine Klassendefinition aufzunehmen, lohnt sich z.B. dann, wenn unterschiedlich umfangreiche Parameterlisten angeboten werden sollen, sodass zwischen einer bequem aufrufbaren Standardausführung (z.B. mit leerer Parameterliste) und einer individuell gestalteten Ausführungsvariante gewählt werden kann.

Bevor mit Java 5.0 *generische* Methoden und Typen (siehe unten) eingeführt wurden, erhielten viele Klassen Familien analog arbeitender Methoden für Parameter unterschiedlichen Typs. So gibt es z.B. in der Klasse **Math** im API-Paket **java.lang** folgende Methoden, die für Argumente unterschiedlichen Typs den Betrag liefern:

public static double abs(double *value*)
public static float abs(float *value*)
public static int abs(int *value*)
public static long abs(long *value*)

4.4 Objekte

4.4.1 (Wieder)verwendung von Klassen bzw. Objekten

Ein zentrales Ziel der OOP ist die Produktion von Software mit hohem Recycling-Potential. Daher wurde im Bruchrechnungs-Demoprojekt 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 zum Leben erwecken und nutzen kann.

Jede Java – Anwendung benötigt bekanntlich eine Startklasse mit statischer **main()**-Methode, die vom Interpreter beim Programmstart automatisch aufgerufen wird und das (möglichst) objektorientierte Leben in Gang setzt. Statt die `Bruch`-Definition mit einer **main()**-Methode auszustatten, bevorzugen wir im Bruchrechnungsprojekt eine separate „Hauptprogrammsklasse“, z.B. `BruchRechnung` genannt. Deren **main()**-Methode erzeugt `Bruch`-Objekte und versorgt sie mit Aufträgen, z.B.:

Quellcode	Ausgabe
`class BruchRechnung {` `  public static void main(String[] args) {` `  Bruch b = new Bruch();` `  b.setzeZaehler(4);` `  b.setzeNenner(16);` `  b.kuerze();` `  b.setzeEtikett("Der gekuerzte Bruch:");` `  b.zeige();` `  }` `}`	`  1` `Der gekuerzte Bruch: -----` `  4`

Hinsichtlich der Dateiverwaltung wird vorgeschlagen:

- Auch die `BruchRechnung`-Klassendefinition sollte in einer eigenen Datei gespeichert werden.
- Den Namensstamm dieser Datei sollte der Klassenname bilden.
- Als Namenserweiterung ist in jedem Fall **.java** zu verwenden.

Beim Übersetzen der Quellcode-Dateien des Projektes entstehen die Bytecode-Dateien **Bruch.class** und **BruchRechnung.class**. Der Compiler erzeugt übrigens grundsätzlich für jede Klasse eine eigene Bytecode-Datei, so dass aus einer Quellcode-Datei eventuell mehrere Bytecode-Dateien entstehen.

4.4.2 Referenzvariablen definieren, Klassen als Datentypen

Um ein 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`-Objektes
  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 Referenz-Literal 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. Wir haben also folgende Datentypen zur Verfügung:

- **Primitive Typen** (**boolean**, **char**, **byte**, ..., **double**)
- **Klassentypen**
  Es kommen Klassen aus dem Java-API und selbst definierte Klassen in Frage.

4.4.3 Objekte erzeugen

Steht die Deklarationsanweisung

```
Bruch b;
```

in einem Methodenrumpf, handelt es sich bei `b` um eine *lokale* Variable ohne Initialisierungswert. Damit der Variablen `b` ein Verweis auf ein `Bruch`-Objekt als Wert zugewiesen werden kann, muss ein solches Objekt erst erzeugt werden, was der **new**-Operator übernimmt, z.B. in folgendem Ausdruck:

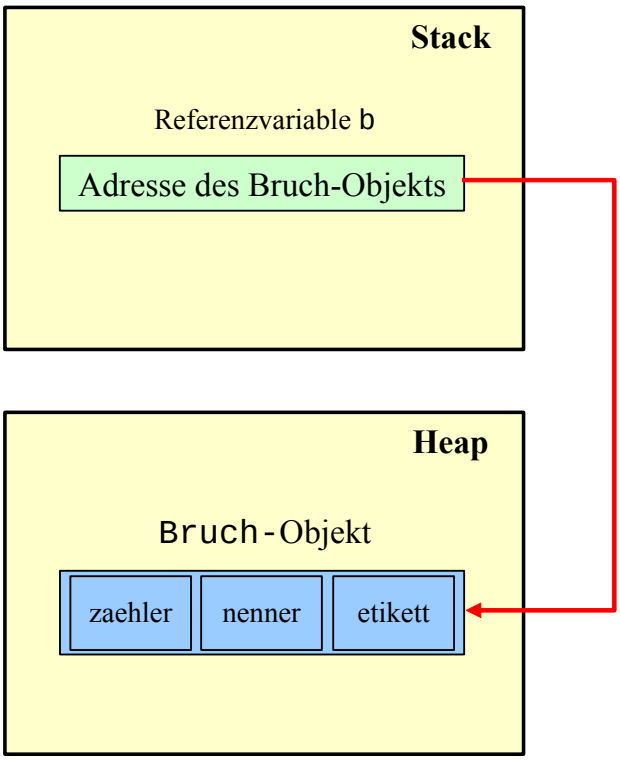
```
new Bruch()
```

Als Operanden erwartet **new** einen Klassennamen, dem eine Parameterliste zu folgen hat, weil er hier als *Konstruktor* (s. u.) aufzufassen ist. Als Wert des Ausdrucks resultiert eine Referenz, die einen Zugriff auf das neue Objekt (seine Instanzvariablen und Methoden) erlaubt.

In der **main()**-Methode des Bruchrechnungsprogramms schreiben wir die vom **new**-Operator gelieferte Referenz mit dem Zuweisungsoperator in die lokale Referenzvariable `b`:

```
Bruch b = new Bruch();
```

Es resultiert folgende Situation:



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.

4.4.4 Überflüssige Objekte entfernen, 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. Im Unterschied zu vielen anderen Programmiersprachen (z.B. C++) sind in Java zwei wichtige Fehlerquellen eliminiert:

- 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). Diese kommen z.B. in C++ dann zu Stande, wenn dynamisch angeforderter Speicher später nicht explizit frei gegeben wird.

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 Objektes kaum vorhersehbar ist.

Mehr müssen die meisten Programmierer 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()** definiert werden, die 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 diese in einer abgeleiteten Klasse, so 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 entweder abgefangen oder weitergereicht werden, was durch den Zusatz **throws Throwable** im Methodenkopf anzumelden ist.
- `this`
  Mit dem Schlüsselwort **this** (vgl. Abschnitt 4.6.3) wird 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 ressourcen-intensiven Aktionen eines Programms. Allerdings ist nicht sicher, ob der Garbage Collector tatsächlich tätig wird. Erst recht 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.

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 4.6.2).

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.:

Quellcode	Ausgabe
`class BruchRechnung {` `  public static void main(String[] args) {` `  Bruch b1 = new Bruch(), b2 = new Bruch();` `  b1.setzeZaehler(1); b1.setzeNenner(2);` `  b2.setzeZaehler(3); b2.setzeNenner(4);` `  b1.zeige(); b2.zeige();` `  b1 = null;` `  b2 = new Bruch();` `  b2.setzeZaehler(5); b2.setzeNenner(6);` `  System.gc();` `  b2.zeige();` `  }` `}`	`  1` `-----` `  2` `  3` `-----` `  4` `Bruch@1a16869 finalisiert` `Bruch@1cde100 finalisiert` `  5` `-----` `  6`

Die Ausgaben

```
Bruch@1a16869 finalisiert
Bruch@1cde100 finalisiert
```

stammen von obiger `finalize()`-Methode. Versuche, den abgebildeten Programmablauf zu reproduzieren, können recht frustrierend verlaufen, weil ein **System.gc()**-Aufruf nur einen Vorschlag darstellt, auf den das Laufzeitsystem eventuell nicht reagiert.

4.5 Konstruktoren

In diesem Abschnitt werden spezielle Methoden behandelt, die beim Erzeugen von neuen Objekten automatisch aufgerufen werden, um diese zu initialisieren.
Wie Sie bereits wissen, wird zum Erzeugen von Objekten der **new**-Operator verwendet. Als Operand ist der vom Laufzeitsystem auszuführende 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 Standard-Konstruktor ist übrigens *nicht* für die oben erwähnten automatischen Initialisierungen von Instanzvariablen verantwortlich und entfaltet auch sonst keine große Aktivität.
In aller Regel ist es sinnvoll, *explizit* einen Konstruktor zu definieren, der das individuelle Initialisieren der Instanzvariablen von neuen Objekten erlaubt. 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.
- Es sind generell beliebig viele Konstruktoren möglich, die alle denselben Namen und jeweils eine individuelle Parameterliste haben müssen. Das Überladen von Methoden (vgl. Abschnitt 4.3.3) 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:

```
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;}

  . . .
  . . .

}
```

In diesem Testprogramm werden beide Konstruktoren eingesetzt:

Quellcode	Ausgabe
`class BruchRechnung {` `  public static void main(String[] args) {` `  Bruch b1 = new Bruch(1, 2, "b1 = ");` `  Bruch b2 = new Bruch();` `  b1.zeige();` `  b2.zeige();` `  }` `}`	`  1` `b1 =  -----` `  2` `  0` `-----` `  1`

Von der Regel, dass Konstruktor-Aufrufe nur über den **new**-Operator möglich sind, gibt es eine Ausnahme: Im Anweisungsblock eines Konstruktors darf ein anderer Konstruktor derselben Klasse über das Schlüsselwort **this** aufgerufen werden, z.B.:

```
public Bruch() {
  this(0, 1, "unbenannt");
}
```

4.6 Referenzen

In diesem Abschnitt wird sich erneut zeigen, dass gute Referenzen nützlich sind.

4.6.1 Referenzparameter

Wir haben schon festgehalten, dass die Formalparameter einer Methode wie *lokale Variablen* funktionieren, die mit den Werten der Aktualparameter initialisiert werden. Bei Parametern von *primitiven Datentyp* wirken sich Änderungen innerhalb einer Methode daher *nicht* auf die rufende Programmeinheit aus.
Bei einem Parameter vom *Referenztyp* wird ebenfalls der Wert des Aktualparameters (eine Objektadresse) beim Methodenaufruf in den Formalparameter kopiert. Es wird jedoch keinesfalls eine Kopie des referenzierten Objekts (auf dem Heap) erstellt, so dass Formal- und Aktualparameter auf dasselbe Objekt zeigen.
Speziell bei speicherintensiven Objekten hat die in Java realisierte Referenzparametertechnik den Vorteil, dass beim Methodenaufruf Zeit und Speicherplatz gespart werden.

Die folgende `addiere()`-Variante aus dem `Bruch`-Projekt verfügt über einen Referenzparameter:

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

Mit dieser Botschaft wird ein 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`-Objektes 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.

Bei einem `addiere()`–Aufruf 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 könnten z.B. die `Bruch`-Klasse um eine Methode `duplW()` erweitern, die ein Objekt beauftragt, seinen Zähler und Nenner auf ein anderes `Bruch`-Objekt zu übertragen, das per Referenzparameter bestimmt wird:

```
  public void duplW(Bruch bc) {
    bc.zaehler = zaehler;
    bc.nenner = nenner;
  }
```

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

Quellcode	Ausgabe
`class BruchRechnung {` `  public static void main(String[] args) {` `  Bruch b1 = new Bruch(1, 2, "b1 = ");` `  Bruch b2 = new Bruch(5, 6, "b2 = ");` `  b1.zeige();` `  b2.zeige();` `  b1.duplW(b2);` `  System.out.println("Nach duplW():\n");` `  b2.zeige();` `  }` `}`	`  1` `b1 =  -----` `  2` `  5` `b2 =  -----` `  6` `Nach duplW():` `  1` `b2 =  -----` `  2`

Hier liegt *kein* Verstoß gegen das Prinzip der Datenkapselung vor, weil der Zugriff durch eine klasseneigene Methode erfolgt, die vom Klassendesigner gut konzipiert sein sollte.[15]

4.6.2 Rückgabewerte vom Referenztyp

Bisher haben die innerhalb von Methoden erzeugten Objekte das Ende der Methode nicht wesentlich überlebt. Weil keine Referenz außerhalb der Methode existierte, wurden die Objekte dem Garbage Collector überlassen. Soll ein methoden-intern kreiertes Objekt weiter existieren, muss eine

[15] Man könnte durchaus darüber diskutieren, ob die Instanzvariablen einer Klasse auch in den eigenen Methoden vorsichtshalber nur über abgesicherte **set**-Methoden geändert werden sollten. Solche Vorsicht bzw. Selbstzweifel eines Klassendesigners scheinen mir allerdings übertrieben.

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 eigenen Instanzvariablen zu initialisieren und die Referenz an den Aufrufer abzuliefern:[16]

```
public Bruch klone() {
  return new Bruch(zaehler, nenner, etikett);
}
```

In der **main()**-Methode des folgenden Programms wird nur *ein* Objekt erzeugt. Das von `b2` referenzierte Objekt entsteht in der von `b1` ausgeführten Methode `klone()`:

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`

4.6.3 this als Referenz auf das aktuelle Objekt

Gelegentlich ist es erforderlich, dass ein Objekt sich selbst ansprechen kann. Dies ist mit dem Schlüsselwort **this** möglich, das innerhalb einer Methode wie eine Referenzvariable benutzt werden kann. In folgendem Beispiel ermöglicht die **this**-Referenz die Verwendung von Formalparameternamen, die mit den Namen von Instanzvariablen übereinstimmen:

```
public Bruch(int zaehler, int nenner, String etikett) {
  this.zaehler = zaehler;
  this.nenner  = nenner;
  this.etikett = etikett;
}
```

Später werden Sie noch weit relevantere **this**-Verwendungsmöglichkeiten kennen lernen.

4.7 Klassenbezogene Eigenschaften und Methoden

4.7.1 Klassenvariablen

Neben den Instanzvariablen und -methoden unterstützt Java auch *klassenbezogene* Variablen und Methoden. In unserem Beispiel kann eine klassenbezogene Variable dazu dienen, die Anzahl der bisher erzeugten `Bruch`-Objekte aufzunehmen:

[16] Im Zusammenhang mit den Schnittstellen (Interfaces) werden Sie später noch eine etwas bequemere Möglichkeit zum Klonen kennen lernen.

```
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++;}

  . . .
  . . .

}
```

Syntaktisch wird eine Klassenvariable durch den Modifikator **static** gekennzeichnet. Sie kann in allen Methoden der eigenen Klasse angesprochen werden. In unserem Beispiel wird die (automatisch auf 0 initialisierte) Klassenvariable `anzahl` in den beiden Konstruktoren inkrementiert.

Im 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");
```

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 3.2.6), wie die **out**-Deklaration aus der **System**-Klassendefinition zeigt:

```
public final static PrintStream out = nullPrintStream();
```

Mit dieser Technik lassen sich offenbar ReadOnly-Variablen realisieren, wobei allerdings auch klasseneigene Methoden nach der Initialisierung keine Veränderung mehr vornehmen dürfen.

Soll der direkte Zugriff auf eine statische Variable den klasseneigenen Methoden vorbehalten bleiben, ist bei der Deklaration der Modifikator **private** anzugeben (siehe obige Quelle). In diesem Zusammenhang begegnet uns übrigens erstmals eine Variablendeklaration mit *zwei* Modifikatoren, wobei die Reihenfolge beliebig ist.

Während jedes Objekt einer Klasse über einen eigenen Satz mit allen Instanzvariablen verfügt, die beim Erzeugen des Objektes 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).

In der folgenden Tabelle werden wichtige Unterschiede zwischen Klassen- und Instanzvariablen erläutert:

	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 Objektes angelegt und initialisiert. Sie werden ungültig, wenn das Objekt nicht mehr referenziert ist.	Klassenvariablen werden beim Laden der Klasse angelegt und initialisiert. Sie werden beim Entladen der Klasse ungültig.

4.7.2 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. Das Erzeugen von Objekten gehört gerade zu den typischen Aufgaben der Methode **main()**, wobei es sich nicht unbedingt um Objekte der eigenen Klasse handeln muss.
Sofern Klassenmethoden vorhanden sind, kann man übrigens auch eine Klasse als *Akteur* auf der objektorientierten Bühne betrachten.
Sind *ausschließlich* Klassenmethoden vorhanden, ist das Erzeugen von Objekten kaum sinnvoll. Es kann durch den Zugriffsmodifikator **private** für die Konstruktoren verhindert werden.
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 in Abschnitt 3.4.1 kennen gelernt. Dort wurde auch demonstriert, wie die **Math**-Klassenmethode **pow()** von einer fremden Klasse benutzt werden kann:

```
System.out.println(4 * Math.pow(2, 3));
```

Dem Methodennamen wird ein Präfix aus Klassennamen und Punktoperator vorangestellt.

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.:

```
static public int hanz() {return anzahl;}
```

Mit Hilfe der `Bruch`-Klassenmethode `hanz()` kann die fremde Klasse `BruchRechnung` nun feststellen, wie viele Brüche bereits erzeugt wurden:

Quellcode	Ausgabe
`class BruchRechnung {` `  public static void main(String[] args) {` `  System.out.println(Bruch.hanz() + " Brueche erzeugt");` `  Bruch b1 = new Bruch(1, 2, "Bruch 1");` `  Bruch b2 = new Bruch(5, 6, "Bruch 2");` `  b1.zeige();` `  b2.zeige();` `  System.out.println(Bruch.hanz() + " Brueche erzeugt");` `  }` `}`	`0 Brueche erzeugt` `  1` `Bruch 1 -----` `  2` `  5` `Bruch 2 -----` `  6` `2 Brueche erzeugt`

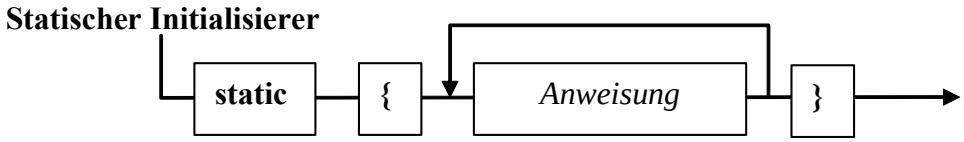
Eine Klassenmethode kann natürlich auch von anderen Methoden der eigenen Klasse (objekt- oder klassenbezogen) aufgerufen werden, wobei der Klassenname nicht angegeben werden muss.

Oft wird fälschlich behauptet, in einer statischen Methode könnten keine Instanzmethoden aufgerufen werden (z.B. Deitel & Deitel 2005, S. 384; Mössenböck 2005, S. 153). Sofern eine statische Methode eine Referenz zu einem Objekt (der eigenen Klasse) besitzt, das sie eventuell selbst erzeugt hat, kann sie im Rahmen der eingeräumten Zugriffsrechte (bei Objekten der eigenen Klasse also uneingeschränkt) auf Instanzvariablen zugreifen und Instanzmethoden aufrufen, wobei selbstverständlich ein konkretes Objekt anzusprechen ist.

In früheren Abschnitten waren mit *Methoden* stets *objektbezogene* Methoden (*Instanzmethoden*) gemeint. Dies soll auch weiterhin so gelten.

4.7.3 Statische Initialisierer

Analog zur Initialisierung von Instanzvariablen durch Konstruktoren, die beim Kreieren eines Objekts ausgeführt werden, 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 der parameterfreie Instanzkonstruktor zufallsabhängige, aber pro Programmlauf identische Werte zur Initialisierung der Felder `zaehler` und `nenner` verwenden. Dazu werden statische Felder ergänzt, die von einem statischen Initialisierer mit Zufallswerten initialisiert werden sollen:

```
private static int zaehlerVoreinst;
private static int nennerVoreinst;
```

Im statischen Konstruktor wird ein Objekt der Klasse **Random** aus dem Paket **java.util** erzeugt und dann per **nextInt()**-Methodenaufruf mit der Produktion von **int**-Zufallswerten beauftragt:

```
  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 {` `  public static void main(String[] args) {` `  Bruch b1 = new Bruch(), b2 = new Bruch();` `  b1.zeige();` `  b2.zeige();` `  }` `}`	`Klasse Bruch geladen` `  5` `  -----` `  9` `  5` `  -----` `  9`

4.7.4 Wiederholung zur Kategorisierung von Variablen

Mittlerweile haben wir verschiedene Variablentypen 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,
  existieren, bis der innerste Block endet.
- **Instanzvariablen ...**
  werden außerhalb jeder Methode deklariert,
  landen (als Bestandteile von Objekten) auf dem Heap,
  werden automatisch initialisiert,
  existieren, solange das Objekt referenziert ist.
- **Klassenvariablen ...**
  werden außerhalb jeder Methode mit dem Modifikator **static** deklariert,
  landen (als Bestandteile von Klassen) in der Method Area,
  werden automatisch initialisiert,
  existieren, solange die Klasse geladen ist.
- **Referenzvariablen ...**
  sind durch ihren speziellen *Inhalt* (Referenz auf ein Objekt) gekennzeichnet. Es kann sich sowohl um lokale Variablen (z.B. `b1` in der **main()**-Methode von `BruchRechnung`) als auch um Instanzvariablen (z.B. `etikett` in der `Bruch`-Definition) handeln oder um Klassenvariablen handeln.

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 und Initialisierung festgelegt (siehe oben).

Aus den Dimensionen Datentyp und Zuordnung ergibt sich eine (2 × 3)-Matrix zur Einteilung der Java-Variablen, die schon im Abschnitt über elementare Sprachelemente zu sehen war:

		Einteilung nach **Zuordnung**		
		Lokale Variable	Instanzvariable	Klassenvariable
Einteilung nach **Datentyp** (Inhalt)	Primitiver Datentyp			
	Referenzvariable			

4.8 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 reduzierten 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 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 3.6.4) durch einen effizienteren Algorithmus ersetzt haben. In diesem Abschnitt betrachten wir die effizientere Variante, wobei zur Vereinfachung der Darstellung der GGT-Algorithmus vom restlichen Kürzungsverfahren getrennt 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;
  }
}
```

Die iterative GGT-Methode `ggTi()` kann durch 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`.
- Anderenfalls wird das Problem, den GGT von *a* und *b* zu finden, auf das einfachere Problem zurückgeführt, den GGT von `b` und `rest` zu bestimmen, und die Methode `ggTr()` ruft sich selbst mit neuen Aktualparametern auf.

Bei der Reduktion des Problems, die übrigens auch im iterativen Algorithmus zum Einsatz kommt, wird ein einfacher Satz der mathematischen Zahlentheorie ausgenutzt:

Für natürliche Zahlen *a*, *b* und *x* mit $a > b$ gilt:

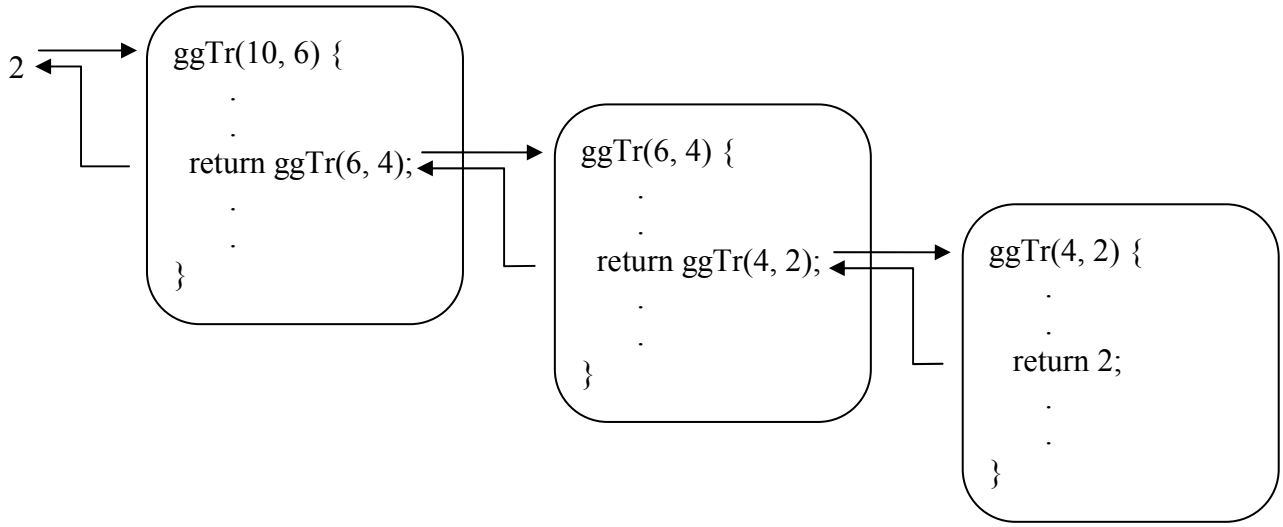
x ist gemeinsamer Teiler von *a* und *b*

$\Leftrightarrow$

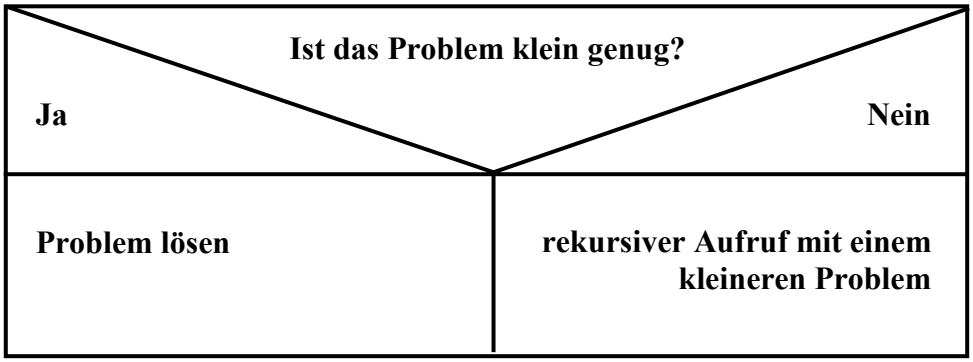
x ist gemeinsamer Teiler von *b* und $a \% b$

Damit ist der GGT von *a* und *b* identisch mit dem GGT von *b* und $a \% b$.

Wird die Methode `ggTr()` z.B. mit den Argumenten 10 und 6 aufgerufen, kommt es zu folgender Aufrufverschachtelung:



Generell läuft ein rekursiver Algorithmus nach folgender Logik ab:



Wird bei einem fehlerhaften Algorithmus der linke Zweig in diesem *Struktogramm* nie erreicht, erschöpfen die geschachtelten Methodenaufrufe bald 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 höhere Zahl von Methodenaufrufen in der Regel mehr Rechenzeit.

4.9 Aggregation

Bei Instanz- und Klassenvariablen sind beliebige Datentypen zugelassen, auch Referenztypen. In der aktuellen `Bruch`-Definition ist z.B. eine Instanzvariable vom Referenztyp **String** vorhanden. Es ist also möglich, vorhandene 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[17] (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. Wir erweitern das Bruchrechnungsprogramm um eine Klasse namens `Aufgabe`, die Trainingssitzungen unterstützen soll. In der `Aufgabe`-Klassendefinition tauchen mehrere Instanzvariablen vom Typ `Bruch` auf:

```
public class Aufgabe {
  private Bruch b1, b2, lsg, antwort;
  private char op = '+';
  private double dlsg;

  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;
    }
    dlsg = (double) lsg.gibZaehler() / lsg.gibNenner();
  }

  public boolean korrekt() {
    double dwert = (double) antwort.gibZaehler()/antwort.gibNenner();
    return Math.abs(dwert-dlsg) < 1.0e-5 ? true : false;
  }

  public void zeige(int was) {
    switch (was) {
      case 1:System.out.println("   " + b1.gibZaehler() +
           "                " + b2.gibZaehler());
           System.out.println(" ------     " + op + "     -----");
           System.out.println("   " + b1.gibNenner() +
           "                " + b2.gibNenner());
           break;
      case 2: lsg.zeige(); break;
      case 3: antwort.zeige(); break;
    }
  }
```

[17] Die betroffenen Personen mögen den Fachterminus *Objekt* nicht persönlich nehmen.

```
    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 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();
    }
}
```

In einer `Aufgabe` dienen die `Bruch`-Objekte folgenden Zwecken:

- `b1` und `b2` werden dem Anwender (in der Methode `frage()`) im Rahmen eines Arbeitsauftrags vorgelegt, z.B. zum Addieren.
- In `antwort` landet der Lösungsversuch des Anwenders.
- In `lsg` steht das korrekte Ergebnis

In folgendem Programm wird die neue Klasse für ein Bruchrechnungstraining eingesetzt:

```
class BruchRechnung {
  public static void main(String[] args) {
    Aufgabe auf = new Aufgabe('+', 1, 2, 2, 5);
    fragen(auf);
    auf.neueWerte('*', 3, 4, 2, 3);
    fragen(auf);
  }
  static void fragen(Aufgabe auf) {
    auf.frage();
    if (auf.korrekt())
      System.out.println("\n Gut!");
    else {
      System.out.println();
      auf.zeige(2);
    }
  }
}
```

Zur Strukturierung und Vereinfachung des Quellcodes wird hier die statische Methode `fragen()` definiert und in **main()** aufgerufen.
Man kann immerhin schon ahnen, wie die praxistaugliche Endversion 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!
```

4.10 Übungsaufgaben zu Abschnitt 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** für Instanzvariablen) werden die Objekte einer Klasse darin gehindert, Attribute anderer Objekte derselben Klasse zu verändern.
3. Auf die statischen Variablen einer Klasse haben auch die Instanzmethoden derselben Klasse vollen Zugriff.
4. Referenzvariablen werden automatisch mit den Wert **null** initialisiert.

2) 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 zur formatierten Ausgabe, , z.B. mit dem Namen `toString()`
  Tipp: Seit Java 5.0 steht in der Klasse **String** die statische Methode **format()** zur Verfügung. Ihr erster Parameter ist vom Typ **String** und kann neben normalen Ausgabezeichen auch Platzhalter mit Formatangaben enthalten. Korrespondierend zu den Platzhaltern folgen als weitere Parameter Ausdrücke von passendem Typ, deren Werte ausgegeben werden sollen. Im folgenden Beispiel enthält der Formatierungsparameter 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:

- Sinnvolle Reaktionen der beiden Klassen auf Versuche zur Vereinbarung irregulärer Werte (z.B. Uhrzeit mit einer Stundenangabe größer als 23) lernen Sie erst im Abschnitt über die Ausnahmebehandlung kennen.
- 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.

3) 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?

4) Diese Übungsaufgabe eignet sich nur für Leute mit Grundkenntnissen in linearer Algebra: Erstellen Sie eine Klasse für zweidimensionale Vektoren, die mindestens über Instanzmethoden 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 1 **normieren**
  Dazu dividiert man beide Komponenten durch die Länge des Vektors, denn mit

  $\tilde{x} := (\tilde{x}_1, \tilde{x}_2)$ sowie $\tilde{x}_1 := \dfrac{x_1}{\sqrt{x_1^2 + x_2^2}}$ und $\tilde{x}_2 := \dfrac{x_2}{\sqrt{x_1^2 + x_2^2}}$ gilt:

  $$|\tilde{x}| := \sqrt{\tilde{x}_1^2 + \tilde{x}_2^2} = \sqrt{\left(\frac{x_1}{\sqrt{x_1^2 + x_2^2}}\right)^2 + \left(\frac{x_2}{\sqrt{x_1^2 + x_2^2}}\right)^2} = \sqrt{\frac{x_1^2}{x_1^2 + x_2^2} + \frac{x_2^2}{x_1^2 + x_2^2}} = 1$$

- Vektoren (komponentenweise) **addieren**

  Die Summe der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

  $$\begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

- **Skalarprodukt** zweier Vektoren ermitteln

  Das Skalaprodukt 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:

  $$\cos(x, y) = \frac{x \cdot y}{|x||y|}$$

Um daraus den Winkel in Grad zu ermitteln, können Sie die Klassenmethoden **Math.acos()** und **Math.toDegrees()** verwenden.

- **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:

$$x' = D\,x = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \cos(\alpha)\,x_1 - \sin(\alpha)\,x_2 \\ \sin(\alpha)\,x_1 + \cos(\alpha)\,x_2 \end{pmatrix}$$

Die trigonometrischen Funktionen können mit den Klassenmethoden **Math.cos()** und **Math.sin()** berechnet werden. Für die Umwandlung von Winkelgraden in Bogenmaß steht die Methode **Math.toRadians()** bereit.

Erstellen Sie ein Demonstrationsprogramm, das Ihre Vektor-Klasse verwendet.
In folgendem Beispiel wurde für eine formatierte Ausgabe der **double**-Werte gesorgt:

```
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)
```

Dazu wurde die seit Java 5.0 in der Klasse **PrintStream** definierte und damit auch vom Objekt **System.out** beherrschte Methode **printf()** benutzt, die sich analog zur **String**-Methode **format()** verhält (siehe oben). Im folgenden Beispiel enthält der Formatierungsparameter den Platzhalter **%.2f** für eine Fließkommazahl mit zwei Dezimalstellen:

```
System.out.printf("\nSumme der Vektoren      (%.2f; %.2f)\n", v1.x, v1.y);
```

Entsprechend folgen dem **String**-Parameter zwei Aktualparameter von Typ **double**, die in der Ausgabe an Stelle der Platzhalter im gewünschten Format (mit dem länderspezifischen Dezimaltrennzeichen!) erscheinen sollen.

5) 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.

5 Elementare Klassen

In diesem Abschnitt wird gewissermaßen die objektorientierte Fortsetzung der elementaren Sprachelemente aus Abschnitt 0 präsentiert. Es werden wichtige Bausteine für Java-Programme (z.B. Felder, Zeichenketten) behandelt, die als *Klassen* realisiert sind. Sie gehören entweder zur Sprache Java selbst, oder zum **Java-API** (**A**pplication **P**rogramming **I**nterface), einer mächtigen Bibliothek mit Klassen für praktisch alle Standardaufgaben der Programmentwicklung (z.B. Stringverarbeitung, Dateizugriffe, Netzverbindungen), die integraler Bestandteil des Java-SDK ist. Die in Abschnitt 5 vorgestellten Klassen stehen *immer* zur Verfügung, so dass es zur akademischen Frage wird, ob man sie der *Programmiersprache* Java oder der Java-*Bibliothek* zuordnet.
Weil die API-Klassen in so genannten *Paketen* organisiert sind, folgt erst im Zusammenhang mit dieser wichtigen Software-Strukturierungsmöglichkeit eine systematische Beschreibung der Java-Klassenbibliothek. U.a. wird dort geklärt, welche API-Klassen stets verfügbar sind, und welche importiert werden müssen.

5.1 Arrays (Felder)

Ein Feld bzw. Array ist ein Objekt, das als Instanzvariablen eine feste Anzahl von Elementen desselben Datentyps enthält. Hier ist als Beispiel ein eindimensionales Feld namens `uni` mit 5 Elementen vom Typ **int** zu sehen:

uni[0]	uni[1]	uni[2]	uni[3]	uni[4]
1950	1991	1997	2057	2005

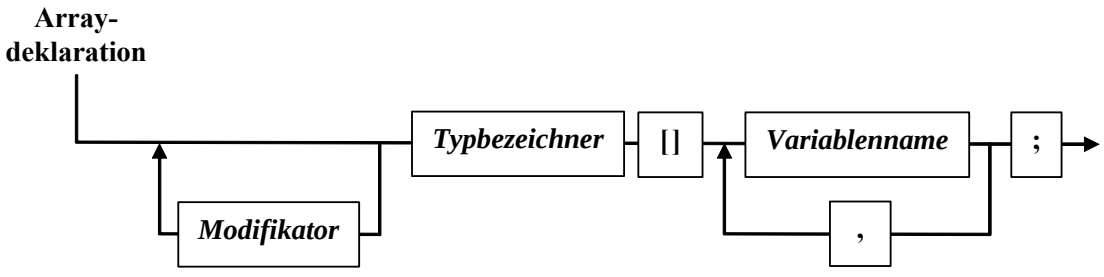
Beim Zugriff auf ein einzelnes Element gibt man nach dem Arraynamen den durch eckige Klammern begrenzten Index an, wobei die Nummerierung mit der 0 beginnt.
Im Vergleich zur Verwendung einer entsprechenden Anzahl von Einzelvariablen ergibt sich eine erhebliche Vereinfachung der Programmierung:

- Bei der *gemeinsamen* Verarbeitung aller Elemente (z.B. Array als Aktualparameter beim Methodenaufruf)
- Bei der Ansprache der einzelnen Elemente über einen *Index* (z.B. in Wiederholungsanweisungen), wobei nur *ein* Variablenname (für den Array) benötigt wird und der Index flexibel über einen Ausdruck (z.B. eine Laufvariable) geliefert werden kann.

Wir befassen uns zunächst mit *eindimensionalen* Arrays, behandeln später aber auch den mehrdimensionalen Fall.

5.1.1 Array-Referenzvariablen deklarieren

Arrays sind in Java stets *Objekte*, so dass diese eigentlich recht elementaren Programmbestandteile erst jetzt behandelt werden. Eine Array-Variable ist in Java vom Referenztyp und wird folgendermaßen deklariert:



Im Vergleich zu der bisher bekannten Variablendeklaration (ohne Initialisierung) ist hinter dem Typbezeichner zusätzlich ein Paar mit eckigen Klammern anzugeben.[18] In unserem Beispiel könnte die Array-Variable `uni` also z.B. folgendermaßen deklariert werden:

```
int[] uni;
```

Bei der Deklaration entsteht nur eine Referenzvariable, nicht jedoch der Array selbst!
Daher ist auch keine Array-Größe anzugeben.

5.1.2 Array-Objekte erzeugen

Mit Hilfe des **new**-Operators erzeugt man ein Array-Objekt mit einem bestimmten Typ 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`:

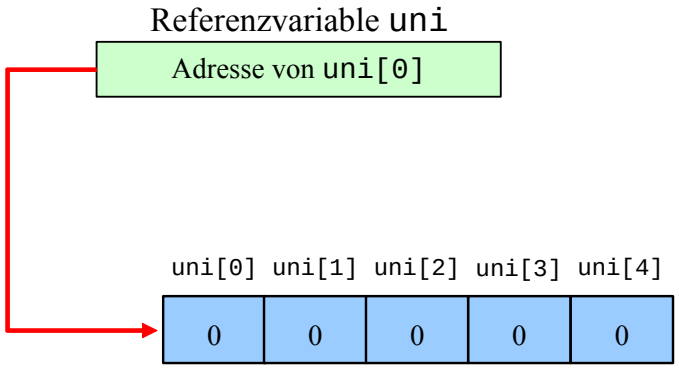
```
uni = new int[max+1];
```

Im **new**-Operanden muss hinter dem Datentyp zwischen eckigen Klammern die Anzahl der Elemente festgelegt werden, wobei ein beliebiger Ausdruck mit ganzzahligem Wert (Typ **byte**, **short**, **int, long** oder **char**) erlaubt ist.
In Java kann man 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 **Fehler! Verweisquelle konnte nicht gefunden werden.**) oder selbst eine so genannte *verkettete Liste* aufbauen.

Die Deklaration einer Array-Referenzvariablen und die Erstellung des Array-Objektes 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:



Array-Objekt mit 5 **int**-Elementen

Weil es sich bei den Array-Elementen um Instanzvariablen eines *Objektes* handelt, erfolgt eine automatische Initialisierung nach den Regeln von Abschnitt 4.1.3. Die **int**-Elemente im letzten Beispiel werden also mit 0 initialisiert.

---

[18] Alternativ dürfen die eckigen Klammern auch *hinter* dem Variablennamen stehen, wobei dann aber andere, in derselben Deklarationsanweisung auftretende Variablen *nicht* zu Array-Referenzen erklärt werden.

Aus der Objekt-Natur eines Arrays folgt unmittelbar, dass er vom Garbage Collector entsorgt wird, wenn keine Referenz mehr auf ihn zeigt. Um eine Referenzvariable aktiv von einem Array zu „entkoppeln“, kann man ihr z.B. den Wert **null** (Zeiger auf nichts) oder aber eine alternative Referenz zuweisen. Gemäß ihrer Deklaration kann eine Array-Referenzvariable auf einen beliebigen Array mit Elementen vom vereinbarten Typ zeigen. Es ist auch ohne weiteres möglich, dass mehrere Referenzvariablen auf dasselbe Array-Objekt zeigen.

5.1.3 Arrays verwenden

Der Zugriff auf die Elemente des Array-Objektes geschieht über eine Referenzvariable, an die zwischen eckigen Klammern ein passender Index angehängt wird. In Indexposition ist ein beliebiger Ausdruck mit ganzzahligem Wert erlaubt (Typ **byte**, **short**, **int, long** oder **char**), 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 nach

```
int[] uni = new int[5];
```

kein Element `uni[5]`. Hier können sich leicht Fehler einschleichen, die das Laufzeitsystem veranlassen, eine Ausnahme zu melden und die verantwortliche Methode zu beenden, z.B.:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at UniRand.main(UniRand.java:14)
```

Man kann sich in Java generell darauf lassen, dass jede Überschreitung von Feldgrenzen verhindert wird, so dass es nicht zur unbeabsichtigten Verletzung anderer Speicherbereiche kommt. In anderen Programmiersprachen (z.B. C/C++) fehlt eine garantierte Feldgrenzenüberwachung. Hier führt ein entsprechender Programmierfehler im günstigen Fall zu einem Absturz, weil das Betriebssystem eine Speicherschutzverletzung feststellt. In ungünstigen Fällen kann es aber auch zu einem kaum nachvollziehbaren Fehlverhalten kommen.

Warum das erste Array-Element die Nummer 0 besitzt, wird durch folgende Überlegungen verständlich:

- Die Referenzvariable `uni` zeigt auf das *erste* Element des Array-Objektes.
- Der Operator **[]** wird auch als *Offset-Operator* bezeichnet. Bei einem Offset von 0 bewegt man sich *nicht* vom Anfang des Array-Objektes weg, spricht also das erste Element an.

Einem Programm muss natürlich die Länge aller Arrays bekannt sein. Damit bei Arrays mit problemabhängig (z.B. durch Benutzerentscheidung) festgelegter Länge keine lästige Routinearbeit entsteht, speichert Java die Länge eines Array-Objekts automatisch in einer finalisierten Instanzvariablen namens **length**:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.print("Laenge des Vektors: ");` `  int[] wecktor = new int[Simput.gint()];` `  System.out.println();` `  for(int i = 0; i < wecktor.length; i++) {` `  System.out.print("Wert von Element "+i+": ");` `  wecktor[i] = Simput.gint();` `  }` `  System.out.println();` `  for(int i = 0; i < wecktor.length; i++)` `  System.out.println(wecktor[i]);` `  }` `}`	`Laenge des Vektors: 3` `Wert von Element 0: 7` `Wert von Element 1: 13` `Wert von Element 2: 4711` `7` `13` `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*-Präfix weg.

Nach der folgenden Anweisung zeigt die Referenzvariable `zzg` auf ein Objekt der Klasse **Random** aus dem API-Paket **java.util**:

```
java.util.Random zzg = new java.util.Random();
```

Durch Verwendung des parameterfreien **Random**-Konstruktors entscheidet man sich für die Anzahl der Sekunden seit dem 1.1.1970, 00.00 Uhr, als Starwert für den Pseudozufall. Lieferant dieses Wertes ist die statische Methode **currentTimeMillis()** der Klasse **System** im API-Paket **java.lang**.[19] Obige Anweisung ist also äquivalent mit:

```
java.util.Random zzg = new java.util.Random(System.currentTimeMillis());
```

[19] Diese Klasse leistet uns bekanntlich über die **println()**-Methode ihrer Klassenvariablen **out** bei der Konsolenausgabe schon geraume Zeit wertvolle Dienste. Ihre Methode **currentTimeMillis()** ist u.a. bei der Messung von Zeitdifferenzen sehr nützlich.

Das angekündigte Programm zieht 10000 Zufallszahlen und überprüft deren 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 + " ");
  }
}
```

In folgender Schleife werden mit der **Random**-Methode **nextInt()** Zufallszahlen aus der Menge {0, 1, 2, 3, 4} gezogen, die dann als Array-Index dienen. Folglich wird bei jedem Schleifendurchgang ein zufällig gewähltes `uni`-Element inkrementiert:

```
for (i = 1; i <= drl; i++)
  uni[zzg.nextInt(5)]++;
```

Ein guter Pseudozufallszahlengenerator liefert eine annähernde Gleichverteilung über der Trägermenge. Das **Random**-Objekt wird diesem Anspruch durchaus gerecht, wie das folgende Ergebnis-Beispiel zeigt:

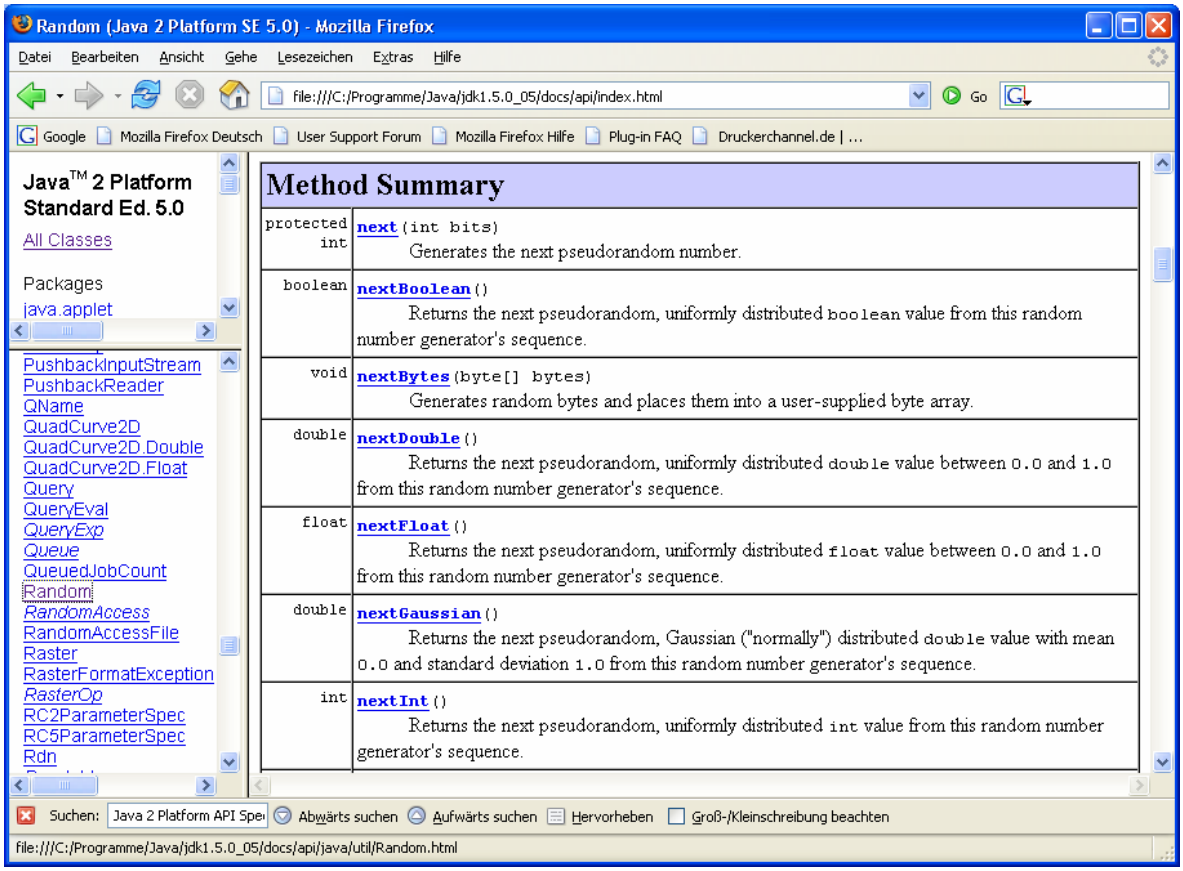
```
Absolute Haeufigkeiten:
1950 1991 1997 2057 2005

Relative Haeufigkeiten:
0.195 0.1991 0.1997 0.2057 0.2005
```

Über die im Beispielprogramm verwendeten Java-Klassen **java.util.Random** und **java.lang.System** können Sie sich mit Hilfe der JDK-Dokumentation informieren:

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

  C:\Programme\Java\jdk1.5.0_05\docs\index.html

  Auf den Pool-PCs an der Universität Trier steht zum Starten der Dokumentation in der **Java-2-SDK** – Programmgruppe der Eintrag **Java 2 SDK Documentation** zur Verfügung.
- Klicken Sie auf den Link **Java 2 Platform API Specification**.
- Klicken Sie im linken oberen Frame auf **All Classes**.
- Klicken Sie im linken unteren Frame auf den Klassennamen **Random**.
  Anschließend erscheinen im rechten Frame detaillierte Informationen über die Klasse **Random**, z.B.:



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.:[20]

```
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 {` `  public static void main(String[] args) {` `  int[] wecktor = {1, 2, 3};` `  System.out.println(wecktor[2]);` `  }` `}`	3

Die Deklarations- und Initialisierungsanweisung

```
int[] wecktor = {1, 2, 3};
```

ist äquivalent zu:

---

[20] Im Hintergrund erzeugt die Methode beim ihrem ersten Aufruf ein **Random**-Objekt über den parameterfreien Konstruktor:

```
new java.util.Random()
```

```
int[] wecktor = new int[3];
wecktor[0] = 1;
wecktor[1] = 2;
wecktor[2] = 3;
```

Initialisierungslisten können sind nur bei der Deklaration erlaubt, sondern auch bei der Objektkreation, z.B.:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  int[] wecktor;` `  wecktor = new int[] {1, 2, 3};` `  System.out.println(wecktor[2]);` `  }` `}`	`3`

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 Felder entstehen.

5.1.7 Mehrdimensionale Felder

In der linearen Algebra und in vielen anderen Anwendungsbereichen werden auch *mehrdimensionale* Arrays benötigt. Ein zweidimensionales Feld 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.

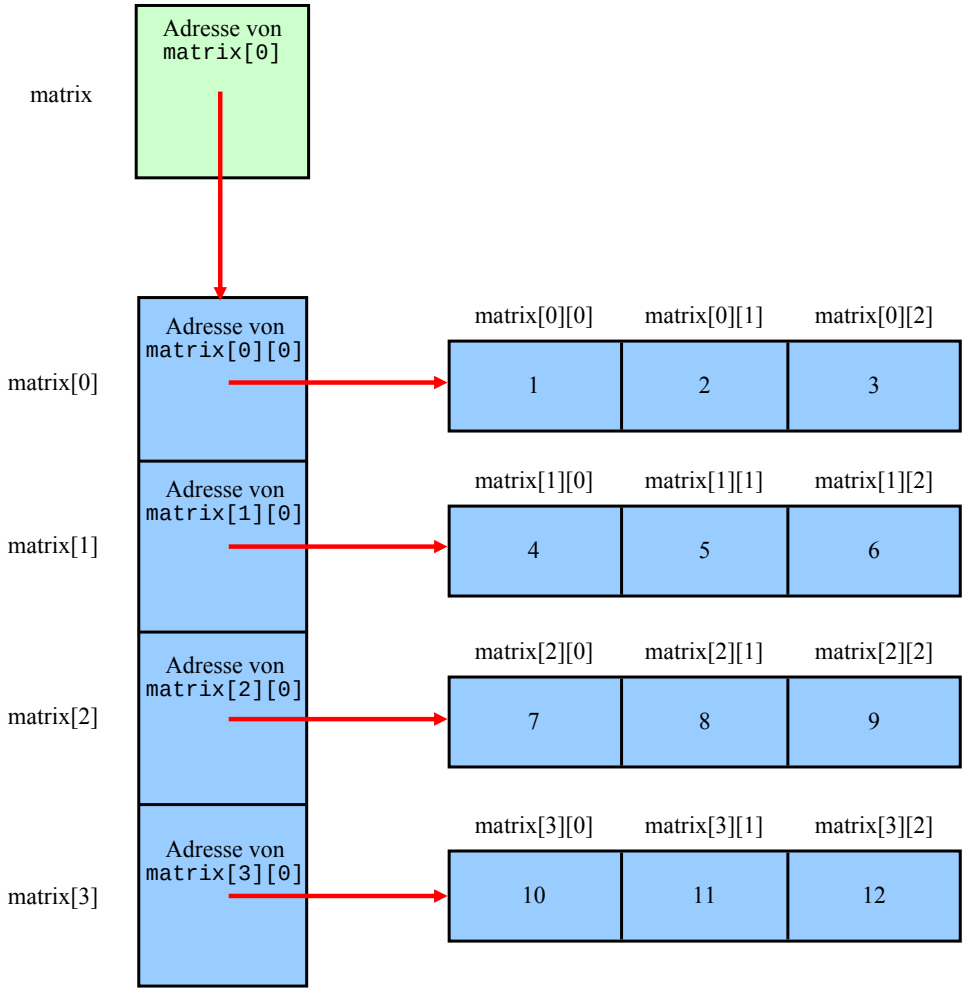
Im Beispiel wird ein Array-Objekt namens `matrix` mit den vier Elementen `matrix[0]` bis `matrix[3]` erzeugt, bei denen es sich jeweils um einen Array mit drei **int**-Elementen handelt. Wir haben eine zweidimensionale Matrix zur Verfügung, auf deren Zellen man per Doppelindizierung zugreifen kann, wobei 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 z.B. geschieht, um die Länge der eindimensionalen Zeilen-Arrays zu ermitteln:

```
matrix[i].length
```

In der folgenden Abbildung, die sich an Mössenböck (2005, S. 102) orientiert, wird mit einem Pfeil die Relation *Referenzvariable zeigt auf Array-Objekt* ausgedrückt:



Auch im mehrdimensionalen Fall können Initialisierungslisten eingesetzt werden, z.B.:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  int[][] matrix = {{1}, {1,2}, {1, 2, 3}};` `  for(int i=0; i < matrix.length; i++) {` `  for(int j=0; j < matrix[i].length; j++)` `  System.out.print(matrix[i][j]+" ");` `  System.out.println();` `  }` `  }` `}`	`1` `1 2` `1 2 3`

Im letzten Beispielprogramm wird auch die in Java gegebene Flexibilität genutzt, eine Matrix mit unterschiedlich langen Zeilen anzulegen. So kann man sich etwa bei einer symmetrischen Matrix Platz sparend darauf beschränken, die untere Dreiecksmatrix abzuspeichern.

5.1.8 Übungsaufgaben zu Abschnitt 5.1

1) 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, .., *k-1* wird das Minimum gesucht und an den linken Rand befördert. Dann wird der Vektor mit den Elementen 1 bis *k-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.

2) Erstellen Sie ein Programm zur Primzahlensuche mit dem **Sieb des Eratosthenes**. 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.
- Dann geschieht iterativ folgendes:
  - Als neue Basis *b* wird die kleinste Zahl gewählt, welche folgende 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.

- Das Verfahren kann enden, wenn für eine neue Basis *b* gilt:
  $$b > \sqrt{K}$$
  Es kann keine Streichkandidaten mehr geben, denn: Ist eine natürliche Zahl $n \leq K$ *keine* Primzahl, dann gibt es natürliche Zahlen $x, y \in \{2, 3, ...\}$ mit
  $$n = x \cdot y, x \leq \sqrt{K} \text{ oder } y \leq \sqrt{K}$$
  Wegen $\sqrt{K} < b$ ist *n* bereits gestrichen worden. 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. Die Primzahlen kleiner oder gleich 18 lauten also:

1, 2, 3, 5, 7, 11, 13 und 17

3) Erstellen Sie eine Klasse für zweidimensionale Matrizen mit Elementen vom Typ **float**. Implementieren Sie eine Methode zum Transponieren einer Matrix und vielleicht noch andere Methoden für klassische Aufgaben der Matrixalgebra.

5.2 Zeichenketten

Java bietet für den Umgang mit Zeichenketten zwei Klassen an:

- **String**
  String-Objekte können nach dem Erzeugen nicht mehr geändert werden. Diese „Read Only“ - 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 beliebig 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:

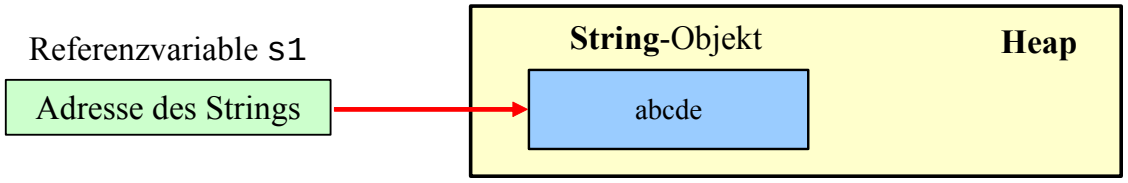
- eine **String**-Referenzvariable namens `s1` angelegt,
- ein neues **String**-Objekt auf dem Heap erzeugt,
- die Adresse des Heap-Objektes in der Referenzvariablen abgelegt

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**-Objektes.
Die Anweisung erzeugt im Hauptspeicher folgende Situation:



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: Existiert dort bereits ein String-Objekt mit demselben Inhalt, wird dessen Adresse verwendet. Anderenfalls wird ein neues Objekt im String-Pool angelegt. Damit gilt für Variablen mit Referenzen auf String-Pool-Objekte: Zwei Variablen stehen genau dann für dieselbe Zeichenfolge, wenn sie denselben Referenzwert haben.

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:

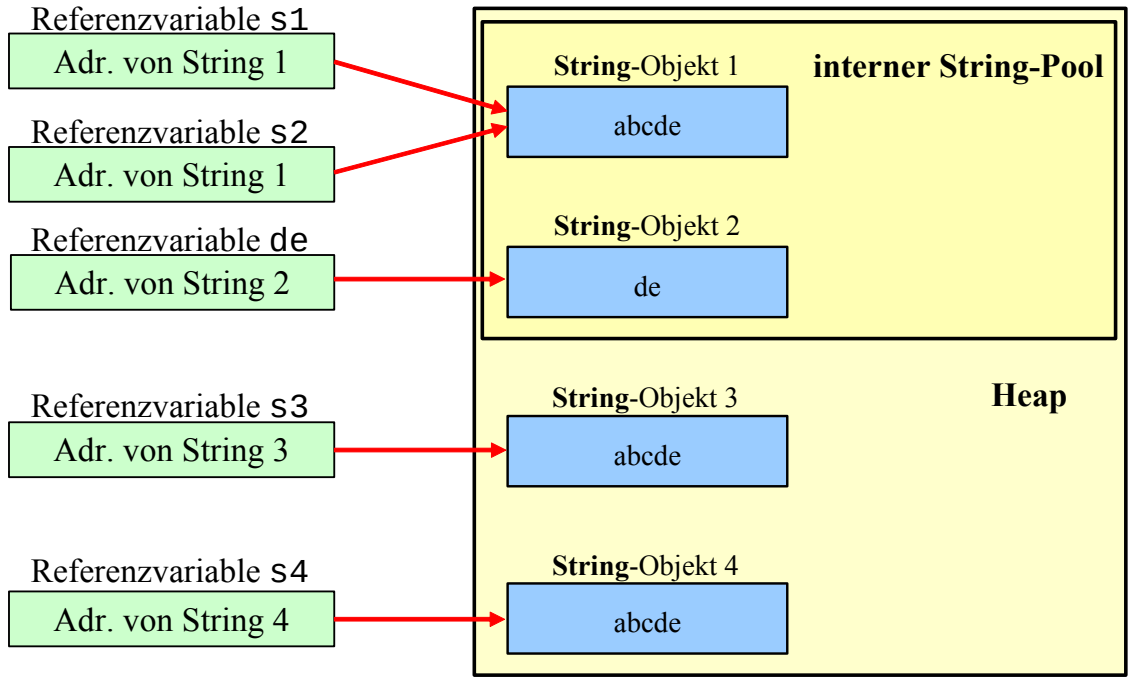
Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  String s1 = "abcde";` `  String s2 = "abc"+"de";` `  String de = "de";` `  String s3 = "abc"+de;` `  String s4 = new String("abcde");` `  System.out.print("(s1 == s2) = "+(s1==s2)+"\n"+` `  "(s1 == s3) = "+(s1==s3)+"\n"+` `  "(s1 == s4) = "+(s1==s4));` `  }` `}`	`(s1 == s2) = true` `(s1 == s3) = false` `(s1 == s4) = false`

Das merkwürdige[21] Verhalten des Programms hat folgende Ursachen:

[21] „Merkwürdig“ bedeutet hier, dass sich eine Aufnahme in das Langzeitgedächtnis lohnt.

- 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, 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:



In Abschnitt 5.2.1.4.2 lernen Sie die **String**-Methode **equals()** kennen, die auf jeden Fall einen Vergleich der Zeichenfolgen 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 {` `  public static void main(String[] args) {` `  String testr = "abc";` `  System.out.println("testr = " + testr);` `  testr = testr + "def";` `  System.out.println("testr = " + testr);` `  }` `}`	`testr = abc` `testr = abcdef`

Was sich hier ändert, ist der Inhalt der Referenzvariablen `testr`: Die Referenz auf das **String**-Objekt mit dem Text *abc* wird ersetzt durch die Referenz auf ein neues **String**-Objekt mit dem Text abcdef. Das alte Objekt ist noch vorhanden, aber nicht mehr referenziert. Sobald das Laufzeitsystem 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 {` `  public static void main(String[] args) {` `  System.out.println("4 + 3 = " + (4 + 3));` `  }` `}`	`4 + 3 = 7`

Es ist übrigens eine Besonderheit, dass **String**-Objekte mit dem + - Operator verarbeitet werden können. Bei anderen Java-Klassen ist das aus C++ bekannte Überladen von Operatoren *nicht* erlaubt.

5.2.1.4.2 Vergleichen von Strings

Für den Test auf identischen **Inhalt** kann die **String**-Methode

boolean equals(String *vergl***)**

verwendet werden, 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 {` `  public static void main(String[] args) {` `  String s1 = "abc";` `  String s2 = new String("abc");` `  System.out.println(s1==s2);` `  System.out.println(s1.equals(s2));` `  }` `}`	`false` `true`

Zum Testen auf **lexikographische Priorität** (z.B. beim Sortieren) kann die **String**-Methode

int compareTo()

dienen, z.B.:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  String a = "Müller, Anja", b = "Müller, Kurt",` `  c = "Müller, Anja";` `  System.out.println("< : " + a.compareTo(b));` `  System.out.println("= : " + a.compareTo(c));` `  System.out.println("> : " + b.compareTo(a));` `  }` `}`	`< : -10` `= : 0` `> : 10`

compareTo() liefert folgende Rückgabewerte:

		compareTo()-Ergebnis
String, 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 *Instanzvariablen* **length** zu finden ist (vgl. Abschnitt 5.1), wird die aktuelle Länge einer Zeichenkette über die *Instanzmethode* **length()** ermittelt:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  char[] vek = {'a', 'b', 'c'};` `  String str = "abc";` `  System.out.println(vek.length);` `  System.out.println(str.length());` `  }` `}`	`3` `3`

5.2.1.4.4 Zeichen extrahieren, suchen oder ersetzen

Im Abschnitt 3.6.2.2 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 Kommandozeilen-Parametern haben. Im folgenden Beispielprogramm wird das erste Element dieses Arrays untersucht:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  if (args.length > 0) {` `  System.out.println(args[0].substring(0,args[0].length()));` `  System.out.println(args[0].indexOf("t"));` `  System.out.println(args[0].indexOf("x"));` `  System.out.println(args[0].startsWith("r"));` `  System.out.println(args[0].charAt(0));` `  }` `  }` `}`	`rot` `2` `-1` `true` `r`

Mit der Methode

String substring(int *start***, int** *ende***)**

lassen sich alle Zeichen zwischen *start* (inklusive) und *ende* (exklusive) extrahieren. Wie man am Beispiel sieht, harmoniert die „exklusive“ Endregel (aufgrund der nullbasierten Indexierung) gut mit dem Verhalten der **length()**-Methode.

Mit der Methode

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

Mit der Methode

boolean startsWith(String *start*)

lässt sich feststellen, ob ein **String** mit einer bestimmten Zeichenfolge beginnt.

Weil ein **String** kein Array ist, kann auf die einzelnen Zeichen *nicht* per Offset-Operator ([]) zugegriffen werden. Mit der **String**-Methode

char charAt()

steht aber ein Ersatz zur Verfügung, wobei die Nummerierung der Zeichen wiederum bei 0 beginnt.

Wenn auf jeden Fall mit dem Offset-Operator gearbeitet werden soll, kann aus einem **String** über die Methode

char[] toCharArray()

ein neuer **char**-Array mit identischem Inhalt erzeugt werden, z.B.:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  String s = "abc";` `  char[] c = s.toCharArray();` `  for (int i = 0; i < c.length; i++)` `  System.out.println(c[i]);` `  }` `}`	`a` `b` `c`

Mit der Methode

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');
```

5.2.1.4.5 Groß-/Kleinschreibung normieren

Mit den Methoden

String toUpperCase()

bzw.

String toLowerCase()

erhält man einen neuen **String**, der im Unterschied zum angesprochenen Original auf Groß- bzw. Kleinschreibung normiert ist, was vor Vergleichen oft sinnvoll ist, z.B.:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  String a = "Otto", b = "otto";` `  System.out.println(a.toUpperCase().equals(b.toUpperCase()));` `  }` `}`	`true`

Im Aufruf der **equals()**-Methode 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.:

- **StringBuffer**(**String** *str*)
  Beispiel: `StringBuffer sb = new StringBuffer("abc");`
- **StringBuffer**()
  Beispiel: `StringBuffer sb = new StringBuffer();`

In folgendem Programm wird eine Zeichenkette 20000-mal verlängert, zunächst mit Hilfe der **String**-Klasse, dann mit Hilfe der **StringBuffer**-Klasse:

```
class Prog {
  public static void main(String[] args) {
    String s = "*";
    long vorher = System.currentTimeMillis();
    for (int i = 0; i < 20000; i++)
      s = s + "*";
    long diff = System.currentTimeMillis() - vorher;
    System.out.println("Zeit fuer String-Manipulation: " + diff);

    s = "*";
    StringBuffer t = new StringBuffer(s);
    vorher = System.currentTimeMillis();
    for (int i = 0; i < 20000; i++)
      t.append("*");
    s = t.toString();
    diff = System.currentTimeMillis() - vorher;
    System.out.println("Zeit fuer StringBuffer-Manipulation: " + diff);
  }
}
```

Die Laufzeiten (gemessen in Millisekunden auf einem PC mit 3-GHz-CPU) unterscheiden sich erheblich:

```
Zeit fuer String-Manipulation:       656
Zeit fuer StringBuffer-Manipulation: 0
```

Ein **StringBuffer**-Objekt beherrscht u.a. die folgenden 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.: `sb.append("*");` Es sind **append()**-Überladungen für zahlreiche Datentypen vorhanden.
insert()	Die Stringrepräsentation des Argumentes, das von nahezu beliebigem

StringBuffer-Methode	Erläuterung
	Typ sein kann, wird an einer bestimmten Stelle eingefügt, z.B.: `sb.insert(4, 3.14);`
delete()	Die Zeichen von einer Startposition (einschließlich) bis zu einer Endposition (ausschließlich) werden gelöscht, in folgendem Beispiel also gerade 2 Zeichen, falls der **StringBuffer** mindestens 3 Zeichen enthält: `sb.delete(1,3);`
replace()	Ein Bereich des **StringBuffer**-Objekts wird durch den Argument-**String** ersetzt, z.B.: `sb.replace(1,3, "xy");`
String toString()	Es wird ein **String**-Objekt mit dem Inhalt des **StringBuffer**-Objekts erzeugt. Dies ist z.B. erforderlich, um zwei **StringBuffer**-Objekte mit Hilfe der **String**-Methode **equals()** vergleichen zu können: `sb1.toString().equals(sb2.toString())`

5.2.3 Übungsaufgaben zu Abschnitt 5.2

1) 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 2 Kommandozeilenparameter übergeben wurden.
Tipp: Um ein Programm spontan zu beenden, kann die Methode **exit()** der Klasse **System** verwendet werden.

2) Die Klassen **String** und **StringBuffer** besitzen beide eine Methode namens **equals()**, doch bestehen gravierende Verhaltensunterschiede:

Quellcode

```
class Prog {
  public static void main(String[] args) {
    StringBuffer sb1 = new StringBuffer("abc");
    StringBuffer sb2 = new StringBuffer("abc");
    System.out.println("sb1 = sb2 = "+sb1);
    System.out.println("StringBuffer-Vergl.: "+
      sb1.equals(sb2));

    String s1 = sb1.toString();
    String s2 = sb1.toString();
    System.out.println("\ns1  = s2  = "+s1);
    System.out.println("String-Vergl.:       "+
      s1.equals(s2));
  }
}
```

Ausgabe

```
sb1 = sb2 = abc
StringBuffer-Vergl.: false

s1  = s2  = abc
String-Vergl.:       true
```

Ermitteln Sie mit Hilfe der JDK-Dokumentation die Ursache für das unterschiedliche Verhalten.

3) 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 können 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 {` `  public static void main(String[] args) {` `    String s = "Dies ist der Satz, der zerlegt werden soll.";` `    java.util.StringTokenizer stob =` `      new java.util.StringTokenizer(s, " ", false);` `    while (stob.hasMoreTokens())` `      System.out.println(stob.nextToken());` `  }` `}`	`Dies` `ist` `der` `Satz,` `der` `zerlegt` `werden` `soll.`

Was als Token interpretiert werden soll, legt man über den letzten Parameter im Konstruktor der Klasse **java.util.StringTokenizer** fest (**boolean returnDelims**):

- **false**: Ein Token ist die maximale Sequenz von Zeichen, die keine Trennzeichen sind. Die Trennzeichen dienen nur zum Separieren der Tokens und werden ansonsten *nicht* zurück geliefert.
- **true**: Auch die Trennzeichen werden als Tokens behandelt.

5.3 Verpackungs-Klassen 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

void	Void

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 (jeweils klassenbezogen).

In der Regel verfügen die Wrapper-Klassen über zwei Konstruktoren mit jeweils einem Parameter, der vom primitiven „Basistyp“ oder vom Typ **String** sein darf, z.B. bei **Integer**:

- **Integer**(**int** *value*)
  Beispiel: `Integer iw = new Integer(4711);`
- **Integer**(**String** *str*)
  Beispiel: `Integer iw = new Integer(args[0]);`

Für den Einsatz in numerischen Ausdrücken sind die Wrapper-Klassen wenig geeignet, weil sie im Einweg- bzw. read-only – Sinn funktionieren: Der beim Erzeugen für ein Wrapper-Objekt festgelegte Wert kann nicht mehr geändert werden. Für diesen Zweck sind schlicht keine Methoden vorhanden.

5.3.1 Autoboxing

Seit der Java-Version 5 kann der Compiler das Einpacken automatisch erledigen, so dass man die Wrapper-Konstruktoren mit Basistypargument nicht mehr explizit aufrufen muss, 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.

Um Werte primitiver Typen in einen Vector 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().toString());
  }
}
```

Wie die Programmausgabe zeigt, sind tatsächlich diverse Wrapper-Objekte 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. An dieser Stelle wird die vor Java 5 übliche **Integer**-Methode **intValue()** nicht mehr benötigt.

Dank Autoboxing sind die elementaren Typen nun auch zuweisungskompatibel mit der 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 Strings in einen Wert des jeweiligen Basistyps zur Verfügung. Mit der Klassenmethode **valueOf(String** *str***)** erhält man ein neues Wrapper-Objekt, falls die Konvertierung des Strings gelingt.
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.valueOf(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 **valueOf()**, deren Einsatz allerdings gleich kritisiert werden muss.
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
```

Die Anweisung

```
summe += Double.valueOf(s);
```

produziert bei jeder Ausführung ein neues **Double**-Objekt, dessen Wert dann per Autounboxing entnommen wird.
Zur Vermeidung der sinnlosen Objektproduktion sollte an dieser Stelle die die Methode

double parseDouble(String *str***)**

verwendet werden, die als Rückgabe einen Wert des primitiven Datentyps **double** liefert, z.B.:

```
summe += Double.parseDouble(s);
```

Auch die Klasse **String** bietet eine Methode **valueOf()**, in diesem Fall aber zur Wandlung primitiver oder sonstiger Datentypen in Strings, z.B.:

```
String s = String.valueOf(summe);
```

5.3.3 Konstanten mit Grenzwerten

In den numerischen Wrapper-Klassen sind 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 {` `  public static void main(String[] args) {` `  System.out.println("Max. double-Zahl:\n"+` `  Double.MAX_VALUE);` `  }` `}`	`Max. double-Zahl:` `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**.
boolean isWhitespace(char *ch*)	Die Methode liefert den Wert **true** zurück, wenn ein Trennzeichen übergeben wurde, sonst **false**. Zu den Trennzeichen gehören u.a.: • Leerzeichen (\u0020) • Tabulatorzeichen (\u0009) • Wagenrücklauf (\u000D) • Zeilenvorschub (\u000A)

Methode	Erläuterung
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**.

5.3.5 Übungsaufgaben zu Abschnitt 5.3

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 **Integer**) die Werte der Objekte nach der Erzeugung zu fixieren.

4) Erweitern Sie die im Abschnitt 5.2.3 vorgeschlagene Klasse `StringUtil` um eine statische Methode namens `countChars()`, die für einen **String**–Parameter die enthaltenen Zeichen mit der Häufigkeiten des Auftretens protokolliert. In folgendem Programm wird die Verwendung der Methode demonstriert:

Quellcode	Ausgabe
`class StringUtilTest {` `  public static void main(String[] args) {` `  StringUtil.countChars("Otto spielt Lotto.");` `  }` `}`	`L: 1` `O: 1` `e: 1` `i: 1` `l: 1` `o: 3` `p: 1` `s: 1` `t: 5`

5.4 Aufzählungstypen

Angenommen, Sie wollten in einer Adressdatenbank auch den *Charakter* der erfassten Personen notieren und sich dabei an den vier Temperamentstypen des griechischen Arztes 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
  Dabei wird relativ viel Speicherplatz benötigt.
- Eine **int**-Variable mit der Kodierungsvorschrift 0 = melancholisch, 1 = cholerisch, etc.
  Es wird wenig Speicher benötigt, allerdings ist der Quellcode nur für Eingeweihte zu verstehen, z.B.:

```
if (otto.temp == 3) ...
```

Bei beiden Vorschlägen wird außerdem nicht garantiert, dass eine Variable zur Temperamentsverwaltung ausschließlich die vier vorgesehenen Werte aufnehmen kann (mangelnde **Typsicherheit**).

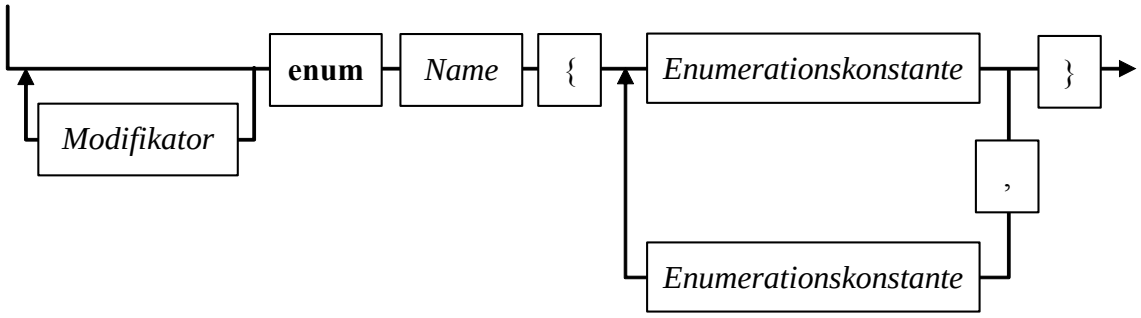
Java bietet seit der Version 5 mit den **Enumerationen** (**Aufzählungstypen)** eine Lösung, die folgende Vorteile bietet:

- Perfekte Typsicherheit
- Gut lesbarer Quellcode
- Geringer Speicherbedarf

5.4.1 Einfache Enumerationen

In ihrer einfachsten Form besteht eine Enumeration aus einer Anzahl von Konstanten, die jeweils eine Ordinalzahl (0, 1, 2, …) gemäß Definitionsreihenfolge repräsentieren und über einen eindeutigen Namen angesprochen werden können. Bei der Definition einer einfachen Enumeration folgt nach den optionalen Modifikatoren auf das Schlüsselwort **enum** und den Typbezeichner eine geschweift eingeklammerte Liste mit den Enumerationskonstanten:

Einfache Enumerationsdefinition



Wie bei den präzisen, aber nicht unbedingt mit einem Blick zu erfassenden Syntaxdiagrammen üblich, 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 generell eingebürgert hat.
Objekte der Klasse `Person` (mit schlecht geschützten Eigenschaften!) erhalten eine Instanzvariable vom Typ `Temperament`:

```
public class Person {
  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;
  }
}
```

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 anzugeben ist, z.B.:

```
switch (otto.temp) {
  case SANGUIN:    System.out.println("Lustiger Typ!"); break;
  case CHOLERISCH: System.out.println("Mit Vorsicht zu genießen!"); break;
}
```

Enumerationen sind in Java als Klassen realisiert, wobei im Vergleich zu einer gewöhnlichen Klasse einige Besonderheiten zu beachten sind:

- Die in der Enumerationsdefinition angegebenen Konstanten zeigen als statische und finalisierte Referenzvariablen auf Objekte der Enumerationsklasse, die beim Laden der Klasse 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 obigem Beispiel ist die `Person`-Eigenschaft `temp` eine Referenzvariable ist, die auf ein Objekt der Klasse `Temperament` zeigen kann. Sofern der Programmierer nicht auf eigene Verantwortung eine Wertzuweisung mit expliziter Typumwandlung vornimmt, zeigt die Instanzreferenzvariable `temp` …

- entweder auf eines der vier `Temperament`-Objekte
- oder auf **null**.

Alle Enumerationsobjekte beherrschen die Instanzmethode **ordinal()**, welche die zum Objekt gehörige Ordnungszahl liefert, z.B.:

Quellcode	Ausgabe
`class PersonTest {` `  public static void main(String[] args) {` `  Person otto = new Person("Otto", "Hummer",` `  35, Temperament.SANGUIN);` `  System.out.println(otto.temp.ordinal());` `  }` `}`	`4`

Bei jeder Enumerationsklasse kann man mit der statischen Methode **values()** einen Array mit ihren Objekten anfordern, z.B.:

Quellcode	Ausgabe
`class PersonTest {` `  public static void main(String[] args) {` `  for (Temperament t : Temperament.values())` `  System.out.println(t.name());` `  }` `}`	`MELANCHOLISCH` `CHOLERISCH` `PHLEGMATISCH` `SANGUIN`

5.4.2 Erweiterte Enumerationen

Gelegentlich bietet es sich an, eine Enumerationsklasse mit Instanzvariablen, Methoden und Konstruktoren auszustatten. Objekte der folgenden erweiterten `Temperament` - Enumeration geben

über die Methoden `stable()` bzw. `extra()` Auskunft darüber, ob die zugehörige Persönlichkeit emotional stabil bzw. extravertiert ist:[22]

```
public enum Temperament {MELANCHOLISCH(false, false),
                         CHOLERISCH(false, true),
                         PHLEGMATISCH(true, false),
                         SANGUIN(true, true);
  private boolean stable, extra;
  Temperament(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.

[22] 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.

6 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-5-API gibt es ca. 130 Pakete mit zusammengehörigen Klassen.
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.

Vielfach werden Java-Pakete auch als *Klassenbibliotheken* bezeichnet. 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. In der Regel befinden sich alle **class**-Dateien eines Paketes 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 unten).
- **Namenskonflikte vermeiden**
  Jedes Paket bildet einen eigenen Namensraum. Identische Bezeichner stellen also kein Problem dar, solange sie sich in verschiedenen Paketen befinden.
- **Zugriffskontrolle steuern**
  Pakete spielen in Java eine wichtige Rolle bei der Abschottung von Klassen gegen Zugriffe, welche die Programmstabilität gefährden könnten. 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 6.3 wird die Rolle der Pakete bei der Zugriffsverwaltung genauer erläutert.

Wird eine Klasse keinem Paket explizit zugeordnet (siehe unten), gehört sie zusammen mit allen anderen ebenfalls nicht zugeordneten Klassen im selben Dateiverzeichnis 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 Klasse 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 vielen Klassen ist aber *kein* Paketname erforderlich:

- Klassen aus dem selben 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.

6.1 Pakete erstellen

Vor der *Verwendung* von Paketen behandeln wir deren *Produktion*, weil dabei die technischen Details gut veranschaulicht werden können.

6.1.1 package-Anweisung und Paketordner

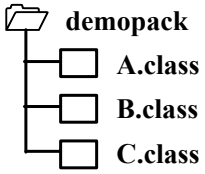
Wir erstellen zunächst ein einfaches Paket namens `demopack` mit den Klassen `A`, `B` und `C`. An den Anfang jeder einzubeziehenden Quellcodedatei setzt man die **package**-Anweisung mit dem Paketnamen, der üblicherweise *komplett klein* geschrieben wird, z.B.:

```
// Inhalt der Datei A.java
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 Paketes 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 Variablen 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 Variablen werden wir uns in Abschnitt 6.3 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. Für die Verwendung eines Paketes sind ausschließlich die **class**-Dateien erforderlich.
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**:

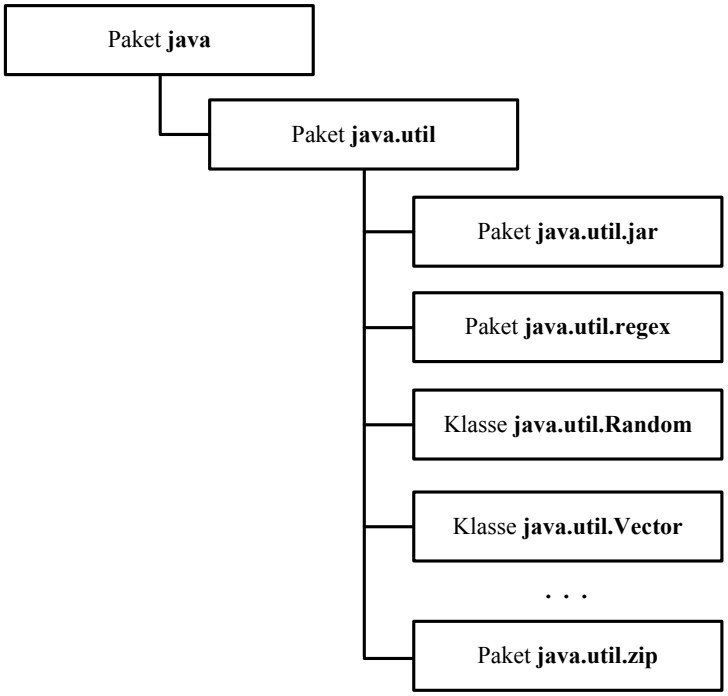


In Abhängigkeit von der verwendeten Java-Entwicklungsumgebung geschieht das Erstellen des Paketordners und das Einsortieren der Bytecode-Dateien eventuell automatisch (siehe Abschnitt 6.1.3 für Eclipse 3.x und Abschnitt 18.3.6 für JCreator 3.x).
Für die Weitergabe von Programmen ist es oft sinnvoll, den Paketordner mit dem JDK-Werkzeug **jar.exe** in eine Java-Archivdatei zu verpacken (siehe Abschnitt 6.4).

Ohne **package**-Definition am Beginn der Quellcode-Datei bilden die resultierenden Klassen zusammen mit allen anderen ebenfalls nicht zugeordneten Klassen im selben Dateiverzeichnis ein anonymes **Standardpaket** (engl. *default package*).

6.1.2 Unterpakete

Ein Paket kann hierarchisch in **Unterpakete** eingeteilt werden, was bei den API-Paketen in der Regel geschehen ist, z.B.:



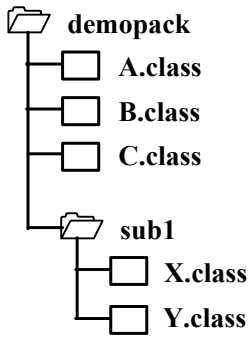
Auf jeder Stufe der Pakethierarchie können sowohl Klassen als auch Unterpakete enthalten sein. So enthält z.B. das Paket **java.util** u.a.

- die Klassen **Random**, **Vector**, ...
- die Unterpakete **jar**, **regex**, **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.
Dies ist z.B. der Quellcode der Klasse `X`, die 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:



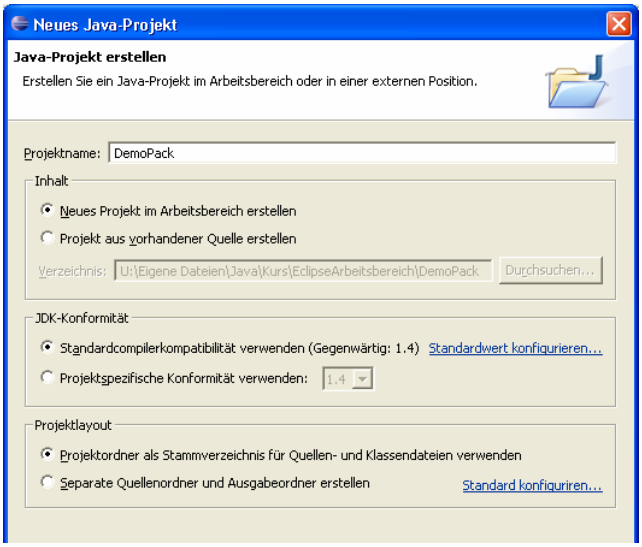
Mit dem JDK-Werkzeug **jar.exe** (siehe Abschnitt 6.4) lassen sich auch ganze Paket- bzw. Verzeichnisbäume in eine einzige Java-Archivdatei verpacken (z.B. **rt.jar** beim Java-API).

6.1.3 Paketunterstützung in Eclipse 3.x

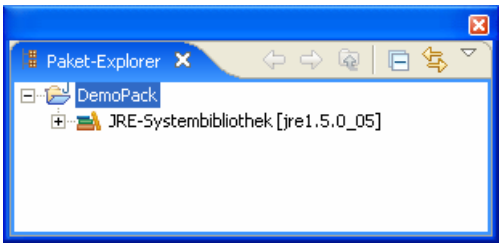
Wir starten in Eclipse über

Datei > Neu > Projekt

ein neues Java-Projekt mit dem Namen `DemoPack`:



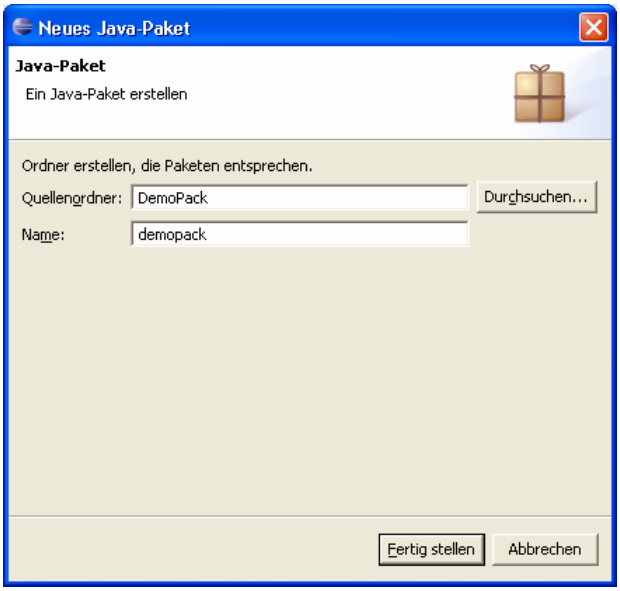
Zunächst zeigt der Paket-Explorer nur die API-Klassenbibliothek an:



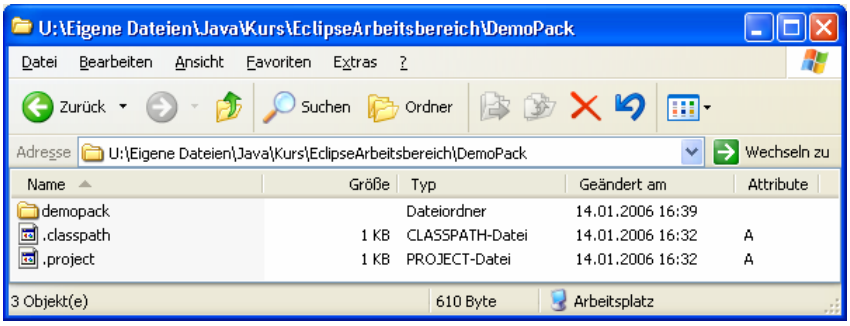
Aus dem Kontextmenü zum Projekteintrag im Paket-Explorer wählen wir die Option

Neu > Paket

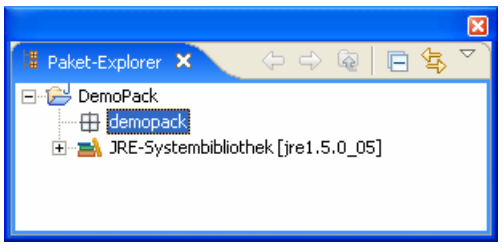
und geben in folgender Dialogbox den gewünschten Namen für das neue Paket an:



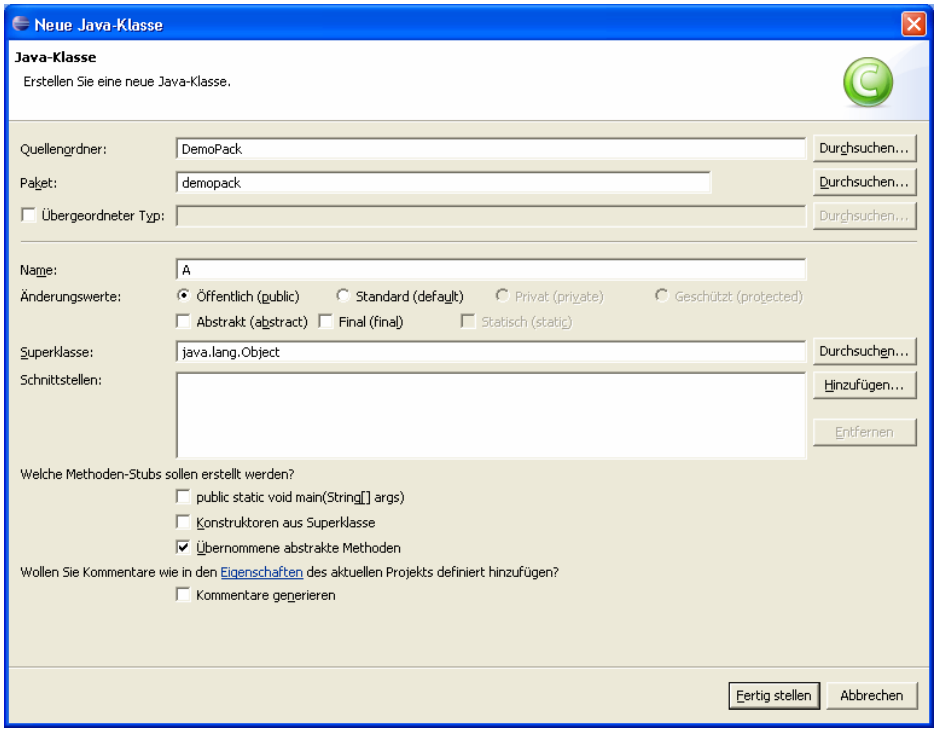
Anschließend erzeugt Eclipse den Ordner **demopack** im Projektverzeichnis



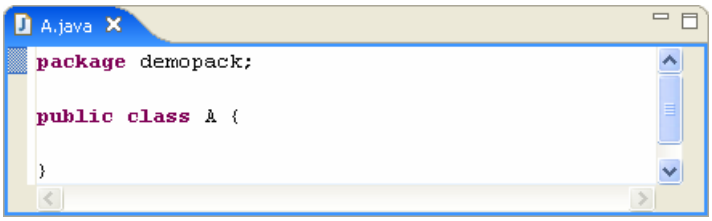
und zeigt ihn im Paket-Explorer an:



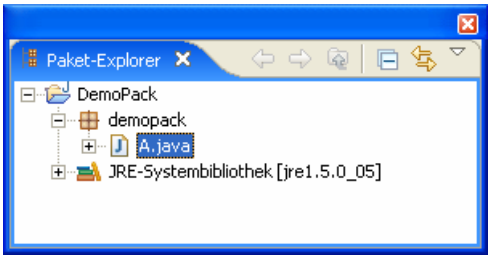
Nun legen wir im Paket `demopack` die Klasse `A` an:



Eclipse startet im Editor eine Klassendefinition mit **package**-Anweisung



und zeigt im Paket-Explorer den aktuellen Entwicklungsstand:



Wir vervollständigen den Quellcode der Klasse `A` (siehe Abschnitt package-Anweisung und Paketordner) und legen analog auch die Klassen `B` und `C` im Paket `demopack` an:

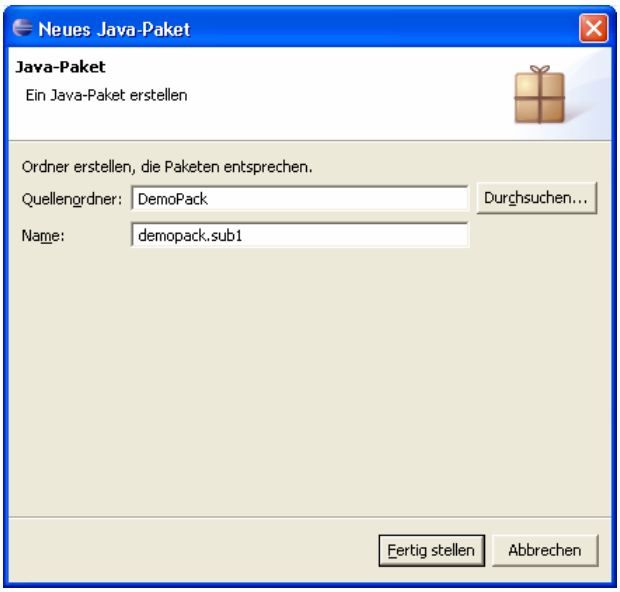
```
package demopack;

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);
  }
}
```

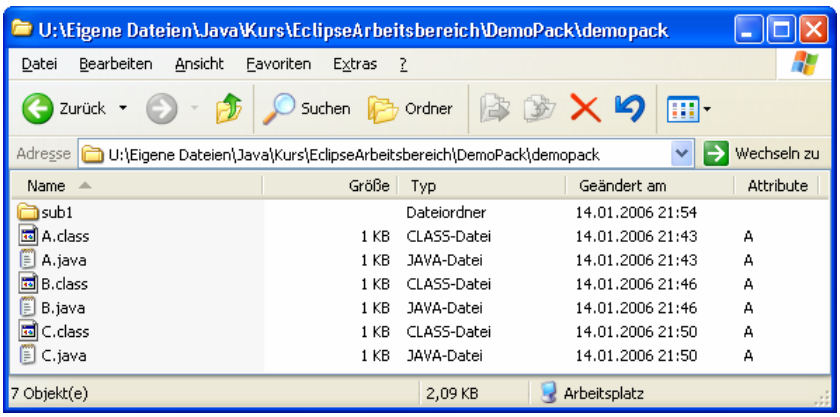
Um das Unterpaket `sub1` zu erzeugen, wählen wir erneut

Neu > Paket

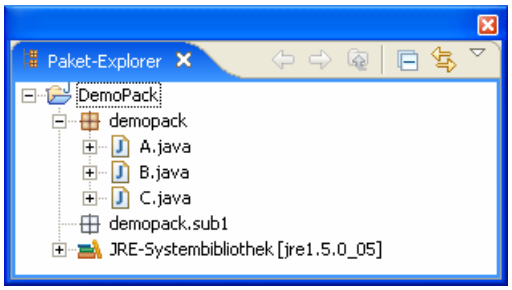
aus dem Kontextmenü zum Projekteintrag im Paket-Explorer und geben in folgender Dialogbox den gewünschten Namen für das neue Paket an:



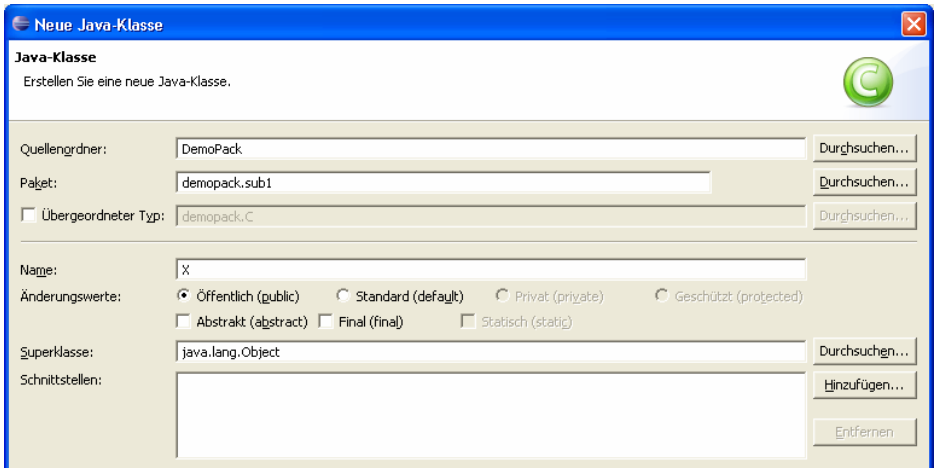
Anschließend erzeugt Eclipse den Ordner **demopack\sub1** im Projektverzeichnis



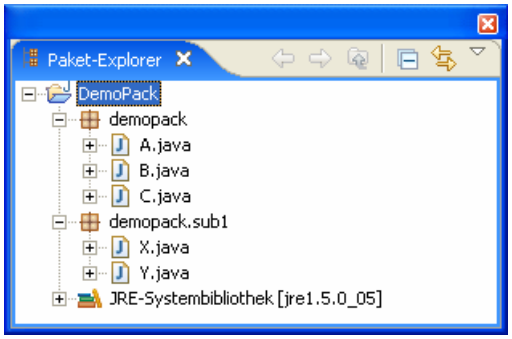
und zeigt ihn im Paket-Explorer an:



Nun legen wir im Unterpaket `demopack.sub1` die Klasse `X` an



und danach die analog aufgebaute Klasse Y.
Schließlich zeigt der Paket-Explorer folgendes Bild:

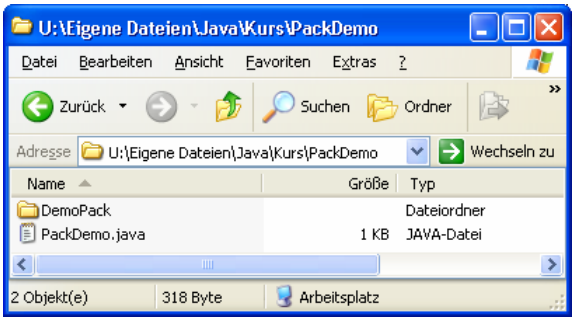


6.2 Pakete verwenden

6.2.1 Verfügbarkeit der 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 seit Java 1.2 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 6.4) und beschränken uns passend zum Entwicklungsstadium des `DemoPack`-Beispiels auf Pakete in Verzeichnissen:

- Paket im aktuellen Verzeichnis ablegen
  Das oberste Paketverzeichnis (in unserem Beispiel: **demopack**) muss sich im selben Ordner befinden wie die zu übersetzende oder auszuführende Klassendatei (im Beispiel: **PackDemo.java**), z.B.:



  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 verfügbar gemacht werden, sondern auch komplette Pakete. Wenn sich das oberste Paketverzeichnis (in unserem Beispiel: **demopack**) im Ordner

  U:\Eigene Dateien\Java\Kurs\MyLib

  befindet, kann die CLASSPATH-Definition etwa lauten:

  ```
  >set classpath=".;U:\Eigene Dateien\Java\Kurs\MyLib"
  ```

  Eclipse berücksichtigt diese Umgebungsvariable allerdings nicht. Nach meinen Erfahrungen arbeitet man bei dieser Entwicklungsumgebung am besten mit Java-Archiven (siehe Abschnitt 6.4) und spricht diese über Klassenpfadvariablen an (vgl. Abschnitt 3.3.2).
- **classpath**-Option bei Aufruf des Compilers bzw. Interpreters angeben
  Beispiel:

  ```
  >javac -cp ".;U:\Eigene Dateien\Java\Kurs\MyLib" PackDemo.java
  ```

Wie Sie bereits wissen, stehen nur die **public**-Klassen eines Paketes für paketfremde Klassen zur Verfügung.

6.2.2 Namensregeln

Für den Zugriff auf die Klassen eines fremden, von **java.lang** verschiedenen, Paketes 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(System.currentTimeMillis());
  ```

- **Import einer benötigten Klasse oder eines kompletten Paketes**
  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.currentTimeMillis());
      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 Variablen 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 Elemente 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) = "+Math.sin(Math.PI/2));
  }
}
```

  Mit Java 5 bietet folgende Alternative:

```
import static java.lang.Math.*;
class Prog {
  public static void main(String[] args) {
    System.out.println("Sin(Pi) = "+sin(PI/2));
  }
}
```

In folgendem Programm wird das Beispielpaket `demopack` samt Unterpaket `sub1` importiert:

Quellcode	Ausgabe
`import demopack.*;` `import demopack.sub1.*;` `class PackDemo {` `  public static void main(String[] args) {` `  A a = new A(), aa = new A();` `  a.prinr();` `  aa.prinr();` `  B b = new B();` `  b.prinr();` `  C c = new C();` `  c.prinr();` `  X x = new X();` `  x.prinr();` `  Y y = new Y();` `  y.prinr();` `  }` `}`	`Klasse A, Objekt Nr. 1` `Klasse A, Objekt Nr. 2` `Klasse B, Objekt Nr. 1` `Klasse C, Objekt Nr. 1` `Klasse X, Objekt Nr. 1` `Klasse Y, Objekt Nr. 1`

Das Programm ist auf folgende Weise übersetzt und ausgeführt worden (mit dem `demopack`-Paket im Ordner **U:\Eigene Dateien\Java\Kurs\MyLib**):

```
>javac -cp "U:\Eigene Dateien\Java\Kurs\MyLib" PackDemo.java
>java -cp ".;U:\Eigene Dateien\Java\Kurs\MyLib" PackDemo
```

6.3 Zugriffsschutz

Nach der Beschäftigung mit Paketen kann endlich präzise erläutert werden, wie in Java die für objektorientierte Softwareentwicklung außerordentlich wichtigen Zugriffsrechte für Klassen, Instanzvariablen und Methoden festgelegt werden.

6.3.1 Zugriffsschutz für Klassen

Die 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 6.2.2 vorgestellten Programms `PackDemo` mit folgender Meldung:

```
> 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.

6.3.2 Zugriffsschutz für Variablen und Methoden

Bei der Deklaration bzw. Definition von Variablen bzw. Methoden (objekt- oder klassenbezogen) können die Modifikatoren **private**, **protected** und **public** angegeben werden, um die Zugriffsrechte festzulegen. In der folgenden Tabelle wird die Wirkung bei einer Klasse beschrieben, die selbst als **public** definiert ist:

Modifikator	**Der Zugriff ist erlaubt für ...**			
	die eigene Klasse	**Klassen im Paket**	**abgeleitete Klassen**	**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 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` 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.

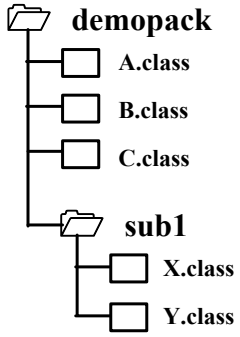
6.4 Java-Archivdateien

Wenn zu einem Programm zahlreiche **class**-Dateien und zusätzliche Hilfsdateien (z.B. mit Multimedia-Inhalten) gehören, dann sollten diese in einer Java-Archivdatei (Namenserweiterung **.jar**) zusammengefasst werden.
Einige Eigenschaften von Java-Archivdateien:

- 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).
- Eine Archivdatei kann analog zu einem Verzeichnis in den Suchpfad für Pakete und **class**-Dateien aufgenommen werden, z.B.:
  ```
  CLASSPATH = .;c:\Programme\DemoPack\demopack.jar
  ```
- Verkürzte Zugriffs- bzw. Übertragungszeiten
  Eine einzelne (am besten unfragmentierte) Archivdatei reduziert im Vergleich zu zahlreichen einzelnen Dateien die Zugriffszeiten beim Laden von Klassen und Ressourcen.
  Bei Java-Applets wird Übertragungszeit gespart, weil nicht für viele einzelne Dateien jeweils eine GET-Transaktion stattfinden muss.
- 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.
  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 unten) überlassen.
- Sicherheit
  Bei *signierten* JAR-Dateien kann sich der Anwender Gewissheit über den Urheber verschaffen und der Software entsprechende Rechte einräumen.

6.4.1 Archivdateien mit jar erstellen

Zum Erstellen und Verändern von Java-Archivdateien dient das JDK-Werkzeug **jar.exe**. Wir verwenden es dazu, aus dem `demopack`-Paket (siehe Abschnitt 6.1)



eine Archivdatei zu erzeugen. Dazu öffnen wir ein Konsolenfenster und wechseln zum Verzeichnis mit dem `demopack`-Ordner.
Dann wird mit folgendem **jar**-Aufruf das Archiv `demopack.jar` mit der gesamten Paket-Hierarchie erstellt:[23]

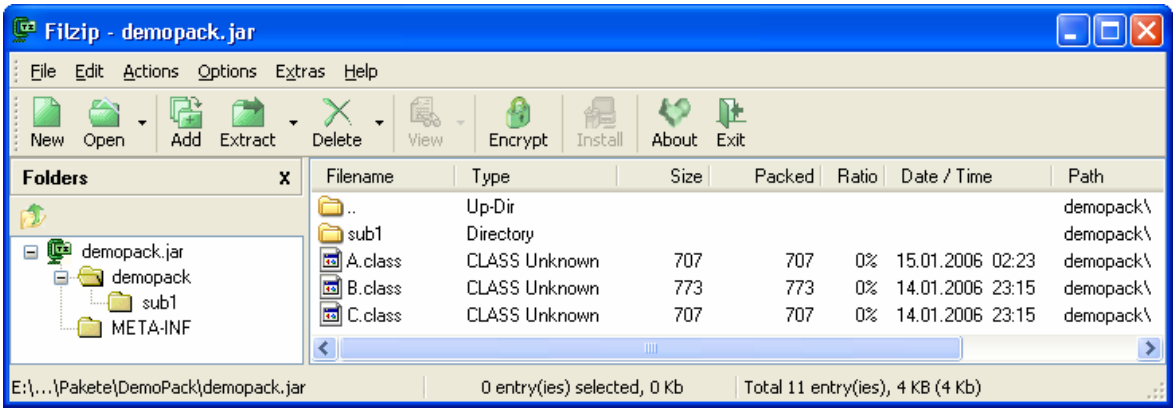
```
jar cf0 demopack.jar demopack
```

Darin bedeuten:

- 1. Parameter: Optionen
  Die Optionen bestehen aus jeweils einem Buchstaben und müssen unmittelbar hintereinander geschrieben werden:
  - **c**
    Mit einem **c** (für *create*) wird das Erstellen eines Archivs angefordert.
  - **f**
    Mit **f** (für *file*) wird die Ausgabe in eine Datei geleitet, wobei der Dateiname als zweiter Kommandozeilenparameter anzugeben ist.
  - **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**):

[23] Sollte der Aufruf nicht klappen, befindet sich vermutlich das JDK-Unterverzeichnis **bin** (z.B. C:\Programme\Java\jdk1.5.0_05\bin) nicht im Suchpfad für ausführbare Programme.



6.4.2 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 `demopack.jar` im Ordner **U:\Eigene Dateien\Java\Kurs\MyLib** 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\Kurs\MyLib\demopack.jar" PackDemo.java
>java -cp ".;U:\Eigene Dateien\Java\Kurs\MyLib\demopack.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 Archivdateien unter Eclipse ist das Setzen von Klassenpfadvariablen sehr zu empfehlen (siehe Abschnitt 3.3.2).

6.4.3 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 folgender Signatur besitzen:

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

Außerdem muss die Klasse im so genannten **Manifest** des Archivs, dem wir bisher keine Beachtung geschenkt haben, als **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`:

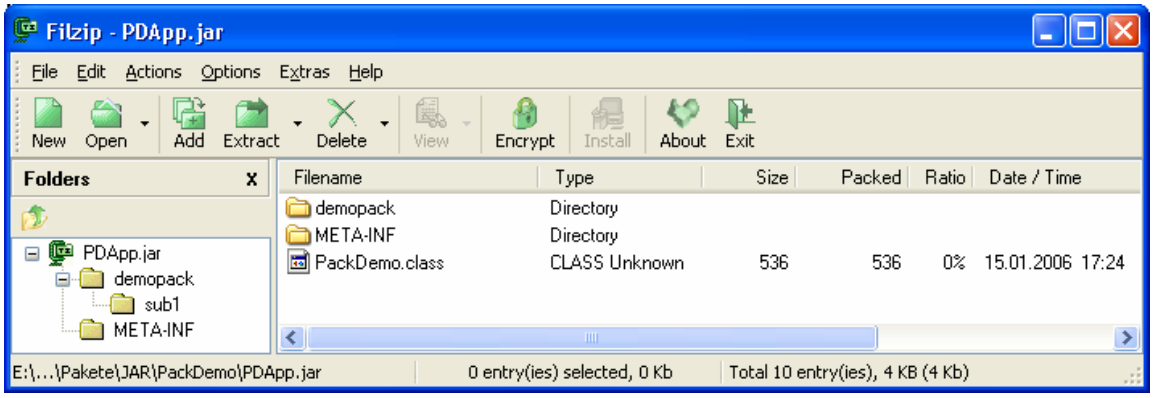
```
>jar cmf0 PDManifest.txt PDApp.jar PackDemo.class demopack
```

Beachten Sie bitte, 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.:

```
<DIR>          .
<DIR>          ..
<DIR>          demopack
           536 PackDemo.class
            22 PDManifest.txt
```

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 die Java-Laufzeitumgebung installiert sein.

Wenn mit dem Betriebssystem die Behandlung von **jar**-Dateien passend vereinbart wird, klappt der Start sogar per Doppelklick auf die Archivdatei.

Wenn Sie im **jar**-Aufruf ein Paket oder eine Klasse *mit Pfad* angeben, dann wird dieser Pfad in das Archiv übernommen, und ohne Klassensuchpfad im Manifest werden betroffene Klassen vom Laufzeitsystem nicht gefunden. So wird mit

```
>jar cmf0 PDManifest.txt PDApp.jar PackDemo.class ..\demopack
```

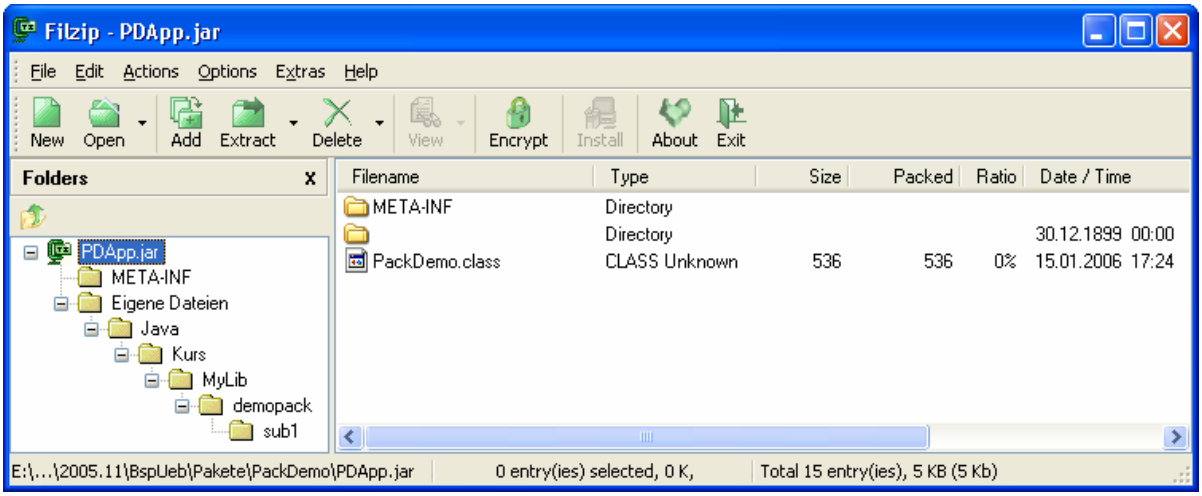
bei passender Verzeichnisorganisation zwar erfolgreich ein Archiv produziert, doch der anschließende Aufruf

```
>java -jar PDApp.jar
```

führt zur Fehlermeldung

```
Exception in thread "main" java.lang.NoClassDefFoundError: demopack/A
        at PackDemo.main(PackDemo.java:6)
```

Bei einer Inspektion der **jar**-Datei wird die Ursache klar:



Außerdem ist zu beachten, dass beim Starten einer JAR-Applikation außer den im Manifest angegebenen Klassenpfaden nur noch die JRE-Standardpfade bekannt sind. Die **CLASSPATH**-Umgebungsvariable ist ebenso wirkungslos wie ein **classpath**-Startparameter.

6.5 Das API der Java 2 Standard Edition

Zur Java-Plattform gehören zahlreiche Pakete, die Klassen für wichtige Aufgaben der Programmentwicklung (z.B. String-Verarbeitung, 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 2 Standard Edition (J2SE)**, auf die wir uns beschränken, bietet die Firma SUN noch weitere Java-APIs an, z.B.:

- Java 2 Enterprise Edition (J2EE)
- Java 2 Micro Edition (J2ME)

Nähere Informationen finden Sie z.B. auf der Webseite http://java.sun.com/.
In der JDK-Dokumentation zur Java 2 Standard Edition (J2SE) sind deren Pakete umfassend dokumentiert (siehe Abbildung in Abschnitt 5.1.4):

- Klicken Sie nach dem Start auf den Link **Java 2 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 Paketes.
  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 wird diese im Hauptrahmen ausführlich erläutert (z.B. mit einer Beschreibung der öffentlichen Methoden).

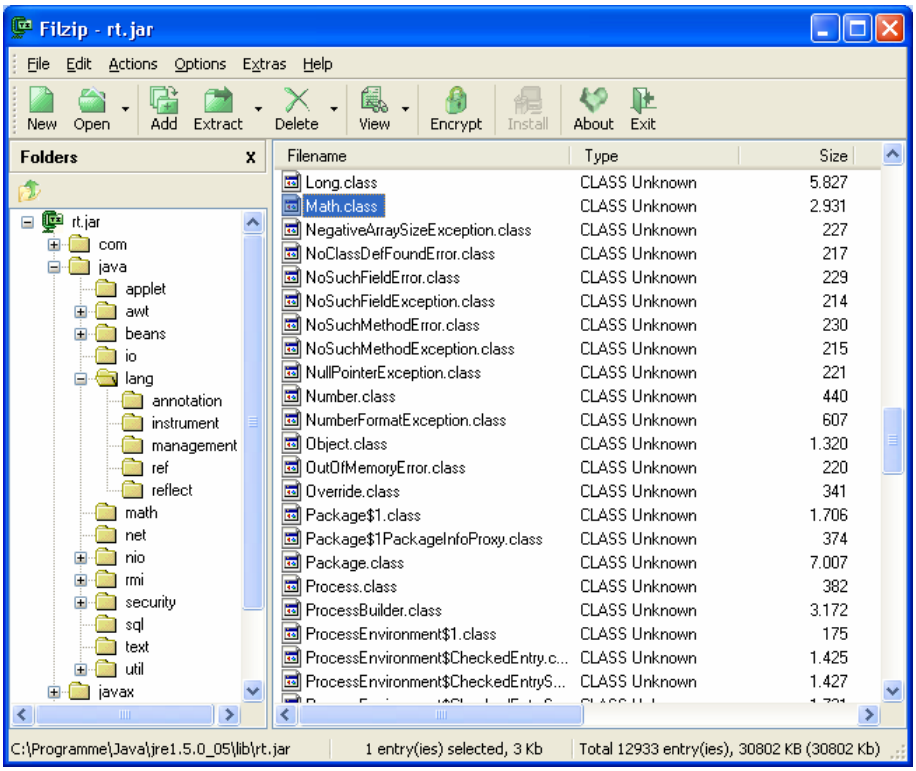
Sie haben vermutlich schon mehrfach von diesem Informationsangebot Gebrauch gemacht.

Die zum Java-API gehörigen Bytecode-Dateien sind auf mehrere Java-Archivdateien (*.jar) verteilt, die sich im Verzeichnis

...\jre\lib

befinden. Den größten Brocken bildet die Datei **rt.jar**. Hier befindet sich z.B. das Paket **java**, dessen Unterpaket **lang** z.B. die Bytecode-Datei zur Klasse **Math** enthält, deren Klassenmethoden wir schon mehrfach benutzt haben:[24]

[24] Zur Anzeige der Datei **rt.jar** wird hier das kostenlose Programm **Filzip** verwendet.



Es folgen kurze Beschreibungen wichtiger Pakete im API der Java 2 Standard Edition (Version 1.5):

- **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 das aktuelleren Swing-Paket bevorzugt (s. u.).
- **java.awt.event**
  Das AWT-Event-Paket enthält Klassen zur Ereignisverwaltung (siehe unten) für die Komponenten aus den Paketen **java.awt** und **javax.swing**.
- **java.beans**
  Das Paket enthält Klassen zum Programmieren von Java-Komponenten (**beans** genannt), welche in der Java-Welt eine ähnliche Rolle spielen wie die OCX- bzw. ActiveX-Komponenten in der Windows-Welt.
- **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 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 enthalten Klassen für die Netzwerkprogrammierung.

- **java.text**
  Hier geht es um die Formatierung von Textausgaben und die Internationalisierung von Programmen. Wir haben mit **DecimalFormat** bereits eine Klasse aus diesem Paket kennen gelernt.
- **java.util**
  Dieses Paket enthält neben Kollektions-Klassen (z.B. **Vector**) 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**.

Neben den Paketen im Kern-API bietet SUN auch noch etliche **Optional Packages** (früher als *Standard Extensions* bezeichnet) an, die ebenfalls kostenlos verfügbar sind, z.B.:

- Java Media Framework (JMF)
- Java 3D

6.6 Übungsaufgaben zu Abschnitt 6

1) Legen Sie das seit Abschnitt 6.1 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 6.2 wiedergegebenen Klasse `PackDemo`.

2) Wir haben früher in Mehrklassen-Anwendungen nie den Zugriffsmodifikator **public** eingesetzt, konnten aber trotzdem (z.B. in den „Hauptprogrammen“) Objekte anderer Klassen erzeugen und mit der Ausführung von Methoden beauftragen. Wieso war die **public**-Deklaration von Klassen und Methoden nicht erforderlich?

3) Erstellen Sie ein ausführbares Archiv mit unserem Währungskonvertierungsprogramm (DM in Euro). Einzupacken sind die Dateien:

- **DM2Euro.class**
  Im Ordner

  …\BspUeb\Elementare Sprachelemente\DM2EuroS

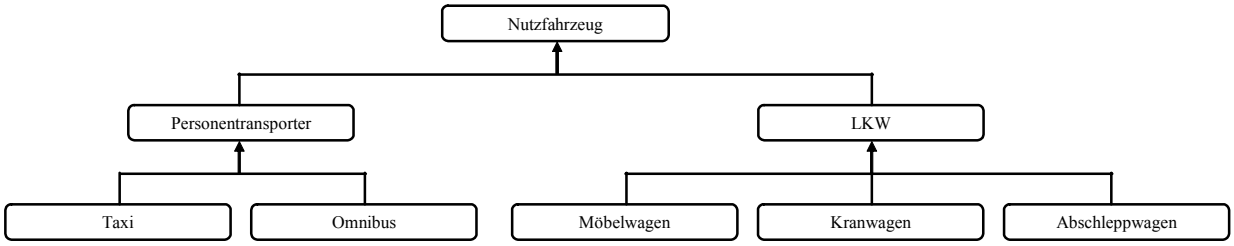
  finden Sie eine Variante, die dank Schleifenkonstruktion nicht für jeden Betrag neu gestartet werden muss.
- **Simput.class**
  `DM2EuroS` benutzt eine Klassenmethode von `Simput`. Sie finden die Datei **Simput.class** im Ordner:

  …\BspUeb\Simput

Außerdem wird noch eine Manifest-Datei benötigt, welche die auszuführende Klasse angibt (siehe Abschnitt 6.4.3).

7 Vererbung und Polymorphie

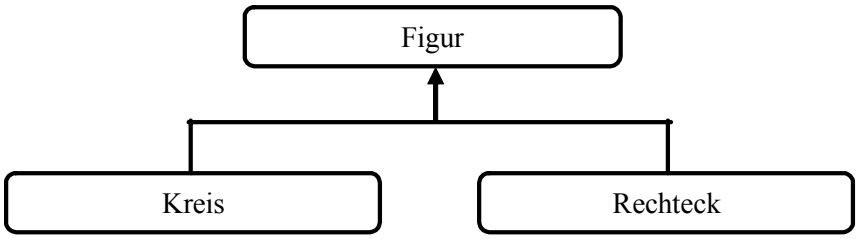
Beim Modellieren eines Gegenstandbereiches durch Klassen, die durch Eigenschaften (Instanz- und Klassenvariablen) und Handlungskompetenzen (Instanz- und Klassenmethoden) gekennzeichnet sind, müssen auch die Spezialisierungs- bzw. Generalisierungsbeziehungen zwischen real existierenden Klassen abgebildet werden. Eine Firma für Transportaufgaben aller Art mag die Nutzfahrzeuge in ihrem Fuhrpark 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.

Demonstrationsbeispiele

Bei unseren Demonstrationsbeispielen 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 Variablen und Methoden ihrer **Basis**- oder **Superklasse** und kann nach Bedarf Anpassungen bzw. Erweiterungen zur Lösung spezieller Aufgaben vornehmen:

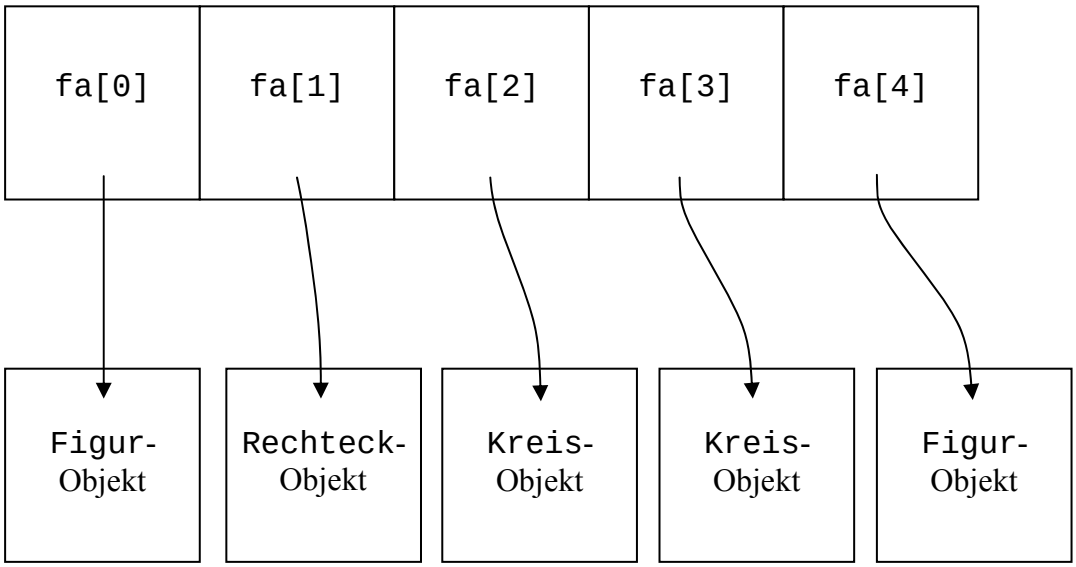
- zusätzliche Variablen deklarieren
- zusätzliche Methoden definieren
- geerbte Methoden überschreiben, d.h. unter Beibehaltung des Namens umgestalten
- geerbte Variablen überdecken, z.B. unter Beibehaltung des Namens den Datentyp ändern

Polymorphie

Die Vererbungstechnologie der OOP erlaubt nicht nur die bequeme Definition von Klassenhierarchien, sondern bietet auch eine spezielle Flexibilität: Über Variablen vom Basisklassentyp lassen sich auch Objekte aus einer abgeleiteten Klasse verwalten. Zwar können dabei nur Methoden aufgerufen werden, die schon in der Basisklasse bekannt sind, doch kommen ggf. die überschrei-

benden Realisationen der *abgeleiteten* Klasse zum Einsatz. Es ist also nicht ungewöhnlich, dass z.B. ein Array vom Basisklassentyp Objekte aus verschiedenen (abgeleiteten) Klassen aufnimmt, die sich beim Aufruf „derselben" Methode unterschiedlich verhalten.
Im Rahmen eines Grafikprogramms kommt vielleicht ein Array mit dem Elementtyp `Figur` zum Einsatz, dessen Elemente auf Objekte aus der Basisklasse oder aus einer abgeleiteten Klasse wie `Kreis` oder `Rechteck` zeigen:

Array `fa` mit Elemententyp `Figur`



Befragt man ein Objekt z.B. über die Methode `verdecktPunkt()`, ob es eine bestimmte Position überlagert, dann ermittelt es die Antwort je nach Gestalt auf unterschiedliche Weise.
Die Fähigkeit, mit Variablen vom Basisklassentyp auch Objekte aus abgeleiteten Klassen zu verwalten, bezeichnet man als **Polymorphie**.

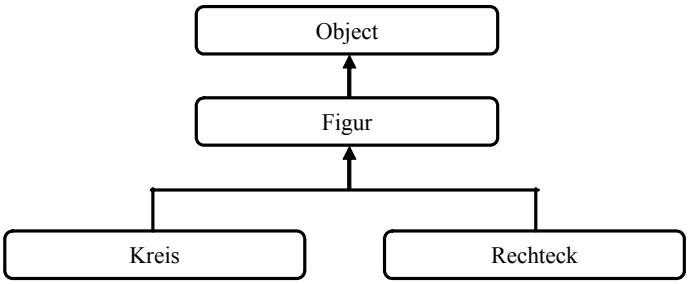
Software-Recycling

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 Superklasse kommen allen (direkt oder indirekt) abgeleiteten Klassen zu Gute.
Die noch weit verbreitete Praxis, vorhanden Code per *Cut & 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 bis in alle Ewigkeit von einer stetig wachsenden Programmierergemeinde eingesetzt wird.

Auch Polymorphie verbessert die Chancen für produktives Software-Recycling. Dank dieser OOP-Technik kann ein Programm auch Objekte aus solchen Klassen verarbeiten, die zum Zeitpunkt seiner Übersetzung noch gar nicht existierten.

Hierarchie der Java-Klassen

In Java stammen *alle* Klassen (sowohl die im Java-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(, falls sich keine bessere, bereits definierte Basisklasse findet):



7.1 Definition einer abgeleiteten Klasse

Im folgenden Programm wird die Klasse `Figur` definiert, die Instanzvariablen für die X- und die Y-Position der linken oberen Ecke einer zweidimensionalen Figur sowie zwei Konstruktoren besitzt:

```
public class Figur {
   int xpos = 100, ypos = 100;
   public Figur(int xpos_, int ypos_) {
      xpos = xpos_;
      ypos = ypos_;
      System.out.println("Figur-Konstruktor");
   }
   public Figur() {}
}
```

Die Klasse `Kreis` wird mit Hilfe des Schlüsselwortes

extends

als Spezialisierung der Klasse `Figur` definiert. Sie erbt die beiden Positionsvariablen und ergänzt eine zusätzliche Instanzvariable für den Radius:

```
public class Kreis extends Figur {
   int radius = 75;
   public Kreis(int xpos_, int ypos_, int rad_) {
      super(xpos_, ypos_);
      radius = rad_;
      System.out.println("Kreis-Konstruktor");
   }
   public Kreis() {}
}
```

Es wird ein initialisierender `Kreis`-Konstrukor definiert, der über das Schlüsselwort

super

den Konstruktor der übergeordneten Klasse aufruft, anstatt dessen Arbeit selbst zu erledigen. Das Schlüsselwort hat übrigens den oben eingeführten Begriff *Superklasse* motiviert. In Abschnitt 7.3 werden wir uns mit einigen Regeln für **super**-Konstruktoren beschäftigen.
Konstruktoren werden generell *nicht* vererbt, so dass in der `Kreis`-Klasse ein parameterfreier Konstruktor neu definiert wird, obwohl auch die Basisklasse einen solchen Konstruktor besitzt (natürlich mit anderem Namen!).
Das folgende Testprogramm erzeugt ein Objekt aus der Basisklasse und ein Objekt aus der abgeleiteten Klasse:

Quellcode	Ausgabe
`class Test {` `  public static void main(String[] args) {` `  Figur fig = new Figur(50, 50);` `  Kreis krs = new Kreis(10, 10, 5);` `  }` `}`	`Figur-Konstruktor` `Figur-Konstruktor` `Kreis-Konstruktor`

7.2 Der Zugriffsmodifikator protected

In folgender Variante des Beispielprogramms wird der Effekt des Zugriffsmodifikators **protected** demonstriert. Zunächst wird dafür gesorgt, dass die Klassen `Figur` und `Kreis` zu verschiedenen Paketen gehören, weil *innerhalb* eines Paketes die abgeleiteten Klassen dieselben großzügigen Zugriffsrechte haben wie beliebige andere Klassen.

```
package fipack;

public class Figur {
  protected int xpos=100, ypos=100;

  public Figur(int xpos_, int ypos_) {
    xpos = xpos_;
    ypos = ypos_;
    System.out.println("Figur-Konstruktor");
  }
  public Figur() {}
}
```

Weil die Basisklasse `Figur` die Instanzvariablen `xpos` und `ypos` als **protected** deklariert, können ihre Werte in der `Kreis`-Methode `setzePos()`verändert werden, obwohl `Kreis` nicht zum selben Paket gehört:

```
import fipack.*;

public class Kreis extends Figur {
  int radius = 75;
  public Kreis(int xpos_, int ypos_, int rad_) {
    super(xpos_, ypos_);
    radius = rad_;
    System.out.println("Kreis-Konstruktor");
  }

  public void setzePos(int xpos_, int ypos_) {
    xpos = xpos_;
    ypos = ypos_;
  }
}
```

Es ist zu beachten, dass hier eine *geerbte Instanzvariable* von `Kreis`-Objekten verändert wird. Auf die `xpos`-Eigenschaft von `Figur`-Objekten haben auch die Methoden der `Kreis`-Klasse *keinen* Zugriff.
Während Objekte aus abgeleiteten Klassen ihre geerbten **protected**-Elemente direkt ansprechen können, haben fremde Klassen auf Elemente mit dieser Schutzstufe grundsätzlich keinen Zugriff:

```
class Test {
  public static void main(String[] args) {
         Kreis  k1 = new Kreis(50, 50, 10);
//verboten:   k1.xpos = 77;
         k1.setzePos(77, 99);            //so klappts
  }
}
```

7.3 super-Konstruktoren und Initialisierungs-Sequenzen

Abgeleitete Klassen erben die Basisklassen-Konstruktoren *nicht*, können sie aber in eigenen Konstruktoren über das Schlüsselwort **super** benutzen. Dadurch wird z.B. es möglich, geerbte Instanzvariablen zu initialisieren, die in der Basisklasse als **private** deklariert wurden.
Wird in einem Konstruktor einer abgeleiteten Klasse kein Basisklassen-Konstruktor 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 Basisklassen-Konstruktor aufrufen. Dieser Aufruf muss am Anfang des Konstruktors stehen.
- In der Basisklasse einen parameterfreien Konstruktor definieren.

Der parameterfreie Basisklassen-Konstruktor wird übrigens auch vom Standard-Konstruktor der abgeleiteten Klasse aufgerufen.

Beim Erzeugen eines Unterklassen-Objektes laufen damit folgende Initialisierungs-Maßnahmen ab (vgl. Gosling et al. 2005, Abschnitt 12.5):

- Alle Instanzvariablen werden mit den typspezifischen Nullwerten initialisiert.
- Der Unterklassen-Konstruktor beginnt seine Tätigkeit mit dem (impliziten oder expliziten) Aufruf eines Basisklassen-Konstruktors. Falls in der Vererbungshierarchie der Urahn **Object** noch nicht erreicht ist, wird am Anfang des Basisklassen-Konstruktors wiederum ein Konstruktor des nächst höheren Vorfahren 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 beim Erzeugen eines `Kreis`-Objektes mit dem Konstruktor-Aufruf `Kreis(150, 200, 50)`:

- Die Instanzvariablen von `Kreis`-Objekten (auch die geerbten) werden angelegt und mit Nullen initialisiert.
- Aktionen für die Klasse **Object**:
  - Die Instanzvariablen der Klasse **Object** werden initialisiert.
    Derzeit sind mir zwar keine **Object**-Instanzvariablen bekannt, doch ist die Existenz von gekapselten Exemplaren durchaus möglich.
  - Der Rumpf des **Object**-Konstruktors wird ausgeführt.
    Weil der `Figur`-Konstruktor keinen expliziten Basisklassen-Konstruktoraufruf vornimmt, kommt der parameterfreie **Object**-Konstruktor zum Einsatz.
- Aktionen für die Klasse `Figur`:

  - Die Instanzvariablen `xpos` und `ypos` erhalten den Initialisierungswert laut Deklaration (jeweils 100).
  - Der Rumpf des Konstruktoraufrufs `Figur(150, 200)` wird ausgeführt, wobei `xpos` und `ypos` die Werte 150 bzw. 200 erhalten.
- Aktionen für die Klasse `Kreis`:
  - Die Instanzvariable `radius` erhält den Initialisierungswert 75 aus der Deklaration.
  - Der Rumpf des Konstruktoraufrufs `Kreis(150, 200, 50)` wird ausgeführt, wobei `radius` den Wert 50 erhält.

7.4 Überschreiben und Überdecken

7.4.1 Methoden überschreiben

Eine Basisklassen-Methode darf in einer Unterklasse durch eine Methode mit gleichem Namen und gleicher Parameterliste überschrieben werden, die dann 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*.
Unsere `Figur`-Basisklasse wird um eine Methode `wo()` erweitert, welche die Position der linken oberen Ecke ausgibt:

```
public class Figur {
  protected int xpos = 100, ypos = 100;
  public Figur(int xpos_, int ypos_) {
    xpos = xpos_;
    ypos = ypos_;
  }
  public Figur() {}
  public void wo() {
    System.out.println("\nOben Links: (" + xpos + ", " + ypos + ") ");
  }
}
```

In der `Kreis`-Klasse kann eine bessere Ortsangaben-Methode realisiert werden, weil hier auch die rechte untere Ecke definiert ist[25]:

```
public class Kreis extends Figur {
  int radius = 75;
  public Kreis(int xpos_, int ypos_, int rad_) {
    super(xpos_, ypos_);
    radius = rad_;
  }
  public Kreis() {}
  public void wo() {
    super.wo();
    System.out.println("Unten Rechts: (" + (xpos+2*radius) +
                       ", " + (ypos+2*radius) + ")");
  }
}
```

[25] 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 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 artgerechtes Verhalten:

Quellcode	Ausgabe
`class Test {` `  public static void main(String[] ars) {` `  Figur f = new Figur(10, 20);` `  f.wo();` `  Kreis  k = new Kreis(50, 100, 25);` `  k.wo();` `  }` `}`	`Oben Links: (10, 20)` `Oben Links: (50, 100)` `Unten Rechts: (100, 150)`

Auch bei den vom Urahn **Object** geerbten Methoden kommt ein Überschreiben in Frage. Die **Object**-Methode **toString()** liefert neben dem Klassennamen eine kodierte Objekt-Identität. Sie wird z.B. von der **String**-Methode **println()** automatisch genutzt, um die Zeichenketten-Repräsentation zu einer Objektreferenz zu ermitteln:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  Prog tst1 = new Prog();` `  Prog tst2 = new Prog();` `  System.out.println(tst1);` `  System.out.println(tst2);` `  }` `}`	`Prog@16f0472` `Prog@18d107f`

In der folgenden Klasse `Mint` (ein **int**-Wrapper) wird die **Object**-Methode **toString()** überschrieben:

```
public class Mint {
    private int val;
    public Mint(int val_) {
        val = val_;
    }
    public String toString() {
      return String.valueOf(val);
    }
}
```

Ein `Mint`-Objekt antwortet auf die **toString**()-Botschaft mit der Zeichenfolgendarstellung seines gekapselten Wertes:

Quellcode	Ausgabe
`class Test {` `  public static void main(String[] args) {` `  Mint zahl = new Mint(4711);` `  System.out.println(zahl);` `  }` `}`	`4711`

7.4.2 Finalisierte Methoden und Klassen

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 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 (Polymorphie!), steht eine **final**-Methode schon beim Übersetzen fest.

Die eben für das Finalisieren von *Methoden* genannten Sicherheitsüberlegungen können auch zum Entschluss führen, eine komplette Klasse mit dem Schlüsselwort **final** als *endgültig* zu deklarieren, so dass sie zwar verwendet (z.B. instantiiert), aber nicht mehr beerbt werden kann.

Finalisierte Klassen eignen sich auch zum Aufbewahren von Konstanten, z.B.:

```
package maipack;
final public class Konst {
  public final static int MAX = 333;
  public final static int MORITZ = 500;
  private Konst(){}
}
```

Um das unsinnige Erzeugen von Objekten zu einer Konstantenklasse zu verhindern, sollte man den Konstruktor als **private** definieren.
Das folgende Programm benutzt die eben definierte Klasse `Konst` aus dem Paket `maipack`:

Quellcode	Ausgabe
`import maipack.Konst;` `class Test {` `  public static void main(String[] args) {` `  System.out.println(Konst.MORITZ);` `  }` `}`	`500`

7.4.3 Instanzvariablen überdecken

Wird in der abgeleiteten Klasse `Sohn` für eine Instanzvariable ein Name verwendet, der bereits eine Variable der beerbten Klasse `Vater` bezeichnet, dann wird die Basisvariable überdeckt. Sie ist jedoch weiterhin vorhanden und kommt in folgenden Situationen zum Einsatz:

- Vom `Vater` geerbte Methoden greifen weiterhin auf die `Vater`-Variable zu, während die zusätzlichen Methoden der `Sohn`-Klasse auf die `Sohn`-Variable zugreifen.
- Analog zu einer überschriebenen Methode kann die überdeckte Variable mit Hilfe des Schlüsselwortes **super** weiterhin benutzt werden.

Im folgenden Beispielprogramm führt ein `Sohn`-Objekt eine `Vater`- und eine `Sohn`-Methode aus, um die beschriebenen Zugriffsvarianten zu demonstrieren:

Quellcode	Ausgabe
`// Datei Vater.java` `class Vater {` `  String x = "Vast";` `  void vm() {` `    System.out.println("x in Vater-Methode:    "+x);` `  }` `}` `// Datei Sohn.java` `class Sohn extends Vater {` `  int x = 333;` `  void sm() {` `    System.out.println("x in Sohn-Methode:     "+x);` `    System.out.println("super-x in Sohn-Meth.: "+` `                       super.x);` `  }` `}` `// Datei Test.java` `class Test {` `  public static void main(String[] args) {` `    Sohn so = new Sohn();` `    so.vm();` `    so.sm();` `  }` `}`	`x in Vater-Methode:    Vast` `x in Sohn-Methode:     333` `super-x in Sohn-Meth.: Vast`

7.5 Polymorphie

Referenzvariablen sind in Java wie in den meisten anderen objektorientierten Sprachen *polymorph*, d.h. sie können auch auf Objekte aus abgeleiteten KLassen zeigen. Dies ermöglicht es, Basisklassen-Referenzvariablen zur flexiblen Verwaltung von Objekten aus beliebigen Unterklassen zu verwenden.
Gegen die unvermeidlichen Gewöhnungsprobleme beim Umgang mit diesem Konzept hilft nur praktische Erfahrung. In welchem Ausmaß durch Polymorphie die Programmierpraxis erleichtert wird, kann leider durch die notwendigerweise kurzen Demonstrationsbeispiele nur ansatzweise vermittelt werden.
Wir definieren zunächst eine Klasse namens `Vater`:

```
class Vater {
  int iv = 1;
  void hallo() {
    System.out.println("hallo-Methode des Vaters");
  }
}
```

Daraus leiten wir eine Klasse namens `Sohn` ab, welche die väterliche `hallo()`-Methode überschreibt und zusätzliche Elemente ergänzt:

- die Instanzvariable `is`
- einen parameterfreien Konstruktor, der die geerbte Instanzvariable `iv` auf den Wert 2 setzt.
- die Methode `nurso()`

Weil die aktuellen Beispielklassen sicher nur im Rahmen ihres Standardpaketes genutzt werden, verzichten wir auf Zugriffsmodifikatoren:

```
class Sohn extends Vater {
  int is = 3;
  Sohn() {iv = 2;}
  void hallo() {
    System.out.println("hallo-Methode des Sohnes");
  }
  void nurso() {
    System.out.println("nurso-Methode des Sohnes");
  }
}
```

Ein Array vom Typ `Vater[]` kann Referenzen auf Väter und Söhne aufnehmen, wie das folgende Beispielprogramm zeigt:

```
class Test {
  public static void main(String[] args) {
    Vater[] varray = new Vater[3];
    varray[0] = new Vater();
    varray[1] = new Sohn();
    System.out.println("iv-Wert von varray[1]: "
      + varray[1].iv);
    varray[0].hallo();
    varray[1].hallo();

    System.out.print("\nWollen Sie zum Abschluss noch einen"+
     " Vater oder einen Sohn erleben?"+
     "\nWaehlen Sie durch Abschicken von \"v\" oder \"s\": ");
    if (Character.toUpperCase(Simput.gchar()) == 'V')
      varray[2] = new Vater();
    else
      varray[2] = new Sohn();
    System.out.println();
    varray[2].hallo();
  }
}
```

Beim Ausführen der `hallo()`-Methode, stellt das Laufzeitsystem die tatsächliche Klassenzugehörigkeit fest und wählt die passende Methode aus (späte bzw. dynamische Bindung):

```
iv-Wert von varray[1]: 2
hallo-Methode des Vaters
hallo-Methode des Sohnes

Wollen Sie zum Abschluss noch einen Vater oder einen Sohn erleben?
Waehlen Sie durch Abschicken von "v" oder "s": s

hallo-Methode des Sohnes
```

Zum „Beweis“, dass der Compiler die Klassenzugehörigkeit nicht kennen muss, darf im Beispielprogramm die Klasse des Array-Elementes `varray[2]` vom Benutzer festgelegt werden.

Über eine `Vater`-Referenzvariable, die auf ein `Sohn`-Objekt zeigt, sind Erweiterungen der `Sohn`-klasse (zusätzliche Elementvariablen und Methoden) nicht unmittelbar zugänglich. Wenn (auf eigene Verantwortung des Programmierers) eine Basisklassen-Referenz als Unterklassen-Referenz behandelt werden soll, um eine unterklassenspezifische Methode oder Variable anzusprechen, dann muss der **cast**-Operator verwendet werden, z.B.:

```
System.out.println(((Sohn) varray[2]).is);
```

Im Zweifelsfall sollte man sich über den **instanceof**-Operator vergewissern, ob das referenzierte Objekt tatsächlich zur vermuteten Klasse gehört.

```
if (varray[2] instanceof Sohn)
  ((Sohn) varray[1]).nurso();
```

7.6 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, kann man die Methode dort als **abstract** deklarieren und auf eine Implementierung verzichten, z.B.:

```
abstract class Vater {
  abstract void hallo();
}
```

Enthält eine Klasse mindestens eine abstrakte Methode, dann handelt es sich um eine **abstrakte Klasse**, und die Klassendefinition muss mit dem Modifikator **abstract** eingeleitet 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 dem abstrakten `Vater` zwei „konkrete“ Klassen abgeleitet:

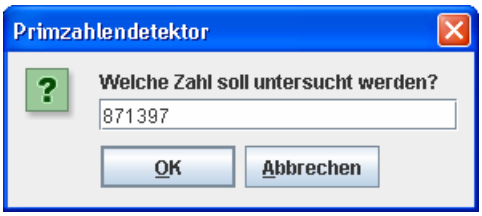
```
class Tochter extends Vater {
  void hallo() {
    System.out.println("hallo-Methode der Tochter");
  }
}
class Sohn extends Vater {
  void hallo() {
    System.out.println("hallo-Methode des Sohnes");
  }
}
```

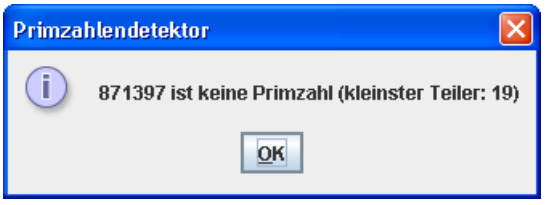
Aus einer abstrakten Klasse lassen sich zwar keine Objekte erzeugen, doch kann man sie als Datentyp verwenden. Referenzen dieses Typs sind ja auch unverzichtbar, wenn Objekte diverser Unterklassen polymorph verwaltet werden sollen:

Quellcode	Ausgabe
`class Test {` `  public static void main(String[] args) {` `  Vater[] varray = new Vater[2];` `  varray[0] = new Tochter();` `  varray[1] = new Sohn();` `  varray[0].hallo();` `  varray[1].hallo();` `  }` `}`	`hallo-Methode der Tochter` `hallo-Methode des Sohnes`

7.7 Entspannungs- und Motivationseinschub: GUI-Standarddialoge

Nach etlichen recht anstrengenden Themen, deren Praxisrelevanz an den notwendigerweise kleinen Beispielprogrammen nicht immer sehr deutlich wurde, soll in diesem Abschnitt zur Entspannung und zur Regeneration Ihrer Motivation eine angenehm einfache Möglichkeit zur Interaktion mit dem Benutzer über die von der Klasse **JOptionPane** realisierten Standarddialoge vorgestellt werden.
Wir greifen auf ein früheres Beispielprogramm zur Primzahldiagnose zurück und rüsten eine graphische Benutzerschnittstelle nach:





Mit statischen Methoden der Klasse **JOptionPane** aus dem Namensraum **javax.swing** ist es sehr einfach, Standarddialoge zu erzeugen, um Nachrichten auszugeben oder Informationen abzufragen. Im folgenden Programm wird per **showInputDialog()** ein Argument für die Primzahlendiagnose erfragt. Wir erhalten von der Methode als Rückgabewert eine Zeichenfolge und lassen diese von der statischen **Long**-Methode **parseLong()** in einen **long**-Wert wandeln:

```
import javax.swing.JOptionPane;
class PrimitivGui {
   public static void main(String[] args) {
      String s;
      boolean tg;
      long i, mpk, argumemt;
      while (true) {
         argumemt = Long.parseLong(
               JOptionPane.showInputDialog(null, "Welche Zahl soll untersucht werden?",
                  "Primzahlendetektor", JOptionPane.QUESTION_MESSAGE));
         tg = false;
         mpk = (int) Math.sqrt(argumemt);  //maximaler Primzahlenkandidat
         for (i = 2; i <= mpk; 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);
      }
   }
}
```

Weil der Klassenname **JOptionPane** an mehreren Stellen auftaucht, wird er am Anfang des Programms importiert.
Die verwendete **showInputDialog()**-Überladung kennt folgende Parameter:

Typ	Name	Erläuterung
Component	parentComponent	Standarddialoge werden oft von einem Anwendungs- bzw. Rahmenfenster (siehe unten) aufgerufen. Die Angabe einer Elternkomponente (an Stelle der Alternative **null**) hat zur Folge, dass die Dialogbox in der Nähe ihrer Elternkomponente erscheint.
Object	message	die auszugebende Nachricht bzw. Frage
String	title	der Text für die Fensterzeile der Dialogbox
int	messageType	Dieser Parameter legt den Typ der Nachricht fest, der wiederum über das Icon am linken Rand der Dialogbox entscheidet. Als Werte sind die folgenden Konstanten der Klasse **JOptionPane** (Datentyp **int**) erlaubt: • ERROR_MESSAGE • INFORMATION_MESSAGE • WARNING_MESSAGE • QUESTION_MESSAGE • PLAIN_MESSAGE

Dieselben Parameter finden sich auch bei der Methode **showMessageDialog()**, die im Beispielprogramm das Ergebnis präsentiert.

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

```
>java PrimitivGui
Exception in thread "main" java.lang.NumberFormatException: null
        at java.lang.Long.parseLong(Unknown Source)
        at java.lang.Long.parseLong(Unknown Source)
        at PrimitivGui.main(PrimitivGui.java:8)
```

Sie werden bald erfahren, wie man solche Ausnahmen abfangen und die Zwangsbeendigung des Programms verhindern kann.

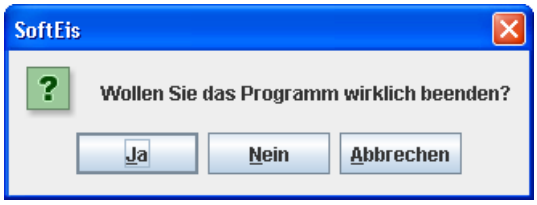
Startet man den Primzahlendetektors konsolenfrei (mit dem JRE-Werkzeug **javaw.exe**), bemerkt der Benutzer wenig von der fehlenden Ausnahmebehandlung:

```
>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.:

```
import javax.swing.JOptionPane;
class Prog {
  public static void main(String[] args) {
    while (true)
      if (JOptionPane.showConfirmDialog(null,
                  "Wollen Sie das Programm wirklich beenden?",
                  "SoftEis", JOptionPane.YES_NO_CANCEL_OPTION) ==
          JOptionPane.YES_OPTION)
        System.exit(0);
  }
}
```

Über den **optionType**-Parameter wählt man die angebotenen Schaltflächen:



Als Rückgabewerte liefert die Methode:

Vom Benutzer gewählter Schalter	**JOptionPane**-Konstante	**int**-Wert
Schließkreuz in der Titelzeile	CLOSED_OPTION	-1
Ja	YES_OPTION	0
Nein	NO_OPTION	1
Cancel	CANCEL_OPTION	2

7.8 Übungsaufgaben zu Abschnitt 7

1) Erweitern Sie das Beispiel mit den geometrischen Figuren (siehe Abschnitt 7.1) um eine aus `Figur` abgeleitete Klasse `Rechteck` mit zusätzlichen Instanzvariablen für Breite und Höhe und einer `wo()`-Methode, welche die geerbte `Figur`-Version überschreibt. Erstellen Sie ein Testprogramm, das polymorphe Objekt-Verwaltung und entsprechende Methodenaufrufe demonstriert.

2) Warum kann der folgende Quellcode nicht übersetzt werden?

```
class Vater {
   int iv;
   Vater(int iv_) {
      iv = iv_;
   }
   void hallo() {
      System.out.println("hallo-Methode des Vaters");
   }
}
class Sohn extends Vater {
   int is = 3;
   void hallo() {
      System.out.println("hallo-Methode des Sohnes");
   }
}
```

3) Im folgenden Beispiel wird die Klasse `Kreis` aus der Klasse `Figur` abgeleitet:

```
// Datei Figur.java
package fipack;
public class Figur {
   int xpos, ypos;
}
// Datei Kreis.java
import fipack.*;
class Kreis extends Figur {
   int radius;
   Kreis(int xpos_, int ypos_, int rad_) {
      xpos = xpos_;
      ypos = ypos_;
      radius = rad_;
   }
}
```

Trotzdem erlaubt der JDK-Compiler dem `Kreis`-Konstruktor keinen Zugriff auf die geerbten Instanzvariablen `xpos` und `ypos` eines neuen `Kreis`-Objekts:

```
Kreis.java:6: xpos is not public in fipack.Figur; cannot be accessed from outside package
                xpos = xpos_;
                ^
Kreis.java:7: ypos is not public in fipack.Figur; cannot be accessed from outside package
                ypos = ypos_;
                ^
```

Wie ist das Problem zu erklären und zu lösen?

8 Ausnahmebehandlung

Zu Störungen im Programmablauf können neben Designfehlern (z.B. versuchter Feldzugriff mit falschem Indexwert) auch besondere Umstände außerhalb des Programms führen (z.B. Speichermangel, unterbrochene Netzverbindung).
Java unterstützt das Design von robusten Programmen, die auf besondere Umstände intelligent reagieren, durch eine ausgefeilte Technik zur Ausnahmebehandlung.
Man kann von keinem Programm erwarten, dass es trotz widriger Umstände normal funktioniert. Doch müssen Datenverluste soweit als möglich verhindert werden, doch der Benutzer sollte eine nützliche Information erhalten.

8.1 Unbehandelte Ausnahmen

Verhindert ein Problem die normale Ausführung einer Methode, dann wird eine so genannte **Ausnahme** (engl.: **exception**) „ausgelöst" bzw. „geworfen". Wir wollen etwas näher betrachten, was sich dabei abspielt: Eine Ausnahme ist in Java ein Objekt aus der Klasse **Exception** oder aus einer problemspezifischen Unterklasse, das Informationen über den aufgetretenen Fehler enthält und über Instanzmethoden zugänglich macht. Tritt bei der Ausführung einer Methode ein Fehler auf, dann wird ein **Exception**-Objekt erzeugt, und das Laufzeitsystem sucht nach einer Routine der aufrufenden Methode, die sich um das Problem kümmern kann. Wird eine solche Routine gefunden, erhält sie eine Referenz auf das Ausnahme-Objekt und kann dessen Methoden nutzen, um Informationen mit Relevanz für das weitere Vorgehen zu gewinnen. Existiert keine zum Ausnahme-Objekt passende Behandlungsroutine, untersucht das Laufzeitsystem entlang der Aufrufgeschichte die jeweils nächst höhere Methode. Findet sich letztlich auch in der **main()**-Methode keine Behandlungsroutine, dann erzeugt das Laufzeitsystem eine Fehlermeldung und beendet das Programm.

Die Initiative beim Auslösen einer Ausnahme kann ausgehen

- vom **Laufzeitsystem**
  Es löst etwa bei arithmetischen Fehlern (z.B. Division durch 0) eine **RuntimeException** aus.
- vom **Programm**, wozu auch die verwendeten Bibliotheks-Klassen gehören
  In jeder Methode kann mit der **throw**-Anweisung (siehe Abschnitt 8.5) eine Ausnahme erzeugt werden. Dies stellt eine effiziente Technik dar, den Aufrufer über ein Problem zu informieren und mit relevanten Informationen zu versorgen.

Wir werden uns anhand der verschiedenen Versionen eines Beispielprogramms damit beschäftigen,

- was bei unbehandelten Ausnahmen geschieht,
- wie man Ausnahmen abfängt,
- wie man selbst Ausnahmen wirft.

Das folgende Programm soll die Fakultät zu einer Zahl ausrechnen, die beim Start als Kommandozeilenparameter übergeben wird. Dabei beschränkt sich die **main()**-Methode auf die eigentliche Fakultätsberechnung und überlässt die Konvertierung und Validierung des übergebenen Strings der Methode `kon2int()`. Diese wiederum stützt sich bei der Konvertierung auf die Methode **Integer.parseInt()**:

```
class Fakul {
  static int kon2int(String instr) {
    int arg = Integer.parseInt(instr);
    if (arg < 0  || arg > 170) {
      System.out.println("Unzulaessiges Argument: "+arg);
      System.exit(1);
      return 0;
    }
    else
      return arg;
  }

  public static void main(String[] args) {
    int argument = -1;

    if (args.length > 0)
      argument = kon2int(args[0]);
    else {
      System.out.println("Kein Argument angegeben");
      System.exit(1);
    }

    double fakul = 1.0;
    for (int i = 1; i <= argument; i++)
        fakul = fakul * i;
    System.out.println("Fakultaet: " + fakul);
  }
}
```

Das Programm ist durchaus bemüht, kritische Situationen zu vermeiden:

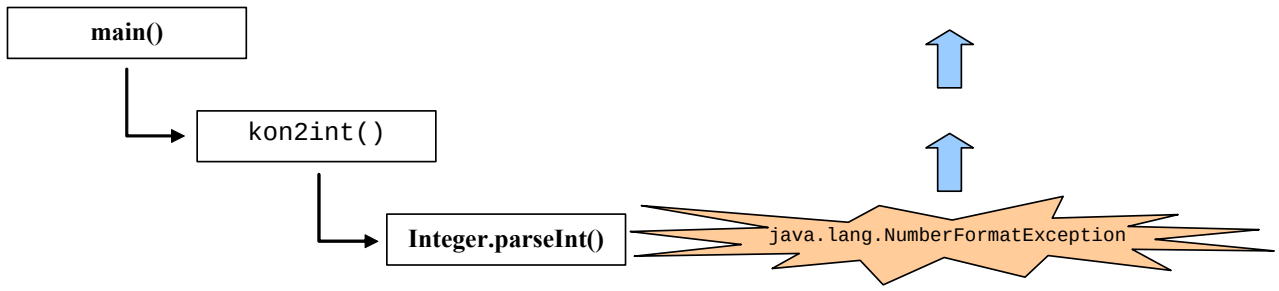
- **main()** überprüft, ob `args[0]` tatsächlich vorhanden ist, bevor dieser String beim Aufruf der Methode `kon2int()` als Parameter verwendet wird. Damit wird verhindert, dass es zu einer **ArrayIndexOutOfBoundsException** kommt, wenn der Benutzer das Programm ohne Kommandozeilen-Argument startet.
- `kon2int()` überprüft, ob die ermittelte **int**-Zahl im zulässigen Wertebereich liegt. Damit werden unsinnige Ergebnisse verhindert (z.B. Fakultät(-4) = 1).

Beide Methoden reagieren auf Probleme mit dem sofortigen Beenden des Programms. Dazu rufen sie die Methode **System.exit()** auf, der als Aktualparameter ein **Exitcode** übergeben wird. Diese Vorgehensweise kann als klassische Technik zur Ausnahmebehandlung betrachtet werden, die bei groben Problemen nach wie vor in Frage kommt.
Der an **System.exit()** übergebene Aktualparameter wird dem Betriebssystem mitgeteilt und steht unter Windows in der Umgebungsvariablen ERRORLEVEL zur Verfügung, z.B.:

```
>java Fakul
Kein Argument angegeben

>echo %ERRORLEVEL%
1
```

Trotz der 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[26] **Integer.parseInt()** wirft daraufhin eine **NumberFormatException**. Diese wird vom Laufzeitsystem entlang der Aufrufreihenfolge an `kon2int()` und dann an **main()** gemeldet:

[26] Aufrufverschachtelungen *innerhalb* der Java-Klassenbibliothek ignorieren wir an dieser Stelle. In Abschnitt 8.2.4 wird die Angelegenheit mit Hilfe des API-Quellcodes genauer untersucht.



Weil beide Methoden keine Behandlungsroutine bereithalten, endet das Programm mit folgender Fehlermeldung:

```
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:447)
      at java.lang.Integer.parseInt(Integer.java:497)
      at Fakul.kon2int(Fakul.java:3)
      at Fakul.main(Fakul.java:17)
```

8.2 Ausnahmen abfangen

8.2.1 Die try-Anweisung

In Java wird die Behandlung von Ausnahmen über die **try**-Anweisung unterstützt:

try {
    Überwachter Block mit Anweisungen für den normalen Programmablauf
}
catch (*Ausnahmeklasse Parametername***) {**
    Anweisungen für die Behandlung von Ausnahmen der ersten Klasse
}
// Optional können weitere Ausnahmen abgefangen werden:
catch (*Ausnahmeklasse Parametername***) {**
    Anweisungen für die Behandlung von Ausnahmen der zweiten Klasse
}

    . . .

// Optionaler Block mit Abschlussarbeiten.
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.

Wird im **try**-Block eine Ausnahme ausgelöst, wird seine Ausführung abgebrochen und das Laufzeitsystem sucht nach einer **catch**-Klausel, welche eine Ausnahme aus dieser Klasse behandeln kann.
Jede **catch**-Klausel verfügt in formaler Analogie zu einer Methode in ihrem Kopfbereich über eine einelementige Parameterliste mit einem formalen Referenzparameter. Das Laufzeitsystem sucht nach einer **catch**-Klausel, deren Formalparameter vom Typ der zu behandelnden Ausnahme ist, und führt dann den zugehörigen Anweisungsblock aus.
catch-Blöcke werden oft auch als *Exception Handler* bezeichnet.

In der folgenden Variante der Methode `kon2int()` wird eine von **Integer.parseInt()** ausgelöste Exception abgefangen:

```
static int kon2int(String instr) {
  int arg;
  try {
    arg = Integer.parseInt(instr);
  }
  catch (NumberFormatException e) {
    System.out.println("Fehler beim Konvertieren von \""+instr+"\"");
    System.exit(1);
    return 0;
  }

  if (arg < 0  || arg > 170) {
    System.out.println("Unzulaessiges Argument: "+arg);
    System.exit(1);
    return 0;
  }
  else
    return arg;
}
```

Beim Programmstart mit Kommandozeilenparameter „vier“ kommt es nun zu keinem Laufzeitfehler, sondern das Programm beendet seine Arbeit mit der informativen Meldung:

```
Fehler beim Konvertieren von "vier"
```

Nach Ausführung eines **catch**-Blocks wird das Programm hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**–Block ausgeführt wird. Im Beispiel kommt es allerdings nicht dazu, weil der **catch**-Block das Programm per **System.exit()** beendet.

Am Ende der **try**-Anweisung können in einem **finally**-Block solche Anweisungen stehen, die auf jeden Fall ausgeführt werden sollen, ob Ausnahmen aufgetreten sind oder nicht. Dies ist z.B. der ideale Platz, um die im **try**-Block geöffneten Dateien wieder zu schließen (siehe unten).

Gleich folgen ausführlichere Erläuterungen zum Programmablauf bei verschiedenen Ausnahme-Konstellationen.

8.2.2 Programmablauf bei der Ausnahmebehandlung

Findet das Laufzeitsystem für eine Ausnahme in der aktuellen Methode keinen **catch**-Block, dann sucht es entlang der Aufrufsequenz 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 Ereignisse 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 ausgeführt:

```
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 erhalten wir folgende 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()
```

Zum wenig überzeugenden Resultat

```
10 % acht = 0
```

kommt es, weil die **NumberFormatException** in `calc()` nicht *sinnvoll* behandelt wird. Wir werden bald elegante Möglichkeiten kennen lernen, derartige Pannen zu vermeiden. 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.
Im Aufrufer **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
        at Except.main(Except.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.

8.2.3 Vergleich mit der traditionellen Ausnahmebehandlung

Eine konventionelle Ausnahmebehandlung stützt sich wesentlich auf die *Rückgabewerte*, mit denen Funktionen bzw. Methoden den Aufrufer über das Ergebnis ihrer Bemühungen informieren. Meist wird ein ganzzahliger *Returncode* verwendet, wobei die 0 einen erfolgreichen Ablauf meldet, während andere Zahlen für einen bestimmten Fehlertyp stehen. Nun ist im Prinzip nach jedem Funktionsaufruf der Returncode auszuwerten, so dass eine harmlose Sequenz von drei Aufrufen:

```
f1();
f2();
f3();
```

nach Ergänzen der Fehlerbehandlung zu einer länglichen und recht unübersichtlichen Konstruktion wird (nach Mössenböck 2005, S. 254):

```
returncode = f1();
if (returncode == 0) {
  returncode = f2();
  if (returncode == 0) {
    returncode = f3();
    if (returncode == 0) {
      . . .
    }
    else {
      Ausnahmebehandlung für diverse f3()-Fehlercodes}
  }
  else {
    Ausnahmebehandlung für diverse f2()-Fehlercodes}
}
else {
  Ausnahmebehandlung für diverse f1()-Fehlercodes}
```

Ein gut gesetzter Rückgabewert nutzt natürlich nichts, wenn sich der Aufrufer nicht darum kümmert. Ebenso unbeachtet kann eine *Statusvariable* bleiben, die oft alternativ zum Returncode als Kommunikationsmittel genutzt wird.

Neben der ungesicherten Beachtung eines Returncodes ist am klassischen Verfahren aus praktischer Sicht auch zu bemängeln, dass eine Fehlerinformation aufwändig entlang der Aufrufersequenz nach oben gemeldet werden muss, wenn sie nicht an Ort und Stelle behandelt werden kann/soll.

Gegenüber der konventionellen Ausnahmebehandlung hat die in Java realisierte Technik u.a. folgende Vorteile:

- Zwang zur Behandlung von Ausnahmen
  Im Unterschied zu Returncodes oder Fehlerstatusvariablen können Ausnahmen nicht ignoriert werden. Methoden, die Ausnahmen aus bestimmten Klassen werfen können, dürfen nur im Rahmen einer **try**-Anweisung mit geeignetem **catch**-Block ausgeführt werden. Für welche Ausnahmeklassen *kein* Behandlungszwang besteht, erfahren Sie in Abschnitt 8.4.
- 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.

- Relativ geringer Aufwand
  Eine sorgfältige Ausnahmebehandlung kann sehr aufwändig werden und den Programmumfang erheblich erweitern. Daher ist es vorteilhaft, wenn *alle* Anweisungen in einem (beliebig langen **try**-Block) einer gemeinsamen Ausnahmebehandlung unterworfen werden können.
- 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, meist ganzzahligen, Rückgabewert nicht der Fall ist.

Es soll nicht verschwiegen werden, dass ein *Zwang* zur Ausnahmebehandlung lästig werden kann, weshalb die Java-Designer z.B. in den Methoden der Klassen **PrintStream** und **PrintWriter** (vgl. Abschnitt 11) keine **IOException** werfen und stattdessen ein Fehlersignal setzen, das über die Methode **checkError**() abgefragt werden kann.

8.2.4 Diagnostische Ausgaben

Statt im **catch**-Block eine *eigene* Fehlermeldung zu formulieren, kann man auch die **toString()**-Methode des übergebenen Ausnahme-Objektes 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 Ausnahmeobjektes via Konstruktor erzeugt wurde, z.B.:

```
java.lang.NumberFormatException: For input string: "acht"
```

Wer nur die situationsspezifische Fehlerinformation, aber nicht den Namen der Exception-Klasse sehen möchte, verwendet die Methode **getMessage()**, z.B.:

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

In unserem Beispiel erscheint nur noch:

```
For input string: "acht"
```

Eine weitere nützliche Information, die ein Ausnahmeobjekt parat hat, ist die **Aufrufersequenz** (**stack trace**) von der ursprünglich betroffenen Methode bis zur **main()**-Methode, die von der virtuellen Maschine gestartet worden war. Mit dem Methodenaufruf

```
e.printStackTrace(System.out);
```

erhält man im Beispiel von Abschnitt 8.2.2:

```
java.lang.NumberFormatException: For input string: "acht"
      at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
      at java.lang.Integer.parseInt(Integer.java:447)
      at java.lang.Integer.parseInt(Integer.java:497)
      at Sequenzen.calc(Sequenzen.java:6)
      at Sequenzen.main(Sequenzen.java:23)
```

Die Ausnahme wird von der Methode **java.lang.Integer.parseInt(String s, int radix)** (Quellcodezeile 447 in der Datei **Integer.java**) geworfen und erhält im Konstruktor die nicht konvertierbare Zeichenfolge als Wert des **String**-Parameters **message**:

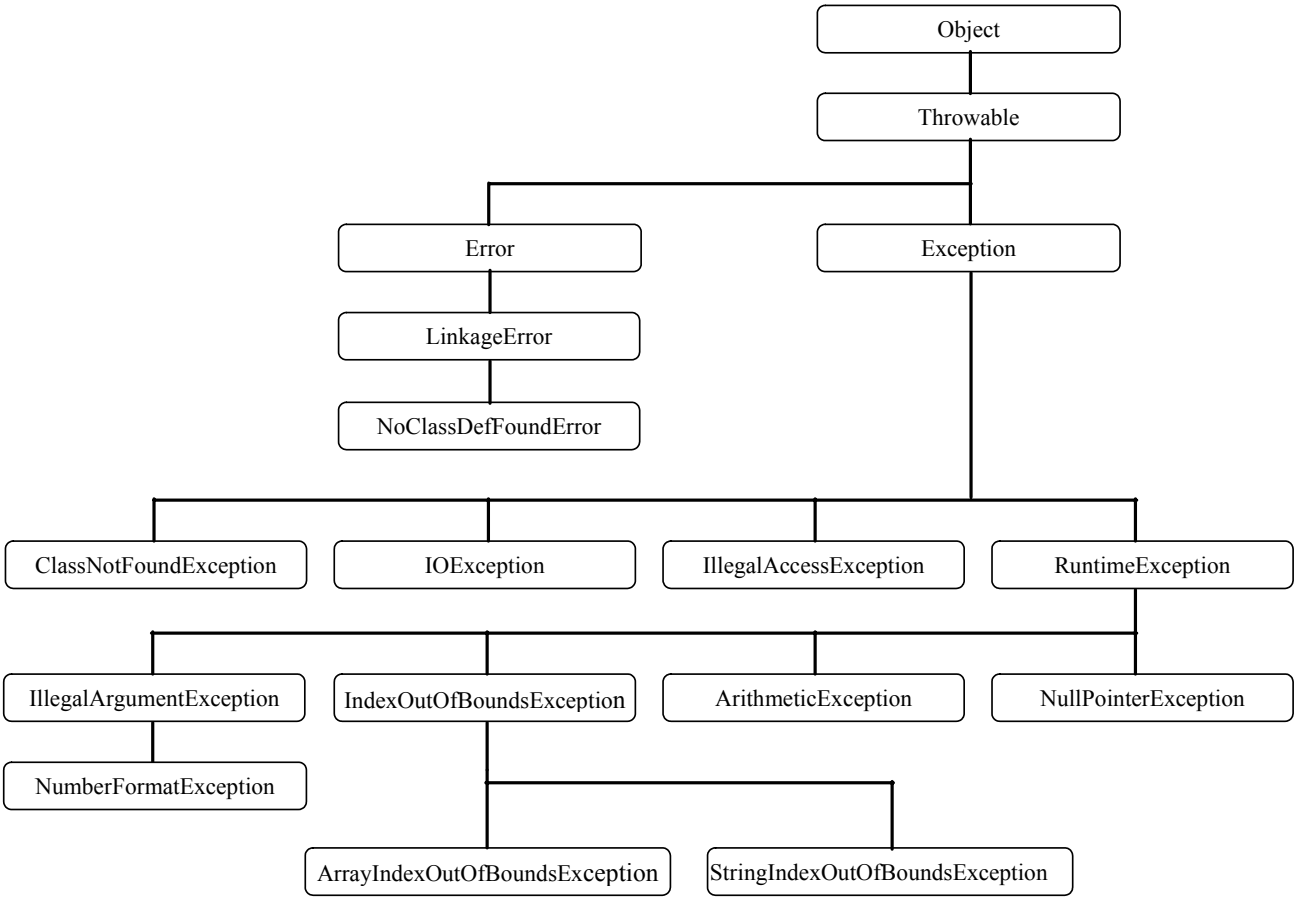
```
.
.
if (digit < 0) {
    throw new NumberFormatException(s);
} else {
    result = -digit;
}
.
.
```

Gerufen wurde **Integer.parseInt(String s, int radix)** von der Methode **Integer.parseInt(String s)**, die zwar denselben Namen besitzt, aber eine andere Signatur (Quellcodezeile 497 in der Datei **Integer.java**).
Weitere Stationen waren:

- `calc()` (Quellcodezeile 6 in der Datei **Sequenzen.java**)
- **main()** (Quellcodezeile 23 in der Datei **Sequenzen.java**)

8.3 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. In obigem Beispielprogramm zur Fakultätsberechnung (siehe Abschnitt 8.2) könnte etwa statt der speziellen **NumberFormatException** auch die maximal breite Ausnahmeklasse **Exception** angegeben werden:

```
static int kon2int(String instr) {
  int arg;
  try {
    arg = Integer.parseInt(instr);
  }
  catch (Exception e) {
    System.out.println("Fehler beim Konvertieren von \""+instr+"\"");
    System.exit(1);
    return 0;
  }
. . .
. . .
}
```

Sind mehrere **catch**-Blöcke vorhanden, dann werden diese beim Auftreten einer Ausnahme sequentiell von oben nach unten auf Zuständigkeit untersucht. Folglich ist darauf zu achten, dass speziellere Ausnahmeklassen *vor* allgemeineren stehen.

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 **Errors** nicht sinnvoll und nicht vorgesehen.
Kann der Interpreter 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)
```

8.4 Obligatorische und freiwillige Ausnahmebehandlung

Bei Ausnahmen aus der Klasse **RuntimeException** und aus daraus abgeleiteten Klassen (siehe obige Klassenhierarchie) ist es dem Programmierer *freigestellt*, ob er sie abfangen möchte (**unchecked exceptions**).
Alle übrigen Ausnahmen (z.B. **IOException**) *müssen* hingegen abgefangen werden (**checked exceptions)**.
In folgendem Programm soll mit der **read()**-Methode aus der Klasse **InputStream**, zu der das Standardeingabe-Objekt **System.in** gehört, ein Zeichen (bzw. ein Byte) von der Tastatur gelesen werden:

```
class ChEx {
  public static void main(String[] args) {
    int key = 0;
    System.out.print("Beliebige Taste + Return: ");
    key = System.in.read();
    System.out.println(key);
  }
}
```

Weil **read()** potentiell eine **IOException** auslöst (siehe JDK-Dokumentation), protestiert der Compiler:

```
ChEx.java:5: unreported exception java.io.IOException; must be
caught or declared to be thrown
    key = System.in.read();
                    ^
```

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);
  }
}
```

Gleich lernen Sie eine Möglichkeit kennen, auf die Behandlung einer obligatorischen Ausnahme zu verzichten und der nächst höheren Methode das Problem zu überlassen.

8.5 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 folgender Variante unseres Beispielprogramms zur Fakultätsberechnung wird in der Methode `kon2int()` ein Ausnahme-Objekt 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 kon2int(String instr) {
  int arg;
  try {
    arg = Integer.parseInt(instr);
    if (arg < 0  || arg > 170)
      throw new IllegalArgumentException ("Unzulaessiges Argument: "+arg);
    else
      return arg;
  }
  catch (NumberFormatException e) {
    throw new IllegalArgumentException ("Fehler beim Konvertieren");
  }
}
```

Zum Auslösen einer Ausnahme dient die **throw**-Anweisung. Sie enthält nach dem Schlüsselwort **throw** eine Referenz auf ein Ausnahme-Objekt. Wie im Beispiel erzeugt man oft das Ausnahme-Objekt an Ort und Stelle per **new**-Operator.

Die meisten Ausnahmeklassen besitzen *zwei* Konstruktoren:

- einen parameterfreien Konstruktor
- einen Konstruktor mit einem **String**-Parameter für eine Fehlermeldung (nähere Beschreibung der Ausnahme)

Wer eine fremde Methode benutzt, sollte wissen, welche Ausnahmen diese Methode auslöst und dann *nicht* lokal behandelt, so dass die aufrufende Methode eventuell zur Ausnahmebehandlung aufgefordert wird. Mit der **throws**-Klausel im Methodenkopf kann man darüber informieren, z.B.:

```
static int kon2int(String instr) throws IllegalArgumentException
```

Durch Kommata getrennt können in der **throws**-Klausel auch *mehrere* Ausnahmeklassen angemeldet werden.

Bei **unchecked exceptions** (**RuntimeException** und Unterklassen, siehe Abschnitt 8.3) ist es dem Programmierer *freigestellt*, ob er die in einer Methode ausgelösten, aber nicht behandelten, Exceptions deklarieren möchte. Für alle übrigen Ausnahmen (z.B. **IOException**) gilt: Wenn eine Methode eine solche Ausnahme auslöst, *muss* sie diese entweder selbst abfangen oder aber in der **throws**-Liste ihres Methodenkopfs auflisten.

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. Diese Technik wird im folgenden Beispiel „ohne störenden Sinn“ demonstriert:

Quellcode	Ausgabe
`class Prog {` `  static void nonsense(boolean cond) {` `  try {` `  if (cond)` `  throw new Exception("A");` `  else` `  throw new Exception("B");` `  }` `  catch (Exception e) {` `  System.out.println(e);` `  }` `  }` `  public static void main(String args[]) {` `  nonsense(true);` `  }` `}`	`java.io.Exception: A`

In Abschnitt 8.4 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 geschützten Block vorzunehmen. Es ist aber auch erlaubt, über das Schlüsselwort **throws** die Verantwortung auf die nächst höhere Ebene der Aufrufhierarchie abzuschieben. Im Beispielprogramm aus Abschnitt 8.4 kann sich die Methode **main()**, welche den potentiellen **IOException**-Absender **read()** ruft, der Pflicht zur Ausnahmebehandlung auf folgende Weise entledigen:

```
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 Ausnahmen *weiterleiten*, die von aufgerufenen Methoden stammen.

8.6 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 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;}
}
```

Durch Verwendung dieser handgestrickten, aus **Exception** abgeleiteten Ausnahmeklasse `BadFakulArgException` kann unsere Methode `kon2int()` beim Auftreten von irregulären Argumenten neben einer Fehlermeldung noch weitere Informationen an aufrufende Methoden übergeben:

- in `instr` die zu konvertierende Zeichenfolge
- in `error` einen numerischen Indikator für die Fehlerart
  - 1: Zeichenfolge kann nicht konvertiert werden
  - 2: konvertierter Wert außerhalb des erlaubten Bereichs
- in `value` das Konversionsergebnis (falls vorhanden, sonst -1)

Außerdem haben wir uns durch die Basisklasse **Exception** für eine *checked exception* entschieden, die im `kon2int()`-Methodenkopf deklariert werden *muss*:

```
static int kon2int(String instr) throws BadFakulArgException {
   int arg;
   try {
      arg = Integer.parseInt(instr);
      if (arg < 0  || arg > 170)
         throw new BadFakulArgException("Unzulaessiges Argument", instr, 2, arg);
      else
         return arg;
   }
   catch (NumberFormatException e) {
      throw new BadFakulArgException("Fehler beim Konvertieren", instr, 1, -1);
   }
}
```

Ebenso ist die **main()**-Methode gezwungen, entweder beim `kon2int()`-Aufruf die `BadFakulArgException` abzufangen oder ihrerseits im Methodenkopf die potentielle Ausnahme zu deklarieren:

```
public static void main(String[] args) {
   int argument = -1;
   if (args.length == 0) {
      System.out.println("Kein Argument angegeben");
      System.exit(1);
   }
   try {
      argument = kon2int(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("Wg. Fehler " + e.getError() + " (" +
                         e.getMessage() + ") keine Berechnung moeglich.");
      switch (e.getError()) {
        case 1 : System.out.println("String: \""+e.getInstr()+"\"");
                 break;
        case 2 : System.out.println("Wert: "+e.getValue());
                 break;
      }
    }
  }
```

Um eine *unchecked exception* zu erzeugen, wählt man eine Basisklasse aus der **RuntimeException**–Hierarchie.

8.7 Übungsaufgaben zu Abschnitt 8

1) Erstellen Sie ausnahmsweise ein Programm, das eine **NullPointerException** auslöst.
Hinweis: Man kann eine Referenzvariable „unbrauchbar“ machen, indem man ihr den Wert **null** zuweist. Sofern keine weiteren Referenzen auf das Objekt zeigen, wird es bei Gelegenheit vom Müllsammler (garbage collector) entsorgt.

2) 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 erst am Ende eines längeren Rechenweges das Ergebnis **NaN** auftaucht.
In folgendem Beispiel wird eine Methode zur Berechnung des dualen Logarithmus[27] verwendet, welche auf die Methode **java.lang.Math.log()** zurück greift und daher bei ungeeigneten Argumenten (≤ 0) als Rückgabewert **Double.NaN** liefert.

Quellcode	Ausgabe
`class DuaLog {` `  static double duaLog(double arg) {` `  return Math.log(arg) / Math.log(2);` `  }` `  public static void main(String[] args) {` `  double a = duaLog(8);` `  double b = duaLog(-1);` `  System.out.println(a*b);` `  }` `}`	`NaN`

Erstellen Sie eine Version, die bei ungeeigneten Argumenten eine **IllegalArgumentException** wirft.

---

[27] Für eine positive Zahl a ist ihr Logarithmus zur Basis b (> 0) definiert durch:

$$\log_b(a) := \frac{\log(a)}{\log(b)}$$

Dabei steht log() für den natürlichen Logarithmus zur Basis e (Eulersche Zahl).

9 Interfaces

Eventuell haben Sie sich schon einmal gefragt, 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**:

java.lang
Class Double

java.lang.Object
  └ java.lang.Number
    └ **java.lang.Double**

All Implemented Interfaces:
Serializable, Comparable<Double>

In diesem konkreten Fall wird über die Klasse ausgesagt:

- **Serializable**
  Weil die Klasse **Double** das Interface **java.io.Serializable** implementiert, können **Double**-Objekte mit Hilfe von Byte-Strömen auf bequeme Weise gespeichert und eingelesen werden. Diese (bei komplexeren Klassen beeindruckende) Option werden wir bald im Abschnitt 11 über Ein- und Ausgabe kennen lernen.
- **Comparable<Double>**
  Weil die Klasse **java.lang.Double** das Interface (die Schnittstelle) **java.lang.Comparable<Double>** implementiert, 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 folgender Signatur besitzen:

  public int compareTo(Double *o***)**

Ein Interface dient in der Regel dazu, Verhaltenskompetenzen von Objekten über eine Liste von Methodensignaturen zu definieren. Man kann dieses in Java erstmals verwendete OOP-Sprachelement in erster Näherung als Klasse mit ausschließlich abstrakten Methoden begreifen. Damit sind keine *Objekte* von diesem Datentyp denkbar, aber *Referenzvariablen* sind erlaubt und als Abstrakationsmittel sehr nützlich. Sie dürfen auf Objekte aus *allen* Klassen zeigen, welche die Schnittstelle implementieren.

Implementiert eine Klasse ein Interface, …

- muss sie Methoden mit den im Interface geforderten Signaturen besitzen oder sich selbst als **abstract** deklarieren,
- können bestimmte Methoden mit Objekten dieser Klasse ausgeführt werden. So setzt z.B. die Methode

  public static void sort(Object[] *array*)

  der Klasse **java.util.Arrays** das Interface **Comparable** voraus.
- werden Variablen vom Typ dieser Klasse vom Compiler akzeptiert, wo der Interface-Datentyp vorgeschrieben ist.

Im aktuellen Abschnitt 9 werden die Begriffe *Interface* und *Schnittstelle* synonym benutzt.

9.1 Interfaces definieren

Wenngleich das *Implementieren* einer Schnittstelle im Programmieralltag eher relevant wird als das *Definieren*, fangen wir mit Letzterem an, weil dabei Struktur und Funktion einer Schnittstelle deutlich werden.
Betrachten wir als Beispiel die Definition der API-Schnittstelle **java.lang.Comparable**:[28]

```
public interface Comparable<T> {
  public int compareTo(T o);
}
```

Im Schnittstellenkopf ist neben dem Schlüsselwort **interface** und dem Typnamen noch der Modifikator **public** mit der üblichen Bedeutung erlaubt.

Seit Java 5 kann dem Interface-Namen in spitzen Klammern eingeschlossen ein Typparameter angehängt werden, so dass für jeden Typ eine eigene Interface-Definition entsteht.

Im Schnittstellenrumpf werden abstrakte Methoden aufgeführt, deren Rumpf durch ein Semikolon ersetzt ist. Dabei kann ein Typparameter wie ein gewöhnlicher Typbezeichner eingesetzt werden.

Nun wird deutlich, was mit *Schnittstelle* bzw. *interface* gemeint ist: Hier wird festgeschrieben, dass Objekte eines Datentyps bestimmte Methodenaufrufe beherrschen müssen, wobei die Implementierung offen bleibt.

Einige allgemeine Regeln für Schnittstellen-Definitionen:

Modifikator **public**	Wird **public** *nicht* angegeben, ist die Schnittstelle nur innerhalb ihres Paketes verwendbar.
Modifikator **abstract**	Weil Schnittstellen grundsätzlich **abstract** sind, wird der Modifikator *nicht* angegeben.
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 dem Interface-Namen in spitzen Klammern eingeschlossen ein Typparameter angehängt werden.
extends *Interface1*, *Interface2, …*	Die von Klassen bekannte Vererbung über das Schlüsselwort **extends** wird auch bei Interfaces unterstützt. 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 sehr sinnvoll). • Die Hierarchie der Java-Interfaces ist *unabhängig* von der Klassenhierarchie.
Methodendeklaration	In einer Schnittstelle sind alle Methoden grundsätzlich **public** und **abstract**. Die beiden Schlüsselwörter können also weggelassen werden. Ein dem Schlüsselwort **public** widersprechender Zugriffsmodifikator ist verboten. Es sind auch Schnittstellen ohne Methoden erlaubt.

[28] 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 Ihre Festplatte befördert werden.

Konstanten	Neben Methoden sind auch Variablen erlaubt, wobei diese implizit als **public**, **final** und **static** deklariert sind. Beispiel: `public interface Farben { int ROT = 1, GRUEN = 2, BLAU = 3; }` Implementierende Klassen können auf die Konstanten ohne Angabe des Schnittstellennamens zugreifen. Andere Klassen müssen den Interface-Namen samt Punktoperator voranstellen.

Die `Demo`-Schnittstelle in folgendem Beispiel verlangt das Implementieren einer `say1()`-Methode und enthält eine **int**-Konstante namens `ONE`:

```
interface Demo {
  int ONE = 1;
  int say1();
}

class ImplDemo implements Demo {
  public int say1() {return ONE;}
}
```

Es sind auch Schnittstellen erlaubt, die weder Methoden noch Konstanten enthalten. Ein besonders wichtiges Beispiel ist die in Abschnitt 11 über Datenströme zu behandelnde Schnittstelle **java.io.Serializable**:

```
public interface Serializable {
}
```

Durch das Implementieren dieser Schnittstelle teilt eine Klasse mit, dass sie gegen das Serialisieren ihrer Objekte nichts einzuwenden hat.

9.2 Interfaces implementieren

Soll für die Objekte einer Klasse angezeigt werden, dass sie auch den Datentyp einer bestimmten Schnittstelle erfüllen, muss diese 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 fi) {
    if (xpos < fi.xpos)
      return -1;
    else if (xpos == fi.xpos)
      return 0;
    else
      return 1;
  }
}
```

Alle Methoden einer im Kopf angemeldeten Schnittstelle müssen im Rumpf der Klassendefinition implementiert werden, sofern keine abstrakte Klasse entstehen soll. Nach der in Abschnitt 9.1 wiedergegebenen **Comparable<T>** - Deklaration ist also im letzten Beispiel eine Methode mit der folgenden Signatur erforderlich:

public int compareTo(T *o***);**

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.

Weil die Methoden einer Schnittstelle grundsätzlich **public** sind, muss diese Schutzstufe auch für die *implementierenden* Methoden gelten, wozu in deren Deklaration der Zugriffsmodifikator **public** explizit anzugeben ist.

Während eine Klasse nur *eine* Super- bzw. Basisklasse besitzt, kann sie *beliebig viele* Schnittstellen implementieren, so dass ihre Objekte entsprechend viele Datentypen erfüllen. Wie wir inzwischen wissen, wird der Klasse aber nichts geschenkt (mal abgesehen von den Konstanten der Schnittstellen), sondern sie schließt Verträge mit dem Compiler ab und muss die vertragsgemäßen Leistungen erbringen.

Auch Schnittstellen ändern nichts daran, dass für Java-*Klassen* eine Mehrfach*vererbung* ausgeschlossen ist. Beim Definieren der Programmiersprache wurde diese Möglichkeit wegen einiger Risiken bewusst *nicht* aus C++ übernommen. 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 Unterklasse die Schnittstellen ihrer Basisklasse erbt. Wird z.B. die Klasse `Kreis` von der oben vorgestellten Klasse `Figur` abgeleitet, so übernimmt sie auch die Schnittstelle **Comparable<Figur>**, und die statische **sort()**-Methode der Klasse **java.util.Arrays.Arrays** kann auf Felder mit `Kreis`-Elementen angewendet werden:

Quellcode	Ausgabe
`class Test {` `  public static void main(String[] args) {` `  Kreis[] ka = new Kreis[3];` `  ka[0] = new Kreis("A", 250, 50, 10);` `  ka[1] = new Kreis("B", 150, 50, 20);` `  ka[2] = new Kreis("C", 50, 50, 30);` `  for (Kreis ko : ka)` `  System.out.print(ko.name+" ");` `  java.util.Arrays.sort(ka);` `  System.out.println();` `  for (Kreis ko : ka)` `  System.out.print(ko.name+" ");` `  }` `}`	`A B C` `C B A`

In der Java-API – Dokumentation sind die Schnittstellen im Navigations-Frame an den kursiv gesetzten Namen zu erkennen, z.B.:

9.3 Interfaces als Referenz-Datentypen 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 DemInt {` `  int sagWas();` `}` `class ImplDemo1 implements DemInt {` `  public int sagWas() {return 1;}` `}` `class ImplDemo2 implements DemInt {` `  public int sagWas() {return 2;}` `}` `class Test {` `  public static void main(String[] args) {` `  DemInt tobj1 = new ImplDemo1(),` `  tobj2 = new ImplDemo2();` `  System.out.println("ImplDemo1-Objekt: " +` `  tobj1.sagWas());` `  System.out.println("ImplDemo2-Objekt: " +` `  tobj2.sagWas());` `  }` `}`	`ImplDemo1-Objekt: 1` `ImplDemo2-Objekt: 2`

Damit wird es z.B. möglich, Objekte aus beliebigen Klassen in einem Array gemeinsam zu verwalten, sofern alle Klassen dasselbe Interface implementieren. Zwar lässt sich derselbe Zweck auch mit **Objekt**-Referenzen erreichen, doch leidet unter so viel Liberalität die Typsicherheit.

10 GUI-Programmierung

Eine Anwendung mit graphischer Benutzeroberfläche (engl.: **G**raphical **U**ser **I**nterface) präsentiert dem Anwender ein oder mehrere Fenster, die neben Bereichen zur Bearbeitung von programmspezifischen Dokumenten (z.B. Texten oder Grafiken) in der Regel mehrere Bedienelemente zur Benutzerinteraktion besitzen (z.B. Menüs, Befehlsschalter, Kontrollkästchen, Textfelder, Auswahllisten). Die von einer Plattform (in unserem Fall also von Java) zur Verfügung gestellten Bedienelemente bezeichnet man oft als *Komponenten*, *controls*, *Steuerelemente* oder *widgets*[29]. 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 nicht nur anschaulicher und intuitiver, sondern vor allem auch ereignisreicher 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 Benutzeroberfläche ist hingegen ein **ereignisorientiertes Paradigma** wesentlich, wobei das Laufzeitsystem als Vermittler von Ereignissen in erheblichem Maße den Ablauf mitbestimmt, indem es Methoden des Programms aufruft. Ausgelöst werden die Ereignisse in der Regel vom Benutzer, der mit der Hilfe von GUI-Komponenten seine Wünsche artikuliert. Ein GUI-Programm präsentiert permanent eine große Anzahl von Bedienelementen und muss eine entsprechende Anzahl von Ereignisbehandlungmethoden bereithalten.
Bei stark 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 ständig verfügbaren Bedienelementen benutzt.

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 7.7. Allerdings kommen nur wenige GUI-Anwendungen mit diesen Gestaltungs- bzw. Interaktionsmöglichkeiten aus.

Anmerkung zum Fenster-Design mit graphischen Entwicklungswerkzeugen

Wir werden in diesem Kurs die GUI-Gestaltung *manuell* vornehmen und dabei die technischen Grundlagen kennen lernen. Im späteren Programmieralltag sollten Sie zur Steigerung der Produktivität eine Entwicklungsumgebung mit graphischem Fenster-Designer verwenden (z.B. Eclipse, NetBeans von Sun, JBuilder von Borland). M.E. ist es erst dann sinnvoll, den zu einem Fenster gehörenden Java-Code automatisch erstellen zu lassen, wenn man die Assistenten-Produkte verstehen und nötigenfalls modifizieren kann.

10.1 Java Foundation Classes

In der Java-Standardbibliothek sind leistungsfähige Klassen zur **plattformunabhängigen GUI-Programmierung** enthalten, die gemeinsam als ***J**ava **F**oundation **C**lasses* (JFC) bezeichnet werden (siehe z.B. http://java.sun.com/products/jfc/index.html). Die JFC-Software verbessert und erweitert

---

[29] Diese Wortkombination aus *window* und *gadgets* steht für ein praktisches Fenstergerät.

die ältere, als ***A**bstract **W**indow **T**oolkit* (AWT) bezeichnete, Java-GUI-Technologie, die aber nicht überflüssig geworden ist und daher in folgender Liste mit JFC-Bestandteilen noch erscheint:

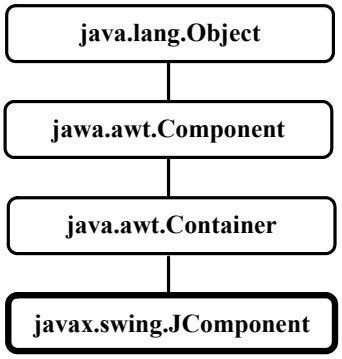
- **Abstract Windowing Toolkit** (AWT, enthalten im Paket **java.awt)**
  Das bereits in Java 1.0 vorhandene AWT ist heute teilweise überholt, stellt aber immer noch wichtige Basisklassen für die aktuellen GUI-Komponenten zur Verfügung. Grundidee beim AWT-Entwurf war die möglichst weitgehende Verwendung von Steuerelementen des *Wirtsbetriebssystems*. Aus der notwendigen Beschränkung auf den kleinsten gemeinsamen Nenner der zu unterstützenden Plattformen resultierte ein beschränkter Funktionsumfang. Die Kopplung jedes Java-Bedienelements an eine Entsprechung des Betriebsystems förderte auch nicht unbedingt die Ausführungsgeschwindigkeit, vielleicht der Grund dafür, die AWT-Komponenten als „schwergewichtig" zu bezeichnen.
  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, die heute in der Regel zu bevorzugen sind. Während die *Top-Level-Fenster* nach wie vor schwergewichtig sind und als Schnittstelle zum Betriebsystem dienen, werden die Steuerelemente komplett von Java verwaltet, was einige gravierende 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** bietet, müssen es aber nicht:
  - Für die Swing-Komponenten kann (sogar vom Benutzer zur Laufzeit) ein **Look & Feel** gewählt werden (verfügbar: Java, Windows, Motif), während die AWT-Komponenten stets auf das GUI-Design des Betriebssystems festgelegt sind.
  - Weitere Vorteile sind: QuickInfo-Fenster (Tool Tips), Steuerung per Tastatur, Unterstützung für die Anpassung an verschiedene Sprachen und Konventionen (Lokalisierung).
- **Java 2D** (enthalten im Paket **java.awt**)
  Mit dem Java2D-API lassen stehen leistungsfähige Klassen für ambitionierte 2D-Grafikanwendungen bereit.
- **Drag-&-Drop** (enthalten im Paket **java.awt.dnd**)
  Drag-&-Drop – Techniken werden Java-intern und auch bei der Kooperation mit anderen Anwendungen unterstützt.
- **Accessibility** (enthalten im Paket **javax.accessibility**)
  Über die Integration von zusätzlichen Ein- oder Ausgabegeräten in die Benutzerschnittstelle können Java-Programme leicht an spezielle Anforderungen angepasst werden (z.B. über Bildschirm-Leseprogramme oder Braille-Ausgaben für sehbehinderte Menschen).

10.2 Elementare GUI-Klassen

Dieser Abschnitt soll einen Überblick zu den im Kurs eingesetzten JFC-Klassen vermitteln.

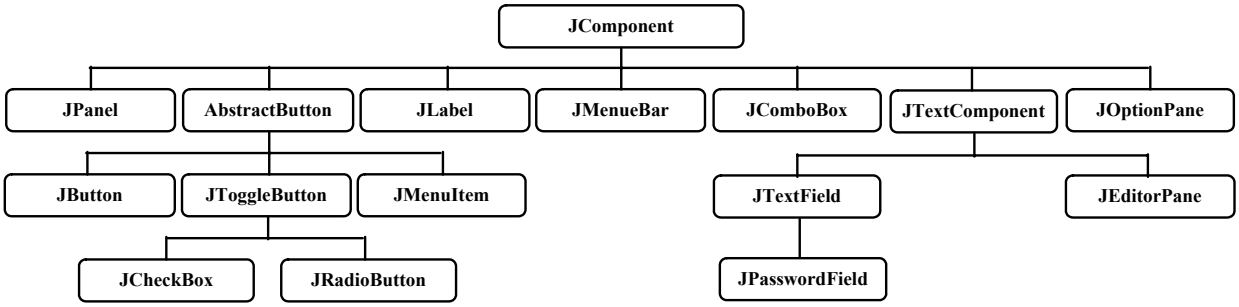
10.2.1 Komponenten und Container

Die fenster-internen Swing-Komponenten stammen meist von der Klasse **javax.swing.JComponent** ab, die wiederum zahlreiche Handlungskompetenzen und Eigenschaften über folgende Ahnenreihe erwirbt:



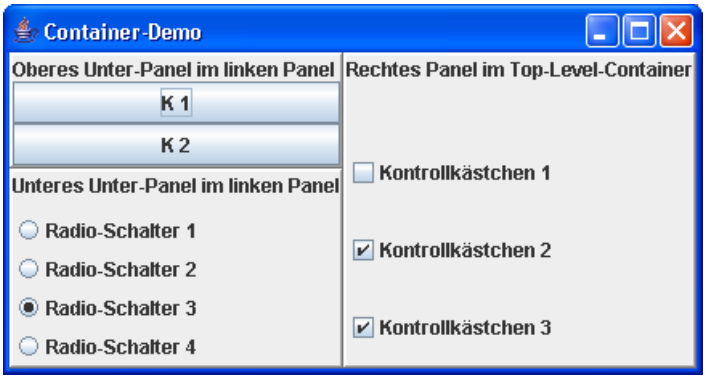
Komponenten sind also auch Objekte, allerdings mit optischer *Präsenz auf dem Bildschirm* und einigen zusätzlichen Kompetenzen: Sie interagieren mit dem Benutzer, wobei *Ereignisse* entstehen, über die sich andere Objekte informieren lassen können, um darauf zu reagieren (siehe unten).

In der folgenden Abbildung sehen Sie die Abstammungsverhältnisse für die in diesem Kurs behandelten **JComponent**-Abkömmlinge:[30]



Wie an den Namen unschwer zu erkennen ist, stehen die meisten Klassen für Steuerelemente, die aus GUI-Systemen wohlbekannt sind (Befehlsschalter, Label etc.).

In der Sammlung befindet sich mit **JPanel** aber auch ein *Container*. Diese Komponente entfaltet keine nennenswerte Interaktion mit dem Benutzer, sondern 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 (siehe unten) insgesamt 4 **JPanel**-Container, um Komponenten aus den Klassen **JLabel**, **JButton**, **JCheckBox** und **JRadioButton** anzuordnen:

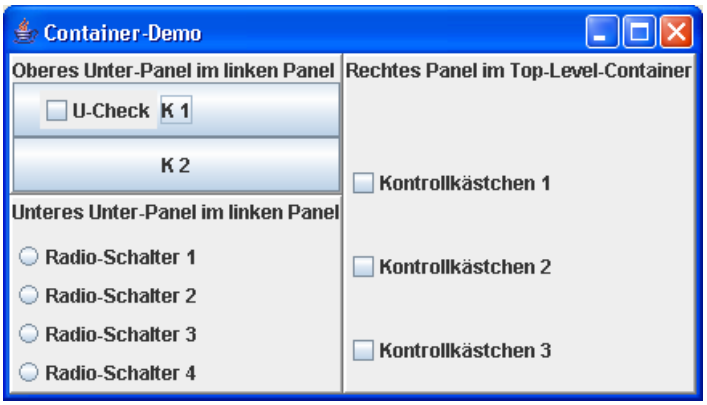


[30] Die Klasse **JTextComponent** stammt aus dem Paket **javax.swing.text**, alle anderen Klassen stammen aus dem Paket **javax.swing**.

Die Frage, welcher von den beiden Begriffen *Komponente* 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 **JComponent**-Unterklassen von **java.awt.Container** die Methode **add()** zum „Einfüllen“ von Komponenten. Die von uns bevorzugten Swing-Komponenten sind also allesamt Container. Man kann z.B. ein Kontrollkästchen in einen Befehlsschalter einbauen:



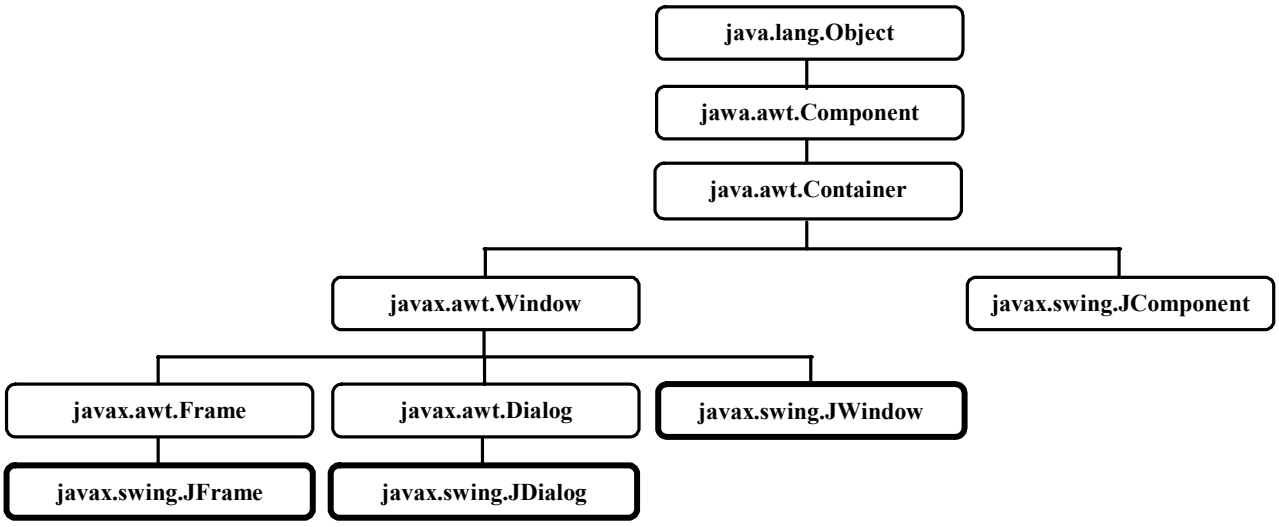
Allerdings fördern derartige Konstruktionen nicht unbedingt die Bedienbarkeit eines Programms. Zur Strukturierung der Bedienelemente einer Dialogbox kommt nur der unsichtbare Container **JPanel** in Frage. Im nächsten Abschnitt geht es um den Top-Level - Container einer Swing-Anwendung.

10.2.2 Top-Level - Container

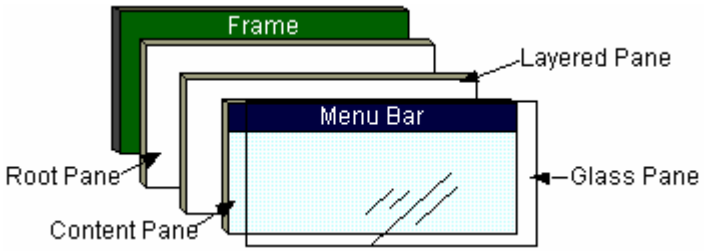
Jedes Programm mit Swing-GUI benötigt mindestens einen **Top-Level - Container**, der als Fenster zu der vom Betriebssystem verwalteten Benutzeroberfläche dient und die leichtgewichtigen Swing-Komponenten aufnimmt. In Abschnitt 10 werden alle Beispielprogramme als Top-Level - Container eine Komponente vom Typ **JFrame** benutzen, die ein **Rahmenfenster** mit folgender Ausstattung realisiert (siehe obige Container-Beispielprogramme):

- Rahmen
- Titelzeile
- Bedienelemente zum Schließen, Minimieren und Maximieren des Fensters
- Systemmenü (über die Java-Tasse am linken Rand der Titelzeile erreichbar)

JFrame und die anderen schwergewichtigen Swing-Komponenten (siehe unten) stammen nicht von **JComponent** ab, sondern haben einen etwas anderen Stammbaum:



JFrame-Container sind mehrschichtig aufgebaut, wie die folgende (aus dem Java-Tutorial entnommene) Abbildung zeigt:



Wir werden uns auf die **Content Pane** (Inhaltsschicht) beschränken, die wir in unseren Programmen über eine von der **JFrame**-Methode **getContentPane()** gelieferte Referenz ansprechen können (siehe unten). Andere Schichten werden z.B. zum Realisieren der Drag-&-Drop - Funktionalität benötigt.

In Swing-Anwendungen sind noch andere Top-Level-Container möglich:

- **JDialog**
  Mit dieser Klasse werden Dialogfenster realisiert, denen im Vergleich zum Rahmenfenster die Bedienelemente zum Maximieren und zum Minimieren fehlen. Sie können zwar auch als selbständige Fenster einer Anwendung arbeiten, werden aber meist einem Rahmenfensters untergeordnet, so dass sie beim Schließen des Besitzers automatisch verschwinden. In der untergeordneten Rolle kann ein **JDialog**-Objekt *modal* arbeiten, so dass während seines Auftritts das übergeordnete Fenster blockiert ist.
- **JWindow**
  Bei den mit dieser Klasse 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 Applets, die im Rahmen eines Web-Browser - Fensters ausgeführt werden, kann die Klasse **JApplet** einen Swing-basierten Top-Level-Container realisieren.

10.3 Beispiel für eine Swing-Anwendung

In folgendem Swing-Programm 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 10 vollständig besprechen:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class SwApp extends JFrame {
    String labelPrefix = "Anzahl der Klicks: ";
    final static String titel = "Swing-Demo";
    JLabel label;
    JPanel panli;
    JButton button;
    int numClicks = 0;
    final byte maxicon = 3;
    ImageIcon icons[];
    JLabel labicon;
    byte iconr = 0;

    SwApp() {
        super(titel);
        Container cont = getContentPane();

        button = new JButton("Ich mag Swing!");
        button.setMnemonic(KeyEvent.VK_S);
        button.setToolTipText("Befehlsschalter");

        label = new JLabel(labelPrefix + "0    ");
        label.setToolTipText("Label mit Text");

        panli = new JPanel();
        panli.setBorder(BorderFactory.createEmptyBorder(30, 30, 30, 30));
        panli.setLayout(new GridLayout(0, 1));
        panli.add(button);
        panli.add(label);
        cont.add(panli, BorderLayout.CENTER);

        icons = new ImageIcon[maxicon];
        icons[0] = new ImageIcon("duke.gif");
        icons[1] = new ImageIcon("fight.gif");
        icons[2] = new ImageIcon("snooze.gif");
        labicon = new JLabel(icons[0]);
        labicon.setToolTipText("Label mit Icon");
        cont.add(labicon, BorderLayout.EAST);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                numClicks++;
                label.setText(labelPrefix + numClicks);
                if (iconr < maxicon-1)
                   iconr++;
                else
                   iconr = 0;
                labicon.setIcon(icon[iconr]);
            }
        });
```

```
        addWindowListener(new Ex());
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
        setSize(300, 150);
        setVisible(true);
    }

    public static void main(String[] args) {
        new SwApp();
    }

    class Ex 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) == 0)
                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 SwApp. Von dieser Rahmenfenster-Sorte wird in der **main()**-Methode *ein* Objekt erzeugt:

```
public static void main(String[] args) {
    new SwApp();
}
```

Weil das Rahmenfenster mit seinen Steuerelementen ein vergleichsweise komplexes Objekt darstellt, hat der SwApp-Konstruktor einige Arbeit:

- Das Fenster erhält seinen Titel durch expliziten Aufruf des Basisklassen-Konstruktors:
  ```
  super(titel);
  ```
- Um bequem auf die Inhaltsschicht (content pane) des Rahmenfensters zugreifen zu können, wird mit der **JFrame**-Methode **getContentPane()** eine lokale Referenzvariable auf diesen **Container** angelegt:
  ```
  Container cont = getContentPane();
  ```
- 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 noch eine initiale Größe für das Fenster fest und mach es dann sichtbar:
  ```
  setSize(300, 150);
  setVisible(true);
  ```

Das SwApp-Objekt sorgt in Kooperation mit dem Benutzer und dem Laufzeitsystem für einen „spannenden und abwechslungsreichen“ Programmablauf. Es aktualisiert bei einem Mausklick auf den Schalter oder nach der Tastenkombination **Alt+S** die beiden Label, lässt sich verschieben, in der Größe ändern, minimieren, maximieren und beenden.

10.4 Swing-Komponenten (Teil 1)

Das vorliegende Manuskript behandelt in zwei Portionen elementare Swing-Komponenten. Eine optische Präsentation *aller* Swing-Komponenten finden Sie im Java-Tutorial (SUN Inc. 2005) über:

Creating a GUI with JFC/Swing > Using Swing Components >
A Visual Index to the Swing Components

10.4.1 Label

Mit Komponenten der Klasse **JLabel** realisiert man Bedienungshinweise in Schrift- und/oder Bildform. Das erste Label aus unserem Beispielprogramm beschränkt sich auf eine Textanzeige:

```
JLabel label;
  . . .
label = new JLabel(labelPrefix + "0    ");
```

Das zweite Label 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*) wird auch das JPEG-Format (*Joint Photographic Experts Group*) und das PNG-Format (*Portable Network Graphics*) unterstützt.

Beim Erzeugen des zweiten Label-Objektes wird ein **ImageIcon** als initiale Anzeige festgelegt:

```
JLabel labicon;
  . . .
labicon = new JLabel(icons[0]);
```

Man kann diverse Swing-Komponenten mit **ImageIcon**-Objekten verschönern, wobei die Auswahl per Konstruktor oder per **setIcon()**-Methode erfolgt.

10.4.2 Befehlsschalter

Die Syntax zum Deklarieren bzw. Erzeugen eines Befehlsschalters bietet keinerlei Überraschungen:

```
JButton button;
  . . .
button = 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**:

```
button.setMnemonic(KeyEvent.VK_S);
```

Den **int**-wertigen Parameter der Methode **setMnemonic()** legt man am besten über die in der Klasse **KeyEvent** definierten VK-Konstanten (**V**irtual **K**ey) fest.

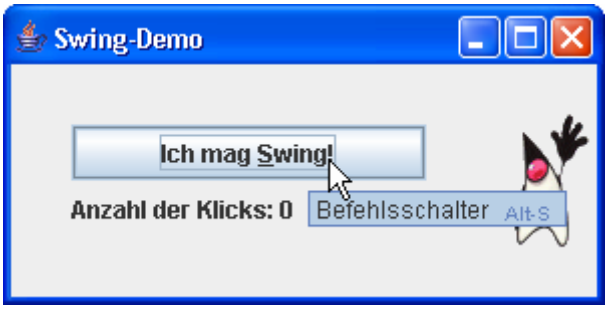
10.4.3 Zubehör für Swing-Komponenten

Es sind zahlreiche Möglichkeiten verfügbar, das optische Erscheinungsbild und die Bedienbarkeit von Swing-Komponenten zu verbessern.

Mit der **JComponent**-Methode **setToolTipText()** kann man Tool-Tipps (QuickInfos) zu einzelnen Steuerelementen definieren, z.B.:

```
button.setToolTipText("Befehlsschalter");
```

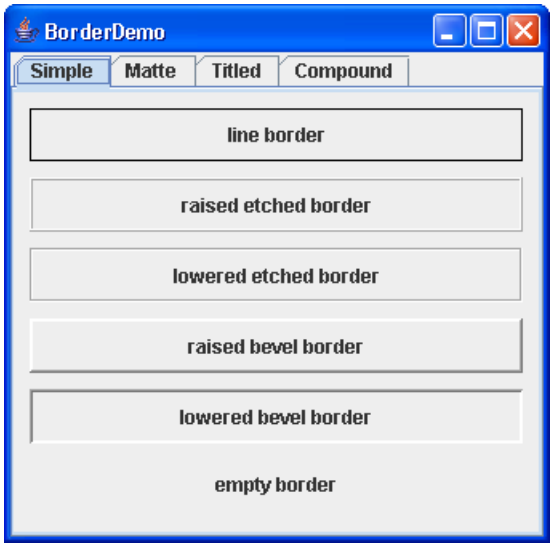
Diese erscheinen in einem PopUp-Fenster, wenn der Mauszeiger über einer betroffenen Komponente verharrt:



Außerdem lässt sich über die Methode **setBorder()** ein Rahmen festlegen, wobei die Klasse **BorderFactory** mit ihren statischen Methoden diverse Modelle herstellen kann. Im Einführungsbeispiel verschaffen wir dem **JPanel**-Container `panli` etwas „Luft“:

```
panli.setBorder(BorderFactory.createEmptyBorder(30, 30, 30, 30));
```

Den Quellcode zu der folgenden Rahmen-Musterschau:



finden Sie auf der Webseite:

http://java.sun.com/docs/books/tutorial/uiswing/misc/example-1dot4/BorderDemo.java

10.5 Die Layout-Manager der Container

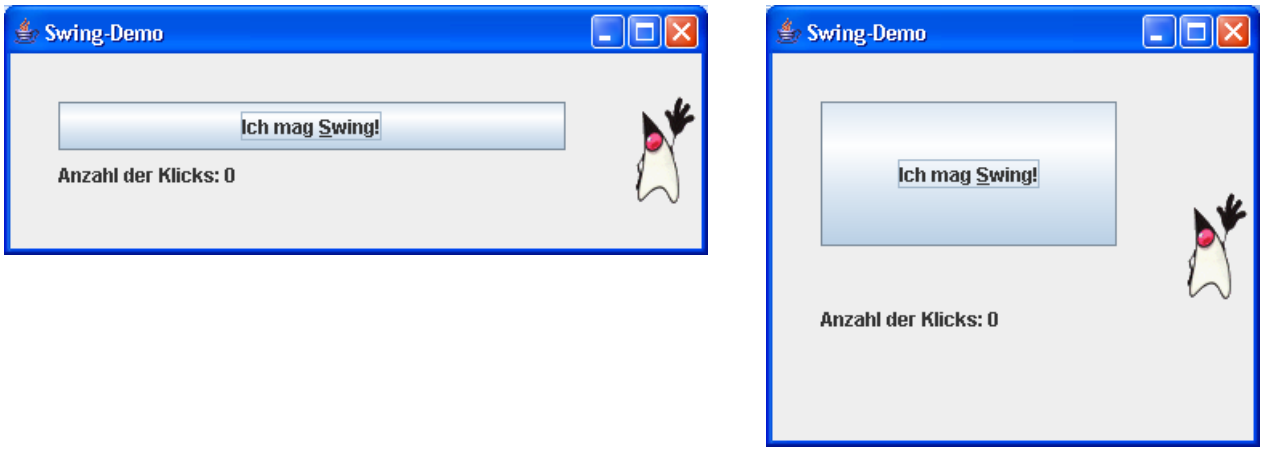
Eine **JPanel**-Komponente kann als Behälter für Swing-Komponenten dienen und dabei selbst in einem übergeordneten Container untergebracht werden, z.B. im Top-Level - Container eines **JFrame**-Fensters. Im unserem Beispiel wird ein **JPanel**-Behälter namens `panli` verwendet:

```
JPanel panli;
  . . .
panli = new JPanel();
```

Um eine Komponente in einen Container aufzunehmen, benutzt man eine der zahlreichen **add()**-Überladungen, z.B.:

```
panli.add(button);
panli.add(label);
```

Die Entscheidung für eine **add()**-Variante hängt auch vom **Layout-Manager** ab, der für eine sinnvolle Anordnung der in einem Container enthaltenen Komponenten sorgt und diese Anordnung bei Größenänderungen dynamisch angepasst, z.B.:



Dank Layout-Manager können sich Java-Programmierer auf die Logik ihrer Anwendung konzentrieren und müssen sich nicht um Positionen und Größen der einzelnen Steuerelemente kümmern. Durch Verschachteln von Containern, für die jeweils ein spezieller Layout-Manager engagiert werden kann, sollte sich fast jede Design-Idee verwirklichen lassen.

Selbstverständlich ist es aber auch möglich, alle Layout-Details pixel-genau festzulegen (siehe Abschnitt 10.5.4), und mit dem folgenden Aufruf der **JFrame**-Methode **setResizable()** kann man eine Änderungen der Fenstergröße durch den Benutzer verhindern:

```
setResizable(false);
```

10.5.1 BorderLayout

In unserem Beispielprogramm bleiben bei nahezu beliebigen Veränderungen des Anwendungsfensters (siehe oben) die folgenden räumlichen Relationen erhalten:

- Das Label mit dem Klickzähler steht senkecht unter dem Befehlsschalter.
- Das Bild erscheint stets rechts neben den beiden anderen Komponenten.

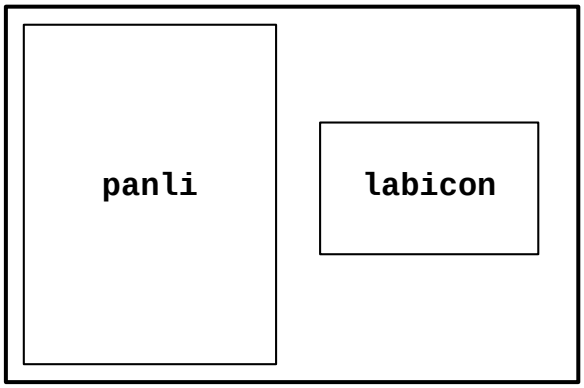
Eine wesentliche Voraussetzung für dieses Verhalten sind die beiden beteiligten **Container**.
Das Anwendungsfenster (Klasse `SwApp`, abgeleitet aus **JFrame**) ist ein (schwergewichtiger) Top-Level - Container und enthält (auf seiner content pane):

- den untergeordneten (leichtgewichtigen) Container `panli` aus der Klasse **JPanel**
- das Label `labicon`

Die beiden leichtgewichtigen Komponenten sind über die **add()**-Methode der **JFrame**–Inhaltsschicht (ein **Container**-Objekt, über die Referenzvariable `cont` ansprechbar) zu ihrem Standort gekommen:

```
cont.add(panli, BorderLayout.CENTER);
cont.add(labicon, BorderLayout.EAST);
```

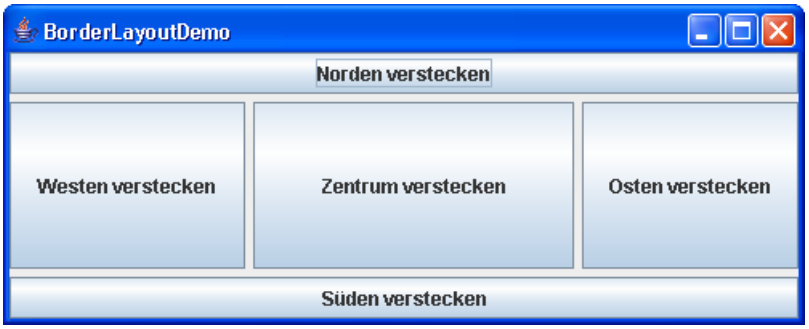
Sie halten folgende räumliche Anordnung ein:



Dafür sorgt der **Layout-Manager** der **JFrame**-Komponente `SwApp` (genauer: der zugehörigen content pane). Weil wir die Voreinstellung nicht geändert haben, handelt es sich um ein Objekt der Klasse **BorderLayout**. Dies wird deutlich in den Positionierungsparametern der oben wiedergegebenen **add()**-Aufrufe:

- `panli` soll das Zentrum des `SwApp`-Containers besetzen, das sich mangels West-Komponente am linken Rand befindet.
- `labicon` soll sich am östlichen `SwApp`-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 durch folgenden Layout-Manager gesorgt:

```
BorderLayout layout = new BorderLayout(5,5);
```

Welche Gestaltungsmöglichkeiten ein **BorderLayout** durch sein Verhalten bei unbesetzten Positionen und bei Veränderungen der Container-Fläche bietet, sollten Sie durch Probieren herausfinden.

10.5.2 GridLayout

Der `panli`-Container enthält zwei Komponenten:

- `button`
- `label`

Für `panli` wurde mit der **JPanel**-Methode **setLayout()**, geerbt von der Klasse **java.awt.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 `panli`-Komponenten bei linksbündiger Ausrichtung übereinander stehen. Dazu wird im Konstruktor *eine* Spalte und (über den Aktualparameter 0) eine unbestimmte Anzahl von Zeilen angekündigt:

```
panli.setLayout(new GridLayout(0, 1));
```

Ist ein solcher Layout-Manager einmal im Dienst, verteilt er die Komponenten automatisch auf die freien Plätze, so dass die **add()**-Methode ohne Platzanweisung auskommt:

```
panli.add(button);
panli.add(label);
```

In der folgendem Anweisung wird dem Inhaltsbereich eines **JFrame**-Rahmenfensters ein (2 × 4) - **GridLayout** – Manager mit Zwischenabständen von jeweils 5 Pixeln zugeordnet:

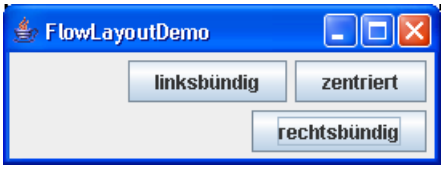
```
getContentPane().setLayout(new GridLayout(2,4,5,5));
```

Der verfügbare Platz wird von einem **GridLayout** gleichmäßig auf alle Komponenten verteilt, wie das folgende Beispielprogramm mit acht phantasielos beschrifteten Schaltern zeigt:

10.5.3 FlowLayout

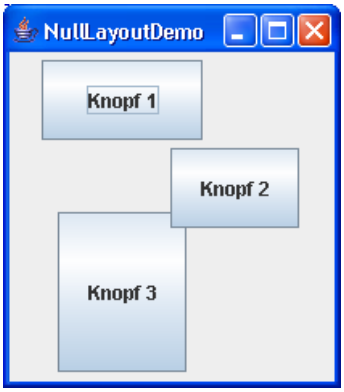
Recht verbreitet ist das **FlowLayout**, das bei vielen Containern (z.B. **JPanel**) als Voreinstellung dient (bei Verzicht auf die explizite Spezifikation eines Layout-Managers per Konstruktor oder **setLayout()**). Es ordnet die Komponenten nebeneinander an, bis ein „Zeilenumbruch“ erforderlich wird:

An Stelle der horizontalen Zentrierung ist alternativ auch eine links- bzw. rechtsbündige Anordnung möglich, z.B.:



10.5.4 Freies Layout

Wer noch mehr Freiheit benötigt, kann auf einen Layout-Manager verzichten und alle Positionen bzw. Größen individuell festlegen, so dass z.B. sogar überlappende Komponenten möglich werden:



Ein freies Layout lässt sich folgendermaßen realisieren:

- Den voreingestellten Layout-Manager abschalten mit **setLayout(null)**
- Positionen und Größen der Komponenten mit **setBounds()** festlegen

Dies wird in folgendem Programmfragment demonstriert:

```
  .   .   .
class NullLayoutDemo extends JFrame {
  JButton k1, k2, k3;
  Container cont = getContentPane();

  NullLayoutDemo() {
    super("NullLayoutDemo");

    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);
  }
  .   .   .
}
```

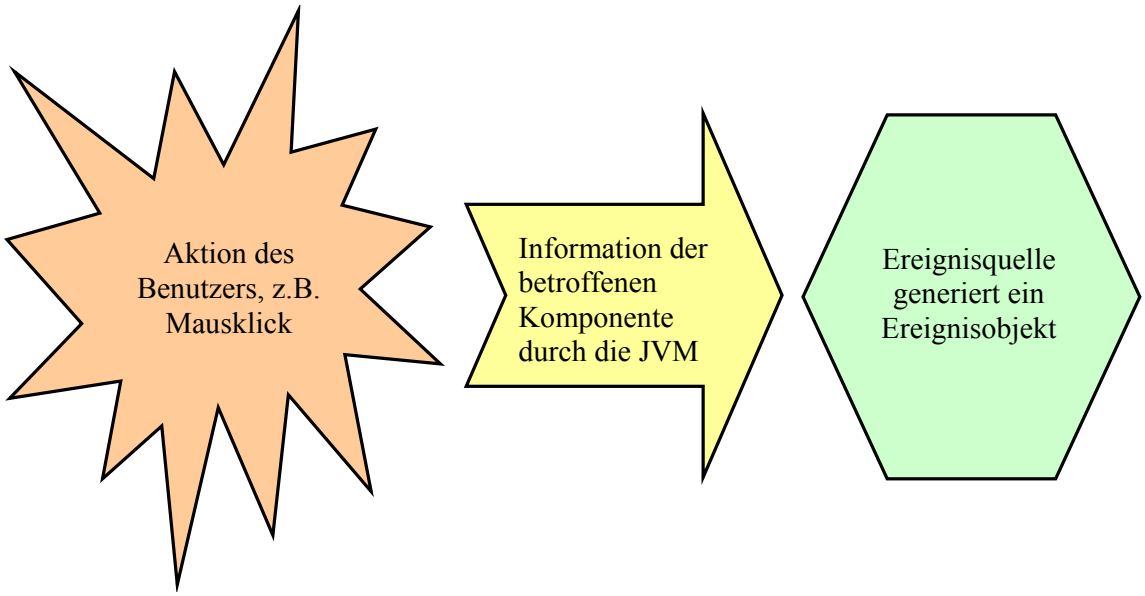
10.6 Ereignisbehandlung

Die Besonderheit einer Komponente im Unterschied zum 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.

10.6.1 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 dem Abschnitt 10.3 besprechen, kann z.B. der Befehlsschalter `button` 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.
Eine Komponente kann von verschiedenen Ereignissen betroffen sein und dementsprechend Ereignisobjekte aus verschiedenen Klassen generieren (z.B. **MouseEvent**, **ActionEvent**, **KeyEvent**).

- **Ereignisobjekt**
  Zu jedem Ereignistyp gehört in Java eine Ereignisklasse, über deren Methoden der anschließend vorzustellende Ereignisempfänger nähere Informationen über das Ereignis abfragen kann. Z.B. kann er nötigenfalls mit **getSource()** die Ereignisquelle in Erfahrung bringen, um angemessen zu reagieren. Bei einem Mausereignis (siehe unten) lässt sich über **getX()** und **getY()** der Ort des Geschehens feststellen.
- **Ereignisempfänger**
  Ein Ereignis wird in der Regel *nicht* „direkt an der Quelle“ behandelt. Statt dessen wird die Bearbeitung an andere Objekte, die Ereignisempfänger (engl.: *event listener*) , *delegiert*.
  Zu jeder Ereignisklasse gehört ein *Interface*, das die Existenz von Methoden mit bestimmter Signatur vorschreibt. Diese Methoden werden auch als *Event Handler* bezeichnet.
  Objekte einer entsprechend gerüsteten Klasse können bei einer Ereignisquelle als Empfänger registriert werden. Oft 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 ihre Vorteile, wenn in einem Programm mehrere Komponenten als Quelle für dieselbe Ereignisklasse in Frage kommen und sich *ein* Empfänger um die Ereignisbehandlung kümmert. In diesem Fall wären getrennt agierende Ereignisbehandlungsmethoden unökonomisch und eventuell sogar unsicher. Außerdem fördert die Trennung von Benutzeroberfläche und Ereignisbehandlung die Wiederverwendbarkeit der Software.

10.6.2 Ereignisklassen und Ereignisarten

Potentielle Empfänger für ein **WindowEvent** müssen das Interface **WindowListener** implementieren, zu dem etliche Methoden gehören, wie die API-Dokumentation zeigt:

Method Summary	
void	**windowActivated**(WindowEvent e) Invoked when the Window is set to be the active Window.
void	**windowClosed**(WindowEvent e) Invoked when a window has been closed as the result of calling dispose on the window.
void	**windowClosing**(WindowEvent e) Invoked when the user attempts to close the window from the window's system menu.
void	**windowDeactivated**(WindowEvent e) Invoked when a Window is no longer the active Window.
void	**windowDeiconified**(WindowEvent e) Invoked when a window is changed from a minimized to a normal state.
void	**windowIconified**(WindowEvent e) Invoked when a window is changed from a normal to a minimized state.
void	**windowOpened**(WindowEvent e) Invoked the first time a window is made visible.

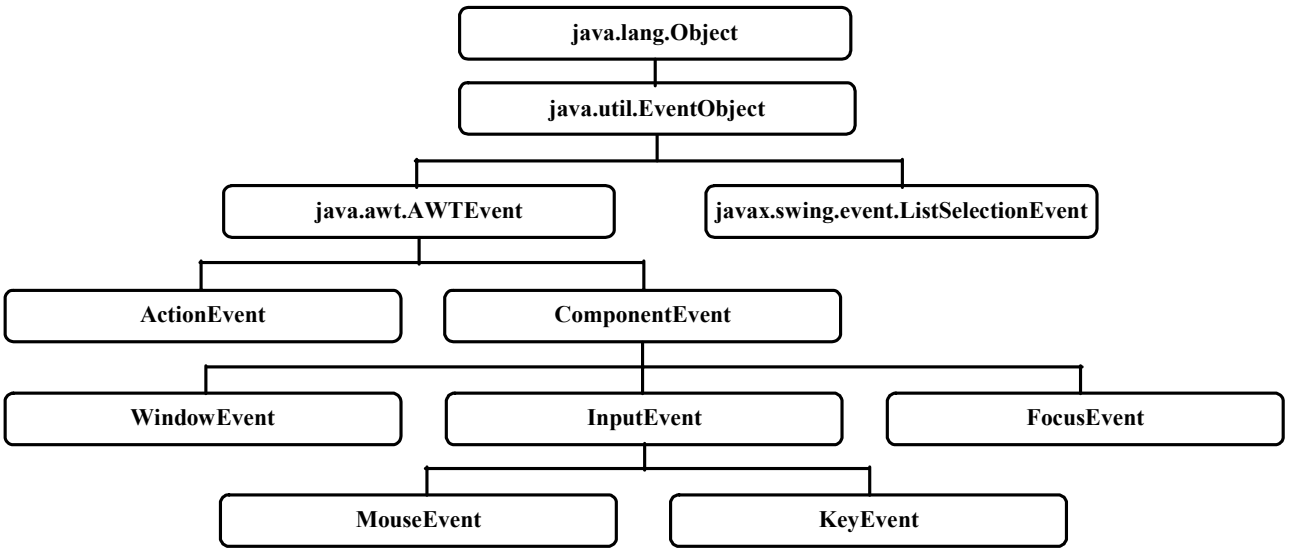
Aus der Anzahl und den Namen der Methoden wird klar, dass die Ereignisklasse **WindowEvent** bei *mehreren* Ereignissen zum Einsatz kommt, z.B. beim Schließen und Verkleinern eines Fensters. Die Java-Designer haben sich für eine überschaubare Anzahl von Ereignisklassen entschieden, die jeweils für mehrere, zusammen gehörige Ereignisarten zuständig sind.

Die von einem Ereignis betroffenen Komponente erfährt von der JVM die exakte Ereignisart (**Event ID**) und kennt nun ...

- die Ereignisklasse
  Damit ist klar, an welche Empfänger das Ereignis weitergeleitet werden muss.
- die bei den Empfängern aufzurufende Ereignisbehandlungsmethode

Das an die Event Handler übergebene Ereignisobjekt enthält die Information über die Ereignisart in der Instanzvariablen **id**.

In der folgenden Abbildung sehen Sie die Abstammungsverhältnisse für die im Kurs behandelten Ereignisklassen:



Klassen ohne Paketangabe gehören zum Paket **java.awt.event**.

10.6.3 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 SwApp (eine **JFrame**-Komponente) im Konstruktor für die Ereignisklasse **WindowEvent** mit der Methode **addWindowListener()** ein neu erzeugtes Objekt aus der Klasse Ex als Empfänger eingetragen:

```
addWindowListener(new Ex());
```

Damit die Klasse Ex den Anforderungen an einen **WindowEvent**-Empfänger genügt, hat sie das Interface **WindowListener** (siehe oben) zu implementieren (direkt oder indirekt).

Zu jeder Ereignisklasse gehört eine spezielle Registrierungsmethode. Welche Ereignisklassen eine Komponente im Umlauf bringen kann, ist also an der Verfügbarkeit von Registrierungsmethoden zu erkennen. So ist z.B. in der Klasse **javax.swing.JList**, nicht aber in der Klasse **javax.swing.JButton**, die Methode **addListSelectionListener()** vorhanden, um Empfänger für das Ereignis **ListSelectionEvent** (Benutzer wechselt in einer Liste den gewählten Eintrag) zu registrieren.

In der folgenden Tabelle ist für einige Ereignisklassen festgehalten:

- zugrunde liegendes Benutzerverhalten
- vom Ereignisempfänger zu implementierendes Interface
  Hier ist festgelegt, welche Event Handler - Methoden eine Listener - Klasse besitzen muss.
- zuständige Empfänger-Registrierungsmethode

Außerdem sind die im nächsten Abschnitt zu beschreibenden *Adapterklassen* angegeben, die für viele Ereignisklassen zur Vereinfachung der Empfänger-Definition verfügbar sind:

Ereignisklasse	**Mögliche Auslöser**	**Empfänger-Interface, zugeh. Adapterklasse, Registrierungsmethode**
ActionEvent	Benutzer klickt auf einen Befehlsschalter, drückt **Enter** in einem Textfeld oder wählt einen Menüeintrag.	ActionListener addActionListener()
ComponentEvent	Eine Komponente wird sichtbar.	ComponentListener ComponentAdapter addComponentListener()
WindowEvent	Benutzer schließt das Anwendungsfenster.	WindowListener WindowAdapter addWindowListener()
MouseEvent[31]	Benutzer klickt, während sich die Maus über einer Komponente befindet.	MouseListener MouseAdapter addMouseListener()
MouseEvent[31]	Benutzer bewegt die Maus über einer Komponente.	MouseMotionListener MouseMotionAdapter addMouseMotionListener()
KeyEvent	Benutzer drückt eine Taste.	KeyListener KeyAdapter addKeyListener()

[31] Ein **MouseEvent** wird ggf. an einen registrierten **MouseListener** *und* an einen registrierten **MouseMotionListener** weitergeleitet.

Ereignisklasse	Mögliche Auslöser	Empfänger-Interface, zugeh. Adapterklasse, Registrierungsmethode
ItemEvent	Das gewählte Item einer Liste oder der Zustand eines Umschalters (z.B. Kontrollkästchen) hat gewechselt.	ItemListener addItemListener()
FocusEvent	Eine Komponente erhält den Eingabefokus.	FocusListener FocusAdapter addFocusListener()
ListSelectionEvent	Benutzer wechselt in Liste den gewählten Eintrag.	ListSelectionListener addListSelectionListener()

Bei einer Ereignisquelle können zu einer Ereignisklasse auch *mehrere* Ereignisempfänger registriert werden, so dass Ereignisobjekte ggf. an mehrere Empfänger weitergeleitet werden.

10.6.4 Adapterklassen

In unserem Beispiel soll der **WindowEvent**-Empfänger zur Ereignisquelle `SwApp` 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 eigentlich nur die **WindowListener**–Methode **windowClosing()** benötigt. Um in solchen Fällen überflüssigen Programmieraufwand zu vermeiden, hat man zu vielen Ereignis-Interfaces jeweils noch so genannten **Adapterklassen** eingeführt. Diese implementieren 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 `Ex` aus der zum Interface **WindowListener** gehörigen Adapterklasse **WindowAdapter** abgeleitet:

```
class Ex 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) == 0)
            System.exit(0);
    }
}
```

Der Event Handler **windowClosing()** fragt beim Benutzer nach, ob allen Ernstes aufhören möchte (vgl. Abschnitt 7.7). Erst nach einer Bestätigung dieser Absicht, wird das Programm beendet.

10.6.5 Schließen von Fenstern und 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 man mit folgender **JFrame**-Methode für diese **WindowEvent**-Behandlung sorgen:

```
setDefaultCloseOperation(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 per Fenstertitelzeilen-Symbol oder Systemmenü schließen, wenn man weder einen **WindowListener** registriert, noch die **setDefaultCloseOperation()**-

Methode benutzt hat. Allerdings verschwindet dabei nur das *Fenster* vom Bildschirm, während die Anwendung nichts von diesem Ereignis erfährt und weiter läuft. War die Anwendung aus einem Konsolenfenster über das Werkzeug **java.exe** gestartet worden, erhält also 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 folgende Anweisung:

```
setDefaultCloseOperation(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 `SwApp` zu erzeugen, und endet folglich mit der Rückkehr des Konstruktoraufrufs:

```
public static void main(String[] args) {
  new SwApp();
}
```

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 in Java eine große Rolle spielt und im weiteren Kursverlauf noch ausführlich zur Sprache kommt. Ein Programm (ein Prozess) kann in mehrere *nebenläufige Ausführungsfäden* zerlegt werden, was bei Java-Programmen mit GUI automatisch geschieht.
Nachdem Sie ein Java-Programm aus einer Konsole gestartet haben, können Sie unter Windows mit der Tastenkombination **Strg+Unterbr** eine Liste seiner aktiven Threads anfordern.

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**[32] hat nichts dagegen.

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.

---

[32] Diesen zentralen Bestandteil der Java-Sicherheitsarchitektur können wir aus Zeitgründen nicht behandeln.

Um ein GUI-Programm per Anweisung zu beenden, muss die Methode **System.exit()** aufgerufen werden, was eine **JFrame**-Komponente aufgrund des oben vorgestellten Methodenaufrufs

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

auch automatisch erledigen kann.

10.6.6 Optionen zur Definition von Ereignisempfängern

In diesem Abschnitt lernen Sie u.a. zwei Prinzipien zur Klassendefinition kennen, die nicht nur bei der Ereignisbehandlung eingesetzt werden können: innere und anonyme Klassen.

10.6.6.1 Innere Klassen als Ereignisempfänger

Wer das vollständige Quellprogramm aufmerksam liest, wird feststellen, dass die `Ex`–Klassendefinition *innerhalb* der `SwApp`-Klassendefinition steht, was `Ex` zur **inneren Klasse** macht. Dank dieser Konstruktion können die privaten `SwApp`-Instanzvariablen in den `Ex`-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 inneren Klassen:

- In den Methoden der inneren Klasse kann man auf die Variablen und Methoden der äußeren Klasse zugreifen.
- Der Compiler erzeugt auch für jede innere Klasse eine eigene **class**-Datei, in deren Namen die Bezeichner für die innere und die umgebende Klasse eingehen, so dass im Beispiel der Name **SwApp$Ex.class** resultiert.
- Innere Klassen dürfen geschachtelt werden.
- Während für die bisher gewohnten Top-Level - Klassen nur der Zugriffsmodifikator **public** erlaubt ist, können bei inneren Klassen dieselben Zugriffsmodifikatoren verwenden werden wie bei Instanzvariablen und –methoden, also auch **private** und **protected**.

10.6.6.2 Anonyme Klassen als Ereignisempfänger

Nun haben wir unser Beispielprogramm fast vollständig durchleuchtet mit Ausnahme der Ereignisbehandlung zur Schaltfläche:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        numClicks++;
        label.setText(labelPrefix + numClicks);
        if (iconr < maxicon-1)
           iconr++;
        else
           iconr = 0;
        labicon.setIcon(icon[iconr]);
    }
});
```

Hier wird bei der Ereignisquelle `button` für die Ereignisklasse **ActionEvent** ein Empfänger registriert, wobei nicht nur das Objekt dynamisch erzeugt, sondern die gesamte Klasse ad hoc definiert wird.
Wie bei den inneren Klassen haben wir auch bei einer solchen **anonymen Klasse** die Möglichkeit, in den Ereignis-Behandlungsmethoden auf Instanzvariablen 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 der fehlende Klassenname durch den Namen der implementierten Schnittstelle oder der beerbten Basisklasse ersetzt wird. Es folgt ein Klassendefinitions-Block, der im Beispiel die implementierte **ActionListener**- Methode **actionPerformed()** enthält.
- Es kann nur eine einzige Instanz erzeugt werden.
  Werden mehrere Instanzen benötigt, ist die innere Klasse (siehe oben) vorzuziehen.
- Es sind keine Konstruktoren, keine statischen Methoden und keine statischen Variablen 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 **SwApp$1.class** resultiert.

10.6.6.3 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 gleich an einem Beispielprogramm zum Umgang mit Mausereignissen demonstriert.

10.6.7 Tastatur- und Mausereignisse

10.6.7.1 KeyEvent

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 {
  JLabel keyCode, uChar;

  KeyEventDemo() {
    super("KeyEvent-Demo");

    keyCode = new JLabel("KeyCode:");
    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("KeyCode: "+String.valueOf(e.getKeyCode()));
      }
      public void keyTyped(KeyEvent e) {
        uChar.setText("Unicode-Zeichen: "+String.valueOf(e.getKeyChar()));
      }
      public void keyReleased(KeyEvent e) {
        keyCode.setText("KeyCode: ");
        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:

Ein Tastendruck löst das Low Level - Ereignis **KEY_PRESSED** aus und bewirkt einen Aufruf der **KeyListener**-Methode **keyPressed()**, die ein **KeyEvent**-Objekt per Aktualparameter erhält. Wird eine Taste losgelassen, kommt das Low Level - Ereignis **KEY_RELEASED** zustande. Mit der **KeyEvent**-Methode **getKeyCode()** bringt man den virtuellen Key Code der betroffenen Taste in Erfahrung (siehe JDK-Dokumentation zur Klasse **KeyEvent**).

Eine Taste(nkombination) führt nur dann zu einem High Level - Ereignis vom Typ **KEY_TYPED** und damit zu einem Aufruf der **KeyListener**-Methode **keyTyped()**, wenn ihr ein Unicode-Zeichen entspricht. Dieses Zeichen kann dann mit der **KeyEvent**–Methode **getKeyChar()** abgerufen werden.

10.6.7.2 MouseEvent

Im folgendem Beispiel wird der Umgang mit dem **MouseEvent** sowie den zugehörigen Interfaces **MouseListener** und **MouseMotionListener**vorgeführt. Außerdem wird gezeigt (wie in Abschnitt 10.6.6.3 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-Interfaces erfüllt:

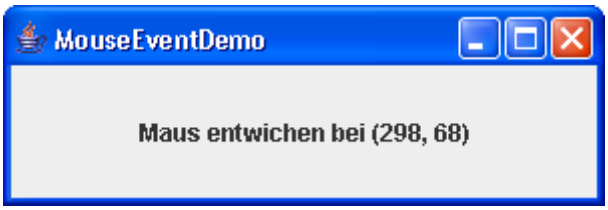
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MouseEventDemo extends JFrame implements MouseListener, MouseMotionListener {
   JLabel status = new JLabel();

   MouseEventDemo() {
      super("MouseEventDemo");
      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 können, z.B.:

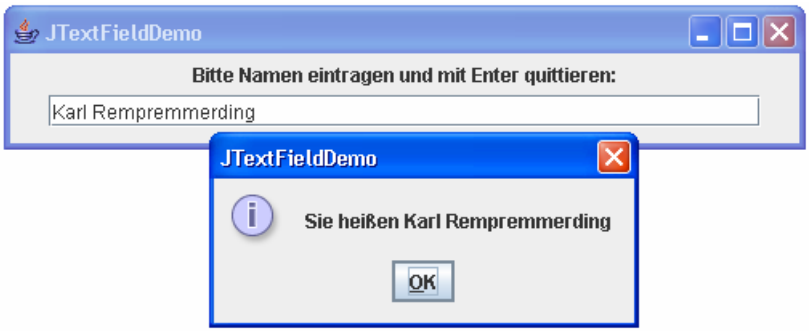


10.7 Swing-Komponenten (Teil 2)

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 2005) alle benötigten Informationen finden.

10.7.1 Einzeilige Textkomponenten

Im Rahmen der **JOptionPane**-Klassenmethode **showInputDialog()** (vgl. Abschnitt 10.6.6.2) konnten Sie schon eine einzeilige Textkomponente besichtigen. In einem **JFrame**-Fenster kann dieses elementare Bedienelement mit Hilfe einer **JTextField**-Komponente implementiert werden, z.B.:



Sobald der Benutzer die **Enter**-Taste drückt, während die **JTextField**-Komponente den Eingabefokus hat, wird ein **ActionEvent** ausgelöst. Im Beispiel präsentiert der im Rahmen einer anonymen Klasse (vgl. Abschnitt 10.6.6.2) realisierte Event Handler daraufhin einen Benachrichtigungs-Standarddialog (vgl. Abschnitt 7.7) mit dem erfassten Text, den er über die **JTextField**-Methode **getText()** ermittelt hat:

```
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 = "JTextFieldDemo";
   JTextFieldDemo() {
      super(titel);
      Container cont = getContentPane();
      cont.setLayout(new FlowLayout());

      name.addActionListener(
         new ActionListener() {
               public void actionPerformed(ActionEvent e) {
                  JOptionPane.showMessageDialog(((JComponent)e.getSource()).getParent(),
                   "Sie heißen "+name.getText(), titel, JOptionPane.INFORMATION_MESSAGE);
               }
            });

      cont.add(label);
      cont.add(name);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(500, 90);
      setVisible(true);
   }

   public static void main(String[] args) {
      new JTextFieldDemo();
   }
}
```

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.

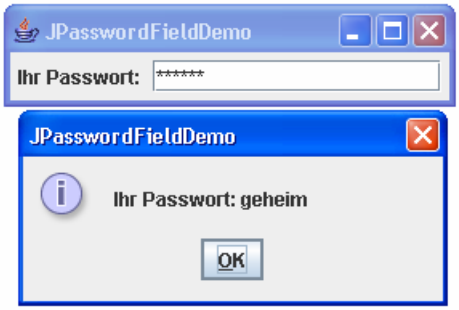
Statt der voreingestellten linksbündigen Ausrichtung der Textfeldinhalte kann mit der **JTextField**-Methode **setHorizontalAlignment()** auch eine zentrierte oder rechtsbündige Ausrichtung gewählt werden, z.B.:

```
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. Sie wird dann benutzt wie eine **JLabel**-Komponente, unterscheidet sich von dieser jedoch im Design.

Zum Erfassen von Passwörtern steht in Java die Swing-Komponente **JPasswordField** bereit, die im Unterschied zu **JTextField** für jedes eingegebene Zeichen ein Sternchen anzeigt, z.B.:



Das erfasste Passwort kann mit der **JPasswordField**-Methode **getPassword()** als **char**-Array extrahiert werden, was im Event Handler des Beispielprogramms zur Weiterverwendung in einem **String**-Konstruktor geschieht:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JPasswordFieldDemo extends JFrame {
   JPasswordField pw = new JPasswordField(16);
   JLabel label = new JLabel("Ihr Passwort: ");
   final static String titel = "JPasswordFieldDemo";

   JPasswordFieldDemo() {
      super(titel);
      Container cont = getContentPane();
      cont.setLayout(new FlowLayout());

      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);
               }
            });

      cont.add(label);
      cont.add(pw);

      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      pack();
      setVisible(true);
   }

   public static void main(String[] args) {
      new JPasswordFieldDemo();
   }
}
```

Ansonsten bringt das letzte 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, es möge seine Größe passend zu den enthaltenen Komponenten einrichten.

10.7.2 Kontrollkästchen und Optionsfelder

In diesem Abschnitt werden zwei **Umschalter** vorgestellt:

- 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:

Die beiden Kontrollkästchen (`bold` und `italic` genannt) werden aus Layout-Gründen in einem eigenen **JPanel**-Container untergebracht, der seinerseits am linken Rand des **JFrame**-Fensters sitzt:

```
JPanel cbPanel, rbPanel;
JCheckBox bold, italic;
   . . .
Container cont = getContentPane();
cont.setLayout(new GridLayout(1, 3));
cbPanel = new JPanel();
cbPanel.setLayout(new GridLayout(0, 1));
cont.add(cbPanel);

bold = new JCheckBox("Fett");
italic = new JCheckBox("Kursiv");
cbPanel.add(bold);
cbPanel.add(italic);
CheckHandler cbHandler = new CheckHandler();
bold.addItemListener(cbHandler);
italic.addItemListener(cbHandler);
```

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 internen Klasse `CheckHandler` als **ItemEvent**-Empfänger:

```
class CheckHandler implements ItemListener {
  public void itemStateChanged(ItemEvent e) {
    Font oldFont = bsp.getFont(), newFont = null;
    int oldStyle = oldFont.getStyle(), newStyle = 0;

    if (e.getSource() == bold) {
      if (e.getStateChange() == ItemEvent.SELECTED) {
        newStyle = oldStyle + Font.BOLD;
      } else {
        newStyle = oldStyle - Font.BOLD;
      }
    } else if (e.getSource() == italic) {
      if (e.getStateChange() == ItemEvent.SELECTED) {
        newStyle = oldStyle + Font.ITALIC;
      } else {
        newStyle = oldStyle - Font.ITALIC;
      }
    }

    newFont = oldFont.deriveFont(newStyle);
    bsp.setFont(newFont);
  }
}
```

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** vertreten sind:

- Die **Component**-Methode **getFont()** stellt die Schriftart einer Komponente fest.
- Die **Font**-Methode **getStyle()** ermittelt den Stil (Schnitt) einer Schriftart, wobei **PLAIN**, **BOLD**, **ITALIC** und **BOLD+ITALIC** in Frage kommen:

Font-Konstante	**int**-Wert
PLAIN	0
BOLD	1
ITALIC	2

- Mit den diversen Überladungen der **Font**-Methode **deriveFont()** gewinnt man eine neue Schriftart als Variante des angesprochenen **Font**-Objekts. Alle nicht per Aktualparameter modifizierten Eigenschaften des angesprochenen Objekts werden übernommen. Man kann z.B. bequem eine neue Schrift erzeugen, die sich von einer bestimmten alten Schrift nur durch die Größe unterscheidet.
- Mit der **JComponent**-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()**.

Auch die drei Optionsschalter haben einen gemeinsamen **ItemListener**. Zudem sorgt ein Objekt aus der Klasse **ButtonGroup** dafür, dass stets nur *ein* Gruppenmitglied aktiviert ist:

```
JRadioButton arial, tiro, courier;
  . . .
courier = new JRadioButton("Courier", false);
tiro = new JRadioButton("Times Roman", true);
arial = new JRadioButton("Arial", false);
rbPanel.add(courier);
rbPanel.add(tiro);
rbPanel.add(arial);

rbGroup = new ButtonGroup();
rbGroup.add(courier);
rbGroup.add(tiro);
rbGroup.add(arial);

RadioHandler rbHandler = new RadioHandler();
courier.addItemListener(rbHandler);
tiro.addItemListener(rbHandler);
arial.addItemListener(rbHandler);
```

Auch im **ItemEvent**-Handler der Optionsschalter kommen die oben vorgestellten Schriftartenverwaltungsmethoden zum Einsatz:

```
class RadioHandler implements ItemListener {
  public void itemStateChanged(ItemEvent e) {
    int style = bsp.getFont().getStyle();
    Font oldFont = null;
    if (e.getSource() == courier) {
      oldFont = courierFont;
    } else if (e.getSource() == tiro) {
      oldFont = tiroFont;
    } else if (e.getSource() == arial) {
      oldFont = arialFont;
    }
    bsp.setFont(oldFont.deriveFont(style));
  }
}
```

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

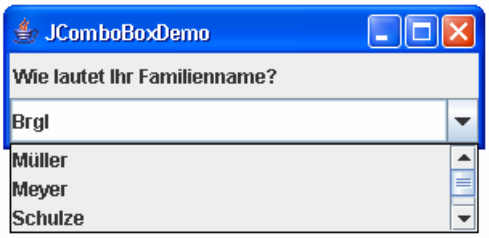
…\BspUeb\Swing\Umschalter

10.7.3 Kombinationsfelder

Die **JComboBox**-Komponente bietet eine Kombination aus einem einzeiligen Textfeld und einer Liste, von der normalerweise nur ein Element sichtbar ist. Um eine 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

In folgendem Programm wird die Angabe des Familiennamens durch eine Liste mit den häufigsten Namen erleichtert:



Per Voreinstellung funktioniert ein **JComboBox**–Objekt als reine DropDown-Liste, bietet also *kein* Texteingabefeld. Dies wird im Beispiel mit dem Methodenaufruf **setEditable(true)** geändert. Außerdem wird mit der Methode **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 {
   JComboBox familienName;
   JLabel prompt;
   final String[] auswahl = {"Müller", "Meyer", "Schulze", "Schmitt"};
   final static String titel = "JComboBoxDemo";

   JComboBoxDemo() {
      super(titel);
      Container cont = getContentPane();
      cont.setLayout(new GridLayout(0,1));

      prompt = new JLabel(" Wie lautet Ihr Familienname?");
      familienName = new JComboBox(auswahl);
      familienName.setMaximumRowCount(3);
      familienName.setEditable(true);
      familienName.setSelectedItem("");

      familienName.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);
              }
            }
         });

      cont.add(prompt);
      cont.add(familienName);
```

```
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(300, 90);
      setVisible(true);
   }

   public static void main(String[] args) {
      new JComboBoxDemo();
   }
}
```

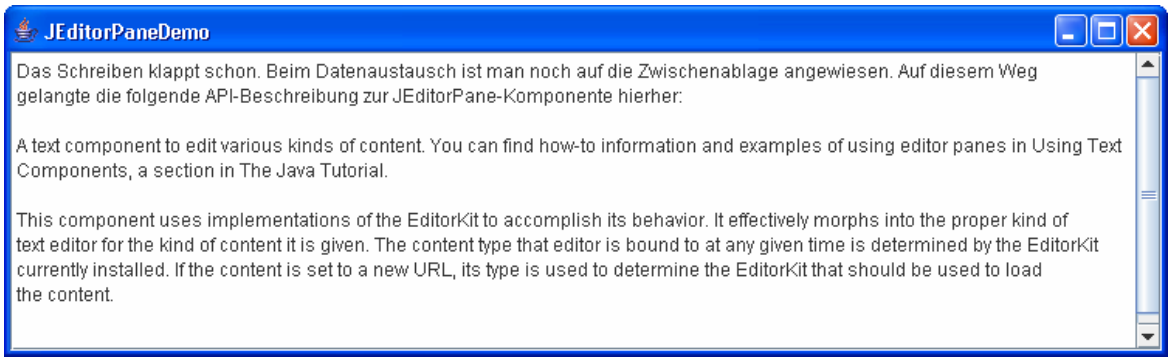
Ein **ItemEvent**-Handler wird bei jeder Selektion *und* bei jeder Deselektion aufgerufen. Im Beispiel wird über den logischen Ausdruck

```
e.getStateChange() == ItemEvent.SELECTED
```

mit Hilfe der **ItemEvent**-Methode **getStateChange()** verhindert, dass der Informations-Standarddialog bei einer Deselektion erscheint.

10.7.4 Ein (fast) kompletter Editor als Swing-Komponente

Die Swing-Komponente **JTextField** hat noch eine große Schwester namens **JEditorPane**, die einen recht brauchbaren Texteditor implementiert:



Für die Rohversion des Editors, die immerhin schon den Datenaustausch via Zwischenablage beherrscht, muss man sehr wenig Aufwand betreiben:

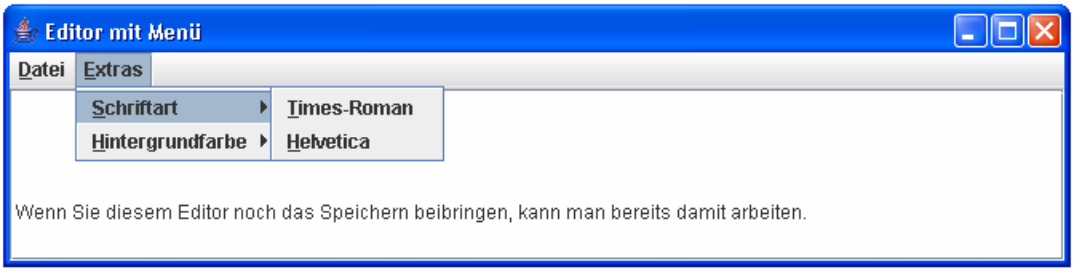
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Editor extends JFrame{
   private JScrollPane roll;
     private JEditorPane text;

   public Editor() {
      super("JEditorPaneDemo");

      text = new JEditorPane();
      roll = new JScrollPane(text);
      getContentPane().add(roll, BorderLayout.CENTER);

      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setSize(300, 200);
      setVisible(true);
   }
}

class Prog {
   public static void main(String[] arg) {
      new Editor();
   }
}
```

Zu erwähnen ist noch, dass die **JEditorPane**-Komponente **Rollbalken** erhält, indem sie in eine in eine **JScrollPane**-Komponente eingepackt wird.

Im nächsten Abschnitt werden wir den Editor um ein Menü mit wichtigen Funktionen erweitern.

10.7.5 Menüs

Um das Potential der **JEditorPane**-Komponente besser auszuschöpfen, erweitern wir das in Abschnitt 10.7.4 begonnene Programm um ein Datei- und ein Extras-Menü:



Als **Menüzeile** verwenden wir einen Spezial-Container der Klasse **JMenuBar**, der im Norden der **ContentPane** unseres **JFrame**-Fensters untergebracht wird:

```
private JMenuBar menueZeile;
  . . .
menueZeile = new JMenuBar();
getContentPane().add(menueZeile, BorderLayout.NORTH);
```

Die Menüs werden als Komponenten der Klasse **JMenu** erstellt, bei Bedarf mit einer **Alt**-Tastenkombination zum schnellen Öffnen versehen und dann auf der Menüzeile untergebracht, wobei man sich auf den voreingestellten Layout-Manager dieses Spezial-Containers verlassen kann, z.B.:

```
private JMenu fileMenue, extrasMenue, fontMenue, farbMenue;
  . . .
extrasMenue = new JMenu("Extras");
extrasMenue.setMnemonic('E');
menueZeile.add(extrasMenue);
```

Eine **JMenu**-Komponente kann über ihre **add()**-Methode andere **JMenu**-Komponenten als Untermenüs aufnehmen, z.B.:

```
fontMenue = new JMenu("Schriftart");
fontMenue.setMnemonic('S');
extrasMenue.add(fontMenue);
```

Menüeinträge, die keine Untermenüs darstellen, werden über Komponenten der Klasse **JMenuItem** realisiert. Diese Klasse stammt von **AbstractButton** ab und löst **ActionEvents** aus.
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('T');
fontMenue.add(timesItem);
```

Für jedes Menüitem wird ein **ActionListener** registriert, z.B.:

```
timesItem.addActionListener(new
  ActionListener() {
    public void actionPerformed(ActionEvent e) {
      text.setFont(new Font("TimesRoman", Font.PLAIN, 14));
    }
  });
```

10.7.6 Dateiauswahldialog

Im **ActionEvent**-Handler zum Menüitem

Datei > Öffnen

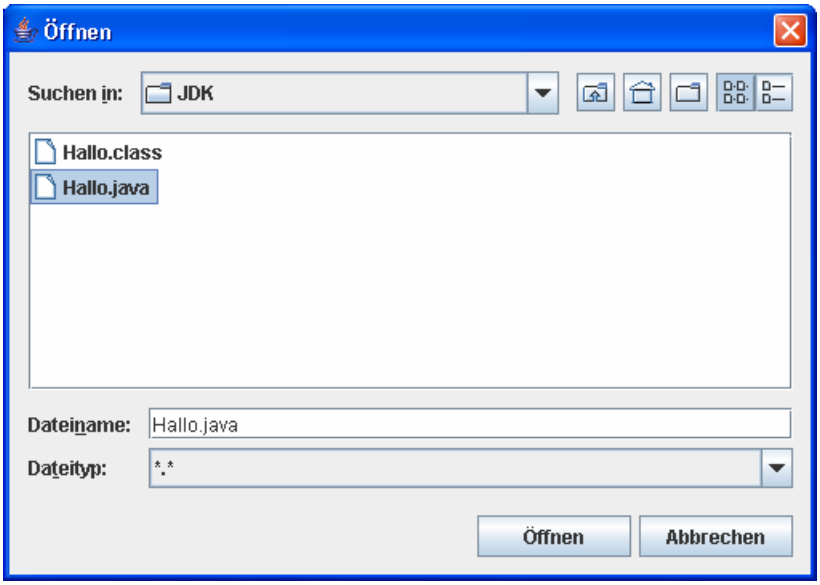
unseres Editors

```
openItem.addActionListener(new
  ActionListener() {
    public void actionPerformed(ActionEvent e) {
      openFile();
      if (datei != null) readText();
    }
  });
```

wird eine private Methode `openFile()` aufgerufen, die für den Standarddialog zum Öffnen einer Datei eine Komponente der Klasse **JFileChooser** einspannt:

```
private void openFile() {
  JFileChooser fc = new JFileChooser();
  int returnCode = fc.showOpenDialog(this);
  if (returnCode == JFileChooser.APPROVE_OPTION)
    datei = fc.getSelectedFile();
  else
    datei = null;
}
```

Das Ergebnis kann sich sehen lassen:



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

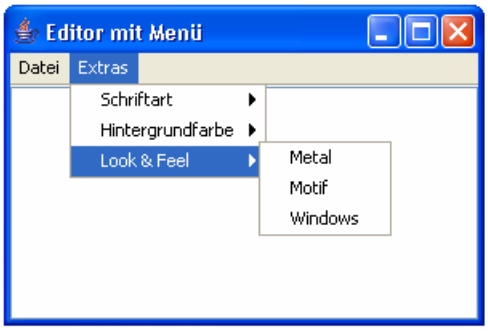
…\BspUeb\Swing\Editor\Mit Menü

10.8 Look & Feel umschalten

Wer den in Abschnitt 10.7.6 gezeigten Dateiauswahldialog lieber im aktuellen Windows-Design erleben möchte, kann die Swing-Option nutzen, das Look & Feel einer Anwendung zwischen folgenden Alternativen umzuschalten:

- **Metal** (Java-Standard)
- **Motif** (ein verbreiteter X-11 – Window-Manager unter UNIX)
- **Windows**
- **Windows - klassisch**

Bevor man ein solches Look & Feel - Menü präsentieren kann, ist etwas Arbeit angesagt.



Neben dem `lookAndFeel`-Menü und den Menüitems (`metalItem`, `motifItem` und `windowsItem`) wird ein Array mit Elementen der Klasse **LookAndFeelInfo** benötigt, die innerhalb der Klasse **UIManager** definiert ist, so dass wir im Kurs erstmals zwischen zwei Klassennamen den Punktoperator verwenden:

```
private JMenu lookAndFeel;
private JMenuItem metalItem, motifItem, windowsItem;
private UIManager.LookAndFeelInfo lafs[];
```

Das Objekt zur Array-Referenzvariablen `lafs` liefert die **UIManager**-Methode **getInstalledLookAndFeels()**:

```
lafs = UIManager.getInstalledLookAndFeels();
```

Zur **ActionEvent**-Ereignisbehandlung für die Look & Feel – Menüitems wird die folgende interne Klasse definiert:

```
class UIChooser implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == metalItem) {
      selUI(0);
    } else if (e.getSource() == motifItem) {
      selUI(1);
    } else if (e.getSource() == windowsItem){
      selUI(2);
    } else if (e.getSource() == windowsClassicItem){
      selUI(3);
    }
  }
}
```

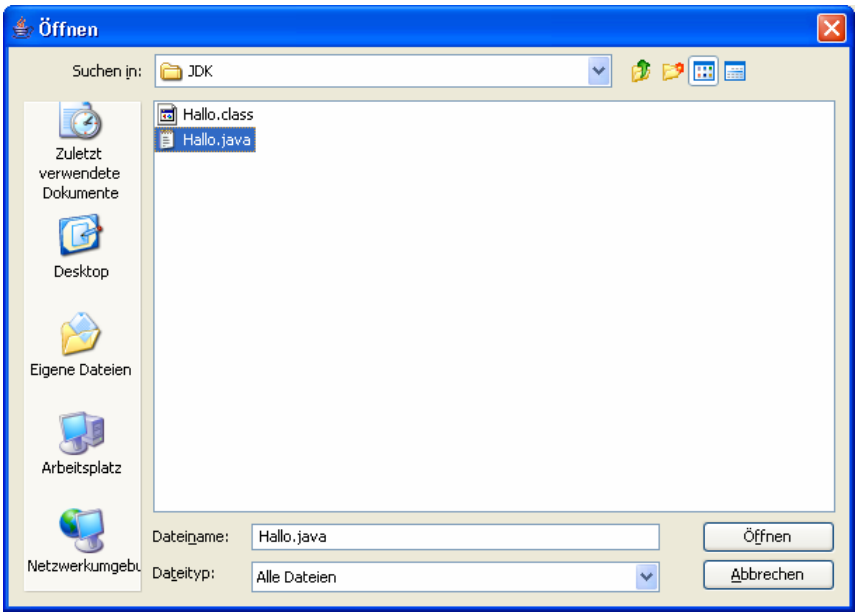
Im Konstruktor der Anwendungsklasse wird ein Objekt aus dieser Klasse erzeugt, das die Ereignisbehandlung für alle Look & Feel – Menüitems übernimmt:

```
UIChooser ui = new UIChooser();
lookAndFeel = new JMenu("Look & Feel");
lookAndFeel.setMnemonic('L');
extrasMenue.add(lookAndFeel);
metalItem = new JMenuItem("Metal");
metalItem.addActionListener(ui);
motifItem = new JMenuItem("Motif");
motifItem.addActionListener(ui);
windowsItem = new JMenuItem("Windows");
windowsItem.addActionListener(ui);
windowsClassicItem = new JMenuItem("Windows - klassisch");
windowsClassicItem.addActionListener(ui);
lookAndFeel.add(metalItem);
lookAndFeel.add(motifItem);
lookAndFeel.add(windowsItem);
lookAndFeel.add(windowsClassicItem);
```

Nachzutragen ist noch die im **ActionEvent**-Handler benutzte Methode `selUI()`:

```
private void selUI(int ui) {
   try {
      UIManager.setLookAndFeel(lafs[ui].getClassName());
      SwingUtilities.updateComponentTreeUI(this);
   } catch (Exception e) {}
   text.setBackground(Color.white);
}
```

Bei aktivem **Windows** - Look & Feel verwendet der Editor den folgenden Dateiauswahldialog:



Den vollständigen Quellcode zur aktuellen Ausbaustufe des Beispielprogramms finden Sie im Ordner

…\BspUeb\Swing\Editor\Mit L&F

10.9 Übungsaufgaben zu Abschnitt 10

1) Ermitteln Sie mit Hilfe eines kurzen Programms die Event IDs zu folgenden Ereignissen:

- Eine Schaltfläche wurde betätigt.
- Über einer Komponente wurde eine Maustaste gedrückt.
- Die über einer Komponente gedrückte Maustaste wurde losgelassen.
- Eine Taste wurde gedrückt bzw. losgelassen.
- Ein Fenster wurde aktiviert.

Sie werden feststellen, dass alle 2 bzw. 3 Maustasten dieselben Ereigniskennungen liefern. Mit Hilfe der **MouseEvent**-Methode **getButton()** kann man die Tasten aber doch unterscheiden.

2) Erstellen Sie ein Tastatur-Demonstrationsprogramm, das für jede gedrückte Taste(nkombination) ausgibt:

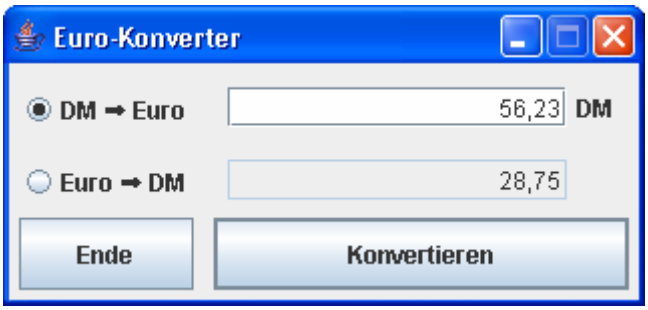
- Key Code der zuletzt gedrückten Taste
- Unicode-Zeichen (falls definiert)
- gedrückte Modifikator-Tasten (Umschalt, Steuerung, Alt)

So ähnlich sollte Ihr Programm z.B. auf die Tastenkombination **Umschalt+x** reagieren:

Verwenden Sie bitte zur Anordnung der Komponenten ein **GridLayout** mit 3 Zeilen und 2 Spalten.

3) Zu einer anonymen Klasse lässt sich nur *eine* Instanz erzeugen. Es ist aber durchaus möglich, ein solches Objekt 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.
Wenngleich die beschriebene Konstellation keine nennenswerten Vorteile bietet, wird doch die Routine beim Umgang mit Ereignissen und anonymen Klassen durch die Anfertigung eines Beispielprogramms (z.B. mit zwei Befehlsschaltern) gefördert.

4) Schreiben Sie einen Euro-DM-Konverter, der in etwa folgende Benutzeroberfläche bietet:



Für den Lösungsvorschlag wurden einige im Manuskript nicht behandelte Techniken eingesetzt:

- Für den mit **Konvertieren** beschrifteten Befehlsschalter wurde mit folgendem Methodenaufruf (gerichtet an die **RootPane**-Schicht des **JFrame**-Fensters) festgelegt, dass sich die **Enter**-Taste an ihn richten soll:

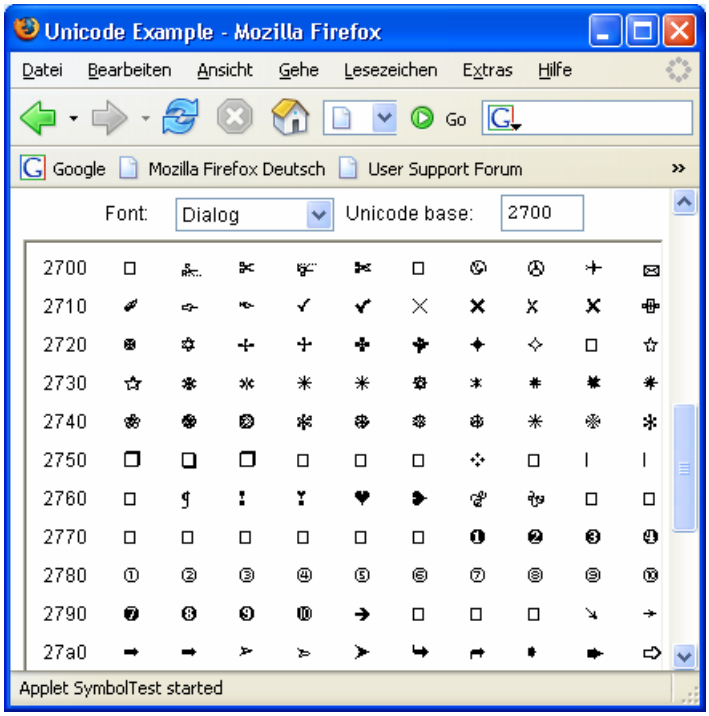
```
getRootPane().setDefaultButton(konvert);
```

- Außerdem wurden Änderungen in der Eingabefokus-Verwaltung vorgenommen, die im Bildschirmphoto nicht sichtbar, aber für den Benutzer recht nützlich sind:

- Beim Start erhält die obere **JTextField**-Komponente den Fokus über den Methodenaufruf:
  ```
  eingabeFeld.requestFocus();
  ```
- Die seltener benötigten Komponenten werden aus der Fokussequenz herausgenommen, z.B.:
  ```
  euro2dm.setFocusable(false);
  ```
  Leider ist die Methode **setFocusable()** erst ab der Java-Version 1.4 verfügbar.
- Der mit **Konvertieren** beschrifteten Befehlsschalter gibt den Fokus spontan weiter (an das Eingabefeld):
  ```
  konvert.transferFocus();
  ```

Weitere Hinweise zum Lösungsvorschlag:

- Die beiden **JPanel**-Komponenten, die den linken bzw. rechten Teil des Fensters besetzen, sind aus ästhetischen Gründen jeweils mit einem Rahmen versehen worden, z.B.:
  ```
  panLinks.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
  ```
- 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 des Optionsfeldes 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);
  ```
  Als Unicode-Browser kann das mit dem JDK ausgelieferte Applet **SymbolTest** wertvolle Dienste leisten:



Ausgehend vom JDK-Wurzelverzeichnis ist es folgendermaßen zu finden:

...\docs\guide\awt\demos\symboltest\actual

11 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 zurealisieren, doch spätestens beim Verlassen des Programms gehen alle Variableninhalte verloren.
In diesem Abschnitt behandeln wir elementare Verfahren zum sequentiellen Datenaustausch zwischen den Variablen eines Java-Programms und externen Datenquellen bzw. –senken, z.B.:

- Primitive Werte, Zeichen 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

Damit werden u.a folgende Themen *nicht* angesprochen:

- Alternative Datenquellen bzw. -senken (z.B. Netzwerkverbindungen, Pipes)
- Wahlfreier Dateizugriff und Datenbanktechniken

Die 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.*;
```

Im Bemühen um eine umfassende und systematische Behandlung der diversen Ein-/Ausgabeproblemstellungen hat sich eine stattliche und auf den ersten Blick recht unübersichtliche Anzahl von Ein-/Ausgabe (EA) - Klassen ergeben. Die folgende Einschätzung von Eckel (2002, TIJ314.htm) ist als Warnung und vorsorglicher Trost gedacht:

> As a result there are a fair number of classes to learn before you understand enough of Java's I/O picture that you can use it properly. In addition, it's rather important to understand the evolution history of the I/O library, even if your first reaction is "don't bother me with history, just show me how to use it!" The problem is that without the historical perspective you will rapidly become confused with some of the classes and when you should and shouldn't use them.

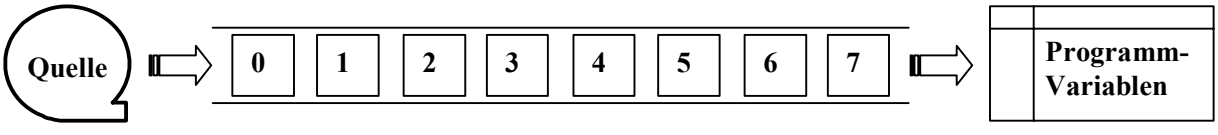
Wie schon im bisherigen Verlauf des Kurses kann auch im aktuellen Abschnitt nur ein erster Eindruck, bestenfalls ein Überblick vermittelt werden. Bei konkreten Aufgaben können Sie z.B. die JDK-Dokumentation konsultieren, die umfassende Informationen über alle EA-Klassen bietet.

11.1 Grundprinzipien

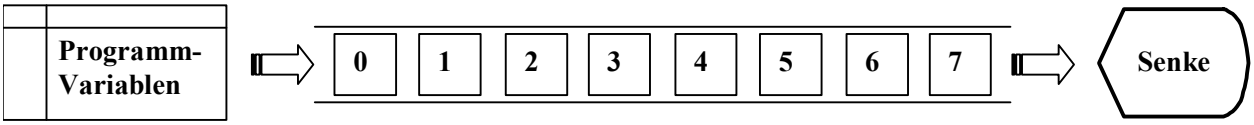
11.1.1 Datenströme

In Java wird die sequentielle Datenein- und –ausgabe über so genannte **Streams** abgewickelt:

Ein Programm **liest** Daten aus einem **Eingabestrom**, der aus einer Datenquelle (z.B. Datei, Eingabegerät, Netzwerkverbindung) gespeist wird:



Ein Programm **schreibt** Daten in einen **Ausgabestrom**, der die Werte von Programmvariablen zu einer Datensenke befördert (z.B. Datei, Ausgabegerät, Netzverbindung):



Ein- bzw. Ausgabeströme werden in Java-Programmen durch Objekte aus geeigneten Klassen des Paketes **java.io** repräsentiert, wobei die Auswahl u.a. von der angeschlossenen Datenquelle bzw. –senke sowie vom Typ der zu transportierenden Daten abhängt.

Datenquellen bzw. –senken können sich innerhalb und außerhalb eines Java-Programms befinden:

- *interne* Quellen bzw. Senken:
  - **byte**-Arrays
  - **char**-Arrays
  - **Strings**
  - Pipes
    Sie dienen der Kommunikation zwischen Threads (siehe unten).
- *externe* Quellen bzw. Senken:
  - Dateien
  - Geräte
  - Netzverbindungen

Ziel des Java-Datenstromkonzeptes ist es, Ein- und Ausgaben möglichst unabhängig von den Besonderheiten konkreter Datenquellen und –senken formulieren zu können.

Nach so vielen allgemeinen bzw. abstrakten Bemerkungen wird es Zeit für ein konkretes Beispiel. Das folgende Programm schreibt zunächst einen **byte**-Array in eine Datei und liest die Daten anschließend wieder zurück:

Quellcode	Ausgabe

```
import java.io.*;

class Einstieg {
  public static void main(String[] args) {
    String name = "demo.txt";
    byte[] arr = {0,1,2,3,4,5,6,7};

    // byte-Array in den Ausgabestrom schreiben
    try {
      FileOutputStream fos = new FileOutputStream(name);
      fos.write(arr);
      fos.close();
    } catch (Exception e) { //Ausnahmebehandlung
    }

    // Eingabestrom öffnen und byteweise in arr einlesen
    try {
      FileInputStream fis = new FileInputStream(name);
      fis.read(arr);
      fis.close();
    } catch (Exception e) { //Ausnahmebehandlung
    }

    for (int i : arr)
      System.out.println(i);
  }
}
```

```
0
1
2
3
4
5
6
7
```

Es benutzt dazu einen Aus- und einen Eingabestrom, die durch Objekte aus geeigneten Klassen des Paketes **java.io** realisiert werden (**FileOutputStream** und **FileInputStream**).

11.1.2 Taxonomie der Stromverarbeitungsklassen

Das Paket **java.io** enthält vier abstrakte Basisklassen, von denen die für uns relevanten Stromverarbeitungsklassen abstammen:

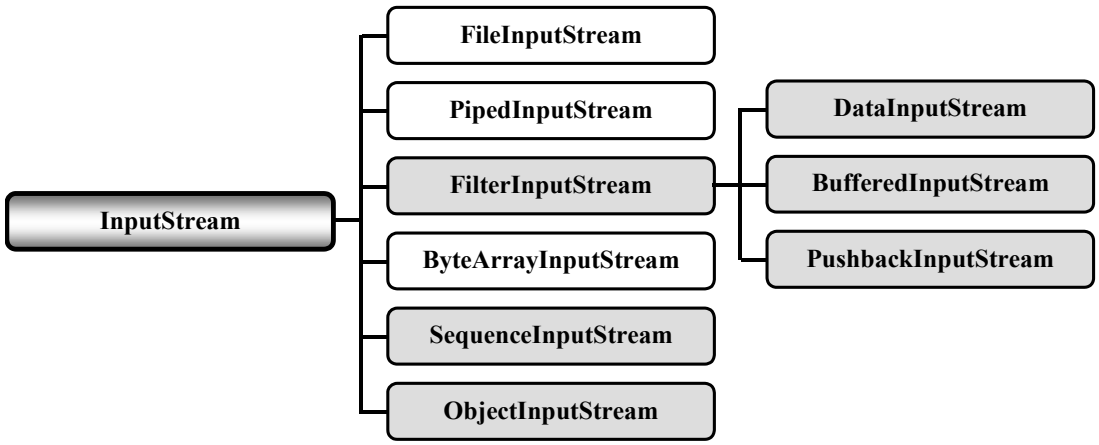
- **InputStream** und **OutputStream**
  Die Klassen aus den zugehörigen Hierarchien verarbeiten **Ströme mit Bytes**[33] als Elementen. Byteströme werden für das Lesen und Schreiben von **binären Daten** verwendet (z.B. Bitmap-Graphiken).
- **Reader** und **Writer**
  Die Klassen aus den zugehörigen Hierarchien verarbeiten **Zeichenströme**, die aus einer Folge von Zeichen in einer bestimmten Kodierung (meist 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 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 mal einen Blick auf die **InputStream**–Hierarchie. In der folgenden Abbildung sind die Eingabeklassen mit einen weißen, und die Eingabetransformationsklassen mit einem grauen Hintergrund dargestellt:



[33] Wenn in Abschnitt 11 der Namensteil *Byte* auftaucht, ist keine Java-Wrapper-Klasse gemeint, sondern eine 8 Bit umfassende Informationseinheit der Datenverarbeitung.

Die abgeleiteten Klassen vervollständigen das Protokoll von **InputStream**, d.h. sie implementieren die dort teilweise abstrakten Methoden. Folglich bieten sie über *dasselbe Protokoll* Zugang zu verschiedenen Datenquellen.

11.1.3 Zum guten Schluss

Allen Java-Datenstromklassen gemeinsam ist die Methode **close()**, welche einen Strom schließt und alle damit assoziierten Ressourcen frei gibt. Ein explizites **close()** ist in manchen Fällen überflüssig, weil die vom Garbage Collector oder beim Beenden des Programms ausgeführte Methode **finalize()** einen **close()**-Aufruf enthält (z.B. bei den Klassen **FileInputStream** und **FileOutputStream**). Es gibt jedoch gute Gründe, **close()** explizit aufzurufen:

- Viele Ausgabestromklassen setzen **Puffer** ein, die unbedingt vor dem Entfernen der Datenstromobjekte geleert werden müssen, z.B. durch einen **close()**-Aufruf. Viele betroffene Klassen (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, und es gehen in der Regel gepufferte Daten verloren (siehe z.B. Abschnitt 11.3.1.4).
- Nicht mehr benötigte Ströme sollten so früh wie möglich geschlossen werden, um die assoziierten Ressourcen (z.B. Dateien) für andere Programme oder für andere Aufgaben im selben Programm frei zu geben.
- Viele Methoden zur Dateiverwaltung (z.B. das Umbenennen) sind bei geöffneten Dateien nicht anwendbar.

Um allen Problemen aus dem Weg zu gehen, schließt man jeden Strom so früh wie möglich durch einen expliziten **close()**-Aufruf.

11.1.4 Aufbau und Verwendung der Transformationsklassen

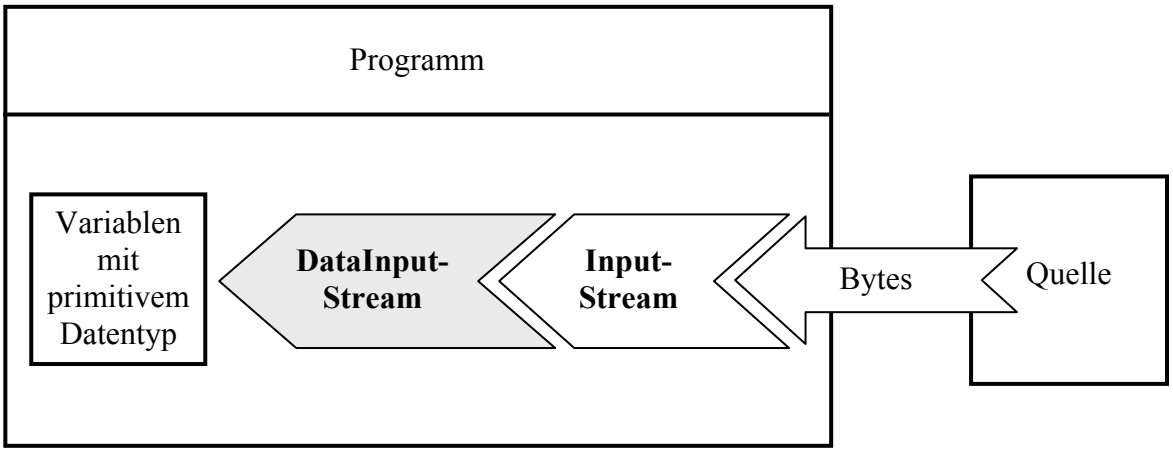
Eine Transformationsklasse baut auf einer Ein- bzw. Ausgabeklasse auf und stellt Methoden für eine erweiterte Funktionalität zur Verfügung. Wie diese Zusammenarbeit organisiert wird, betrachten wir am Beispiel der byteorientierten Eingabe-Transformationsklasse **DataInputStream**.

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.

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 4 Bytes) aus. Aus dem Bytestrom wird ein Strom mit Elementen vom gewünschten primitiven Typ:



Wird für ein **DataInputStream**–Objekt die **close()**-Methode aufgerufen, dann leitet es diese Botschaft an das verbundene **InputStream**-Objekt weiter.

Im folgenden Beispielprogramm kooperiert ein **DataInputStream** mit der byteorientierten Eingabeklasse **FileInputStream**, um **int**-Werte aus einer Datei zu lesen.
Zuvor werden diese **int**-Werte in dieselbe Datei *geschrieben*, wobei das Gespann aus der Ausgabe-Transformationsklasse **DataOutputStream** und der byteorientierten Ausgabeklasse **FileOutputStream** zum Einsatz kommt. Hier zerlegt der Filter die **int**-Werte in einzelne Bytes und schiebt sie in den Ausgabestrom.

Quellcode

```
import java.io.*;

class DataInOutputStream {
   public static void main(String[] egal) {
      String name = "demo.txt";
      int[] arr = {1024,2048,4096,8192};

      // int-Array in den Ausgabestrom schreiben
      try {
         FileOutputStream fos = new FileOutputStream(name);
         DataOutputStream dos = new DataOutputStream(fos);
         for (int i=0; i<arr.length; i++)
            dos.writeInt(arr[i]);
         dos.close();
      } catch (Exception e) {//Ausnahmebehandlung
      }

      // Eingabestrom öffnen und int-Werte einlesen
      try {
         FileInputStream fis = new FileInputStream(name);
         DataInputStream dis = new DataInputStream(fis);
         for (int i=0; i<arr.length; i++)
            arr[i] = dis.readInt();
         dis.close();
      } catch (Exception e) {//Ausnahmebehandlung
      }

      for (int i : arr)
         System.out.println(i);
   }
}
```

Ausgabe

```
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** mit dem Beschaffen von Bytes in hinreichender Menge. Dann werden Bytes aus dem Eingabestrom entnommen und zu Werten eines primitiven Datentyps zusammengesetzt.
- **DataInputStream**-Objekte können mit jedem **InputStream**-Objekt kooperieren. Bei Lesen von primitiven Datenwerten aus einer Datei kann man die Eingabeklasse **FileInputStream** verwenden.
- Ein Aufruf der **DataInputStream**-Methode **close()** wird an das verbundene **InputStream**-Objekt durchgereicht.

11.2 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 wichtige Methoden dieser Klasse werden im weiteren Verlauf des aktuellen Abschnitts anhand von Codefragmenten aus dem folgenden Rahmenprogramm vorgestellt:

```
import java.io.*;

class FileDemo {
   public static void main(String[] args) {
      byte[] arr = {1, 2, 3};
   . . .
   }
}
```

Während beim Erzeugen eines Objekts aus der Klasse **FileOutputStream** die zugeordnete Datei in der Regel automatisch geöffnet wird, führt das Erzeugen eines **File**-Objekts *nicht* zum Öffnen einer Datei.

11.2.1 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(dname+" existiert, ist aber kein Verzeichnis.");
      System.exit(1);
   }
}
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).

11.2.2 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 statische 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);
   }
}
```

11.2.3 Informationen über Dateien und Ordner

Neben **isDirectory()** kennen **File**-Objekte noch weitere Informations-Methoden, z.B.:

- **String getAbsolutePath()**
  Absoluten Pfadnamen feststellen
- **long length()**
  Dateigröße in Bytes feststellen
- **boolean canWrite()**
  Prüft, ob das Programm schreibend zugreifen darf

Im Beispiel 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());
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
  Groesse in Bytes:   3
  Schreiben moeglich: true
```

11.2.4 Verzeichnisinhalte auflisten

Im folgenden Codefragment wird ein namenloses **File**-Objekt mit der Botschaft **listFiles()** beauftragt, für jeden Eintrag im aktuellen Verzeichnis ein Element im **File**-Array `names` anzulegen:

```
File names[] = new File(".").listFiles();
System.out.println("Dateien im akt. Verzeichnis:");
for (File fi : names)
  System.out.println(" "+fi.getName());
names = new File(".").listFiles(new FileFilter("java"));
System.out.println("\nDateien im akt. Verzeichnis mit Extension .java:");
for (File fi : names)
  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
 FileDemo.jcp
 FileDemo.jcw
 FileFilter.class
 FileFilter.java
 src_filedemo.txt
```

Eine alternative **listFiles()**-Überladung liefert eine *gefilterte* Liste mit **File**-Verzeichniseinträgen, benötigt dazu aber 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()** überschreiben.

11.2.5 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 mit keinem offenen Ein- oder Ausgabestrom-Objekt verbunden sein.

11.2.6 Löschen

Mit der **File**-Methode **delete()** löscht man eine Datei bzw. einen Ordner:

```
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);
}
```

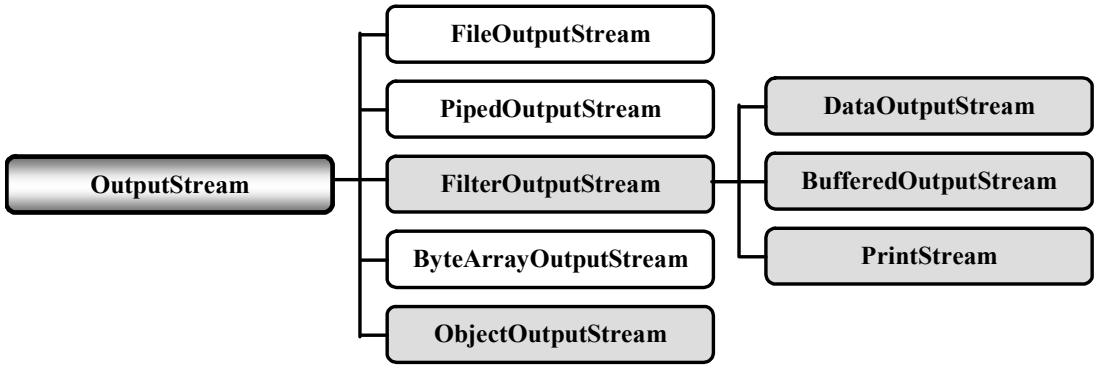
11.3 Klassen zur Verarbeitung von Byteströmen

In Java 1.0 stammten *alle* Ein-/Ausgabeklassen von **InputStream** oder **OutputStream** ab. Während sich diese Lösungen 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 byteorientierten Klassen adäquat. An einigen Stellen (z.B. bei der Standardausgabe) haben außerdem die alten Lösungen zur Zeichenverarbeitung überlebt.

11.3.1 Die OutputStream-Hierarchie

11.3.1.1 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:

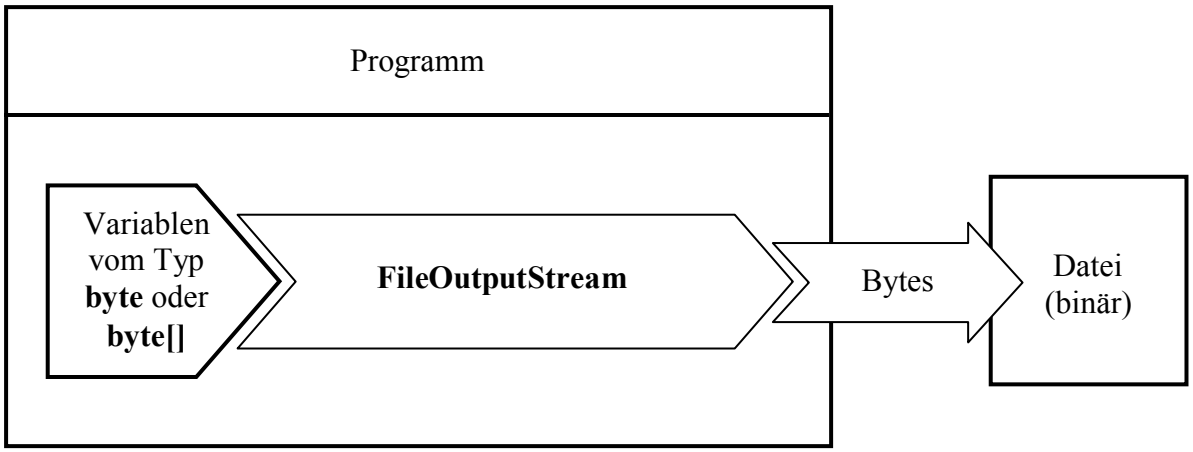


Mit der Transformationsklasse **ObjektOutputStream** können komplette Objekte in einen byteorientierten Ausgabestrom geschrieben werden. Sie wird zusammen mit ihrem Gegenstück **ObjektInputStream** in Abschnitt 11.6 behandelt.
Die folgenden **OutputStream**-Unterklassen werden in diesem Manuskript *nicht* 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.

11.3.1.2 FileOutputStream

Ein **FileOutputStream**-Objekt ist mit einer Datei verbunden, die vom Konstruktor im Schreibmodus geöffnet wird. Die verfügbaren **write()**-Methoden befördern die Inhalte von **byte**–Variablen oder –Arrays in die Ausgabedatei, so dass **FileOutputStream**-Objekte oft mit Filterobjekten (z.B. aus der Klasse **DataOutputStream**) kombiniert werden, die reichhaltigere Ausgabemethoden bieten (siehe unten):



Im **FileOutputStream**-Konstruktor wird die anzusprechende Datei über ein **File**-Objekt (siehe Abschnitt 11.2) oder über einen **String** spezifiziert:

- **FileOutputStream**(**File** *file*)
- **FileOutputStream(File** *file***, boolean** *append***)**
- **FileOutputStream(String** *name***)**
- **FileOutputStream**(**String** *name*, **boolean** *append*)

Existiert eine Datei bereits, wird sie von den **FileOutputStream**-Methoden gnadenlos überschrieben, wenn man nicht im Konstruktor durch den **append**-Aktualparameter **true** dafür sorgt, dass die Ausgaben an den vorhandenen Inhalt *angehängt* werden.

Obwohl wir die Klasse **FileOutputStream** in den Abschnitten 11.1.1 und 11.1.4 bereits eingesetzt haben, folgt noch ein kleines Beispielprogramm zum *Kopieren* von Dateien:

```
import java.io.*;

class FileCopy {
  public static void main(String[] args) {
    String quelle = "quelle.dat", ziel = "ziel.dat";
    final int buflen = 1048576;
    byte[] buffer = new byte[buflen];
    int nread;
    long total = 0;
    long zeit = System.currentTimeMillis();
    try {
      FileInputStream  fis = new FileInputStream(quelle);
      FileOutputStream fos = new FileOutputStream(ziel);
      System.out.println("Kopieren von "+quelle+" in "+ziel+
                         " startet (* = "+buflen+" Bytes):");
```

```
        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("*");
          }
        }
        fis.close();
        fos.close();
      } catch (IOException e) {
        System.out.println(e);
        System.exit(1);
      }
      zeit = System.currentTimeMillis() - zeit;
      System.out.println("\nEs wurden "+total+" Bytes kopiert. "+
                         "(Benoetigte Zeit: "+zeit+" Millisekunden.)");
    }
  }
```

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 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.
Während die Ausgabedatei nötigenfalls automatisch neu angelegt wird, führt eine fehlende Eingabedatei zur folgenden Ausnahme:

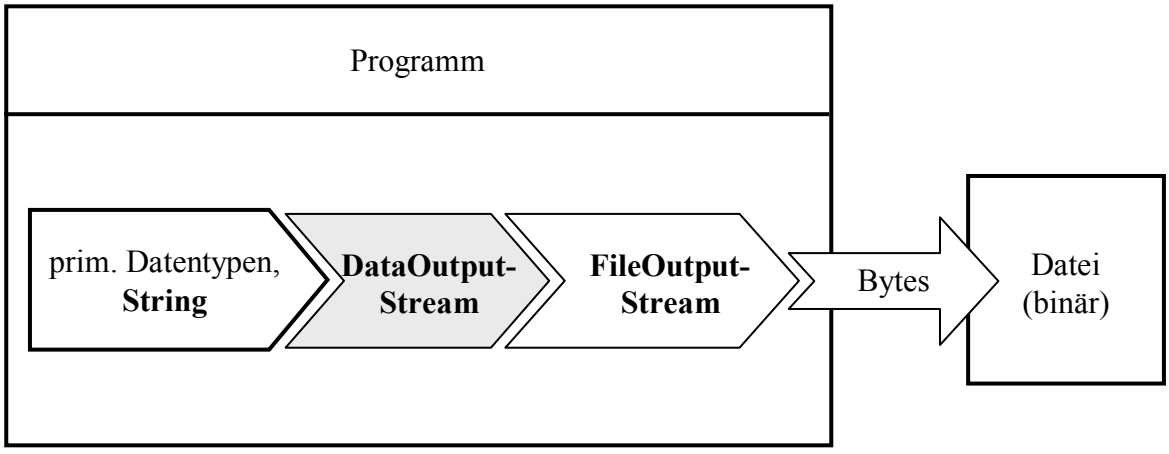
```
java.io.FileNotFoundException: quelle.dat (Das System kann die angegebene
Datei nicht finden)
```

Nach einem erfolgreichen Programmlauf wird die Transportleistung und die benötigte Zeit protokolliert, z.B.:

```
Kopieren von quelle.dat in ziel.dat startet (* = 1048576 Bytes):
********************************************************************************
****************************************
Es wurden 126619130 Bytes kopiert. (Benoetigte Zeit: 2812 Millisekunden.)
```

11.3.1.3 DataOutputStream

Mit einem Objekt aus der Transformationsklasse **DataOutputStream** lassen sich die Werte primitiver Datentypen über einen **OutputStream** (z.B. einen **FileOutputStream**) so in eine Datensenke befördern, dass sie mit dem komplementären Gespann aus einem **InputStream** und einem **DataInputStream** wieder zurück geholt werden können.



Neben primitiven Datentypen kann ein **DataOutputStream** auch **String**-Werte ausgeben, wobei u.a. die Platz sparende UTF8-Kodierung zur Verfügung steht (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**). Nähere Erläuterungen zu den folgenden Schreibmethoden finden Sie in der JDK-Dokumentation:

- **void write(byte[]** *b*, **int** *off*, **int** *len*)
- **void write(int b)**
- **void writeBoolean(boolean** *v*)
- **void writeByte(int** *v*)
- **void writeBytes(String** *s*)
- **void writeChar(int** *v*)
- **void writeChars(String** *s*)
- **void writeDouble(double** *v*)
- **void writeFloat(float** *v*)
- **void writeInt(int** *v*)
- **void writeLong(long** *v*)
- **void writeShort(int** *v*)
- **void writeUTF(String** *str*)

Im folgenden Beispielprogramm wird ein **DataOutputStream** vor einen **FileOutputStream** geschaltet und dann beauftragt, Daten vom Typ **int**, **double** und **String** zu schreiben:

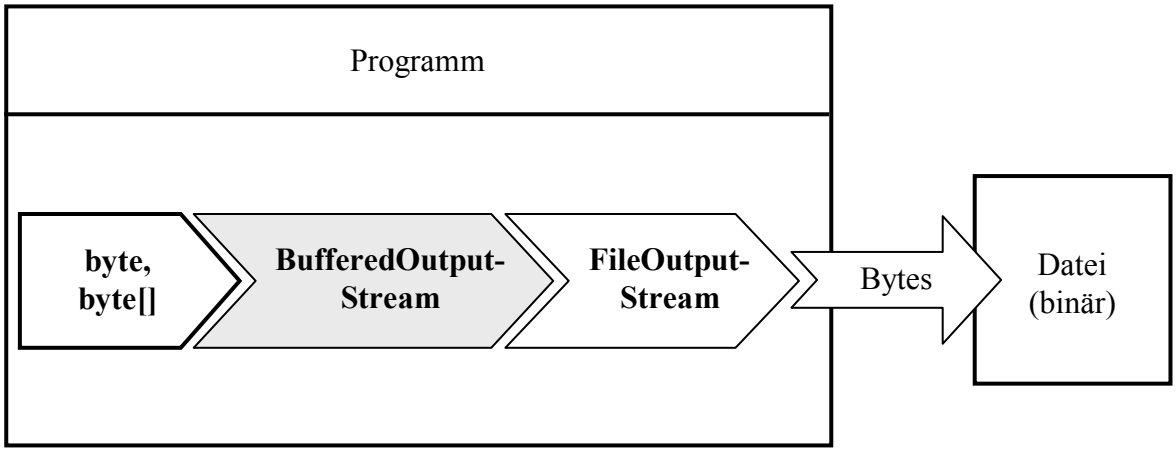
```
import java.io.*;
class DataOutputStreamDemo {
  public static void main(String[] egal) {
    DataOutputStream dos;
    DataInputStream dis;

    // int, double und String schreiben
    try {
      dos = new DataOutputStream(
             new FileOutputStream("demo.dat"));
      dos.writeInt(4711);
      dos.writeDouble(Math.PI);
      dos.writeUTF("DataOutputStream-Demo");
      dos.close();
    } catch (Exception e) {//Ausnahmebehandlung
    }

    // int, double und String lesen
    try {
      dis = new DataInputStream(
             new FileInputStream("demo.dat"));
      System.out.println("readInt()-Ergebnis:\t"+dis.readInt()+
                  "\nreadDouble()-Ergebnis:\t"+dis.readDouble()+
                  "\nreadUTF()-Ergebnis:\t"+dis.readUTF());
      dis.close();
    } catch (Exception e) {//Ausnahmebehandlung
    }
  }
}
```

Ein **DataInputStream** holt in Kooperation mit einem **FileInputStream** die Werte zurück:

```
readInt()-Ergebnis:     4711
readDouble()-Ergebnis:  3.141592653589793
readUTF()-Ergebnis:     DataOutputStream-Demo
```

11.3.1.4 BufferedOutputStream

Zur Beschleunigung von Ein- oder Ausgaben setzt man oft Transformationsklassen ein, die durch Paketierung und Zwischenspeichern die Anzahl der (meist langsamen) Zugriffe auf Datenquellen oder –senken reduzieren. Diese kooperieren mit Ein- bzw. Ausgabeklassen, die direkten Kontakt mit Datenquellen bzw. –senken haben.
Ein **BufferedOutputStream**–Objekt nimmt **byte**-Variablen oder –Arrays entgegen und leitet diese in geeigneten Portionen an ein **OutputStream**-basiertes Ausgabeobjekt weiter (z.B. an ein **FileOutputStream**-Objekt):



Im **BufferedOutputStream**–Konstruktor ist obligatorisch ein **OutputStream**-Objekt zu übergeben (vgl. Abschnitt 11.1.4). Optional kann die voreingestellte Puffergröße von 512 Bytes geändert werden:

- **BufferedOutputStream**(**OutputStream** *out*)
- **BufferedOutputStream**(**OutputStream** *out*, **int** *size*)

Das folgende Beispielprogramm schreibt in 51200 **write()**-Aufrufen jeweils ein einzelnes Byte in eine Datei, zunächst ungepuffert, dann unter Verwendung eines **BufferedOutputStream**-Objekts:

```
import java.io.*;

class BufferedOutputStreamDemo {
   public static void main(String[] args) {
      long time1 = System.currentTimeMillis();
      try {
         FileOutputStream fos = new FileOutputStream("noBuffer.dat");
         for (int i = 1; i <= 51200; i++)
            fos.write(i);
         fos.close();
      } catch (Exception e) {//Ausnahmebehandlung
      }
      long time2 = System.currentTimeMillis();
      System.out.println("Zeit fuer die ungepufferte Ausgabe: " +
                         (time2-time1));

      try {
         FileOutputStream fos = new FileOutputStream("Buffer.dat");
         BufferedOutputStream bos = new BufferedOutputStream(fos, 8192);
         for (int i = 1; i <= 51200; i++)
            bos.write(i);
         bos.close();
      } catch (Exception e) {//Ausnahmebehandlung
      }
      long time3 = System.currentTimeMillis();
      System.out.println("Zeit fuer die gepufferte Ausgabe:   " +
                         (time3-time2));
   }
}
```

Durch den Einsatz des **BufferedOutputStream**-Objekts (mit einer Puffergröße von 8192 Bytes) kann der Zeitaufwand beim Schreiben erheblich reduziert werden (Angaben in Millisekunden):

```
Zeit fuer die ungepufferte Ausgabe: 313
Zeit fuer die gepufferte Ausgabe:   31
```

Wegen des erheblichen Performanzvorteils sollte also ein Ausgabepuffer eingesetzt werden, wenn zahlreiche Schreibvorgänge mit jeweils kleinem Volumen stattfinden. Das `FileCopy`-Beispielprogramm in Abschnitt 11.3.1.2 wird hingegen nicht profitieren, weil das dortige **FileOutputStream**-Objekt ohnehin nur sehr große Datenblöcke erhält.

Achtung: 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:[34]

```
public void close() throws IOException {
  try {
    flush();
  } catch (IOException ignored) {
  }
  out.close();
}
```

Im Unterschied zu **FileOutputStream** überschreiben weder **BufferedOutputStream** noch **FilterOutputStream** die von **Object** geerbte **finalize()**-Methode, so dass beim automatischen Terminieren eines **BufferedOutputStream**-Objekts kein **close()**- und insbesondere kein **flush()**-Aufruf erfolgt.
Das obige Programm schreibt exakt 51200 Bytes in beide Ausgabedateien. Entfernt man die Anweisung

```
bos.close();
```

dann befinden sich nach Programmende in der Ausgabedatei **Buffer.dat** nur 49152 (= 6 · 8192) Bytes!

11.3.1.5 PrintStream

Die Transformationsklasse **PrintStream** dient dazu, Werte beliebigen Typs in einer für Menschen lesbaren Form auszugeben, z.B. auf der Konsole. Während ein **DataOutputStream**-Objekt dazu dient, Variablen beliebigen Typs in eine *Binärdatei* zu schreiben, kann man mit einem **PrintStream**-Objekt eine *Textdatei* erstellen.
Nach Abschnitt 11.1.2 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 wesentlichen Vorteil, dass für die Umsetzung des Java-internen Unicodes in die bevorzugte Textkodierung der Ausgabe ein Kodierungsschema gewählt werden kann (siehe Abschnitt 11.4.1.2).
Allerdings steht die Klasse **PrintStream** trotzdem *nicht* zur Disposition, weil z.B. der per **System.out** ansprechbare Standard-Ausgabestrom ein **PrintStream**-Objekt ist. Dies gilt auch für den Standard-*Fehler*ausgabestrom, der über die Klassenvariable **System.err** ansprechbar ist.[35]

---

[34] Sie finden diese Definition in der Datei **FilterOutputStream.java**, die wiederum im Archiv **src.zip** mit den API-Quelltexten steckt. Das Quelltextarchiv kann bei der JDK-Installation auf Ihre Festplatte befördert werden.

[35] Wird z.B. ein Ausnahmeobjekt über die Methode **printStackTrace()** beauftragt, die Aufrufsequenz auszugeben, dann landet diese im Fehlerausgabestrom.

Ein **PrintStream**-Objekt kann mit Hilfe seiner vielfach überladenen Methoden **print()** und **println()** Werte mit beliebigem Datentyp ausgeben, z.B.:

Quellcode	Ausgabe

```
class PrintStreamDemo {
   public static void main(String[] args) {
    PrintStreamDemo wob = new PrintStreamDemo();
    System.out.println("Ein PrintStream kann Variablen\n"
     +"bel. Typs verarbeiten.\n"
     +"Z.B. die double-Zahl\n"+"   "+Math.PI
     +"\noder auch das Objekt\n"+"   "+wob);
   }
}
```

```
Ein PrintStream kann Variablen
bel. Typs verarbeiten.
Z.B. die double-Zahl
   3.141592653589793
oder auch das Objekt
   PrintStreamDemo@16f0472
```

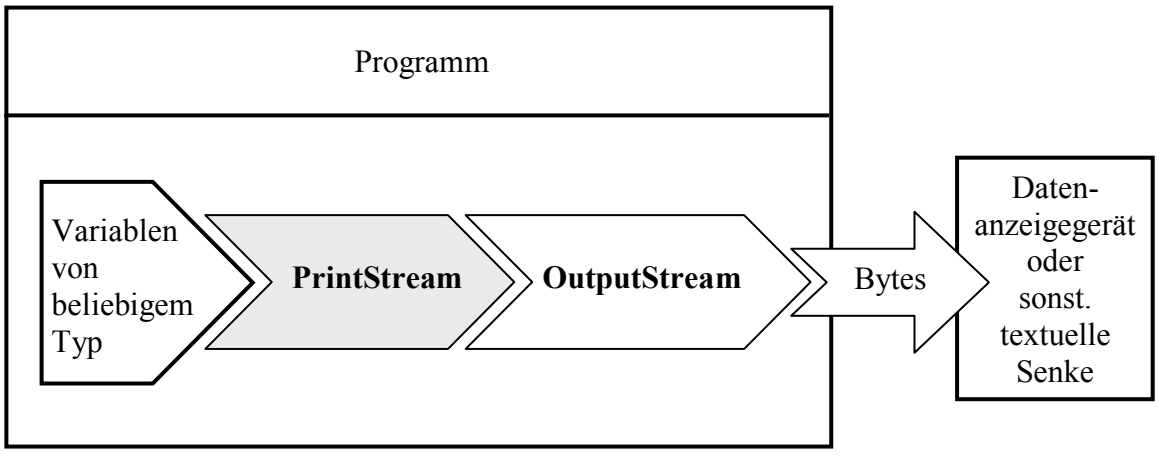
Seit Java 5.0 (alias 1.5) ist (zusammen mit der analog operierenden Methode **format()** der Klasse **String**) die sehr leistungsfähige Methode **printf()** zur formatierten Ausgabe hinzugekommen. Ihr erster Parameter ist vom Typ **String** und kann neben normalen Ausgabezeichen auch Platzhalter mit Formatangaben enthalten. Korrespondierend zu den Platzhaltern folgen als weitere Parameter Ausdrücke von passendem Typ, deren Werte ausgegeben werden sollen. Im folgenden Beispiel enthält der Formatierungsparameter den Platzhalter %02**d** für eine ganze Zahl, die bei Werten kleiner als 10 mit einer führenden Null ausgestattet wird:

```
System.out.printf("%02d:%02d:%02d %s", hours, minutes, seconds, "h:m:s");
```

Zur Bequemlichkeit trägt außerdem bei, dass im Unterschied zu anderen **OutputStream**-Unterklassen von den **PrintStream**-Methoden *keine* **IOException** geworfen wird. Stattdessen wird ein, von uns bisher nicht beachtetes Fehlersignal gesetzt, 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.

Hinter das per **System.out** ansprechbare **PrintStream**-Filterobjekt ist noch ein **FileOutputStream-Objekt** geschaltet, das mit der Konsole verbunden ist (siehe unten).

Generell kann man die **PrintStream**-Arbeitsweise folgendermaßen darstellen, wobei an Stelle der abstrakten Klasse **OutputStream** eine konkrete Unterklasse stehen muss:



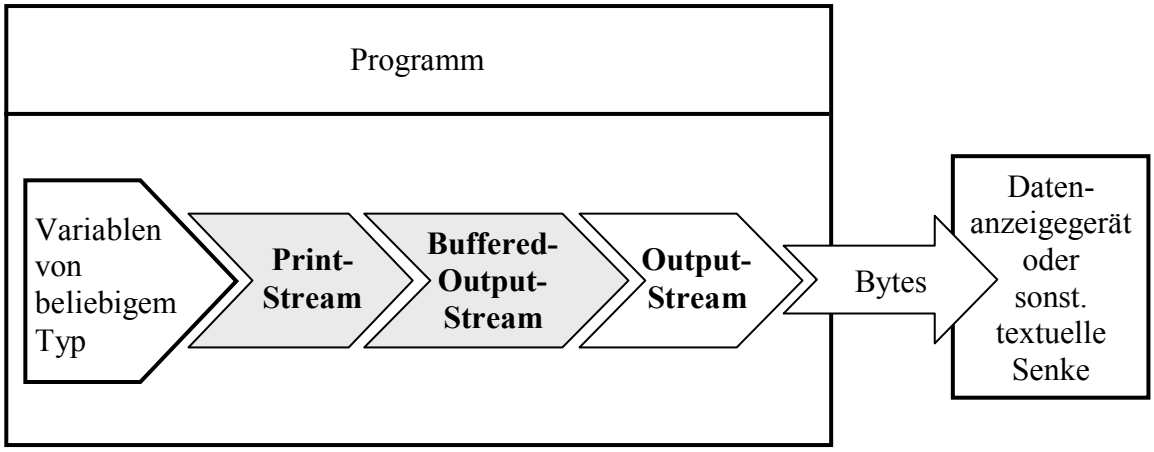
Im nächsten Beispiel ist zu sehen, wie mit Hilfe der Transformationsklasse **PrintStream** Werte primitiver Datentypen in eine *Textdatei* geschrieben werden (über einen **FileOutputStream**):

```
import java.io.*;

class PrinStreamDemoFile {
  public static void main(String[] args) {
    FileOutputStream fos = null;
    try {
      fos = new FileOutputStream("ps.txt");
    } catch (FileNotFoundException e) {System.out.println(e);}

    PrintStream ps = new PrintStream(fos);
    ps.println(64798 + " " + Math.PI);
    ps.close();
  }
}
```

In der Ausgabedatei **ps.txt** landen die Zeichenkettenrepräsentationen des **int**- und des **double**-Wertes:

```
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 sein Puffer geleert wird. Dazu stehen mehrere Möglichkeiten bereit:

- direkter Aufruf der Methode **flush()**
- Aufruf der Methode **close()**
- einen **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 beim automatischen Beenden eines **PrintStream**-Objekts (z.B. per garbage collector) kein **close()**-Aufruf und keine Pufferentleerung statt.

Ein gutes Beispiel für die Kombination aus **PrintStream** und **BufferedOutputStream** ist der per **System.out** ansprechbare Standard-Ausgabestrom, 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. Im Beispiel ist die Standardausgabe allerdings „originalgetreu“ nachgebaut worden.

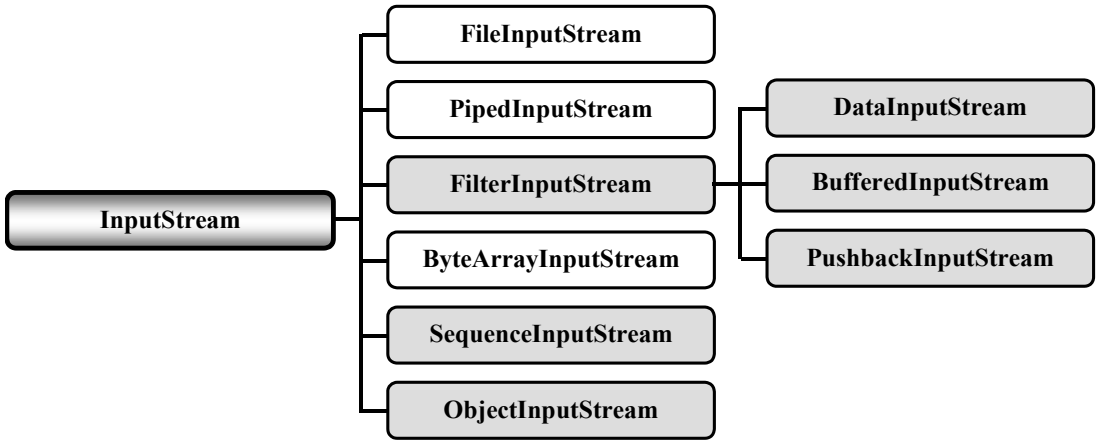
Bei der Ausgabe von *Textdaten* übersetzen **PrintStream**-Objekte den Java-internen Unicode unter Verwendung der plattformspezifischen Standardkodierung in Bytes. Welche Konsequenzen sich daraus ergeben können, zeigt die unglückliche Ausgabe von Umlauten im Konsolenfenster unter MS-Windows, z.B.:[36]

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.println("Ganz schön übel!");` `  }` `}`	`Ganz sch÷n ³bel!`

Daher ist zur Textausgabe oft die modernere und flexiblere Klasse **PrintWriter** vorzuziehen (siehe Abschnitt 11.4.1).

11.3.2 Die InputStream-Hierarchie

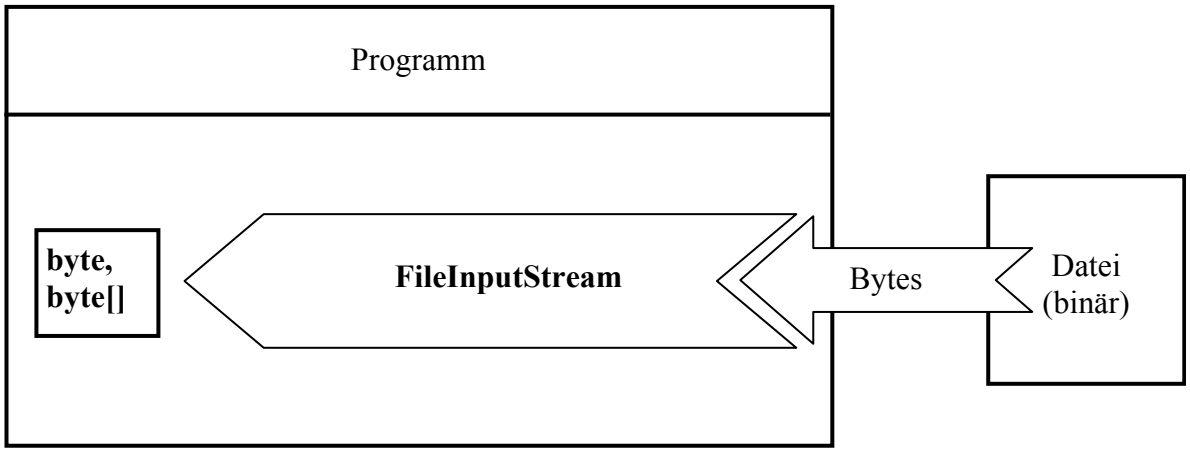
Bei den Klassen der **InputStream**-Hierarchie können wir uns im Wesentlichen mit Verweisen auf analoge Klassen der **OutputStream**-Hierarchie begnügen. Um Ihnen das Blättern zu ersparen, wird die schon in Abschnitt 11.1.2 gezeigte Abbildung zur **InputStream**-Hierarchie wiederholt (Eingabeklassen mit weißem Hintergrund und Eingabe*transformations*klassen mit grauem Hintergrund):



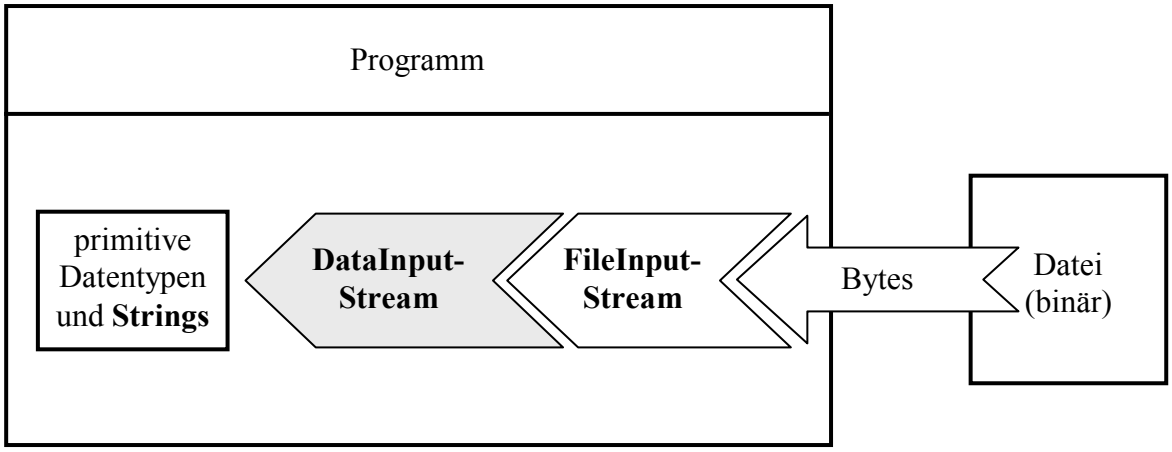
Wie man mit einem **FileInputStream**–Objekt Bytes aus einer Datei liest, wurde schon im Abschnitt 11.3.1.2 über den **FileOutputStream** gezeigt.

---

[36] Auf der Eclipse-Konsole erscheint die erwünschte Ausgabe, beim normalen Start via **java**-Werkzeug jedoch nicht.



Die Transformationsklasse **DataInputStream** liest primitive Datentypen sowie **Strings** aus einem Bytestrom und ist uns zusammen mit ihrem Gegenstück **DataOutputStream** schon in Abschnitt 11.3.1.3 begegnet.



Mit der Transformationsklasse **ObjectInputStream** werden wir uns in Abschnitt 11.6 näher beschäftigen. Sie ermöglicht das Einlesen kompletter Objekte aus einem Bytestrom.

Die restlichen Klassen sollen nur kurz vorgestellt werden:

- **PipedInputStream**
  Objekte dieser Klasse lesen Bytes aus einer Pipe, die zur Kommunikation zwischen Threads dient.
- **ByteArrayInputStream**
  Objekte dieser Klasse lesen Bytes aus einem **byte**-Array.
- **BufferedInputStream**
  Analog zum **BufferedOutputStream** (siehe Abschnitt 11.3.1) wird ein Zwischenspeicher eingesetzt, um das Lesen aus einem Eingabestrom zu beschleunigen.
- **PushbackInputStream**
  Diese Klasse bietet Methoden, um aus einem Eingabestrom entnommenen 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.

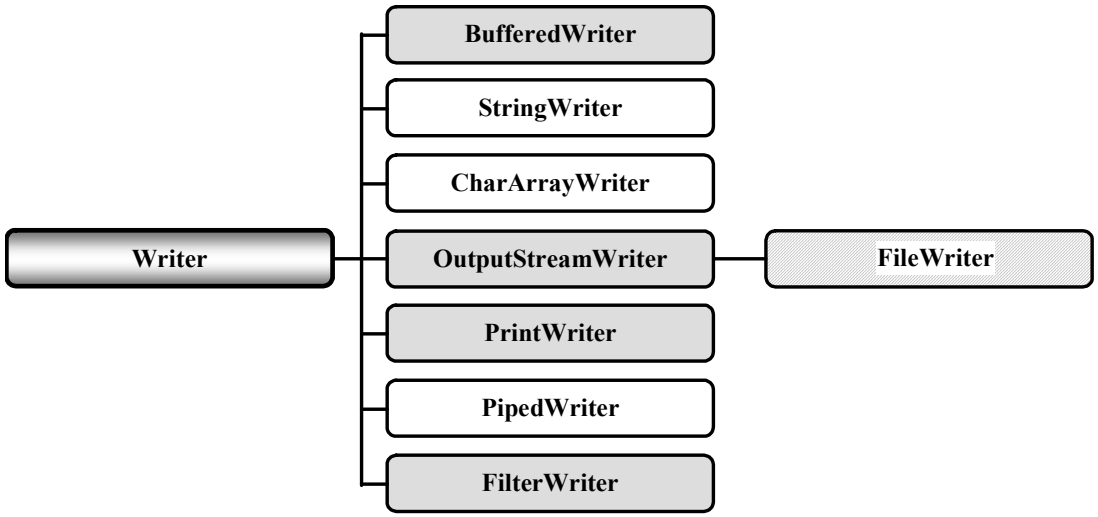
11.4 Klassen zur Verarbeitung von Zeichenströmen

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, wenn konkurrierende Funktionalitäten vorliegen.

11.4.1 Die Writer-Hierarchie

11.4.1.1 Überblick

In der folgenden Darstellung der **Writer**-Hierarchie sind Ausgabeklassen (in direktem Kontakt mit einer Datensenke) 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.

11.4.1.2 Brückenklasse OutputStreamWriter

Die Klasse **OutputStreamWriter** übersetzt als **Brückenklasse** die von Java intern verwendeten 16-bittigen Unicode-Zeichen in 8-bittige Bytes, wobei unterschiedliche Kodierungsschemata (engl. *encodings*) benutzt werden können. Ihre Besonderheit im Unterschied zu gewöhnlichen Transformationsklassen besteht darin, dass sie einen Zeichenstrom in einen Bytestrom überführt.
Die folgenden Kodierungsschemata (bezeichnet nach den Vorschriften der IANA (***I**nternet **A**ssigned **N**umbers **A**uthority*)) werden von allen Java-Implementationen unterstützt:

IANA-Bezeichnung	Beschreibung
US-ASCII	7-bit-ASCII-Code Bei den Unicode-Zeichen \u0000 bis \u007F wird das niederwertige Byte ausgegeben, ansonsten ein Fragezeichen (0x3F).
ISO-8859-1	Erweiterter ASCII-Code (ISO Latin-1) Bei den Unicode-Zeichen \u0000 bis \u00FF wird das niederwertige Byte ausgegeben, ansonsten ein Fragezeichen (0x3F).

<table>
<tr><th>IANA-Bezeichnung</th><th>Beschreibung</th></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>\u0000</td><td>2</td></tr>
<tr><td>\u0001</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>
</td></tr>
<tr><td>UTF-16BE</td><td>Für alle Unicode-Zeichen werden 16 Bit in Big-Endian - Reihenfolge ausgegeben: Das höherwertige Byte zuerst.
In Java ist diese Reihenfolge generell voreingestellt (auch bei anderen Datentypen).
Beim großen griechischen Delta (\u0394) wird ausgegeben: 03 94</td></tr>
<tr><td>UTF-16LE</td><td>Für alle Unicode-Zeichen werden 16 Bit in Little-Endian - Reihenfolge ausgegeben: Das niederwertige Byte zuerst.
Bei großen griechischen Delta (\u0394) wird ausgegeben: 94 03</td></tr>
</table>

Unter MS-Windows verwendet Java folgendes Kodierungsschema:

IANA-Bezeichnung	Beschreibung
Cp1252	Windows Latin-1 (ANSI) Im Unterschied zu ISO-8859-1 werden die Codes von 0x80 bis 0x9F (in ISO-8859-1 reserviert für <control>) mit „höheren“ Unicode-Zeichen assoziiert. Z.B. wird das Eurozeichen (Unicode: \u20AC) auf den Code 0x80 abgebildet und daher bei der Weiterverarbeitung einer Datei durch aktuelle Windows-Textverarbeitungsprogramme korrekt dargestellt.

Auf der Webseite

http://java.sun.com/j2se/1.5.0/docs/guide/intl/encoding.doc.html

werden zahlreiche weitere von Java 5.0 (alias 1.5) unterstützte Kodierungsschemata aufgelistet.

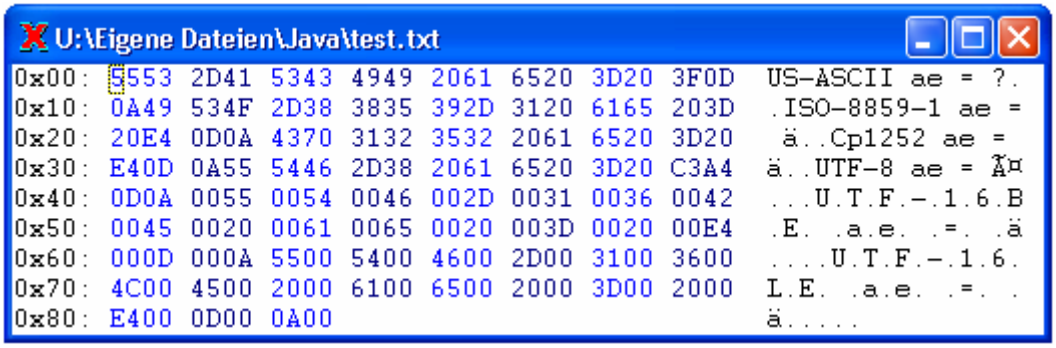
Komplette Übersetzungstabellen zu wichtigen encodings finden sich auf der Webseite:

http://www.ingrid.org/java/i18n/encoding/mapping.html

Im folgenden Programm werden die gerade tabellierten Kodierungsschemata nacheinander dazu verwendet, einen kurzen Text mit dem Umlaut „ä“ (\u00E4) in eine Datei zu schreiben:

```
import java.io.*;

class OSW {
  public static void main(String[] args) throws IOException {
    String[] encodings = {"US-ASCII", "ISO-8859-1", "Cp1252", "UTF-8",
                          "UTF-16BE", "UTF-16LE"};
    FileOutputStream fos = new FileOutputStream("test.txt");
    OutputStreamWriter osw = null;
    PrintWriter pw = null;

    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");
    }
    pw.close();
  }
}
```

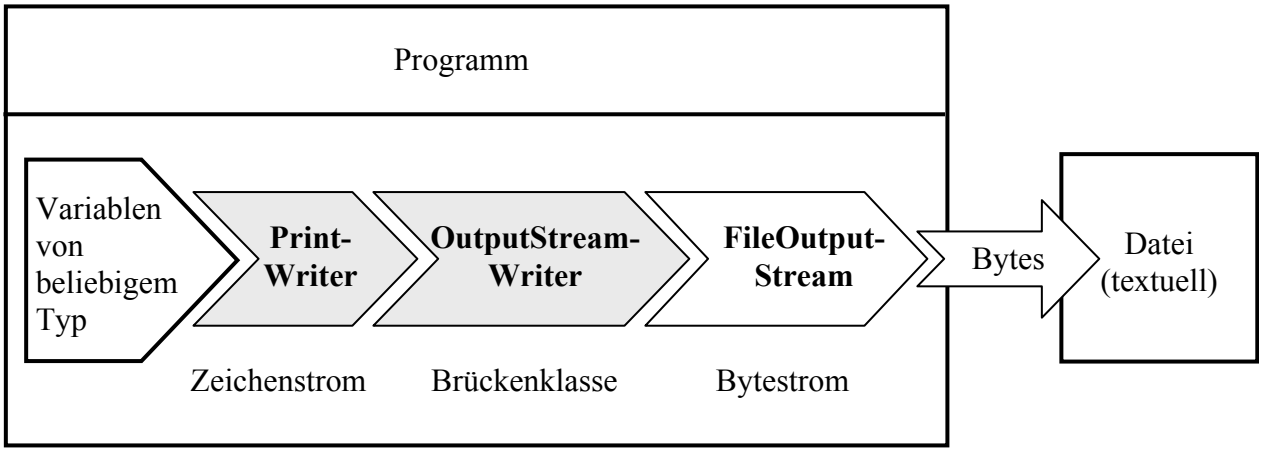
In einem Hex-Editor sieht die Ergebnisdatei folgendermaßen aus:



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:

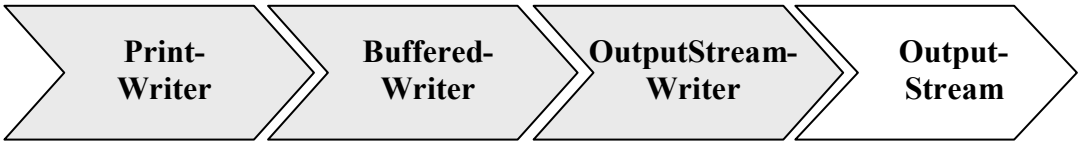


Wie die ältere Klasse **PrintStream** bietet auch die Klasse **PrintWriter** Ausgabemethoden für praktisch alle Datentypen.

OutputStreamWriter–Objekte kommen auch ohne Zutun des Programmierers zum Einsatz, wenn im **PrintWriter**–Konstruktor ein **OutputStream** angegeben wird. In diesem Fall nimmt der Konstruktor insgeheim einen **BufferedWriter** (siehe unten) und einen **OutputStreamWriter** in Betrieb:[37]

```
public PrintWriter(OutputStream out, boolean autoFlush) {
  this(new BufferedWriter(new OutputStreamWriter(out)), autoFlush);
}
```

Damit arbeitet insgesamt folgendes Gespann:



11.4.1.3 Implizite und explizite Pufferung

OutputStreamWriter–Objekte (und damit auch **FileWriter**-Objekte) 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 solchen Objekts (z.B. beim Programmende) der Puffer geleert werden. Dazu stehen mehrere Möglichkeiten bereit:

- direkter Aufruf der Methode **flush()**
- 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 unten) 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 beim automatischen Beenden eines Objekts aus diesen Klassen (z.B. per garbage collector) kein **close()**-Aufruf und keine Pufferentleerung statt.

Für die *explizite* Pufferung von *Zeichen* (statt Bytes) steht in der **Writer**-Hierarchie die Transformationsklasse **BufferedWriter** zur Verfügung, wobei die voreingestellte Puffergröße von 8192 Zeichen per Konstruktor-Parameter verändert werden kann.
Abweichend vom Aufbau der **OutputStream**-Hierarchie ist **BufferedWriter** *nicht* von **FilterWriter** abgeleitet.
Vom folgenden Programm werden mit Hilfe eines **PrintWriter**s und eines **FileWriter**s zweimal jeweils 10000 Zeilen in eine Textdatei geschrieben, zunächst ohne und dann mit zwischengeschaltetem **BufferedWriter**:

---

[37] 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 Ihre Festplatte befördert werden.

```
import java.io.*;

class BufferedWriterDemo {
  public static void main(String[] args) throws IOException  {
    PrintWriter pw = new PrintWriter(new FileWriter("test.txt"));
    long time1 = System.currentTimeMillis();
    for (int i = 1; i <= 10000; i++)
      pw.println("Zeile " + i);
    System.out.println("Benoetigte Zeit ohne BufferedWriter: "
                       +(System.currentTimeMillis()-time1));
    pw.close();

    pw =  new PrintWriter(
          new BufferedWriter(
          new FileWriter("test.txt"), 16384));
    time1 = System.currentTimeMillis();
    for (int i = 1; i <= 10000; i++)
      pw.println("Zeile " + i);
    System.out.println("Benoetigte Zeit mit BufferedWriter: "
                       +(System.currentTimeMillis()-time1));
    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: 62
Benoetigte Zeit mit BufferedWriter: 16
```

Dass **BufferedWriter**–Objekte gelegentlich auch ohne Zutun des Programmierers zum Einsatz kommen, wurde schon in Abschnitt 11.4.1.2 demonstriert: Wird beim Erstellen eines **PrintWriter**s ein **OutputStream**–Objekt übergeben dann nimmt der Konstruktor insgeheim einen **BufferedWriter** in Betrieb:[38]

```
public PrintWriter(OutputStream out, boolean autoFlush) {
  this(new BufferedWriter(new OutputStreamWriter(out)), autoFlush);
}
```

Beim Einsatz einer **Writer**-Subklasse ist unbedingt auf die abschließende Pufferentleerung zu achten.[39]

11.4.1.4 PrintWriter

Die Transformationsklasse **PrintWriter** besitzt diverse **print()**- bzw. **println()**-Überladungen, um Variablen beliebigen 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 Standard-Ausgabestrom **System.out** und im Standard-Fehlerausgabestrom **System.err** weiter lebt (vgl. Abschnitt 11.3.1.5). Seit der Java-Version 5.0 (alias 1.5) beherrschen **PrintWriter**-Objekte auch die Methoden **printf()** und **format()** zur formatierten Ausgabe.

Mit Ausnahme einiger Konstruktoren werfen **PrintWriter**-Methoden *keine* **IOException**, sondern setzen ein Fehlersignal, das mit **checkError**() abgefragt werden kann.

---

[38] 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 Ihre Festplatte befördert werden.

[39] Bei den Klassen **StringWriter** und **CharArrayWriter** ist ein **close()**-Aufruf allerdings überflüssig und wirkungslos.

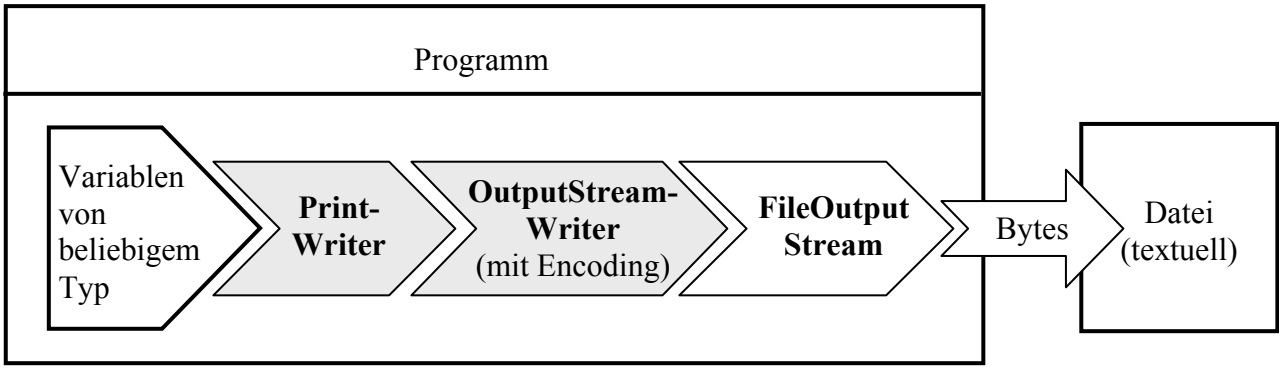
Wie die folgende Auswahl der **PrintWriter**-Konstruktoren zeigt, dürfen die angekoppelten Datenstromobjekte von den Basisklassen **OutputStream** oder **Writer** abstammen:

- **PrintWriter(OutputStream** *out***)**
- **PrintWriter(OutputStream** *out***, boolean** *autoFlush***)**
- **PrintWriter(Writer** *out***)**
- **PrintWriter(Writer** *out***, boolean** *autoFlush***)**

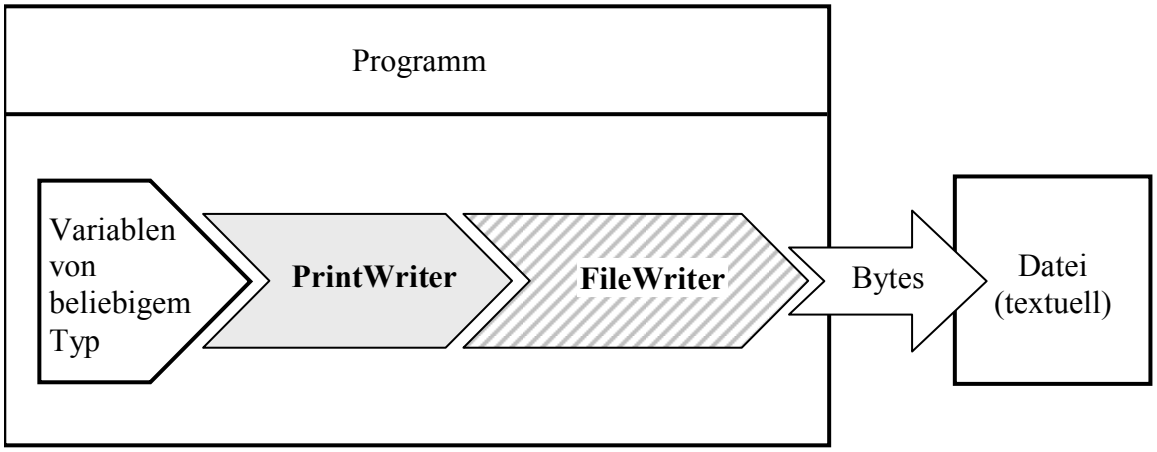
Erhält der Parameter **autoFlush** den Wert **true**, dann wird der Puffer bei jedem Aufruf der Methoden **println()**, **printf()** oder **format()** entleert.

Wird ein **PrintWriter** auf einen Bytestrom aus der **OutputStream**-Hierarchie gesetzt, dann kommt automatisch ein übersetzender **OutputStreamWriter** mit voreingestelltem Kodierungsschema zum Einsatz.
Gibt man im 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 11.4.1.2). Bei der Ausgabe von numerischen Daten in eine Textdatei spielt die Wahlfreiheit beim Kodierungsschema natürlich keine Rolle. Insgesamt ist die Klasse **PrintWriter** beim Erstellen von Textdateien zu bevorzugen, weil sich damit alle Aufgaben bewältigen lassen.

Mit der folgenden Ausgabestromkonstruktion wird die Flexibilität der **Writer**-Subklassen bei der Textausgabe ausgeschöpft:



Wenn die Standardkodierung akzeptabel ist, bietet sich als bequemer Ausgabestrom die von der Brückenklasse **OutputStreamWriter** abgeleitete und damit automatisch puffernde (vgl. Abschnitt 11.4.1.3) Klasse **FileWriter** an (siehe Abschnitt 11.4.1.6):

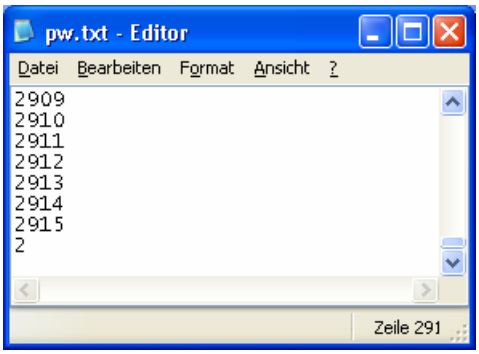


Beim Einsatz der Klasse **PrintWriter** ist zu beachten, dass in der Regel ein Puffer im Spiel ist (vgl. Abschnitt 11.4.1.3). Weil **PrintWriter**–Objekte keine automatische Pufferentleerung über einen

terminalen **finalize()**-Aufruf vornehmen, müssen wir uns selbst darum kümmern, z.B. per **close()**-Aufruf:

```
import java.io.*;
class PrintWriterDemo {
  public static void main(String[] egal) throws IOException {
    FileWriter fw = new FileWriter("pw.txt");
    PrintWriter pw = new PrintWriter(fw);
    for (int i = 1; i <= 3000; i++) {
      pw.println(i);
    }
    pw.close();
  }
}
```

Ohne **close**()-Aufruf fehlen in der Ausgabe des Beispielprogramms ca. 1500 Zeilen:



Soll ein **PrintWriter**–Objekt nach jedem Aufruf der Methoden **println()**, **printf()** oder **format()** seinen Puffer automatisch ausgeben (z.B. bei einer *interaktiven* Anwendung), dann muss im Konstruktor der **autoFlush**-Parameter auf **true** gesetzt werden (siehe Beispiel in Abschnitt 11.4.1.5). Wenn kein **autoFlush** erforderlich ist, sollte aus Performanzgründen darauf verzichtet werden.

Abweichend vom Aufbau der **OutputStream**-Hierarchie ist **PrintWriter** *nicht* von **FilterWriter** abgeleitet.

11.4.1.5 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 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 (genauer: mit dem Kodierungsschema *Cp1252* alias *Windows Latin 1*, siehe Abschnitt 11.4.1.2). Infolgedessen werden Umlaute in Java-Konsolenanwendungen unter Windows falsch dargestellt.

Ein **OutputStreamWriter** - Objekt mit dem Kodierungsschema *Cp850* („IBM-ASCII“, „code page 850“) ermöglicht die korrekte Darstellung der Umlaute:[40]

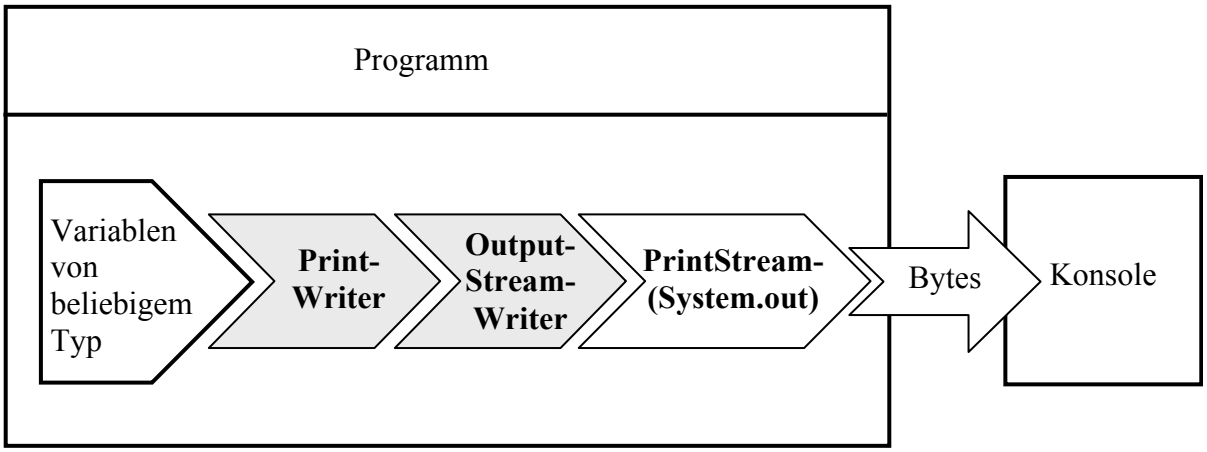
[40] Weil die Entwicklungsumgebung Eclipse 3.x bei der Konsolenausgabe eine automatische Kodierungsanpassung vornimmt, führt nun eine doppelte Korrektur zur fehlerhaften Ausgabe von Umlauten.

Quellcode	Ausgabe
`import java.io.*;` `class Cp850 {` `  public static void main(String[] ars) throws IOException {` `  OutputStreamWriter osw = null;` `  if (` `  System.getProperty("file.encoding").equals("Cp1252"))` `  osw = new OutputStreamWriter(System.out, "Cp850");` `  else` `  osw = new OutputStreamWriter(System.out);` `  PrintWriter pw = new PrintWriter(osw, true);` `  pw.println("Der Ärger war nicht nötig.");` `  }` `}`	`Der Ärger war nicht nötig.`

Die statische Variable **System.out** zeigt auf ein **PrintStream**-Objekt, das mit der Konsole verbunden ist. Um die Plattformunabhängigkeit nicht einzuschränken, wird das spezielle Kodierungsschema nur dann verwendet, wenn der **System**-Methodenaufruf

String getProperty("file.encoding")

als Standard-Kodierungsschema *Cp1252* zurück meldet.
Es liegt in jedem Fall die folgende Datenstrom-Architektur vor:



Die Darstellung ist leicht vereinfacht, denn zwischen dem **PrintStream**-Filterobjekt und der Konsole agieren noch (vgl. Abschnitt 11.3.1.5):

- ein **BufferedOutputStream**
- ein **FileOutputStream**

11.4.1.6 FileWriter

Die von **OutputStreamWriter** abgeleitete Klasse **FileWriter** wird oft an einen **PrintWriter** zur bequemen Ausgabe in eine Textdatei angehängt, wenn das voreingestellte Kodierungsschema (bei MS-Windows: *Cp1252* alias *Windows Latin 1*) und die voreingestellte Puffergröße (8192 Bytes) akzeptabel sind.
In obiger Darstellung der **Writer**-Hierarchie erhielt die Klasse **FileWriter** einen schraffierten Hintergrund, weil sie eine Ausgabe- *und* eine Filterfunktionen hat. Ihre Objekte …

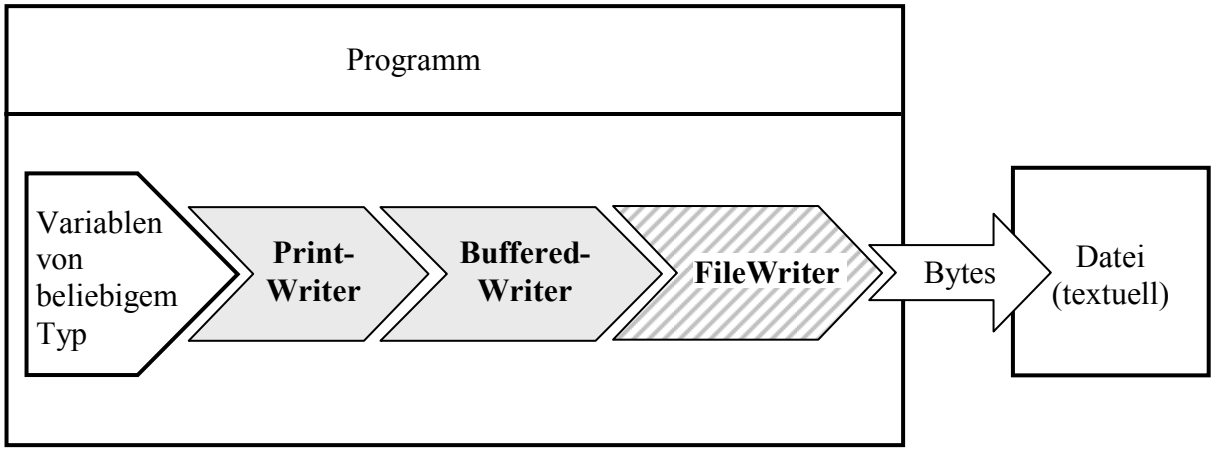
- stehen über ein Instanzobjekt aus der Klasse **FileOutputStream** mit einer Datei in Kontakt, so dass sie als Ausgabestrom arbeiten können.
- transformieren den eingehenden Zeichenstrom in einen Bytestrom, so dass sie als Filterobjekt bezeichnet werden können.

Wichtiger als die akademische Bemerkung zur korrekten Klassifikation der Klasse **FileWriter** sind ihre bequemen Konstruktoren, die das Öffnen einer per **String**- oder **File**-Objekt bestimmten Datei zum Überschreiben oder zum Anhängen weiterer Zeichen erlauben:

- **FileWriter(File** *file***)**
- **FileWriter(File** *file***, boolean** *append***)**
- **FileWriter(String** *fileName***)**
- **FileWriter(String** *fileName***, boolean** *append***)**

Ist das voreingestellte Kodierungsschema ungeeignet, muss man auf die **FileWriter**-Bequemlichkeit verzichten, einen passenden **OutputStreamWriter** wählen und einen **FileOutputStream** dahinter schalten (siehe Abschnitt 11.4.1.2).

Ein Beispiel für den **FileWriter**-Einsatz zusammen mit einem **PrintWriter** haben Sie schon in Abschnitt 11.4.1.4 kennen gelernt. In Abschnitt 11.4.1.3 war sogar das bei hohen Performanz-Ansprüchen empfehlenswerte Gespann aus **PrintWriter**, **BufferedWriter** und **FileWriter** zu sehen:



11.4.1.7 Sonstige Writer-Subklassen

Bei den folgenden **Writer**-Subklassen beschränken wir uns auf kurze Hinweise:

- **StringWriter** und **CharArrayWriter**
  StringWriter-Objekte schreiben ihre Ausgabe in einen dynamisch wachsenden **StringBuffer**. Im folgenden Beispiel werden die Eingaben von einem **PrintWriter** geliefert:

Quellcode	Ausgabe
`import java.io.*;` `class StringWriterDemo {` `  public static void main(String[] args) {` `  StringWriter sw = new StringWriter();` `  PrintWriter pw = new PrintWriter(sw);` `  for (int i = 1; i <= 5; i++)` `  pw.println("Zeile " + i);` `  System.out.println(sw.toString());` `  }` `}`	`Zeile 1` `Zeile 2` `Zeile 3` `Zeile 4` `Zeile 5`

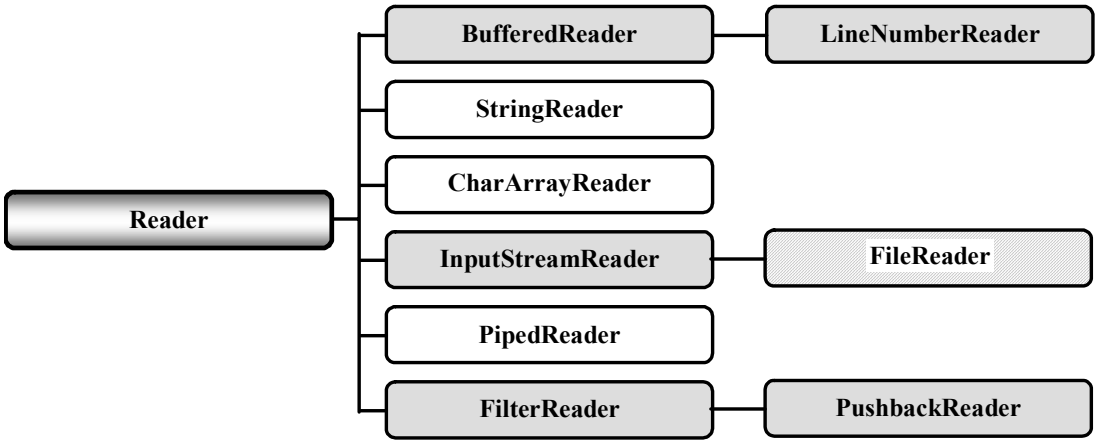
  CharArrayWriter-Objekte verhalten sich im Wesentlichen genauso.
- **PipedWriter**
  Diese Klasse ist das zeichenorientierte Analogon zum **PipedOutputStream**.
- **FilterWriter**
  Diese abstrakte Basisklasse bietet sich dazu an, eigene Transformationsklassen für zeichenorientierte Ausgabeströme abzuleiten.

11.4.2 Die Reader-Hierarchie

In der **Reader**–Hierarchie finden sich diverse Klassen zur Verarbeitung von zeichenorientierten *Eingabe*strömen.

11.4.2.1 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** eine Eingabe- *und* eine Filterfunktionen, ist sie mit schraffiertem Hintergrund gezeichnet.



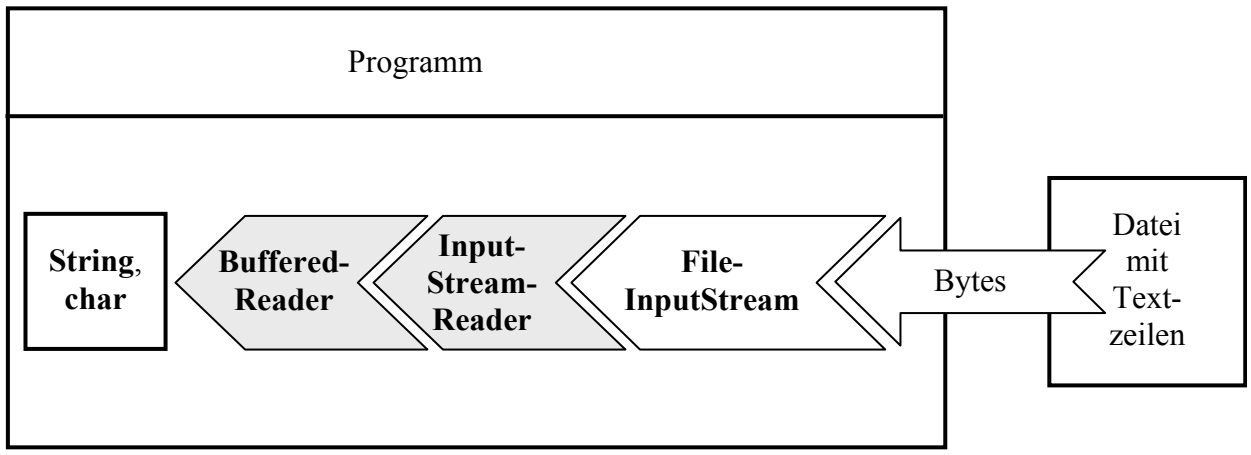
Bei den meisten **Reader**-Unterklassen beschränken wir uns auf kurze Hinweise:

- **BufferedReader**
  Diese Klasse kann die Eingabe durch einen Puffer beschleunigen und ist auch wegen ihrer **readLine()**-Methode zum Einlesen einer kompletten Zeile recht beliebt.
  Die voreingestellte Puffergröße von 8192 Zeichen lässt sich per Konstruktor ändern.
- **LineNumberReader**
  Dieser gepufferte Zeichen-Eingabestrom erweitert seine Basisklasse um Methoden zur Verwaltung der 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 11.5 die (unmittelbar aus **java.lang.Object** abgeleitete) Klasse **Scanner** ansehen.

11.4.2.2 Brückenklasse InputStreamReader

Die Brückenklasse **InputStreamReader** ist das Gegenstück zur Klasse **OutputStreamReader**, wandelt also Bytes unter Verwendung eines einstellbaren Kodierungsschemas (vgl. Abschnitt 11.4.1.2) in Unicode-Zeichen:



Wie beim **OutputStreamReader** findet zur Beschleunigung der Konvertierung automatisch eine Pufferung des Bytestroms statt.

11.4.2.3 FileReader

Die Klasse **FileReader** ist das Gegenstück zur Klasse **FileWriter**. Sie erhielt in obiger Darstellung der **Reader**-Hierarchie einen schraffierten Hintergrund, weil sie eine Eingabe- *und* eine Filterfunktionen hat. 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 Bytestrom in einen Zeichenstrom, so dass sie als Filterobjekt bezeichnet werden können.

Bei der Wandlung von Bytes in Unicode-Zeichen kommt das voreingestellte Kodierungsschema der Plattform zum Einsatz (bei MS-Windows: *Cp1252* alias *Windows Latin 1*). Wer ein alternatives Kodierungsschema benötigt, muss auf die **FileReader**-Bequemlichkeit verzichten, einen passenden **InputStreamReader** wählen und mit einem **FileInputStream** kombinieren.

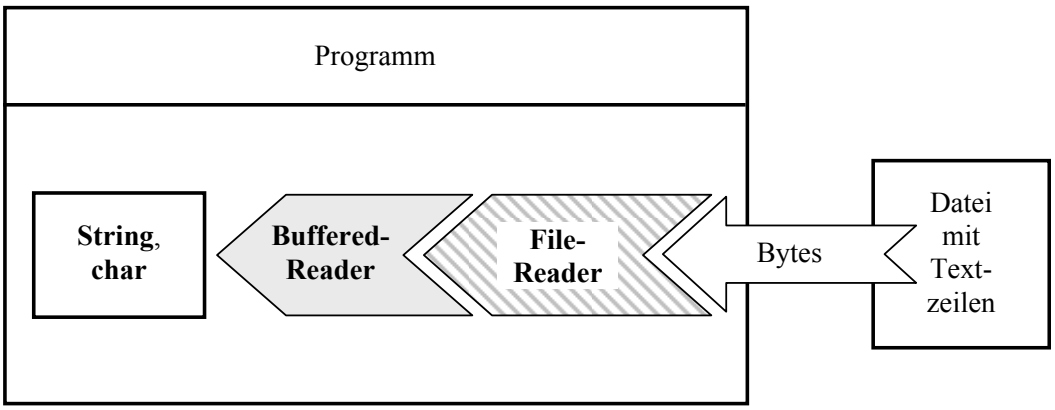
In der Regel setzt man vor den **FileReader** noch einen **BufferedReader**, der für eine Beschleunigung sorgt und außerdem die bei Dateien mit Zeilenstruktur sehr nützliche Methode **readLine()** bietet, z.B.:

Quellcode	Ausgabe

```
import java.io.*;

class FR {
  public static void main(String[] args) throws IOException {
    String[] starry = new String[100];
    BufferedReader br = new BufferedReader(
                new FileReader("test.txt"));
    for (int i = 0; i < 100; i++)
      starry[i] = br.readLine();
    System.out.println(starry[99]);
    br.close();
  }
}
```

```
Zeile 100
```

Das Beispielprogramm arbeitet mit folgender Datenstrom-Architektur:



11.5 Primitive Typen und Zeichenketten aus Textdateien lesen

Seit Java 5.0 (alias 1.5) erleichtert die unmittelbar von **java.lang.Object** abstammende Klasse **Scanner** (Namensraum **java.util**) das Einlesen von primitiven Datentypen und Zeichenketten aus Textdateien.
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.:

- **int nextInt()**
  Versucht, das nächste Token als ganze Zahl zu interpretieren, und wirft bei Misserfolg eine **InputMismatchException**.
- **double nextDouble()**
- **String nextLine()**

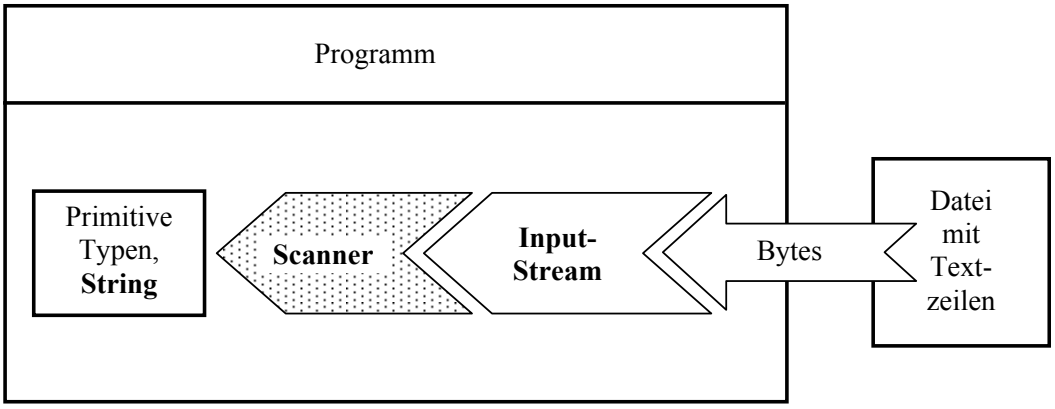
Ob das nächste Token vom gewünschten Typ ist, kann mit einer entsprechenden Methode festgestellt werden, z.B.:

- **boolean hasNextInt()**
- **boolean hasNextDouble()**

Als Trennzeichen für die Zerlegung des Eingabestroms in Tokens gelten per Voreinstellung alle *WhiteSpace*-Zeichen (z.B. Leerzeichen, Tabulator), doch erlaubt die Methode **useDelimiter()** eine alternative Festlegung über einen regulären Ausdruck.

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 dann Strings aus einer Textdatei:

```
import java.util.*;
class ScannerFile {
  public static void main(String[] args) {
    int i = 0;
    try {
      Scanner input = new Scanner(new java.io.FileInputStream("daten.txt"));
      while (input.hasNext())
        System.out.println("Zeile " + i++ +"\nInteger= "+input.nextInt()
                           + "\tDouble= "+input.nextDouble());
      input.close();
      System.out.println();
      i = 0;
      input = new Scanner(new java.io.FileInputStream("text.txt"));
      while (input.hasNext())
        System.out.println("Zeile " +i++ +"\n"+input.nextLine());
      input.close();
    } catch(Exception e) {}
  }
}
```

Mit den Eingabedateien

daten.txt

```
4711   3,1415926
13     9,99
```

text.txt

```
Damit das schon mal klar ist:
Wir mögen keinen üblen Ärger.
```

erhält man die Ausgabe:

```
Zeile 0
Integer= 4711 Double= 3.1415926
Zeile 1
Integer= 13   Double= 9.99

Zeile 0
Damit das schon mal klar ist:
Zeile 1
Wir mögen keinen üblen Ärger.
```

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(String.valueOf(basis)+" hoch "+exponent+
                         " = "+Math.pow(basis, exponent));
    } catch(Exception e) {
      System.out.println("Eingabefehler");
    }
  }
}
```

Nun sind Sie im Stande, die von der Klasse `Simput` 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;
  }
 .   .   .
 .   .   .
}
```

11.6 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 Bytestrom 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*.
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 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 ohne unser Mitwirken abläuft.

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()**):

```
// Datei Kunde.java
import java.io.*;

public class Kunde implements Serializable {
   private String vorname;
   private String name;
   private transient int stimmung;
   private int nkaeufe;
   private double aussen;
   public Kunde(String vorname_, String name_, int stimmung_,
                int nkaeufe_, double 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");
   }
}

// Datei Serialisierung.java
import java.io.*;

class Serialisierung {
   public static void main(String[] args) throws Exception {
      Kunde demo = new Kunde("Fritz", "Orth", 1, 13, 426.89);
      System.out.println("Zu sichern:\n");
      demo.prot();
      int anzahl;
      ObjectOutputStream oos = new ObjectOutputStream(
                               new FileOutputStream("test.bin"));
      oos.writeObject(demo);
      oos.writeInt(1);
      oos.close();
      ObjectInputStream ois = new ObjectInputStream(
                              new FileInputStream("test.bin"));
      Kunde unbekannt = (Kunde) ois.readObject();
      anzahl = ois.readInt();
      System.out.println(anzahl + " Fall eingelesen:\n");
      unbekannt.prot();
      ois.close();
   }
}
```

Beim Schreiben eines Objektes wird auch seine Klasse festgehalten. Beim Lesen eines Objektes wird zunächst die zugehörige Klasse festgestellt und in die virtuelle Maschine geladen (falls noch nicht vorhanden). Dann 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 ...

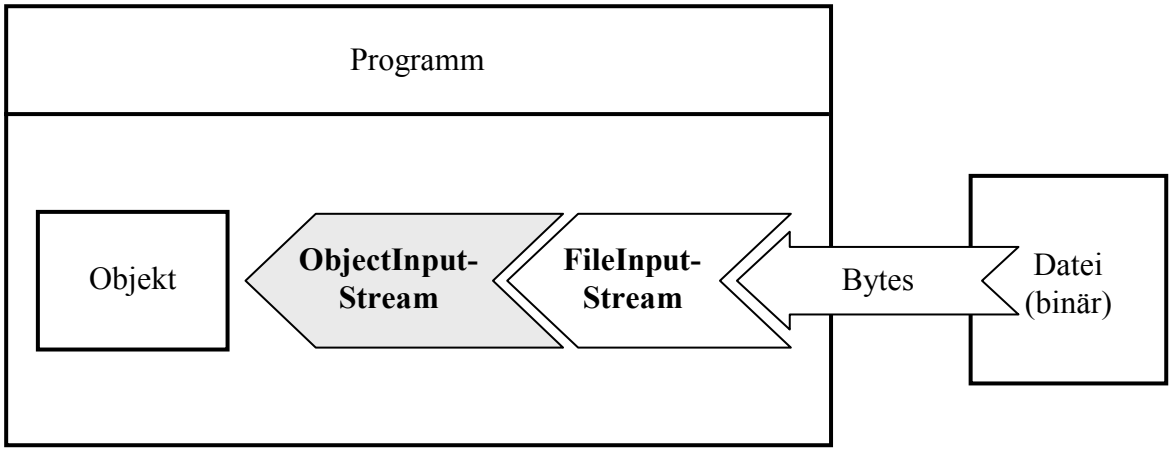
- Variablen, die aus Sicherheitsgründen nicht in eine Datei gespeichert werden sollen,
- Variablen, deren temporärer Charakter ein Speichern nutzlos macht.

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 Objektes skizziert:



11.7 Empfehlungen zur Verwendung der EA-Klassen

Weil die Java-EA-Behandlung durch die Vielzahl beteiligter Klassen auf Anfänger etwas unübersichtlich wirkt, folgt in diesem Abschnitt eine rezeptartige Beschreibung wichtiger Spezialfälle.

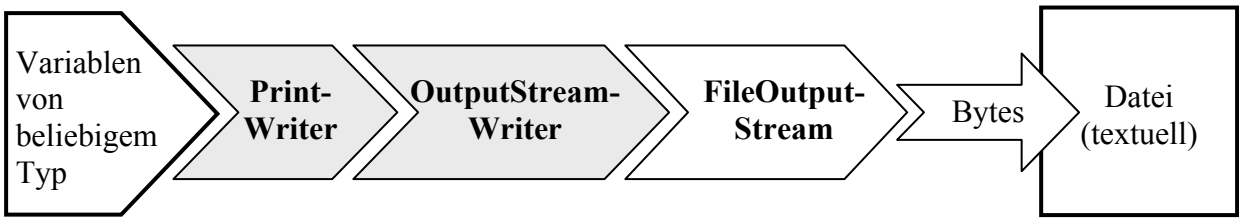
11.7.1 Ausgabe in eine Textdatei

Um Textdaten (Datentypen **String**, **char**) oder die Zeichenfolgen-Repräsentationen beliebiger andere Datentypen in eine sequentiell organisierte Datei zu schreiben, verwendet man in der Regel einen **FileWriter** (siehe Abschnitt 11.4.1.6), dessen Konstruktor die Ausgabedatei öffnet, in Kombination mit einem **PrintWriter** (siehe Abschnitt 11.4.1.4), der bequeme Ausgabemethoden bietet (z.B. **println()**, **printf()**).

Beispiel:

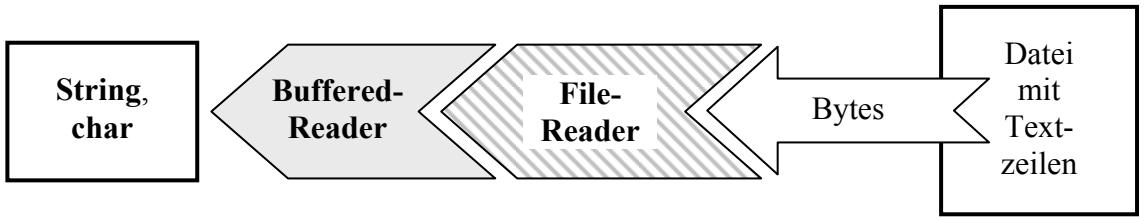
```
PrintWriter pw =  new PrintWriter(new FileWriter("test.txt"));
pw.println("Diese Zeile landet in der Datei test.txt.");
```

Wenn bei der Ausgabe von Textsdaten das voreingestellte Kodierungsschema ungeeignet ist, ersetzt man den **FileWriter** durch die Kombination aus einem **OutputStreamWriter** mit wählbarer Kodierung (siehe Abschnitt 11.4.1.2) und einem **FileOutputStream** (siehe Abschnitt 11.3.1.2).



11.7.2 Textdaten einlesen

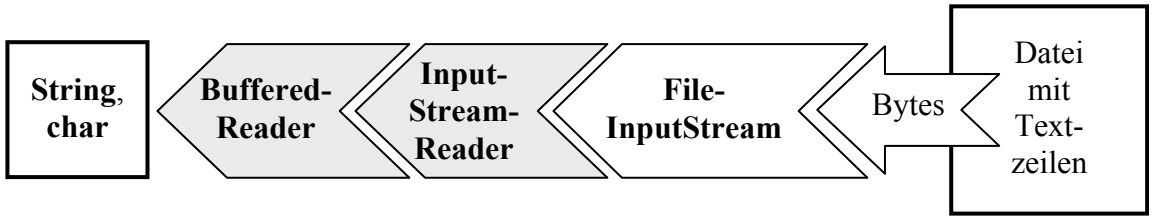
Um Textdaten aus einer sequentiell organisierten Datei zu lesen, verwendet man in der Regel ein Objekt aus der Klasse **FileReader** (siehe Abschnitt 11.4.2.3).
Meist schaltet man hinter den **FileReader** noch einen **BufferedReader**, der die Anzahl der Dateizugriffe reduziert und die nützliche Methode **readLine()** bietet.



Beispiel:

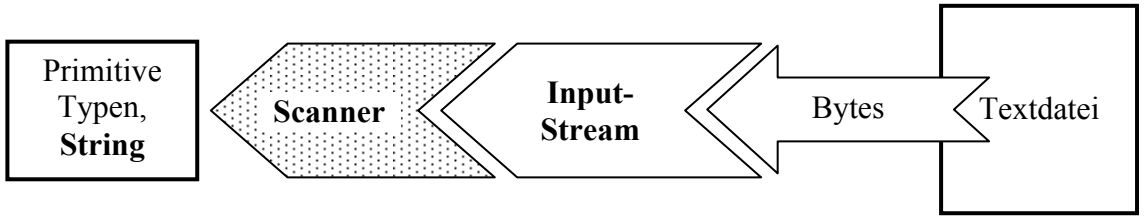
```
BufferedReader br = new BufferedReader(new FileReader("test.txt"));
String s = br.readLine();
```

Wer mit dem voreingestellten Kodierungsschema nicht zufrieden ist, ersetzt den **FileReader** durch die Kombination aus einem **InputStreamReader** mit wählbarer Kodierung und einem **FileInputStream**.



11.7.3 Primitive Typen 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** verwenden (siehe Abschnitt 11.5):



Beispiel:

```
Scanner input = new Scanner(new java.io.FileInputStream("daten.txt"));
double a = input.nextDouble();
int b = input.nextInt();
```

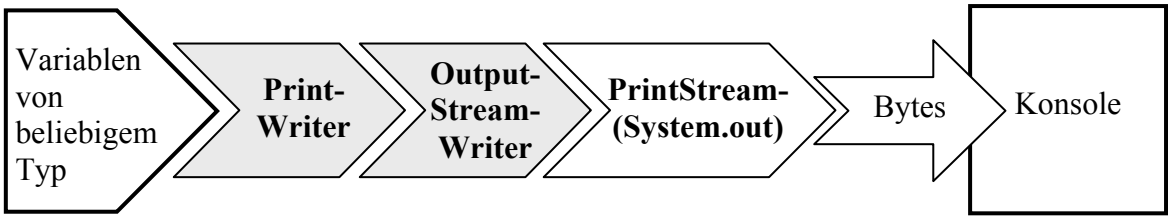
Ein **Scanner** liest aber auch Zeichenfolgen und erlaubt dabei sogar die Wahl eines Kodierungsschemas.

11.7.4 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 11.4.1.5 wird beschrieben, wie eine perfekte Konsolenausgabe mit korrekter Darstellung der Umlaute zu bewerkstelligen ist:[41]



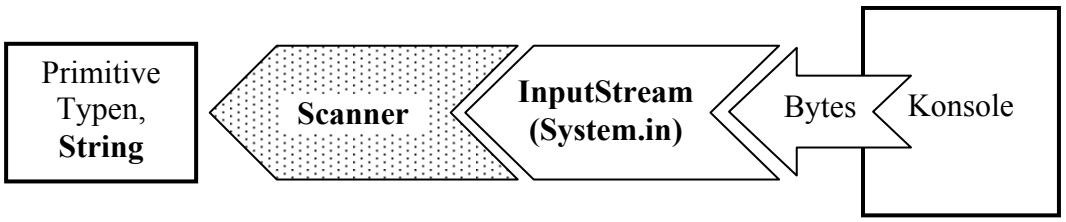
Weil der **OutputStreamWriter** die durch Wandlung von Unicode-Zeichen entstehenden Bytes automatisch puffert (vgl. Abschnitt 11.4.1.3), sollte beim **PrintWriter** die **autoFlush**-Option eingeschaltet werden.

Beispiel:

```
OutputStreamWriter osw = new OutputStreamWriter(System.out, "Cp850");
PrintWriter pw = new PrintWriter(osw, true);
pw.println("Hallo Wörld");
```

11.7.5 Eingabe von der Konsole

In Abschnitt 11.5 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();
```

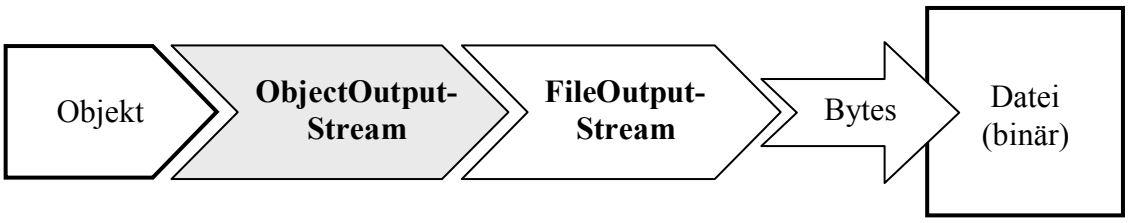
11.7.6 Objekte schreiben und lesen

Um Objekte in eine Datei zu schreiben oder aus einer zu Datei lesen, verwendet man die in Abschnitt 11.6 vorgestellten Klassen **ObjectOutputStream** und **ObjectInputStream**:

---

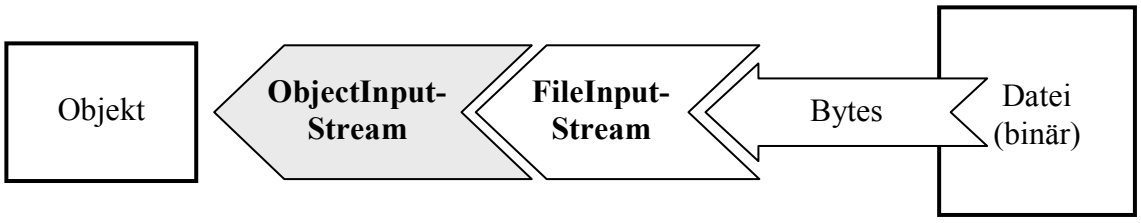
[41] Die Darstellung ist leicht vereinfacht, denn zwischen dem **PrintStream**-Filterobjekt und der Konsole agieren noch:

- ein **BufferedOutputStream**
- ein **FileOutputStream**



Beispiel:

```
ObjectOutputStream oos = new ObjectOutputStream(
                           new FileOutputStream("test.bin"));
oos.writeObject(demobj);
```

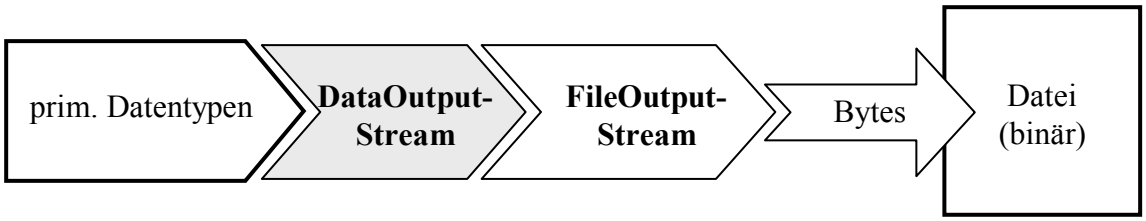


Beispiel:

```
ObjectInputStream ois = new ObjectInputStream(
                          new FileInputStream("test.bin"));
Kunde unbekannt = (Kunde) ois.readObject();
```

11.7.7 Primitive Datentypen in eine Binärdatei schreiben

Um primitive Datentypen (z.B. **int**, **double**) binär in eine sequentiell organisierte Datei zu schreiben, verwendet man ein Filterobjekt aus der Klasse **DataOutputStream** in Kombination mit einem Ausgabeobjekt aus der Klasse **FileOutputStream**:

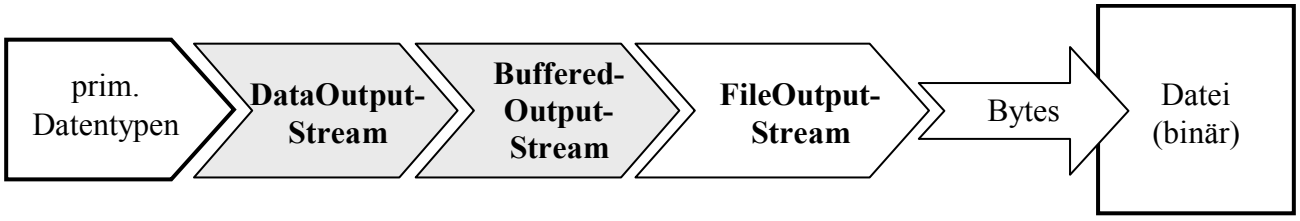


Beispiel:

```
DataOutputStream dos = new DataOutputStream(new FileOutputStream("demo.dat"));
dos.writeInt(4711);
dos.writeDouble(Math.PI);
```

Beim primitiven Datentyp **byte** ist kein **DataOutputStream**-Filterobjekt erforderlich, weil bereits die **OutputStream**-Methoden eine hinreichende Funktionalität bieten.

Soll die Ausgabe gepuffert erfolgen, um die Anzahl der Dateizugriffe gering zu halten, dann muss ein Filterobjekt aus der Klasse **BufferedOutputStream** eingesetzt werden:



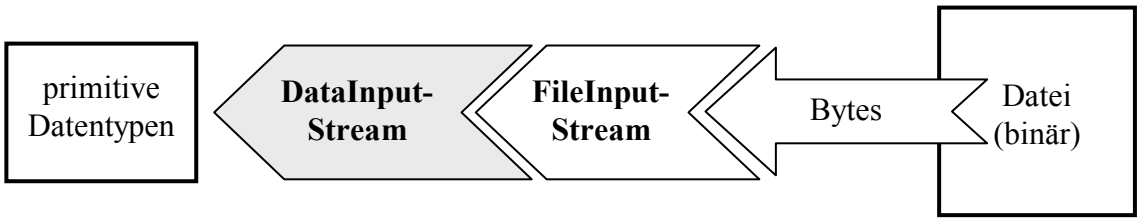
Hier wird ein Puffer mit 16384 Bytes Kapazität verwendet:

```
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.

11.7.8 Primitive Datentypen aus einer Binärdatei lesen

Um primitive Datentypen (z.B. **int**, **double**) aus einer sequentiell organisierten Binärdatei zu lesen, verwendet man ein Filterobjekt aus der Klasse **DataInputStream** in Kombination mit einem Eingabeobjekt aus der Klasse **FileInputStream**:

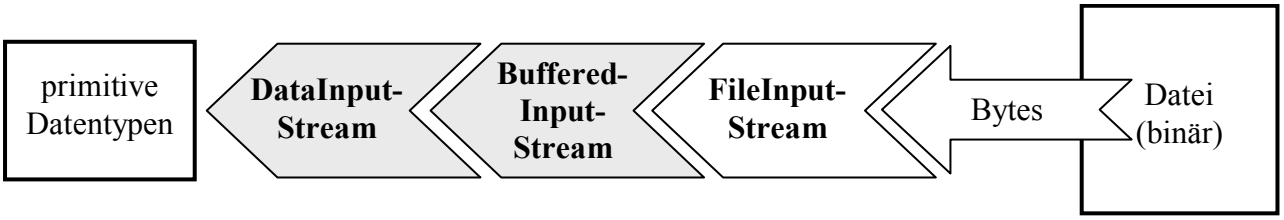


Beispiel:

```
DataInputStream dis = new DataInputStream(new FileInputStream("demo.dat"));
int i = dis.readInt();
double d = dis.readDouble();
```

Beim primitiven Datentyp **byte** ist kein **DataInputStream**-Filterobjekt erforderlich, weil bereits die **InputStream**-Methoden eine hinreichende Funktionalität bieten.

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));
```

11.8 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.*;` `class DataInput{` `  public static void main(String[] args) throws IOException {` `  String wort;` `  DataInputStream dis = new DataInputStream(System.in);` `  System.out.print("Wort eingeben: ");` `  wort = dis.readLine();` `  System.out.println("Sie gaben ein: " + wort);` `  dis.close();` `  }` `}`	`Wort eingeben: egal` `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: DataInput.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.

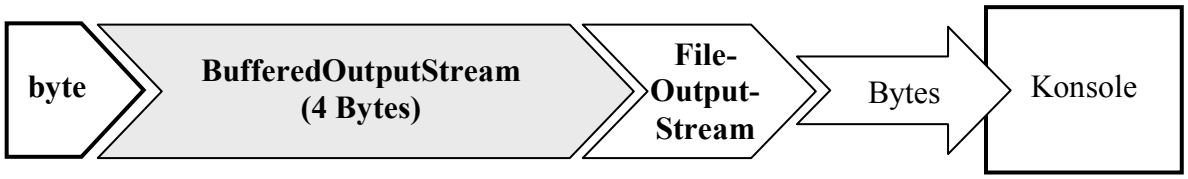
11.9 Übungsaufgaben zu Abschnitt 11

1) Erweitern Sie das Editor-Beispielprogramm in Abschnitt 10.7 um die Möglichkeit, den bearbeiteten Text zu sichern. Analysieren Sie auch die vorgeschlagene Methode zum Lesen einer Textdatei.

2) 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 11.3.1.5 wissen, ist der per **System.out** ansprechbare **PrintStream** bei 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 fdout =
    new FileOutputStream(FileDescriptor.out);
BufferedOutputStream bos =
    new BufferedOutputStream(fdout, 4);
```

Über die statische Variable **out** der Klasse **FileDescriptor** wird der Bezug zur Konsole hergestellt. Dorthin schreibt der **FileOutputStream** `fdout`, 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 aber 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.
Sollten sich bei Programmende noch Bytes im Puffer befinden, müssen diese per **flush()** oder **close()** vor dem Untergang bewahrt werden.

3) Wie kann man beim folgenden Programm den Quellcode vereinfachen und dabei auch noch die Laufzeit erheblich reduzieren?

```
import java.io.*;

class Prog {
  public static void main(String[] egal) throws IOException {
    long time = System.currentTimeMillis();
    FileOutputStream fos = new FileOutputStream("pw.txt");
    PrintWriter pw = new PrintWriter(fos, true);
    for (int i = 1; i < 30000; i++) {
      pw.println(i);
    }
    pw.close();
    System.out.println("Zeit: " + (System.currentTimeMillis()-time));
  }
}
```

4) 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**. Nun kann man hoffen, mit einem **DataInputStream**-Objekt, das auf **System.in** aufsetzt, Werte primitiver Datentypen von der

Tastatur entgegen nehmen zu können. In folgendem Programm wird versucht, beim Benutzer einen **short**-Wert (Ganzzahl mit 2 Bytes) zu erfragen:

Quellcode	Ausgabe
`import java.io.*;` `class Prog {` `  public static void main(String[] args)` `  throws IOException {` `  short zahl;` `  DataInputStream dis = new DataInputStream(System.in);` `  System.out.print("Zahl eingeben: ");` `  zahl = dis.readShort();` `  System.out.println("OK: " + zahl);` `  }` `}`	`Zahl eingeben: 0` `OK: 12301`

Ein Blick auf den Beispieldialog zeigt, dass offenbar eine Transformation stattfindet, deren Ergebnis jedoch wenig brauchbar ist.
Wie ist das Ergebnis zu erklären?
Wie sollte man numerische Daten von der Konsole lesen?
Wozu sollte man die Klasse **DataInputStream** verwenden?

5) Zum Schluss noch eine Aufgabe für besonders aufmerksame Leser(innen): Die beiden folgenden Programme unterscheiden sich darin, dass ein **PrintWriter** bzw. ein **PrintStream** zur Ausgabe von Zeichenfolgen auf der Konsole eingesetzt wird. In beiden Fällen ist die **autoFlush**-Option eingeschaltet, und es kommt die in beiden Klassen vorhandene **print()**-Methode zum Einsatz. Wie ist das unterschiedliche Verhalten zu erklären?

Quellcode	Ausgabe
`import java.io.*;` `class Prog {` `  public static void main(String[] args)` `  throws IOException {` `  PrintStream px = new PrintStream(` `  new FileOutputStream(FileDescriptor.out), true);` `  for (int i = 0; i < 10; i++)` `  px.print("Zeile "+i+"\n");` `  }` `}`	`Zeile 0` `Zeile 1` `Zeile 2` `Zeile 3` `Zeile 4` `Zeile 5` `Zeile 6` `Zeile 7` `Zeile 8` `Zeile 9`

Quellcode	Ausgabe
`import java.io.*;` `class Prog {` `  public static void main(String[] args)` `  throws IOException {` `  PrintWriter px = new PrintWriter(` `  new FileOutputStream(FileDescriptor.out), true);` `  for (int i = 0; i < 10; i++)` `  px.print("Zeile "+i+"\n");` `  }` `}`	

12 Applets

Java war lange vor allem dazu gedacht, das WWW mit kleine Prográmmchen (Applets) aufzuwerten, die von einem Webserver schnell via Internet zum lokalen Rechner transportiert und dort im 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.
Wie ein interaktives, optisch attraktives Java-Applet sich im Browser-Fenster darstellt, war schon in der Einleitung zu sehen. Wer auch die akustischen Qualitäten des **TicTacToe**-Beispiels genießen möchte, muss das zum Java 2 SDK gehörige Applet auf einem multimedia-tauglichen Rechner ausführen.
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, teilweise einfacher zu realisierende, Möglichkeiten zur dynamischen und interaktiven Gestaltung von Webseiten (z.B. Javaskript, Macromedia Flash, animierte Gifs, ActiveX). Nur bei besonders anspruchsvollen Web-Anwendungen (z.B. Internet-Banking, komplexe Visualisierungen) hat sich Java seine Vorrangstellung bewahrt.
- 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. M.E haben Java-Applikationen mittlerweile eine größere Bedeutung als Applets. Außerdem werden Java-Lösungen auf dem Webserver (**J**ava **S**erver **P**ages, Servlets) immer populärer.

Damit haben Applets zwar ihre ursprüngliche Bedeutung verloren, doch bieten sie nach wie interessante Möglichkeiten für viele Szenarien und stellen damit einen Zusatznutzen der Programmiersprache Java dar.

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-Aplikation mit geringem Aufwand in ein Java-Applet umzuwandeln, informiert z.B. Krüger (2002, Abschnitt 40.3).

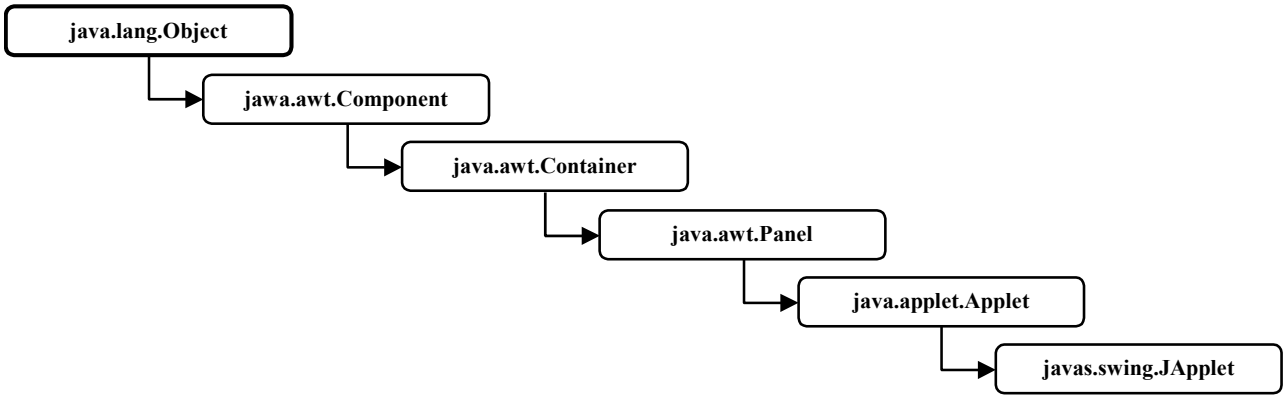
Damit ein Java-Applet ausgeführt werden kann, muss der Browser eine virtuelle Java-Maschine anbieten, die auf entsprechendem Versionsstand ist. Bei der Applet-Entwicklung für *beliebige* Internet-Nutzer mit unbekanntem Browser sollte man sich daher auf den Stand von Java 1.1 beschränken und z.B. beim GUI-Design auf die Swing-Komponenten verzichten.
Günstiger ist die Situation bei *Intranet*-Anwendungen, weil dort die Ausrüstung bzw. Konfiguration der Browser eher unter Kontrolle ist. Weitere Hinweise und Lösungsvorschläge zur Browser-Problematik finden Sie in Abschnitt 12.5.
Während bei Java-*Programmen* die Swing-Komponenten eindeutig zu bevorzugen sind, hängt also bei Applets die Entscheidung von den konkreten Umständen ab. Für den aktuellen Abschnitt des Kurses habe ich mich dafür entschieden, der Kompatibilität halber mit *AWT*-Komponenten zu arbeiten, wobei die eigentlichen Lerninhalte zum Thema *Applets* von der GUI-Frage weitgehend unberührt sind.
Allerdings garantiert der Verzicht auf Swing-Komponenten noch keine Kompatibilität mit älteren virtuellen Maschinen. Im Prinzip muss dazu auch ein Compiler mit passender Version verwendet werden. Z.B. kann die im altehrwürdigen Browser Netscape 4.7 eingebaute VM mit Java-Version 1.1.5 die in diesem Abschnitt vorgestellten Applets in der Regel ausführen, wenn sie mit dem JDK 1.3 erstellt werden, während die Produkte der jüngeren JDK-Versionen eher nicht funktionieren.

Nach den langen Vorbemerkungen und Einordnungsversuchen wollen wir nun den Aufbau und die Verhaltensweisen eines Applets näher betrachten.

12.1 Stammbaum der Applet-Basisklasse

Wie eine Java-*Anwendung* besteht auch ein Java-*Applet* aus mindestens einer Klassendefinition, wobei die beim Applet-Start aktive Hauptklasse stets aus **java.applet.Applet** (bei Beschränkung auf AWT-Komponenten) oder aus **javax.swing.JApplet** (bei Verwendung von Swing-Komponenten) abgeleitet werden muss.

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 Kooperations-Methoden werden gleich mit Hilfe des folgenden Beispiels erläutert, das gemäß obiger Verabredung auf der Klasse **java.applet.Applet** basiert:

```
import java.awt.*;
import java.applet.*;

public class Life extends Applet {
  private String iniStatus;
  private int startCount;
  private int stopCount;
  private int paintCount;

  public void paint(Graphics g) {
    paintCount++;
    g.drawString("Applet-Status:",0, 10);
    g.drawString("-----------------",0, 20);
    g.drawString(iniStatus,0, 40);
    g.drawString("start()-Aufrufe = " + startCount, 0, 60);
    g.drawString("stop()-Aufrufe = " + stopCount, 0, 80);
    g.drawString("paint()-Aufrufe = " + paintCount, 0, 100);
  }

  public void init() {
    setBackground(Color.WHITE);
    iniStatus = "Neu initialisiert";
  }
```

```
    public void start() {
       startCount++;
    }

    public void stop() {
       stopCount++;
       iniStatus = "Altlast";
    }

    public void destroy() {
       AudioClip clip = getAudioClip(getCodeBase(),"Kuckuck.au");
       clip.play();
       long time = System.currentTimeMillis();
       while (System.currentTimeMillis() - time < 600);
    }
}
```

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.

12.2 Applet-Start via Browser oder Appletviewer

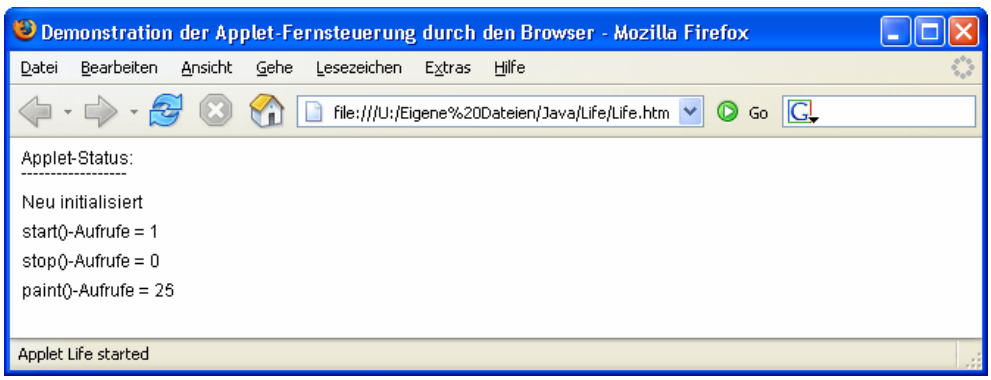
Wesentliche Eigenart eines Applets ist seine Einbettung in eine HTML-Seite, wobei im Normalfall ein so genanntes **applet**-Tag verwendet wird, das im `Life`-Beispiel z.B. folgendermaßen aussehen kann:

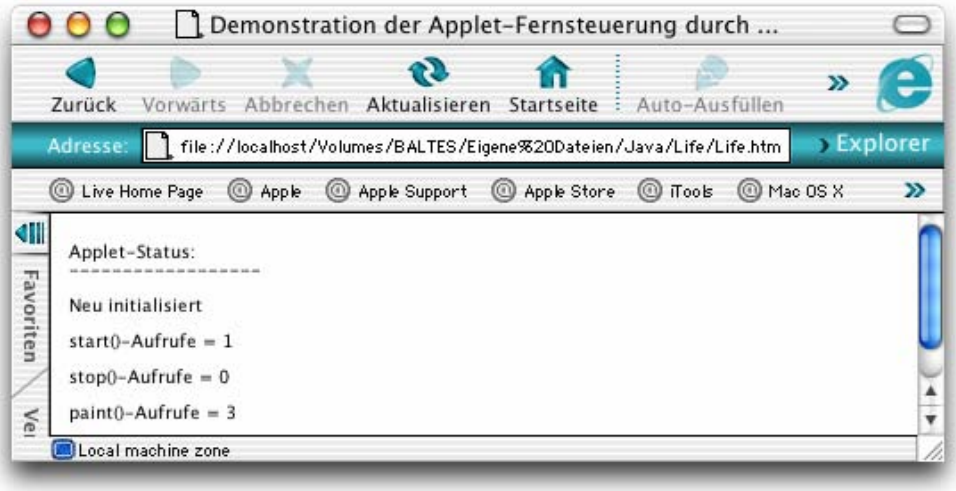
```
<html>
<head>
<title>Demonstration der Applet-Fernsteuerung durch den Browser</title>
</head>
<body>
<applet code="Life.class" width=300 height=120>
</applet>
</body>
</html>
```

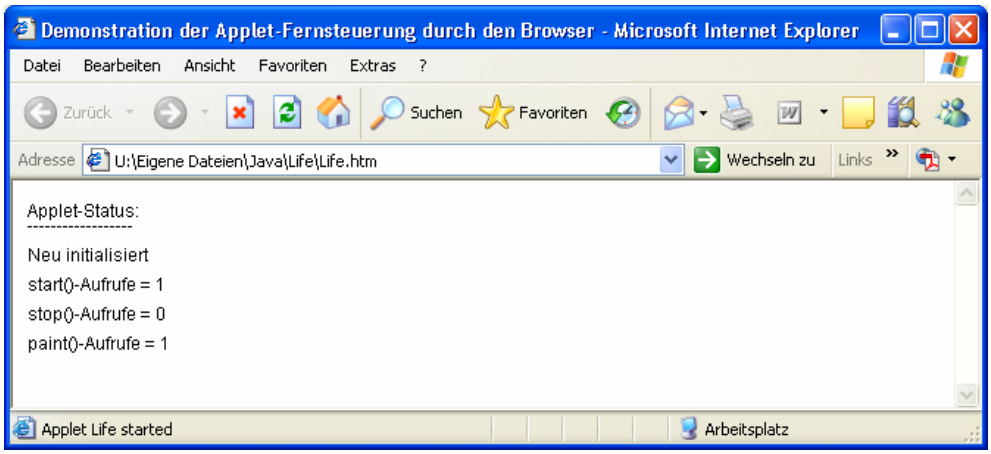
Bei den Attributen des Applet-Tags beschränkt sich das Beispiel auf die wichtigsten Angaben:

- **code**
  Name der Bytecodedatei zur Hauptklasse
- **width**, **height**
  Die Breite und Höhe der Appletfläche in Pixeln

Um ein Applet in einem Browser auszuführen, öffnet man das zugehörige HTML-Dokument, z.B. per URL-Eingabe, über das **Datei**-Menü oder per Drag-&-Drop, z.B.:



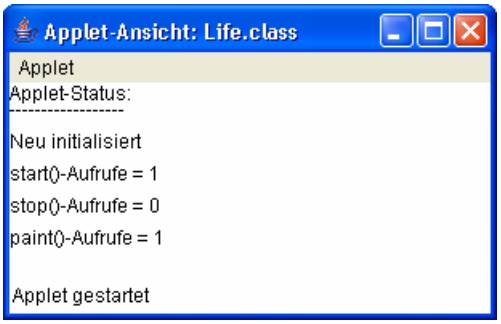




Die Rolle des Browsers beim Starten und Ausführen eines Java-Applets kann auch vom JDK-Hilfsprogramm **Appletviewer** übernommen werden. In der Entwicklungsphase ist der Appletviewer sogar zu bevorzugen, weil die übliche Cache-Strategie der Browser das Testen der aktuellen Applet-Version oft erschwert.
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 im vorigen Abschnitt wiedergegebene Applet `Life` sieht im Fenster des Appletviewers unmittelbar nach dem Start folgendermaßen aus:



In der Entwicklungsumgebung Eclipse 3.x hat man zum Starten eines Applets dieselben Optionen wie beim Starten einer Anwendung. Beim ersten Start wählt man also z.B. die Option

Ausführen als > Java-Applet

aus dem Kontextmenü zur Startklasse im Paket-Explorer. Bei späteren Starts genügt ein Mausklick auf den **Ausführen**-Schalter  . Eclipse verwendet den JDK-Appletviewer und erzeugt die umrahmende HTML-Seite selbst.

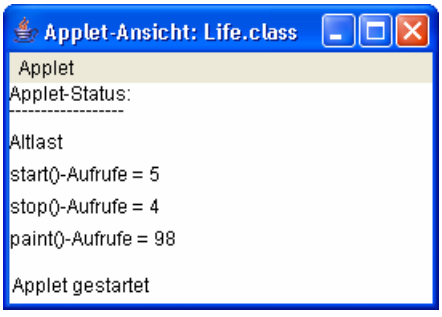
Über weitere Attribute des Applet-Tags und sonstige Optionen beim Einbinden von Applets auf HTML-Seiten informiert z.B. Münz (2005).

12.3 Methoden für kritische Lebensereignisse

Wer das Applet in Abschnitt 12.1 vor dem Hintergrund unserer bisherigen Erfahrungen mit Java-Anwendungen näher betrachtet, wird sicher die **main()**-Methode vermissen, über die eine Java-*Anwendung* vom Interpreter in Gang setzt wird. Bei einem Applet läuft die Initialphase deutlich anders ab:

- Der Browser (oder Appletviewer) erzeugt automatisch ein Objekt der im Applet-Tag angegebenen Klasse, die aus diesem Grund unbedingt als **public** deklariert werden muss.
- Üblicherweise benutzt man *keinen* Konstruktor der Klasse, um das beim Applet-Start erzeugte Objekt zu initialisieren. Dazu verwendet man die **Applet**-Methode **init()**, die vom Browser automatisch ausgeführt wird.
- Nach **init()** führt der Browser (oder Appletviewer) die **start()**-Methode des Applets aus. Während die **init()**-Methode eines Applet-Objektes nur *einmal* aufgerufen wird, führt der Browser die **start()**-Methode auch beim Reaktivieren eines zwischenzeitlich gestoppten Applet-Objektes aus.
- Schließlich wird die Methode **paint()** des Applet-Objektes aufgerufen, mit der die Grafikelemente gezeichnet werden (vgl. Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**). Dies geschieht im weiteren Leben des Objektes immer dann, wenn das Fenster aus irgendeinem Grund neu gezeichnet werden muss, z.B. nach der Rückkehr aus dem ikonisierten Zustand.

In der **init()**-Methode unseres Beispiels wird mit der **Applet**-Methode **setBackground()** die Hintergrundfarbe der Applet-Fläche festgelegt. Außerdem wird eine **String**-Instanzvariable zur Statusanzeige vorbereitet.
Die Methoden **start()** und **paint()** erhöhen im Beispiel jeweils eine Zählvariable, welche die Anzahl ihrer bisherigen Aufrufe festhält.
In der **paint()**-Methode schreibt das via Referenzparameter übergebene **Graphics**-Objekt[42] mit seiner Methode **drawString()** alle Statusinformationen auf die Applet-Fläche, so dass wir über den bisherigen Lebensweg des Applets informiert sind, z.B.:



In Abschnitt 13.1 werden wir uns näher mit der Graphikausgabe beschäftigen.
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 Me-

[42] Streng genommen kann man einem ***Graphics**-Objekt* nicht reden, weil **Graphics** eine abstrakte Klasse ist. Allerdings kennen wie die konkrete, aus **Graphics** abgeleitete, Klasse des Parameter-Objektes *nicht*, so dass wir uns der Einfachheit halber die ungenaue Redeweise genehmigen.

thoden sind in den Basisklassen als **public** definiert, und beim Überschreiben dürfen generell keine Zugriffsrechte eingeschränkt werden.
Die geerbten Varianten der Methoden sind übrigens unterschiedlich aktiv:

- Die **(J)Applet**-Methoden **init()** und **start()** tun *nichts*.
- Die **Container**-Methode **paint()** reicht die Aufforderung zum Zeichnen nötigenfalls an Kindfenster weiter, was in unserem Beispiel nicht erforderlich ist.

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 deaktivieren. Dabei bleiben alle Ressourcen des Applets erhalten, so dass es später fortgesetzt werden kann, wobei die **start()**-Methode erneut ausgeführt wird. Aktuelle Browser machen allerdings wenig Gebrauch von der **stop()**-Methode.
- Ist ein Applet-Objekt am Ende seiner Existenz angelangt, führt der Browser seine **destroy()**-Methode aus, wobei alle Ressourcen freigegeben werden. Dies geschieht z.B. beim Wechsel auf eine andere Webseite.

12.4 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 (2002, Abschnitt 40.2) informieren.

12.4.1 Parameter 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>
```

Ein Applet wird seine Steuerparameter in der Regel schon in der **init()**-Methode ermitteln, z.B.:

```
import java.awt.*;
import java.applet.*;

public class Param extends Applet {
  private String par1;
  private int pari1;

  public void init() {
    setBackground(Color.WHITE);
    par1 = getParameter("Par1");
    try {pari1 = Integer.parseInt(par1);}
    catch (Exception e) {//Ausnahmen behandeln
    }
  }
```

```
    public void paint(Graphics g) {
       g.drawString("Par 1 = " + pari1,30, 30);
    }
}
```

Die zuständige Methode **getParameter()** liefert nur Strings, so dass eventuell mit den entsprechenden Methoden der Wrapper-Klassen noch eine Konvertierung vorzunehmen ist.
Im Beispiel erhalten wir folgendes Ergebnis:

12.4.2 Browser-Statuszeile ändern

Über seine Methode **showStatus()** hat ein Applet Zugriff auf die Browser-Statuszeile. In folgendem Beispiel werden dort Besuche der Maus dokumentiert:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Statuszeile extends Applet {
   public void init() {
      setBackground(Color.ORANGE);
      addMouseListener(new MausDetector());
   }

   class MausDetector 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 10.6 vorgestellte Ereignisbehandlung auch bei AWT-basierten Applets verwendbar:

12.4.3 Andere Webseiten öffnen

Über die Möglichkeit, aus einem Applet heraus HTML-Seiten zu öffnen, kann man z.B. ein Frameset mit Navigationszone oder auch eine Imagemap erstellen. Wir wollen uns mit dem ersten Thema beschäftigen und das folgende Frameset realisieren. Als Browser kommt diesmal Safari unter MacOS 10 zum Einsatz:

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 **Content1.htm** (mit der animierten Java-Tasse) geöffnet.
Das gesamte Frameset ist in der Datei **NaviDemo.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.

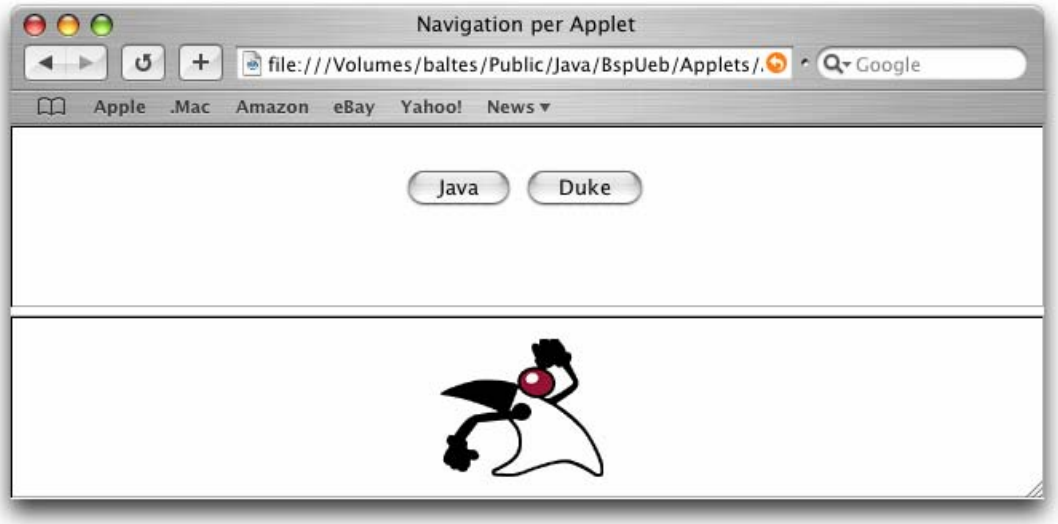
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.net.*;

public class Appligator  extends Applet  implements ActionListener {
   private Button duke = new Button("Duke");
   private Button java = new Button("Java");

   public void init() {
      setBackground(Color.WHITE);
      add(java);
      add(duke);
      java.addActionListener(this);
      duke.addActionListener(this);
   }

   public void actionPerformed(ActionEvent e)
   {
      URL neus;
      try {
         if (e.getSource() == java)
            neus = new URL(getCodeBase(), "Content1.htm");
         else
            neus = new URL(getCodeBase(), "Content2.htm");
         getAppletContext().showDocument(neus, "content");
      }
      catch (Exception ex) {//Ausnahmebehandlung
      }
   }
}
```

In der **init()**-Methode wird zunächst mit **setBackground()** die Hintergrundfarbe des Applets eingestellt.
Schon in Abschnitt 10.2.2 wurde das Applet als Top-Level-Container eingeordnet. Daher kann es mit seiner **add()**-Methode die beiden **Button**-Komponenten aufnehmen. Im Unterschied zur **JFrame**-Komponente wird hier keine spezielle *Schicht* (z.B. Content Pane) angesprochen.

Außerdem ist die Klasse `Appligator` als Ereignisempfänger 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**), welche über die Methode **getCodeBase()** im Pfad des Applets lokalisiert wird.

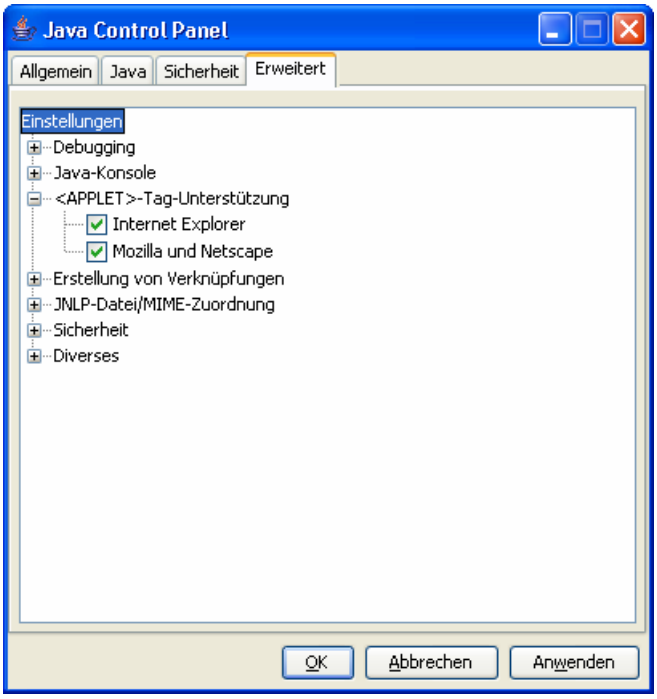
Mit **getAppletContext()** wird ein Objekt erzeugt, welches den Browser vertritt. An dieses Objekt richtet sich dann die Botschaft **showDocument()**, die als Parameter den URL der anzuzeigenden HTML-Seite sowie den Zielframe enthält. Wie alternative Ausgabeziele angesprochen werden können, ist in der API-Dokumentation zur **AppletContext**-Methode **showDocument()** zu erfahren. Z.B. erhält man mit der Zielangabe **_blank** ein neues Toplevel-Browserfenster.

Abschließend soll noch der Zustand nach einem Mausklick auf den Schalter **Duke** gezeigt werden:



12.5 Das Java-Browser-Plugin

Mit dem von Sun kostenlos angebotenen Java-Plugin können der Mozilla/Netscape – Browser sowie Microsofts Internet Explorer mit einer aktuellen JVM versorgt werden. Auf vielen Rechner ist das Java-Plugin schon unbemerkt installiert worden, z.B. zusammen mit der Java Runtime Environment (JRE) ab Version 1.2 oder mit dem entsprechenden Java SDK. Die genannten Sun-Produkte erweitern bei ihrer Installation die vorgefundenen Browser um das Java-Plugin.
Unter MS-Windows wird mit dem Java-Plugin auch ein Systemsteuerungs-Konfigurationsprogramm installiert. Hier kann man z.B. einstellen, ob das Plugin als Java-Standard-Laufzeitumgebung für die unterstützten Browser verwendet werden soll:



Ein aktives Java-Plugin macht sich im Statusbereich der Windows-Taskleiste (neben der Uhr) durch ein Java-Symbol bemerkbar.

12.6 Übungsaufgaben zu Abschnitt 12

1) Erstellen Sie ein Applet zur Primzahlendiagnose, z.B. mit folgender Benutzeroberfläche:

2) Erstellen Sie eine Applet-Variante zum Euro-DM-Konverter, den Sie als Übungsaufgabe zu Abschnitt 10 entwickelt haben.
Bei diesem Vorschlag wird abweichend von unserer bisherigen Praxis ein *Swing*-GUI verwendet:

13 Multimedia

Dieser Abschnitt enthält elementare Hinweise zu zweidimensionaler Grafik und Sound, die für Applets und Anwendungen in gleicher Weise gelten.

13.1 Grafik

Die bereits behandelten GUI-Komponenten bieten etliche Möglichkeiten zur individuellen Gestaltung der Benutzeroberfläche (z.B. durch Texte in diversen Schriftarten, Bilder oder speziell gestaltete Rahmen). Allerdings ist oft eine freie Grafikausgabe unumgänglich, die von Java mit zahlreichen Methoden zum Zeichnen elementarer Gebilde (z.B. Linien, Ellipsen, Rechtecke, Texte) unterstützt wird.
Weil die meisten von **jawa.awt.Component** abstammenden Klassen bemalt werden können, bieten sich dem kreativen Programmierer verschiedene Möglichkeiten, z.B.:

- **Bemalen einer leeren Zeichenfläche**
  In einem Swing-GUI eignet sich vor allem die Komponente **JPanel** als Zeichenfläche für das freie Gestalten. Zwar kann man auch Top-Level-Container wie **JFrame** bemalen, doch insbesondere bei einer Kombination von Swing-Bedienelementen mit freien Grafiken sollten letztere auf **JPanel**-Komponenten untergebracht werden.
- **Eigene Bedienelemente entwerfen**
  Von einer Standardkomponente wie z.B. **JButton** ausgehend kann man eine eigene Bedienelement-Klasse mit dem gewünschten Design ableiten. Wenngleich die Zeichnungsfläche stets rechteckig ist, lassen sich über intelligente Ereignismethoden auch anders geformte Bedienelemente realisieren, z.B. eine achteckige Stopp-Schalfläche.

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). Bei Applet-Beispielen beschränken wir uns trotzdem auch in Abschnitt 13.1 aus Kompatibilitätsgründen auf das AWT-GUI.

Für besonders anspruchsvolle Grafikausgaben empfiehlt sich das in diesem Kurs aus Zeitgründen nicht behandelte **Java 2D API** (siehe z.B. SUN Inc. 2005).

13.1.1 Die Klasse Graphics

Im Abschnitt 12 haben wir in der **paint()**-Methode einiger Applets ohne große Erläuterungen die Methode **drawString()** verwendet, um Text als *Grafikelement* auf die Appletfläche zu bringen, z.B.:

```
g.drawString("Applet-Status:",0, 10);
```

Die Zeichenbefehle haben wir an ein per Referenzparameter zur Verfügung gestelltes Objekt geschickt, das die abstrakte Klasse **java.lang.Graphics** erfüllt.
Zu welcher *konkreten* Klasse das (automatisch erzeugte) Objekt gehört, kann man über die **Object**-Methode **getClass()** ermitteln:

```
g.drawString(g.getClass().toString(), 50,50);
```

Bei Java 1.5.0 stellt man z.B. unter MS-Windows die folgende **Graphics**-Subklasse fest:

sun.java2d.SunGraphics2D

Auf einem Macintosh mit Java 1.3 ist anzutreffen:

com.apple.mrj.internal.awt.graphics.PenGraphics

Während die abstrakte Klasse **Graphics** eine plattformunabhängige Grafik-Schnittstelle definiert, stellt z.B. die konkrete Klasse **com.apple.mrj.internal.awt.graphics.PenGraphics** deren plattformspezifische Implementation auf dem Macintosh dar.
Wir haben uns schon in Abschnitt 12.3 entschieden, der Einfachheit halber die nicht ganz korrekte Bezeichnung **Graphics**-Objekt zu verwenden, um von der Plattform unabhängig zu bleiben und die umständlichen Namen der konkreten Klassen zu vermeiden.

Ein **Graphics**-Objekt bietet nicht nur ca. 50 Ausgabemethoden für diverse graphische Elemente wie Linien, Rechtecke, Ovale, Polygone etc. (siehe API-Referenz), sondern stellt auch einen so genannten **Grafikkontext** mit allen für eine Grafikausgabe relevanten Informationen zur Verfügung. U.a. kennt das **Graphics**-Objekt von der Komponente, deren Oberfläche zu bemalen ist:

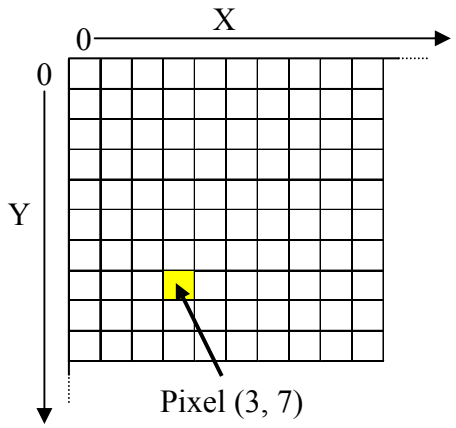
- Bildschirmposition und Größe des zu zeichnenden Rechtecks
- Vorder- und Hintergrundfarbe
- Schriftart

Zu jeder sichtbaren Komponente, also zu jedem sichtbaren Objekt aus einer von **jawa.awt.Component** abstammenden Klasse gehört ein Grafikkontext.[43]

Um auf die Oberfläche einer Komponente zeichnen zu können, beschafft man sich mit der **Component**-Methode **getGraphics()** eine Referenz zum Grafikkontext der Komponente. Bei Verwendung der **Component**-Methode **paint()**, die bei Applets und GUI-Anwendungen z.B. dann aufgerufen wird, wenn die Komponenten-Oberfläche ganz oder teilweise neu zu zeichnen ist, wird automatisch ein **Graphics**-Objekt mitgeliefert.

13.1.2 Das Koordinatensystem der Zeichenfläche

Die Grafikausgabe basiert 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:

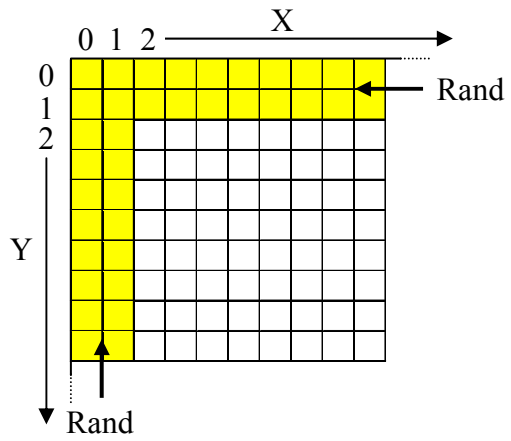


Weil die Positionsangaben in Pixeln gemacht werden, ist die tatsächliche Größe eines Grafikelementes von Größe und Auflösung des Bildschirms abhängen.

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:

---

[43] 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** und **Applet** sowie bei der Komponente **Canvas**.



Über die aktuelle Größe der Zeichenfläche sowie der Ränder kann man sich z.B. mit den folgenden **JComponent**-Methoden informieren (siehe Beispiel in Abschnitt 13.1.3.3):

- **getHeight()**
- **getWidth()**
- **getInsets()**

13.1.3 Organisation der Grafikausgabe

Jede Grafikausgabe findet auf der Oberfläche einer Komponente statt, wobei sich dort auch ohne individuelle Grafikbefehle des Programmierers mehr oder weniger aufwändige „Gemälde" befinden, z.B. eine Hintergrundfarbe und eine Beschriftung.

Eine Komponente muss aus verschiedenen Anlässen neu gezeichnet bzw. aktualisiert werden, z.B.:

- Der erste Auftritt eines Fensters oder die Rückkehr aus dem minimierten Zustand
- Ein zuvor überdeckter Teil der Komponente wird sichtbar.
- Änderungen im Programm erfordern eine Aktualisierung der Anzeige.

Bisher haben wir uns nicht damit beschäftigt, 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 interessant:

- 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 einer GUI-Ausgabe kann keine Ereignisbehandlungsmethode ausgeführt werden.

13.1.3.1 System-initiierte Aktualisierung (paint)

Einer Komponente wird vom Laufzeitsystem automatisch eine **paint()**-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 folgender Signatur:

```
public void paint(Graphics g)
```

Per **Graphics**-Parameter wird der zum Zeichnen erforderliche Grafikkontext übergeben.

Bei **paint()** handelt es sich um eine typische **Callback**-Methode, die vom Programm bereit gehalten und vom Laufzeitsystem aufgerufen wird. Während ein Programm bzw. Applet in der Regel die **paint()**–Methoden der Komponenten nicht direkt aufruft, kann es dem Laufzeitsystem aber nahe legen, dies bei nächster Gelegenheit zu tun (siehe unten).

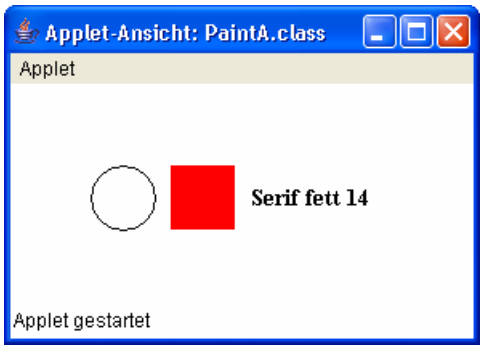
Um die Bemalung einer Komponente zu beeinflussen, definiert man eine eigene Klasse mit geeigneter Basisklasse und überschreibt eine Callback-Methode zur Grafikausgabe. Beim AWT-GUI wird die Methode **paint()** selbst überschrieben, beim komplexeren und leistungsfähigeren Swing-GUI hingegen meist eine nachgeordnete, von **paint()** aufgerufene Methode (siehe 13.1.3.3).

Mit dieser Vorgehensweise haben Sie im Zusammenhang mit den Applets schon Erfahrungen gesammelt, an die wir nun in einem Beispiel mit elementaren Grafikausgaben anknüpfen:

```
import java.awt.*;
import java.applet.*;

public class PaintA extends Applet {

   public void init() {
      setBackground(Color.WHITE);
      setFont(new 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);
   }
}
```

Hier 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. Mit **drawOval()** wird z.B. an der Bildschirmposition (50, 50) ein Oval mit einer Höhe und Breite von 40 Einheiten gezeichnet, also ein Kreis mit diesem Durchmesser. Weil die Hintergrundfarbe und die Schriftart des Applets unverändert bleiben, können sie schon in der **init()**-Methode mit **setBackground()** bzw. **setFont()** festgelegt werden.

Sobald das Applet seine Fläche neu zeichnen muss, ruft das Laufzeitsystem seine **paint()**-Methode auf, wo das übergebene **Graphics**-Objekt mit den erforderlichen Ausgaben beauftragt wird:



Im Beispielprogramm sind zwei wichtige Spezialthemen der Grafikausgabe tangiert, die leider nicht im angemessenen Umfang behandelt werden können:

- **Farben**
  Mit der **Graphics**-Methode **setColor()** kann die aktuelle Zeichenfarbe gesetzt werden, z.B.:[44]
  ```
  g.setColor(Color.RED);
  ```
- **Schriftarten**
  Mit der Methode **setFont()** kann man Schriftart und –auszeichnung für spätere Grafik-Textausgaben festlegen, wobei ein **Font**-Objekt zu übergeben ist, z.B.:
  ```
  g.setFont(new 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.
  Mit **Font**-Objekten haben wir übrigens schon in Abschnitt 10.7.2 gearbeitet.

Grundsätzlich erfolgt die Grafikausgabe bei einer Anwendung mit Swing-GUI nach derselben Logik. Für das folgende Trivialprogramm

wurde die **paint()**-Methode des Applets unverändert übernommen:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class PaintS extends JFrame {
   public PaintS() {
      super("Grafik im Swing-Anwendungen");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setBackground(Color.WHITE);
      setFont(new Font("Serif", Font.BOLD, 14));
      setSize(300, 125);
      setVisible(true);
      setResizable(false);
   }

   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);
   }

   public static void main(String[] args) {
      new PaintS();
   }
}
```

Dieses Beispiel ist allerdings *nicht* vorbildlich für die Grafikausgabe in Anwendungen mit Swing-GUI. In den meisten Fällen wird man dazu eine eigene Klasse aus **JPanel** ableiten, die Methode **paintComponent()** überschreiben und ein Objekt der eigenen Klasse in den **JFrame** - Top-Level-Container einfügen (siehe Abschnitt 13.1.3.3). Die Methode **paint()** bleibt auch bei **JPanel**-

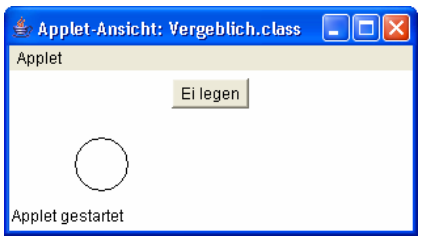
[44] Bis zur API-Version 1.3 waren abweichend von der üblichen Bezeichnungsweise *klein* geschrieben Farbkonstanten zu verwenden (z.B. `Color.red`). Seit der Version 1.4 sind beide Schreibweisen erlaubt.

Objekten im Spiel. Sie ruft **paintComponent()** auf und erledigt darüber hinaus wichtige andere Arbeiten, weshalb sie nicht überschrieben werden sollte.

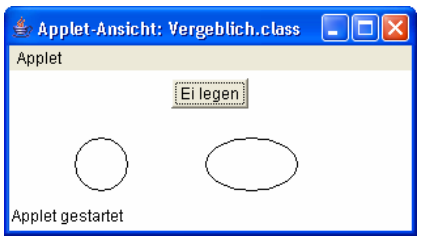
In folgendem Applet (mit AWT-GUI) soll die Rolle der Callback-Methode **paint()** noch einmal demonstriert werden. Um in der Ereignisbehandlungsmethode **actionPerformed()** zu einem Befehlsschalter auf die Fläche des Applets ein Ei zeichnen zu können, wird in der **init()**-Methode eine Referenz auf den zugehörigen Grafikkontext mit **getGraphics()** ermittelt und in der Instanzvariablen `mg` gespeichert:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Vergeblich extends Applet implements ActionListener {
   private Graphics mg;
   private Button ei = new Button("Ei legen");

   public void init() {
      setBackground(Color.WHITE);
      mg = getGraphics();
      ei.addActionListener(this);
      add(ei);
   }

   public void paint(Graphics g) {
      g.drawOval(50, 50, 40, 40);
   }

   public void actionPerformed(ActionEvent e) {
      mg.drawOval(150, 50, 70, 40);
   }
}
```

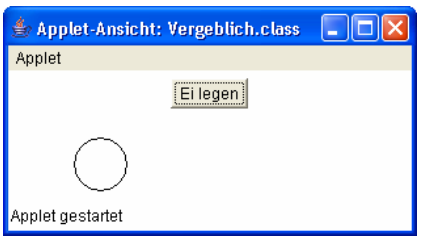
Ein Mausklick auf den Schalter hat zunächst den gewünschten Effekt, doch ist die Freude nur von kurzer Dauer:



Nach dem Start



Unmittelbar nach einem Mausklick auf den Schalter



Nach einer Größenänderung des Fensters

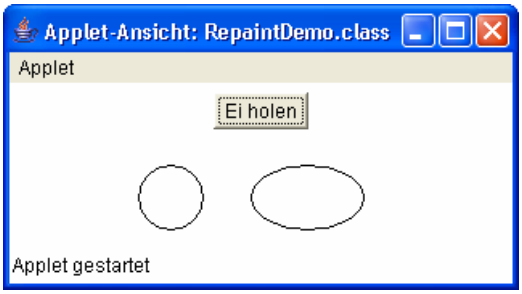
Was dauerhaft auf einer Komponente sichtbar sein soll, muss mit deren **paint()**-Methode gemalt werden. Gleich erfahren Sie, wie ein Programm bzw. Applet das Aktualisieren der Grafikanzeige veranlassen kann, ohne die Methode **paint()** direkt aufzurufen.

13.1.3.2 Programm-initiierte Aktualisierung (repaint)

Ein Programm oder Applet kann jederzeit die Aktualisierung der Grafikanzeige veranlassen, indem es die **Component**-Methode **repaint()** für die betroffene Komponente aufruft. In der folgenden Variante des Hühner-Applets verändert die Befehlsschalter-Ereignisroutine eine boolesche Instanzvariable der Applet-Komponente und ruft dann deren **repaint()**-Methode auf:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class RepaintDemo extends Applet implements ActionListener {
  private boolean eida = false;
  private Button ei = new Button("Ei legen");

  public void init() {
    setBackground(Color.WHITE);
    ei.addActionListener(this);
    add(ei);
  }

  public void paint(Graphics g) {
    g.drawOval(80, 50, 40, 40);
    if (eida)
      g.drawOval(150, 50, 70, 40);
  }

  public void actionPerformed(ActionEvent e) {
    eida = ! eida;
    if (eida)
      ei.setLabel("Ei holen");
    else
      ei.setLabel("Ei legen");
    repaint();
  }
}
```

Diese Konstruktion zeigt ein sinnvolles Verhalten, z.B.:



Zwar führt jeder **repaint()**-Aufruf letztlich zu einem **paint()**-Aufruf, doch entfaltet **repaint()** zuvor weitere Tätigkeiten, so dass ein direkter Aufruf der **paint()**-Methode *nicht* sinnvoll ist. Wer z.B. abenteuerlustig in obigem Applet den **repaint()**-Aufruf durch

```
paint(getGraphics());
```

ersetzt, wird beim Klick auf den mit Ei holen beschrifteten Schalter feststellen, dass der gewünschte Effekt (Ei verschwindet) nicht mehr eintritt.

Bei einem **Applet** hat eine **repaint()**-Anforderung wie bei allen schwergewichtigen Komponenten folgende Konsequenzen:

- **repaint()** ruft die **Component**-Methode **update()** auf.
- **update()** bemalt die Komponentenfläche mit der Hintergrundfarbe, löscht also den bisherigen Inhalt.
- **update()** ruft **paint()** auf.

Beim direkten **paint()**-Aufruf wird also u.a. das automatische Löschen des bisherigen Inhalts durch die **Component**-Methode **update()** übergangen.

Im Swing-API ist die Grafikaktualisierung in vielen Details anders gelöst (siehe Abschnitt 13.1.3.3); z.B. wird die **Component**-Methode **update()** für Swing-Komponenten nicht aufgerufen. Es gilt jedoch für beide APIs:

- Um die Anzeige einer Komponente zu aktualisieren, ruft man deren **repaint()**-Methode auf.
- Keinesfalls sollte man die **paint()**-Methode direkt aufrufen.

An Stelle der parameterfreien **repaint()**-Methode ist zur Beschleunigung der Grafikausgabe oft eine Überladung mit Angabe des tatsächlich aktualisierungsbedürftigen Rechtecks zu bevorzugen:

void repaint(**int** *x*, **int** *y*, **int** *width*, **int** *height*)

13.1.3.3 Details zur Grafikausgabe in Swing

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 (siehe Fowler 2003).
Die **paint()**-Implementation der Klasse **javax.swing.JComponent** ruft nacheinander folgende Methoden auf:

- **paintComponent()**
- **paintBorder()**
  Zeichnet den Rahmen der Komponente.
- **paintChildren()**
  Zeichnet die enthaltenen Komponenten.

Bei den meisten Swing-Komponenten wird das *Look & Feel* (das Erscheinungsbild und das Interaktionsverhalten) durch ein separates Objekt aus der Klasse **ComponentUI** (*UI delegate*) implementiert, auf das die Instanzvariable **ui** zeigt. In diesem Fall hat ein **paintComponent()**–Aufruf folgende Konsequenzen:

- **paintComponent()** ruft **ui.update()** auf.
- Sofern die Komponente *nicht* transparent ist, die Eigenschaft **opaque** also den voreingestellten Wert **true** besitzt, füllt **ui.update()** die Fläche der Komponente mit ihrer aktuellen Hintergrundfarbe.
- **ui.update()** ruft **ui.paint()**, und diese Methode zeichnet den Inhalt der Komponente.

Sofern die bei Swing-Komponenten voreingestellte **Doppelpufferung** nicht abgeschaltet wurde, erhalten **paintComponent()**, **paintBorder()** und **paintChildren()** einen Offscreen-Graphikkontext. Erst die fertig gezeichnete Fläche gelangt auf den Bildschirm, so dass kein lästiges Flackern auftreten kann.

Für individuelle Grafikausgaben in Java-Anwendungen mit Swing-GUI sollte (von Trivialfällen abgesehen) eine eigene Klasse aus **JPanel** abgeleitet werden:

- Bei Grafikausgaben auf der Oberfläche eines Top-Level-Containers kann es zu Überlappungen mit dort enthaltenen Komponenten kommen.
- Zwar unterscheidet sich die Klasse **JPanel** in der Java-Version 1.4 nur noch unwesentlich von ihrer Basisklasse **JComponent**, jedoch wird sie als Zeichnungsgrundlage in der Regel weiterhin bevorzugt.

Im folgenden Beispiel bietet die von **JPanel** abstammende Klasse `PaintPanel` eine Methode `wechsle()` an, um einen Kreis, ein Quadrat und einen Text unabhängig voneinander erscheinen bzw. verschwinden lassen:

```
import java.awt.*;
import javax.swing.*;

public class PaintPanel extends JPanel {
  private boolean kreisDa = false, quadratDa = false, textDa = false;

  public PaintPanel() {
    setFont(new Font("Serif", Font.BOLD, 14));
    setBorder(BorderFactory.createLineBorder(Color.BLUE,10));
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    int x = (getWidth()-174)/2;
    int y = (getHeight()-40)/2;
    if (kreisDa)
      g.drawOval(x, y, 40, 40);
    if (quadratDa) {
      g.setColor(Color.RED);
      g.fillRect(x+50, y, 40, 40);
    }
    if (textDa) {
      g.setColor(Color.BLACK);
      g.drawString("Serif fett 14", x+100, y+25);
    }
    Insets ins = getInsets();
    g.setColor(Color.YELLOW);
    g.fillRect(ins.left, ins.top, 10, 10);
    g.fillRect(ins.left, getHeight()-ins.bottom-10, 10, 10);
    g.fillRect(getWidth()-ins.right-10, ins.top, 10, 10);
    g.fillRect(getWidth()-ins.right-10, getHeight()-ins.bottom-10, 10, 10);
  }

  public void wechsle(int figur) {
    switch (figur) {
      case 1: kreisDa = !kreisDa; break;
      case 2: quadratDa = !quadratDa; break;
      case 3: textDa = !textDa; break;
    }
    repaint();
  }
}
```

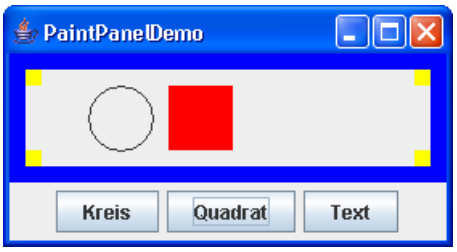
Bei der Definition einer auf **JComponent** zurück gehenden eigenen Klasse mit individueller Grafikausgabe sollte man statt **paint()** grundsätzlich die Methode **paintComponent()** überschreiben. Dann bleibt die oben beschriebene, wesentlich von der **JComponent**-Methode **paint()** realisierte Logik der Swing-Grafikausgabe erhalten (z.B. Doppelpufferung, Zeichnen von eventuell enthaltenen Kind-Komponenten).
Wer die Randzone einer Komponente individualisieren möchte, kann zusätzlich **paintBorder()** überschreiben.

Am Anfang der **paintComponent()**-Überschreibung sollte in der Regel mit

```
super.paintComponent(g);
```

die Basisklassenvariante aufgerufen werden, deren Verhalten oben skizziert wurde. Damit treffen die eigenen Grafikausgaben stets auf einen perfekt vorbereiteten Hintergrund.

Im folgenden Programm



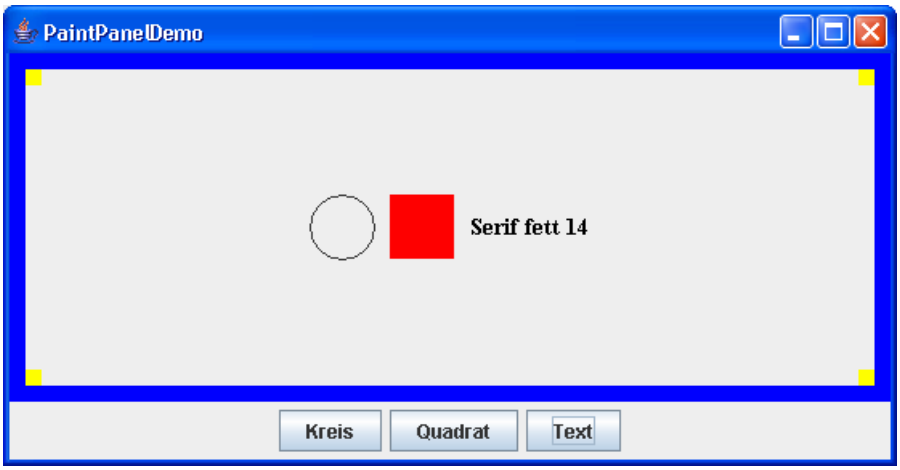
wird eine `PaintPanel`-Komponente eingesetzt und über die **actionPerformed()**-Methode zu drei Befehlsschaltern mit `wechsle()`-Botschaften versorgt:

```
public void actionPerformed(ActionEvent e) {
  if (e.getSource() == kreis) {
    tafel.wechsle(1);
  }
  else if (e.getSource() == quadrat) {
    tafel.wechsle(2);
  }
  else {
    tafel.wechsle(3);
  }
}
```

Wir fassen noch einmal zusammen, wie eine von **JComponent** abstammende eigene Komponente auf den Bildschirm gebracht wird:

- **super.paintComponent(g)** erstellt den Hintergrund.
- Es folgen die individuellen Grafikausgaben in **paintComponent()**.
- **paintBorder()** zeichnet ggf. einen Rand.
- Falls vorhanden, werden via **paintChildren()** die enthaltenen Komponenten gezeichnet.

Damit die schaltbaren Grafikelemente stets zentriert erscheinen, wird in der `PaintPanel`-Methode **paintComponent()** die aktuelle Größe der Zeichenfläche mit den **JComponent**-Methoden **getWidth()** und **getHeight()** ermittelt.



Mit **getInsets()** beschafft sich **paintComponent()** ein **Insets**-Objekt, dessen Instanzvariablen **top**, **bottom**, **left** und **right** über die Breite der Ränder informieren. Damit können die vier gelben Quadrate so positioniert werden, dass sie bei beliebiger Wahl der Umrandung in den Ecken erscheinen.

13.2 Sound

In folgendem Applet wird eine Musikbox realisiert, die derzeit 3 Stücke vorspielen kann, aber beliebig ausbaufähig ist:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Sound extends Applet implements ActionListener {
  private Button start = new Button("Start");
  private Button wechseln = new Button("Wechseln");
  private Button stop = new Button("Stop");
  private Label label = new Label("");
  private final static int maxclip = 7;
  private int actclip = 0;
  private boolean music;
  private AudioClip[] clip = new AudioClip[maxclip];
  private String[] song = new String[maxclip];

  public void init() {
    setBackground(Color.WHITE);
    setLayout(new BorderLayout());
    start.addActionListener(this);
    wechseln.addActionListener(this);
    stop.addActionListener(this);

    add(start, BorderLayout.WEST);
    add(wechseln, BorderLayout.CENTER);
    add(stop, BorderLayout.EAST);
    add(label, BorderLayout.SOUTH);

    java.net.URL cb = getCodeBase();
    clip[0] = getAudioClip(cb,"chirp1.au");
    song[0] = "Vogel-Geschwister (.au)";
    clip[1] = getAudioClip(cb,"Kuckuck.au");
    song[1] = "Kuckuck (.au)";
    clip[2] = getAudioClip(cb,"trippygaia1.mid");
    song[2] = "Trippygaia (mid)";
    clip[3] = getAudioClip(cb,"minuet.mid");
    song[3] = "Menuett (mid)";
    clip[4] = getAudioClip(cb,"reggae.mid");
    song[4] = "Reggae (mid)";
    clip[5] = getAudioClip(cb,"jazz.mid");
    song[5] = "Jazz (mid)";
    clip[6] = getAudioClip(cb,"weissNicht.mid");
    song[6] = "Keine Ahnung (mid)";
    label.setText(song[0]);
  }

  public void stop() {
    clip[actclip].stop();
  }

  public void start() {
    if (music)
      clip[actclip].loop();
  }

  public void destroy () {
    clip[actclip].stop();
  }
```

```
    public void actionPerformed(ActionEvent e) {
      clip[actclip].stop();
      if (e.getSource() == stop)
        music = false;
      else {
        if (e.getSource() == wechseln) {
          if (actclip < maxclip-1)
            actclip++;
          else
            actclip = 0;
          label.setText(song[actclip]);
        } else // Source == start
          music = true;
        if (music)
          clip[actclip].loop();
      }
    }
}
```

Die **Applet**-Methode **getAudioClip()** erstellt aus einer Audiodatei ein Objekt aus einer Klasse[45], die das Interface **AudioClip** erfüllt. Im Beispiel werden die Audiodateien mit Hilfe eines URL-Objektes im Pfad des Applets lokalisiert:

```
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

Ein Überschreiben der **Applet**-Methode **paint()** ist nicht erforderlich, weil sich auf der Oberfläche des Applets nur Komponenten befinden. Diese zeichnen sich selbst, sobald sie dazu aufgefordert werden.

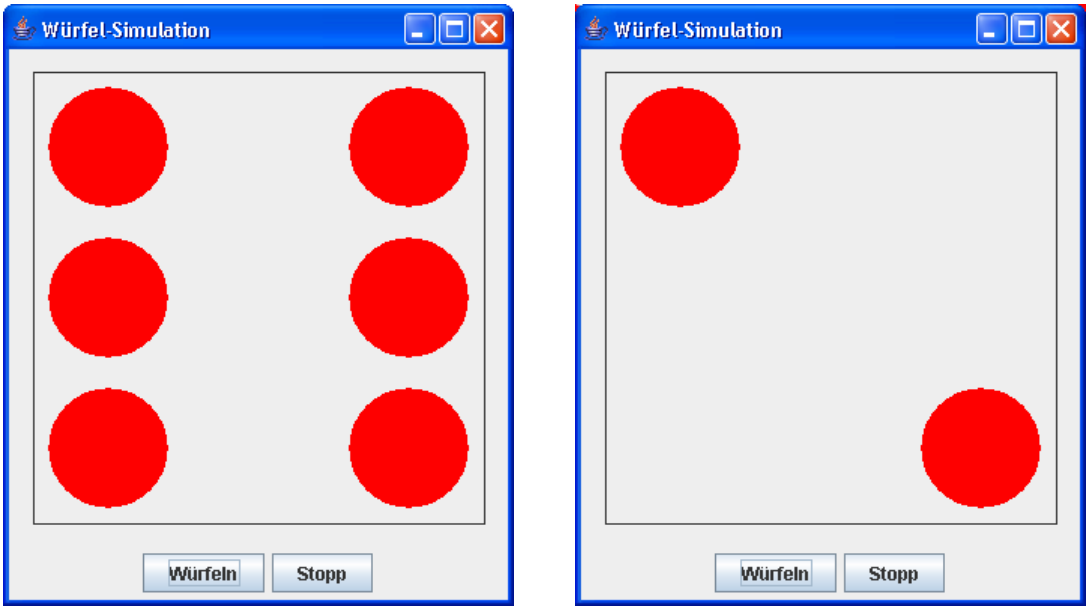
Die Bedienungsoberfläche hat noch kein disko-taugliches Design, sieht aber zumindest auf einem Macintosh schon recht passabel aus:

[45] Es handelt sich um die Klasse **sun.applet.AppletAudioClip**, was man aber als Programmierer normalerweise nicht wissen muss.

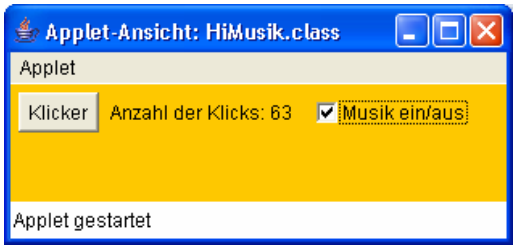
Beim Einsatz des Applets in einem Browser ist zu beachten, dass eine virtuelle Maschine mit der minimalen Java-Version 1.2 benötigt wird (vgl. Abschnitt 12.5).

13.3 Übungsaufgaben zu Abschnitt 13

1) Erstellen Sie ein Würfel-Simulationsprogramm nach folgendem Muster:



2) Erstellen Sie ein Applet, das zum Zählen von Ereignissen beliebiger Art über einen Schalter sowie ein Label mit aktueller Anzeige des Zählerstandes 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:



14 Threads

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 aktuellen Internet-Browser muss man z.B. nicht untätig warten, bis die von einem langsamen Server gelieferte WebSeite aufgebaut ist, sondern kann parallel mit anderen Browser-Fenstern arbeiten.

Die Multithreading-Technik kommt immer dann in Frage, wenn eine Anwendung mehrere Aufgaben gleichzeitig erledigen soll. Allerdings ist eine sorgfältige Einsatzplanung erforderlich, denn:

- Thread-Wechsel sind mit einem gewissen Zeitaufwand verbunden und sollten daher nicht zu häufig stattfinden.
- In der Regel erfordert die Synchronisation von Threads einige Aufmerksamkeit (siehe unten).

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 Multithread-Unterstützung in die Sprache (genauer: in das API) eingebaut und von jedem Programmierer ohne große Probleme zu nutzen.

14.1 Threads erzeugen

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:

```
public class Lager {
  private int bilanz;
  private int anz;
  private final int MANZ = 20;

  public Lager(int start) {
    bilanz = start;
    System.out.println("Der Laden ist offen (Bestand = "+bilanz+")\n");
  }
```

```
    public 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());
    }

    public void ergaenze(int add) {
      bilanz += add;
      anz++;
      System.out.println("Nr. "+anz+":\t"+Thread.currentThread().getName()+
        " ergaenzt\t"+add+"\tum "+formZeit()+" Uhr. Stand: "+bilanz);
    }

    public 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 l = new Lager(100);
      ProThread pt = new ProThread(l);
      KonThread kt = new KonThread(l);
      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 vor allem die **run()**-Methode überschreiben, z.B.:

```
public class ProThread extends Thread {
  private Lager pl;

  public ProThread(Lager pl_) {
    super("Produzent");
    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) {
        System.err.println(ie.toString());}
    }
  }
}
```

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 durch das Laufzeitsystem und erreicht dann den Zustand **running**. Die JVM arbeitet **preemtiv**, d.h. sie kann den Threads die Rechenerlaubnis jederzeit entziehen. Damit wechseln die Threads in Abhängigkeit von den Vergaberichtlinien des Laufzeitsystems zwischen den Zuständen **ready** und **running**. (siehe Abschnitt 14.6.1).

Sobald seine **run()**-Methode abgearbeitet ist, endet ein Thread. Er befindet sich dann im Zustand **dead** 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 beschaffte 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**.

Weil von der Methode **sleep()** potentiell eine **InterruptedException** zu erwarten ist, muss sie in einem **try**-Block ausgeführt werden. Die in Abschnitt 14.5 näher zu beschreibende **Thread**-Methode **interrupt()** wird oft dazu eingesetzt, einen per **sleep()** in den Schlaf oder per **wait()** (siehe Abschnitt 14.2.2) in den Wartezustand geschickten Thread wieder zu aktivieren. In der Ausnahmebehandlung muss der Programmierer dann geeignet reagieren, was bei unseren Übungsprogrammen aber nicht erforderlich ist.

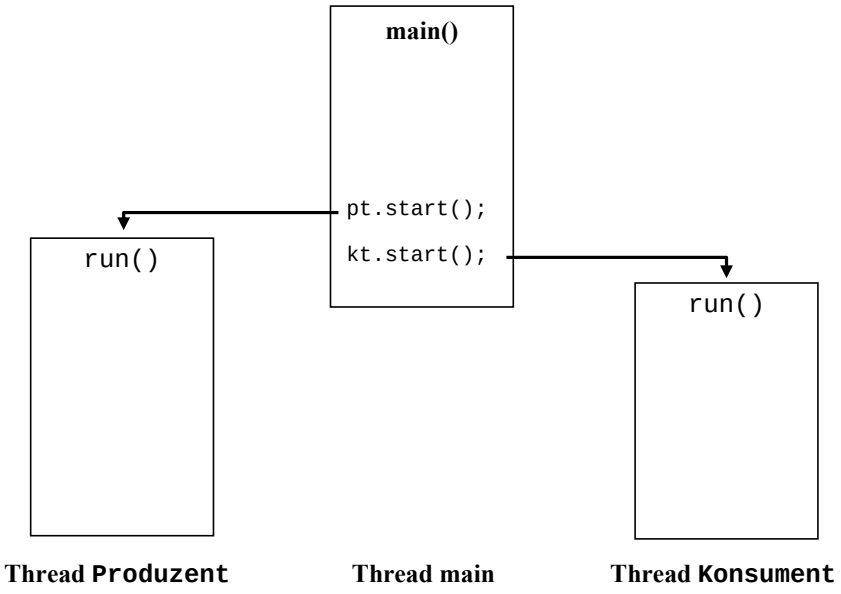
Zum `ProThread`-Konstruktor ist noch anzumerken, dass im Aufruf des Superklassen-Konstruktors ein Thread-Name festgelegt wird.

Der Konsumenten-Thread ist weitgehend analog definiert:

```
public class KonThread extends Thread {
  private Lager pl;

  public KonThread(Lager pl_) {
    super("Konsument");
    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){
        System.err.println(ie.toString());}
    }
  }
}
```

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:[46]



Wenn die drei Benutzer-Threads abgeschlossen sind, endet auch das Programm.

In den beiden Ausführungsfäden `Produzent` bzw. `Konsument`, die von einem `ProThread`- bzw. einem `KonThread`-Objekt begründet werden, kommt das `Lager`-Objekt wesentlich zum Einsatz:

[46] Nachdem Sie ein Java-Programm aus einer Konsole gestartet haben, können Sie unter Windows mit der Tastenkombination **Strg+Unterbr** eine Liste seiner aktiven Threads anfordern.

- In seiner Methode `offen()` 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.

14.2 Threads synchronisieren

14.2.1 Monitore

Am Anfang des eben wiedergegebenen Ablaufprotokolls stehen zwei „wirre“ Einträge, die folgendermaßen zu erklären sind:

- Der (zuerst gestartete) Produzenten-Thread nimmt 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 vom Laufzeitsystem 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.

Offenbar muss verhindert werden, dass zwei Threads simultan auf das Lager zugreifen.

Genau dies 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 betritt, 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 14.6.2).
Sobald der Monitor-Besitzer die **synchronized**-Methode beendet, kann ein wartender Thread den Monitor übernehmen und seine Arbeit fortsetzen.

In unserem Beispiel sollten die `Lager`-Methoden `offen()`, `ergaenze()` und `liefere()` als **synchronized** deklariert werden, z.B.:

```
public 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!
```

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. Diese Möglichkeit wird in Abschnitt 14.4 über das Unterbrechen von Threads vorgestellt.

14.2.2 Koordination per wait() und notify()

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 14.6.2). Die Methode **wait()** kann nur in einem synchronisierten Bereich, aufgerufen werden, z.B.:

```
public 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.toString());
    }

  bilanz -= sub;
  anz++;
  System.out.println("Nr. "+anz+":\t"+Thread.currentThread().getName()+
    " entnimmt\t"+sub+"\tum "+formZeit()+" Uhr. Stand: "+bilanz);
}
```

Mit den **Object**-Methoden **notify()** bzw. **notifyAll()** kann ein Thread aus dem Zustand **waiting** in den Zustand **ready** versetzt werden:

- **notify()** versetzt den Thread mit der längsten Wartezeit in den Zustand **ready**.
- **notifyAll()** versetzt alle wartenden Threads in den Zustand **ready**.

Wie **wait()** können auch **notify()** und **notifyAll()** nur in einem synchronisierten Bereich aufgerufen werden, z.B.:

```
public 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!
```

14.3 Das Interface Runnable

In unseren bisherigen Beispielen waren die Baupläne der als Thread ausgeführten Objekte aus der Klasse **Thread** abgeleitet. Oft benötigt man aber eine Thread-fähige Klasse, die einen anderen Stammbaum besitzt, z.B. aus **(J)Applet** abgeleitet ist. In einer solchen Situation, die in anderen Programmiersprachen möglicherweise durch Mehrfachvererbung gelöst wird, kommt in Java ein Interface zum Einsatz.
In konkreten Fall heißt es **Runnable** und verlangt von einer implementierenden Klasse lediglich eine **run()**-Methode mit dem vom Thread auszuführenden Code, z.B.:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Laufschrift extends Applet implements Runnable{
  private final String TXT = "Wo laufen sie denn? Wo laufen sie denn hin?";
  private final int PAUSE  = 10;
  private final int SPRUNG = 2;
  private final int HOEHE  = 10;

  Thread laufband;
  private int x, y,  // akt. Schreibpos.
              tb;    // Textbreite

  private final String LABELPREFIX = "Anzahl der Klicks: ";
  private Label lab;
  private int numClicks = 0;
  private Button butt;

  public void init() {
    setBackground(Color.YELLOW);
    lab = new Label(LABELPREFIX + "0    ");
    butt = new Button("Klicker");
    y =  getSize().height-HOEHE;
    tb = getFontMetrics(getFont()).stringWidth(TXT);
    add(butt);
    add(lab);
    butt.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        numClicks++;
        lab.setText(LABELPREFIX + numClicks);
      }
    });
    laufband = new Thread(this, "Laufband");
    laufband.start();
  }
```

```
    public void paint(Graphics g){
       g.drawString(TXT, x, y);
    }

    public void run() {
       while (true) {
          try {
             Thread.sleep(PAUSE);
          } catch(InterruptedException ie) {
             System.err.println(ie.toString());
          }
          if (x + tb < 0)
             x = getSize().width;
          else
             x -= SPRUNG;
          repaint();
       }
    }
}
```

Beim *Erzeugen* eines Ausführungsfadens wird aber doch ein **Thread**–Objekt benötigt, wobei man der passenden Konstruktor-Überladung einen Aktualparameter vom Typ **Runnable** übergibt:

Thread(Runnable *target***)**
Thread(Runnable *target***, String** *name***)**

Optional kann man zusätzlich den Namen des neuen Threads festlegen.

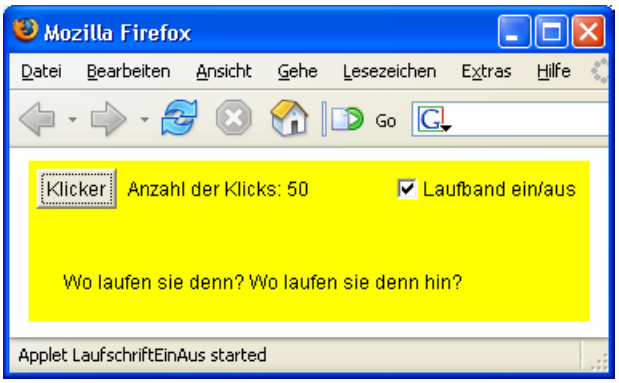
Im aktuellen Beispiel beschäftigt sich ein Thread damit, einen Text kontinuierlich über die Applet-Fläche laufen zu lassen. Gleichzeitig interagiert ein anderer Thread mit dem Benutzer, der z.B. vorbeifahrende Autos per Mausklick zählen kann:

Das Applet ist noch nicht ganz ausgereift; z.B. sollte durch Doppelpufferung das deutliche Flickern des Lauftextes reduziert werden. Wie dies durch Verwendung von Swing-Komponenten sehr bequem geschehen kann, haben Sie im Abschnitt über die zweidimensionale Grafikausgabe erfahren.

14.4 Threads unterbrechen

Zum Unterbrechen und Reaktivieren eines Threads sollten an Stelle der **Thread**-Methoden **suspend()** und **resume()**, die als veraltet (engl.: *deprecated*) eingestuft sind, die **Object**-Methoden **wait()** und **notify()** eingesetzt werden, die wir schon zur Synchronisation von Threads verwendet haben.

In folgender Variante des Applets aus Abschnitt 14.3 wird ein Kontrollkästchen angeboten, um den Laufschrift-Thread ein- oder auszuschalten:



In der booleschen Instanzvariablen `lban` des Applets wird der aktuelle Schaltzustand des Kontrollkästchens gespeichert. Wird in der **run()**-Methode des Laufschrift-Thread der `lban`-Wert **false** angetroffen, dann begibt sich der Thread per **wait()**-Aufruf freiwillig in den Wartezustand.
Wie Sie bereits wissen, darf die Methode **wait()** nur in einem synchronisierten Bereich aufgerufen werden. Die komplette **run()**-Methode des Threads als **synchronized** zu deklarieren, führt zu Problemen beim Reaktivieren. Als Alternative zum **synchronized**-Modifikator für Methoden steht in Java der **synchronisierte 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) {
      System.err.println(ie.toString());
    }
    if (x-SPRUNG + tb < 0)
      x = getSize().width;
    else
      x -= SPRUNG;
    repaint();
  }
}
```

Nach dem Schlüsselwort **synchronized** wird in runden Klammern ein beliebiges Objekt als Monitor (Lock-Objekt) für die Synchronisation angegeben. Im Beispiel übernimmt das Applet-Objekt diese Rolle selbst.
Die **notify()**-Anweisung zum Reaktivieren des wartenden Laufschrift-Threads wird in der Ereignisbehandlungsmethode **itemStateChanged()** zum Kontrollkästchen ausgeführt:

```
public synchronized void itemStateChanged(ItemEvent e) {
  lban = !lban;
  if (lban)
    notify();
}
```

Im Beispiel ist das Applet-Objekt als Monitor zum synchronisierten Block und als **ItemChanged**-Empfänger für das Kontrollkästchen tätig. Folglich wird die Ereignisbehandlungsmethode per **synchronized**-Deklaration in denselben, vom Applet-Objekt überwachten, synchronisierten Bereich einbezogen, und der **notify()**-Aufruf befreit den wartenden Laufschrift-Thread.

14.5 Threads stoppen

Zum Stoppen eines Threads sollte man nicht mehr die unerwünschte **Thread**-Methode **stop()** verwenden, sondern ein behutsames, kommunikatives Prozedere nutzen:

- 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.
- 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 einfach die **run()**-Methode und erreicht damit den Zustand **dead**.

Als Beispiel soll eine weitere Variante des Produzenten-Lager-Konsumenten-Beispiels dienen, wobei der Lagerverwalter den Konsumenten-Thread zum Terminieren auffordert, sobald dieser mit einem Wunsch über den Lagerbestand hinausgeht:

```
public synchronized void liefere(int sub) {
  if (bilanz - sub < 0) {
     anz++;
     System.out.println("Nr. "+anz+":\t"+Thread.currentThread().getName()+
          " wegen Gier beendet \tum "+formZeit()+" Uhr. Stand: "+bilanz);
     Thread.currentThread().interrupt();
  }
  else {
     bilanz -= sub;
     anz++;
     System.out.println("Nr. "+anz+":\t"+Thread.currentThread().getName()+
       " entnimmt\t"+sub+"\tum "+formZeit()+" Uhr. Stand: "+bilanz);
  }
}
```

Trifft der Konsumenten-Thread auf ein Interrupt-Signal, stellt er (wenn auch etwas mürrisch) seine Tätigkeit ein:

```
public void run() {
  while (pl.offen()) {
     if (isInterrupted()) {
        System.out.println(
             "Als Kunde muss ich mir so etwas nicht gefallen lassen!");
        return;
     }
     pl.liefere((int) (5 + Math.random()*100));
     long limit =  System.currentTimeMillis() +
                     (long)(1000 + Math.random()*3000);
     while(System.currentTimeMillis() < limit);
  }
}
```

In dieser **run()**-Methode wird bewusst ein Aufruf von **Thread.sleep()** vermieden, weil ein **interrupt()**-Aufruf in der Schlafenszeit zu einer **InterruptedException** führt. Mit dieser Konstellation werden wir uns gleich beschäftigen.

In der folgenden Lager-Sequenz muss der Konsument frühzeitig aussteigen:

```
Der Laden ist offen (Bestand = 100)

Nr. 1:    Produzent ergaenzt      71        um 19:03:39 Uhr. Stand: 171
Nr. 2:    Konsument entnimmt      39        um 19:03:39 Uhr. Stand: 132
Nr. 3:    Konsument entnimmt      95        um 19:03:41 Uhr. Stand: 37
Nr. 4:    Konsument wegen Gier beendet      um 19:03:43 Uhr. Stand: 37
Nr. 5:    Produzent ergaenzt      20        um 19:03:43 Uhr. Stand: 57
Nr. 6:    Produzent ergaenzt      30        um 19:03:46 Uhr. Stand: 87
Als Kunde muss ich mir so etwas nicht gefallen lassen!
Nr. 7:    Produzent ergaenzt      37        um 19:03:49 Uhr. Stand: 124
```

```
Nr. 8:   Produzent ergaenzt     90       um 19:03:51 Uhr. Stand: 214
Nr. 9:   Produzent ergaenzt     60       um 19:03:53 Uhr. Stand: 274
Nr. 10:  Produzent ergaenzt     14       um 19:03:57 Uhr. Stand: 288
Nr. 11:  Produzent ergaenzt     39       um 19:03:59 Uhr. Stand: 327
Nr. 12:  Produzent ergaenzt     56       um 19:04:01 Uhr. Stand: 383
Nr. 13:  Produzent ergaenzt     8        um 19:04:02 Uhr. Stand: 391
Nr. 14:  Produzent ergaenzt     20       um 19:04:06 Uhr. Stand: 411
Nr. 15:  Produzent ergaenzt     78       um 19:04:08 Uhr. Stand: 489
Nr. 16:  Produzent ergaenzt     56       um 19:04:09 Uhr. Stand: 545
Nr. 17:  Produzent ergaenzt     54       um 19:04:11 Uhr. Stand: 599
Nr. 18:  Produzent ergaenzt     62       um 19:04:15 Uhr. Stand: 661
Nr. 19:  Produzent ergaenzt     7        um 19:04:17 Uhr. Stand: 668
Nr. 20:  Produzent ergaenzt     27       um 19:04:19 Uhr. Stand: 695

Lieber Produzent, es ist Feierabend!
```

Angewandt auf einen per **wait()** in den Wartezustand versetzten oder per **sleep()** ruhig gestellten 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, die vom **wait()**- bzw. **sleep()**-Aufrufer abgefangen werden muss.
- Das Interrupt-Signal wird ggf. *aufgehoben* (auf **false** gesetzt). [47]

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, z.B.:

```
public void run() {
  while (pl.offen()) {
    if (isInterrupted()) {
      System.out.println(
           "Als Kunde muss ich mir so etwas nicht gefallen lassen!");
      return;
    }
    pl.liefere((int) (5 + Math.random()*100));
    try {
      Thread.sleep((int) (1000 + Math.random()*3000));
    } catch(InterruptedException ie) {
        interrupt();
    }
  }
}
```

14.6 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.

14.6.1 Scheduling und Prioritäten

Den Bestandteil der virtuellen Maschine, der die verfügbare Rechenzeit auf die arbeitswilligen und -fähigen Threads (Zustand **ready**) verteilt, bezeichnet man als **Scheduler**.

---

[47] Bei einem Thread, der auf einen *Monitor* wartet, verhält sich **interrupt()** wie zu Beginn des Abschnitts beschrieben, setzt also das Interrupt-Signal des Threads auf **true**.

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.

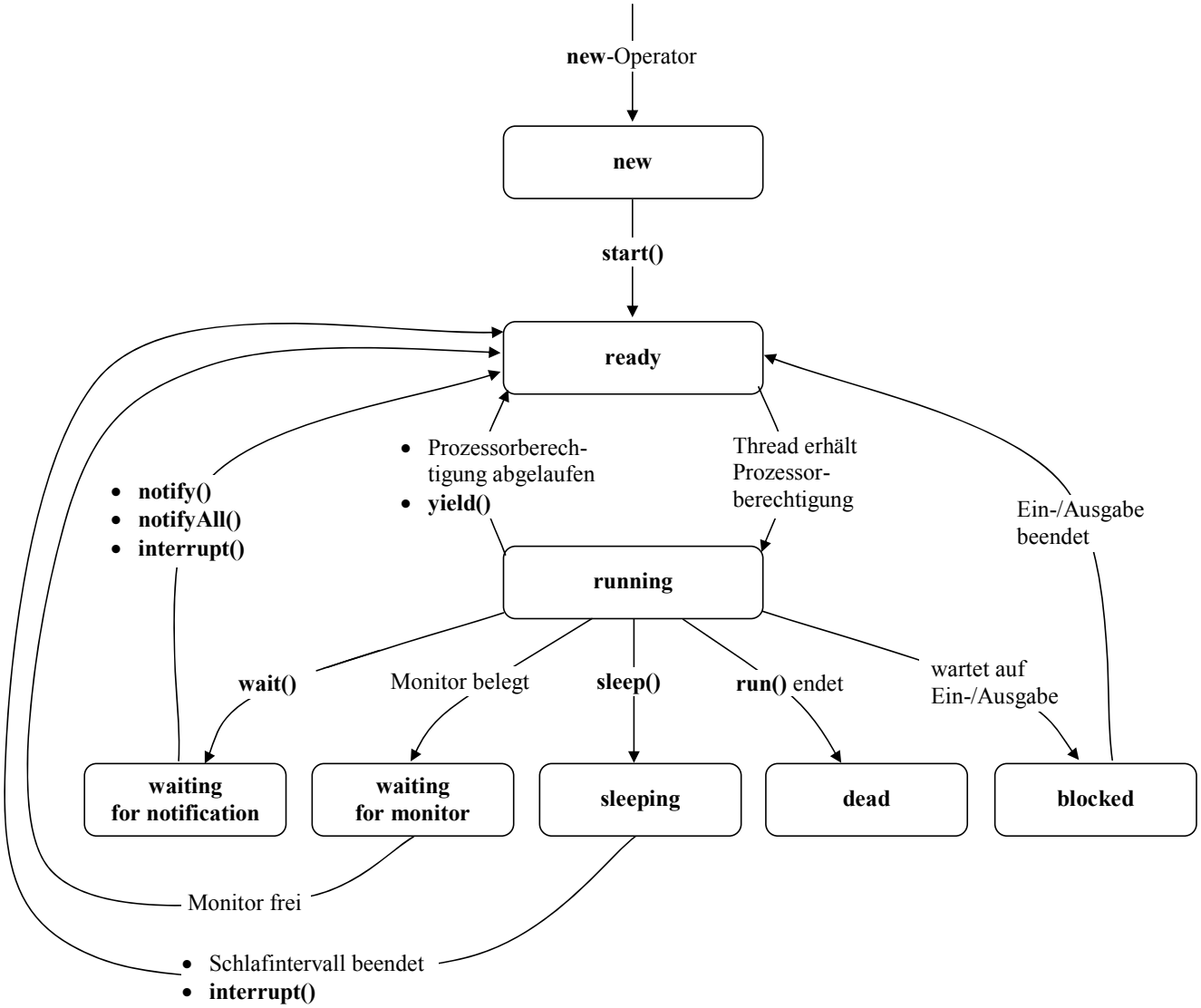
Leider gibt es im Verhalten des Schedulers bei Threads *gleicher* Priorität relevante Unterschiede zwischen den JVMs, so dass diesbezüglich *keine vollständige* Plattformunabhängigkeit besteht. Auf einigen Plattformen (z.B. unter MS-Windows) benutzt die JVM das von modernen Betriebsystemen zur *Prozess*verwaltung verwendete **preemtive Zeitscheibenverfahren** für die Thread-Verwaltung:

- Die 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 zum Prozessor.

Auf anderen Plattformen (z.B. unter Solaris) benutzt die JVM ein rein prioritätenbasiertes Multithreading, so dass ein Thread in der Regel nur von Konkurrenten mit höherer Priorität verdrängt wird.
Freundliche Threads können aber auch ohne Round-Robin-Unterstützung der JVM den gleichrangigen Kollegen eine Chance geben, indem sie bei passender Gelegenheit die **Thread**-Methode **yield()** aufrufen, um den Prozessor abzugeben und sich wieder in die Warteschlage der rechenwilligen Threads einzureihen. Mit Hilfe dieser Methode können also die Plattformunterschiede in der Thread-Verwaltung kompensiert werden.

14.6.2 Zustände von Threads

In der folgenden Abbildung nach Deitel & Deitel (1999, S. 738) werden wichtige Thread-Zustände und Anlässe für Zustandsübergänge dargestellt:



14.7 Sonstige Thread-Themen

14.7.1 Daemon-Threads

Neben den in Abschnitt 14 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 allerdings 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.

14.7.2 Deadlock

Wer sich beim Einsatz von Monitoren zur Thread-Synchronisation ungeschickt anstellt, kann einen so genannten Deadlock produzieren, wobei sich Threads gegenseitig blockieren. Im folgenden Beispiel nehmen `Thread1` und `Thread2` jeweils einen Monitor in Besitz, indem sie eine Klassenmethode mit **synchronized**-Deklaration aufrufen, wobei die jeweilige Klasse (`DeadLock1` bzw. `DeadLock2`) die Rolle des Aufpassers spielt:

```
class DeadLock1 {
  synchronized static void m() {
    System.out.println(Thread.currentThread().getName()+" in DeadLock1.m");
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {}
    System.out.println(Thread.currentThread().getName()+
                       " wartet darauf, dass DeadLock2.m frei wird ...");
    DeadLock2.m();
  }

  public static void main(String[] args) {
    Thread1 t1 = new Thread1();
    Thread2 t2 = new Thread2();
    t1.start();
    t2.start();
  }
}

class DeadLock2 {
  synchronized static void m() {
    System.out.println(Thread.currentThread().getName()+" in DeadLock2.m");
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {}
    System.out.println(Thread.currentThread().getName()+
                       " wartet darauf, dass DeadLock1.m frei wird ...");
    DeadLock1.m();
  }
}

class Thread1 extends Thread {
  Thread1() {super("Thread1");}
  public void run() {
    DeadLock1.m();
  }
}

class Thread2 extends Thread {
  Thread2() {super("Thread2");}
  public void run() {
    DeadLock2.m();
  }
}
```

Nach einem kurzen Schläfchen versuchen beide Threads, eine Methode aus dem jeweils anderen Monitor-Bereich aufzurufen, und geraten dabei in einen Wartezustand, aus dem sie nie wieder frei kommen:

```
Thread1 in DeadLock1.m
Thread2 in DeadLock2.m
Thread1 wartet darauf, dass DeadLock2.m frei wird ...
Thread2 wartet darauf, dass DeadLock1.m frei wird ...
```

14.7.3 Threads und Swing

Das Swing-API wurde (im Unterschied zum AWT-API) aus Performanzgründen für einen *Single-Thread* - Betrieb konzipiert. Greifen trotzdem mehrere Threads simultan auf eine Swing-Komponente zu, kann es zu irregulärem Verhalten (z.B. bei der Bildschirmaktualisierung) kommen. Wegen der fehlenden Thread-Sicherheit sollte man den exklusiven Zugriff auf die Swing-Komponenten unbedingt dem bei GUI–Anwendungen bzw. –Applets automatisch vorhandenen **Ereignisverteilungs-Thread** (*Event Dispatch* - Thread) überlassen. Er informiert Komponenten über Ereignisse und fordert sie ggf. zur Aktualisierung ihrer Oberfläche auf. Weil alle Ereignisbehandlungsmethoden in diesem Thread ablaufen, sollte sich deren Zeitaufwand in Grenzen halten, weil sonst die Benutzeroberfläche zäh reagiert.
Beachten Sie also bei Verwendung des Swing-APIs unbedingt die folgende Single-Thread - Regel: Änderungen mit Auswirkung auf die Darstellung von bereits realisierten Swing-Komponenten dürfen nur im Rahmen des Event Dispatch - Threads vorgenommen werden, also in Ereignisbehandlungsmethoden. Als *realisiert* gilt eine Swing-Komponente dann, wenn für das zugehörige Top-Level-Fenster eine Methoden **setVisible()**, **show()** oder **pack()** aufgerufen worden ist.

Allerdings bietet das Swing-API etliche Möglichkeiten, Änderungen von Komponenten außerhalb des Ereignisbehandlungs-Threads zu veranlassen, ohne die obige Regel zu verletzen, z.B.:

- **repaint()**, **revalidate()**
  Diese **JComponent**-Methoden platzieren eine Aktualisierungs-Anforderung in die Ereigniswarteschlange.

14.8 Zusammenfassung

Ein Thread ist ein selbständiger Ausführungsfaden (Kontrollfluss) innerhalb eines Java-Programms bzw. –Applets. Alle Threads eines Programms laufen im selben Adressraum. Sie teilen sich den Heap (mit den Objekten des Programms), haben jedoch jeweils einen eigenen Stack.
Bei der Vergabe von Rechenzeit an die Threads berücksichtigt der Scheduler der JVM deren Prioritäten und wendet meist für Threads gleicher Priorität ein Zeitscheibenverfahren an.

Zum Erzeugen eines Threads bietet Java zwei Optionen:

- Eine eigene Klasse aus **java.lang.Thread** ableiten und ein Objekt dieser Klasse erzeugen
- In einer eigenen Klasse das Interface **java.lang.Runnable** implementieren und ein Objekt dieser Klasse an einen **Thread**-Konstruktor übergeben

Über *Monitore* kann man verhindern, dass ein Code-Bereich von mehreren Threads parallel durchlaufen wird:

- Sobald ein Thread eine als **synchronized** definierte Instanzmethode für ein Objekt aufruft, ignoriert dieses Objekt jeden Aufruf einer **synchronized**-Instanzmethode durch einen anderen Thread, bis der erste Thread den Monitor verlassen hat.
- Sobald ein Thread eine als **synchronized** definierte statische Methode einer Klasse aufruft, sind alle statischen **synchronized**-Methoden der Klasse für alle anderen Threads gesperrt, bis der erste Thread den Monitor verlassen hat.
- Sobald ein Thread einen **synchronized**-Block betreten hat, der durch ein Lock-Objekt geschützt wird, sind *alle* von diesem Objekt geschützten **synchronized**-Blöcke für andere Threads gesperrt, bis der erste Thread das Lock-Objekt frei gibt.
- Ein Lock-Objekt kann gleichzeitig synchronisierte Instanzmethoden und synchronisierte Blöcken überwachen.

Wichtige Methoden der Klasse **java.lang.Thread**:

currrentThread()	liefert eine Referenz auf das aktuelle **Thread**-Objekt
getName()	liefert den Namen des Threads
getPriority()	ermittelt die Priorität des Threads
interrupt()	Ein per **wait()** deaktivierter oder bei **sleep()** ruhig gestellter Thread wird durch eine **InterruptedException** aufgeweckt. Für andere Threads wird das Interrupt-Signal auf **true** gesetzt.
isAlive()	testet, ob der Thread lebt, d.h. gestartet wurde und noch nicht beendet ist
run()	enthält den vom Thread auszuführenden Code
setPriority()	setzt die Priorität des Threads
sleep()	lässt den Thread eine bestimmte Zeit schlafen
start()	initiiert den Thread und ruft seine **run()**-Methode auf
yield()	gibt Rechenzeit an Threads gleicher Priorität ab (nur relevant, wenn die JVM kein Zeitscheiben-Scheduling unterstützt)

Wichtige Methoden der Klasse **java.lang.Object**:

notify()	Von den auf einen Monitor wartenden Threads wird derjenige mit der längsten Wartezeit in den Zustand **ready** versetzt.
notifyAll()	Alle auf einen Monitor wartenden Threads werden in den Zustand **ready** versetzt.
wait()	Der aktuelle Thread wird in den Wartezustand versetzt.

14.9 Übungsaufgaben zu Abschnitt 14

1) Das folgende Programm startet einen Thread, lässt ihn 10 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 gestarted.");
    Thread.sleep(10000);
    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 Thread prüft in seiner **run()**-Methode 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, ignoriert der Thread das erste Interrupt-Signal, reagiert aber auf das zweite:

```
Thread gestarted.
**************************************************************************************
********************
Thread beendet!?
*********
Thread beendet!?
*
```

Verändert man die 100 ms lange Thread-interne Kunstpause 101 oder 99 ms, dann bleiben sogar sämtliche Interrupt-Signale wirkungslos:

```
Thread gestarted.
************************************************************************
Thread beendet!?
*******
Thread beendet!?
*******
Thread beendet!?
*******
Thread beendet!?
*******
   .   .   .
```

Wie ist das Verhalten zu erklären, und wie sorgt man für eine zuverlässiges Beenden des Threads?

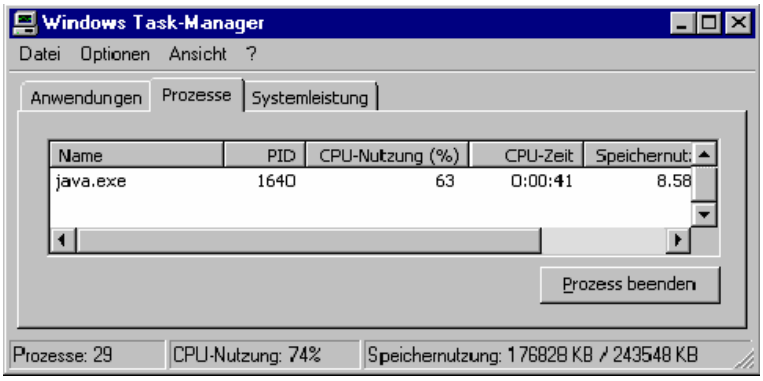
2) Erstellen sie eine Anwendung oder ein Applet, um den Effekt der Prioritäten auf das Verhalten zweier Threads beobachten zu können:

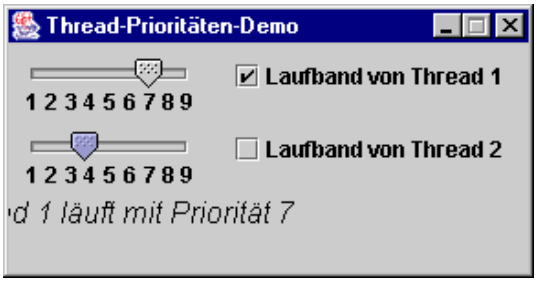
- Der Benutzer soll die Möglichkeit haben, die Prioritäten der beiden Threads zu verändern.
- Die Ausführungsgeschwindigkeiten der Threads sollen als „Rennen“ zu beobachten sein.
- Vielleicht bauen Sie noch Bedienelement ein, um die Threads einzeln unterbrechen und reaktivieren zu können (per **wait()** und **notify()**).

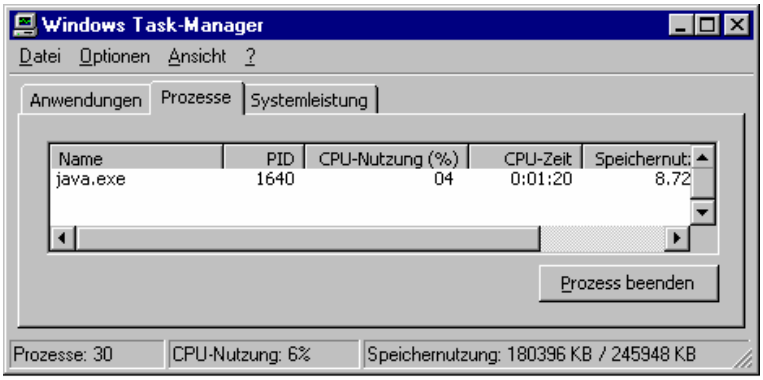
Bei diesem Lösungsvorschlag handelt es sich um eine Anwendung mit Swing-GUI:

Beide Threads zeichnen eine Laufschrift im unteren Bereich eines **JFrame**-Fensters. Zur Einstellung der Prioritäten sind Schieberegler vorhanden, realisiert mit **JSlider**-Komponenten (siehe API-Dokumentation).

Unter Windows NT/2000/XP kann man mit dem **Task Manager** (**Strg+Alt+Entf**, **T**) beobachten, wie sich die CPU-Nutzung der Virtuellen Maschine in Abhängigkeit von den beiden Thread-Aktivitäten bzw. -Prioritäten ändert, z.B.







Lässt man mehrere Threads mit hoher Priorität laufen, reagiert die graphische Benutzeroberfläche eventuell sehr zäh.

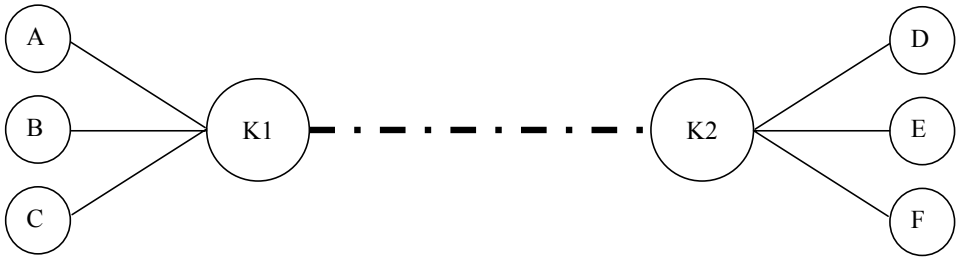
15 Netzwerkprogrammierung

Konform zu ihrem bereits 1982 kreierten Leitsatz *The Network is the Computer* hat sich die Firma Sun beim Java-Design intensiv und erfolgreich um eine möglichst einfache Netzwerkprogrammierung bemüht. Dabei werden in erster Linie Netzwerke auf Basis der TCP/IP - Protokollfamilie (siehe unten) unterstützt.

15.1 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** 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) bis zur physikalischen Ebene (z.B. elektromagnetische Wellen auf einem Ethernetkabel) sind zahlreiche Übersetzungen vorzunehmen bzw. Aufgaben zu lösen, jeweils unter Beachtung der zugehörigen Regeln.

15.1.1 Das OSI-Referenzmodell

Nach dem **OSI – Modell** (*Open System Interconnection*) der ISO (*International Standards Organisation*) werden sieben Schichten mit jeweiligen Zuständigkeiten 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)

Hier wird festgelegt, wie von der Netzwerk-Hardware Bits zwischen zwei direkt verbundenen Stationen zu übertragen sind. Im einfachen Beispiel einer seriellen Verbindung 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.

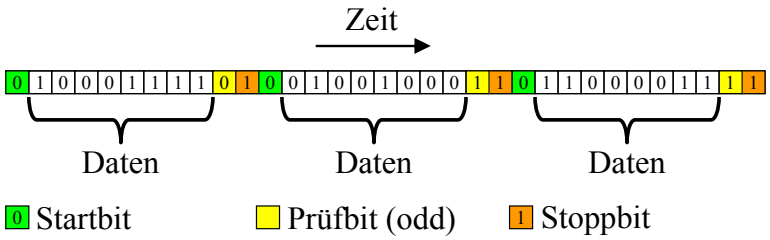
2. Link-Ebene (Frame-Übertragung)

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 3 Frames zu sehen, die nacheinander über eine serielle Leitung gesendet werden:



Das odd-Prüfbit wird so gesetzt, dass es die 8 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 Ebene 2 hängen von der Netzwerktechnik ab, so dass auf der Gesamtstrecke vom Absender bis zum Empfänger in der Regel *mehrere* Frame-Architekturen beteiligt sind (z.B. auf der Telefonstrecke zum Internet-Provider eine andere als auf dem weiteren Weg über Ethernet- oder ATM-Verbindungen). Auf der Ebene 3 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 15.1.2).

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 überwiegend das IP-Protokoll auf Ebene 3 zum Einsatz. Seine Pakete bezeichnet auch als *IP-Datagramme*.
In der noch üblichen IP-Version 4 besteht eine Adresse aus 32 Bits, üblicherweise durch 4 per Punkt getrennte Dezimalzahlen (aus dem Bereich von 0 bis 255) dargestellt, z.B.:

136.199.8.62

4. Transportschicht (Übertragung von Datenströmen, 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 (Kommunikation von Anwendungen, z.B. via 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, die eine kommunikationswillige bzw. -fähige Anwendung identifizieren. So wird es 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 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-Nummer des Serverrechners und die Portnummer des Dienstes
- Die IP-Nummer 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 Rechnern her.
- Auf beiden Seiten steht eine als **Socket** bezeichnete Programmierschnittstelle zur Verfügung, die eine Netzwerkkommunikation analog zum **Datenstrommodell** der seriellen Dateibearbeitung erlaubt. Aus der Sicht des Anwendungsprogrammierers werden per TCP also 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 z.B. für Multimedia-Anwendungen.
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 weit wichtigere TCP-Protokoll beschränken.

Schließlich ist noch das **Internet Control Message Protocol** (ICMP) zu erwähnen, das zur Übermittlung von Fehlermeldungen und verwandten Informationen dient. Wenn z.B. ein Router ein IP-Datagramm verwerfen muss, weil seine maximale Zahl 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.

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.

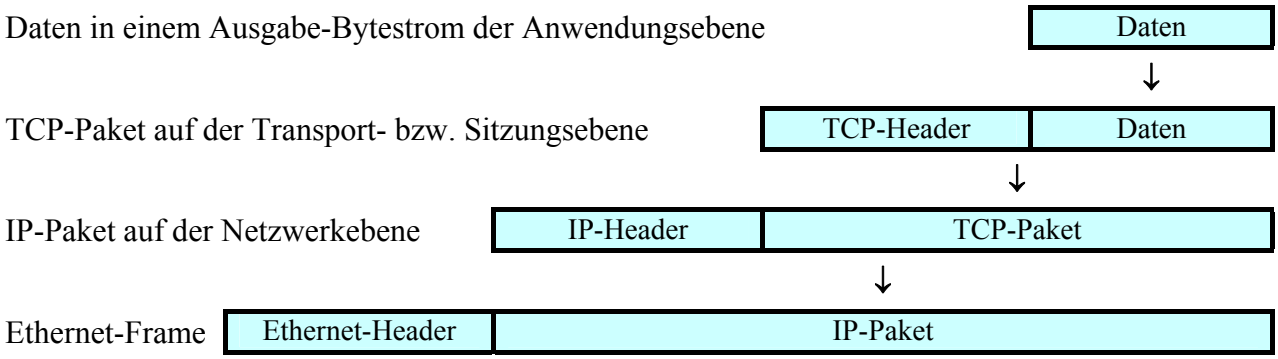
7. Anwendung

Hier wird für verschiedene Dienste festgelegt, wie Anforderungen zu formulieren sind. Einem SMTP-Server, der an Port 25 lauert, kann man z.B. durch folgende Sequenz von Zeichenfolgen eine Mail übergeben:

```
HELO meinpc.uni-trier.de
MAIL FROM: murx@ist-egal.te
RCPT TO: jemand@uni-trier.de
DATA
subject: Hallo
Dies ist der Inhalt!
.
QUIT
```

15.1.2 Zur Funktionsweise von Protokollstapeln

In diesem Abschnitt sollen die *funktionalen* Aspekte der eben erläuterten Struktur vertieft werden. Möchte eine Anwendung (genauer: eine aktive Anwendungsinstanz) auf Rechner A über ein TCP/IP – Netzwerk eine gemäß zugehörigem Protokoll (z.B. POP3, Telnet) zusammengestellte Sendung an eine korrespondierende Anwendung auf Rechner B schicken, dann übergibt sie eine Serie von Bytes an die TCP-Schicht, welche daraus TCP-Pakete erstellt. Wir beschränken uns auf den einfachen Fall, dass alle Daten in *ein* TCP-Paket passen:



Wichtige Bestandteile des TCP-Headers sind:

- Die Portnummern der Quell- und Ziel*anwendung*
- TCP-Flags
  Hierzu gehört z.B. das 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 Ziel*rechner*
- Typ des eingepackten Protokolls (z.B. TCP oder UDP)

- Time-To-Live
  Beim Routing kann es zu Schleifen kommen. Damit ein Paket nicht ewig rotiert, erhält es beim Start einen Time-To-Live – Wert mit der maximalen Anzahl von erlaubten Router-Passagen, der von jedem besuchten Router dekrementiert wird. Beim TTL-Wert 0 verwirft ein Router das Paket und informiert den Absender eventuell per ICMP-Paket über den Vorfall.

Wird ein IP-Paket auf der ersten Teilstrecke per Ethernet-Technik transportiert, muss es in einen Ethernet-Frame verpackt werden, wobei der zusätzliche Header z.B. die MAC-Adresse des ersten Etappenziels (z.B. Gateway im Subnetz) 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.

15.1.3 Optionen zur Netzwerkprogrammierung in Java

Java unterstützt sowohl die Socket-orientierte TCP- bzw. UDP-Kommunikation (auf der Ebene 4/5 des OSI-Modells) als auch die Nutzung wichtiger Protokolle auf der Anwendungsebene. Ein Zugriff auf tiefere Protokollschichten (unterhalb der TCP-Ebene) ist nur über externe Software möglich (per JNI angebunden[48]), was allerdings für die meisten Netzwerkprogrammierer ohne praktische Bedeutung 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
- Klient-Server - Programmierung auf Socket-Ebene
- Einführung in die Erstellung von Web-Applikationen mit Servlet- und JSP-Technik

Das nur für wenige Anwendungen relevante UDP-Protokoll wird *nicht* behandelt.

15.2 Internet-Ressourcen nutzen

Auf Internet-Ressourcen, die über einen so genannten **Uniform Ressource Locator** (URL) ansprechbar sind, kann man in Java genau so einfach zugreifen wie auf lokale Dateien.
Ein URL (z.B. http://www.uni-trier.de/) ist folgendermaßen aufgebaut:

Syntax:	*Protokoll*	://	*Benutzer:Passwort@*	*Rechner*	*:Port*	*Pfad*	*#Referenz*
Beispiel:	http	://		www.uni-trier.de		/	

15.2.1 Die Klasse URL

In vielen Fällen kommt man 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:

---

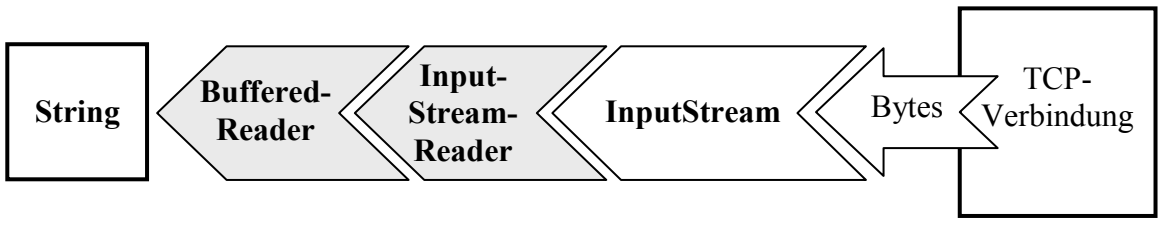
[48] Auf der Webseite **http://www.geocities.com/SiliconValley/Bit/5716/ping/index_eng.html** bieten Isabel García und Óscar Fernández eine Lösung für den Einsatz unter Windows an.

```
import java.net.*;
import java.io.*;

class URLDemo {
   public static void main(String[] args)   {
      try {
         URL url = new URL("http://www.uni-trier.de/");
         BufferedReader in =
            new BufferedReader(new InputStreamReader(url.openStream()));
         String s;
         for (int i = 0; i < 16; i++) {
            s = in.readLine();
            if (s == null)
               break;
            System.out.println(s);
         }
      } catch (MalformedURLException e) {
         e.printStackTrace();
      } catch (IOException e ) {
         e.printStackTrace();
      }
   }
}
```

In Java wird fast jeder Datenaustausch via Netz (mit Ausnahme der UDP-Kommunikation) aus der Perspektive einer Anwendung 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.

Weil der **URL**-Konstruktor ggf. eine **MalformedURLException** – Exception wirft, muss er im Rahmen einer **try-catch** - Anweisung aufgerufen werden.
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. Um die bei Verbindungsproblemen von der Methode **openStream()** zu erwartenden **IOException**s muss sich eine **catch**-Klausel kümmern.

Vom **InputStreamReader**-Objekt werden die angelieferten Bytes in Unicode-Zeichen transformiert, 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 das Transformationsobjekt, so dass sich folgende „Pipeline“ ergibt:



Ein Programmlauf bringt (am 25.04.2006) folgendes Ergebnis:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>

  <meta content="text/html; charset=ISO-8859-1" http-equiv="Content-Type">

  <meta content="Offizielle Homepage der Universit&auml;t Trier" name="description">
```

Über weitere Möglichkeiten der Klasse **URL** informiert die API-Dokumentation. Besonders zu erwähnen sind:

- Konstruktoren für die URL-Spezifikation *relativ* zu einer Basis, z.B.:
  URL(**URL** *basis*, **String** *relSpec*)
- Methoden, die Informationen über URL-Bestandteile liefern, z.B.:
  getHost()

15.2.2 Die Klasse URLConnection

Die angenehm einfach aufgebaute Klasse **URL** verwendet beim Aufruf ihrer Methode **openStream()** klammheimlich ein Objekt der Klasse **URLConnection**. Durch expliziten Einsatz dieser Klasse 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 das **Cookie**-Feld bringt man einem Server Informationen in Erinnerung, die er selbst bei seinem letzten Kontakt über das korrespondierende Response-Header – Feld beim Klienten hinterlassen hat. Über diese (von manchen Benutzern aus Datenschutzgründen abgelehnte) Technik kann ein WWW-Server seine Benutzer wieder erkennen und ein individualisiertes Angebot liefern.
- 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.

Eine Liste mit allen Header-Feldern der aktuellen 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 statt dessen die **openConnection()** - Methode eines passenden **URL**-Objektes auf.

Im folgenden Beispiel wird die Homepage der Universität Trier angefordert, falls sie seit dem 12.02.2006 geändert worden ist:

```
import java.net.*;
import java.io.*;
import java.text.*;
import java.util.*;

class URLConnectionDemo {
  public static void main(String[] args) {
    try {
      String urlString;
      if (args.length == 0)
        urlString = "http://www.uni-trier.de/";
      else
        urlString = args[0];

      // URLConnection-Objekt generieren
      URL url = new URL(urlString);
      URLConnection urlConn = url.openConnection();

      // Request - Header setzen
      DateFormat df = DateFormat.getDateTimeInstance();
      urlConn.setIfModifiedSince(df.parse("12.02.2006 00:00:00").getTime());
```

```
            // Verbindung herstellen und Response-Header entgegennehmen
            urlConn.connect();

            // Response-Header auswerten
            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"+urlConn.getExpiration());
            System.out.println("  Last Modified:\t"+
               df.format(new Date(urlConn.getLastModified()))+"\n");

            // Inhalt ausgeben
            if (urlConn.getContentLength() > 0) {
               System.out.println("Weiter zum Inhalt mit Enter");
               System.in.read();
               BufferedReader 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();
         }
      }
   }
```

Die **URL**-Methode **openConnection()** baut noch keine Verbindung zum Server auf, sondern liefert ein **URLConnection**-Objekt und schafft so die Möglichkeit, die zum URL gehörige Anforderung über Request-Header näher zu spezifizieren. Generell dient dazu die Methode

setRequestProperty(**String** *key*, **String** *value*)

Einige Request-Header können aber auch mit speziellen **URLConnection** - Methoden gesetzt werden, z.B. **If-Modified-Since**:

```
DateFormat df = DateFormat.getDateTimeInstance();
urlConn.setIfModifiedSince(df.parse("12.02.2006 00:00:00").getTime());
```

Im Beispiel wird eine Klartext-Datum/Zeit-Angabe mit Hilfe der Klassen **Date** und **DateFormat** in die benötigte Anzahl von Millisekunden seit dem 1. Januar 1970 (GMT) umgewandelt.

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. Beim Ausführen des Beispielprogramms erfährt man (am 27.04.2006):

```
Response-Header:
  Content-Type:    text/html
  Content-Length:  8131
  Expiration:      0
  Last Modified:   09.03.2006 10:38:08
```

Die **URLConnection**-Methode **getLastModified()** liefert Millisekunden seit dem 1. Januar 1970 (GMT), die im Beispiel mit Hilfe der Klassen **Date** und **DateFormat** zu einer verständlichen Ausgabe werden.

Über die **URLConnection**-Methode **getInputStream()** erreicht man denselben Eingabestrom mit den angeforderten Daten, den auch die **URL**-Methode **openStream()** (siehe oben) liefert.

Sind Verbindungsprobleme zu befürchten, sollte vor dem **connect()**-Aufruf mit der Methode

setConnectTimeout(int *timeout*)

eine maximale Wartezeit festgelegt werden. Ein Überschreiten dieser Zeit wird von **connect()** per **SocketTimeoutException** signalisiert.

15.2.3 Datei-Download

Mit den Klassen **URL** oder **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 **spss13dat.zip** geschieht:

```
import java.net.*;
import java.io.*;

class DateiDownload {
  public static void main(String[] args) {
    try {
      String urlString =
        "http://www.uni-trier.de/urt/user/baltes/docs/spss/v13/spss13dat.zip";
      URL url = new URL(urlString);
      System.out.println("Die Datei "+urlString+" wird herunter geladen ...");
      BufferedInputStream in = new BufferedInputStream(url.openStream());
      BufferedOutputStream out = new BufferedOutputStream(
          new FileOutputStream((new File(url.getFile()).getName())));
      int i, n=0;
      while ((i = in.read()) != -1) {
        out.write(i);
        n++;
      }
      out.close();
      System.out.println("Fertig! (Bytes geschrieben: "+n+")");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Wir kommen mit Byte-orientierten Strömen aus, kombinieren diese aber zur Transportbeschleunigung jeweils mit einem Puffer.
Mit Hilfe des oben vorgestellten Request-Headers **If-Modified-Since** lässt sich z.B. relativ leicht ein Programm erstellen, das von einer Webseite eines Anti-Viren-Software-Herstellers die aktuelle Schädlings-Definitionsdatei ggf. herunter lädt (siehe Übungsaufgabe in Abschnitt 15.5).

15.2.4 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

Um ein **HttpURLConnection**-Objekt zu erzeugen, wendet man eine explizite Typumwandlung auf ein **URLConnection**-Objekt an, eventuell nach vorheriger Typprüfung per **instanceof**, 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
```

15.2.5 Dynamisch erstellte Webinhalte mit Java-Klientenprogrammen anfordern

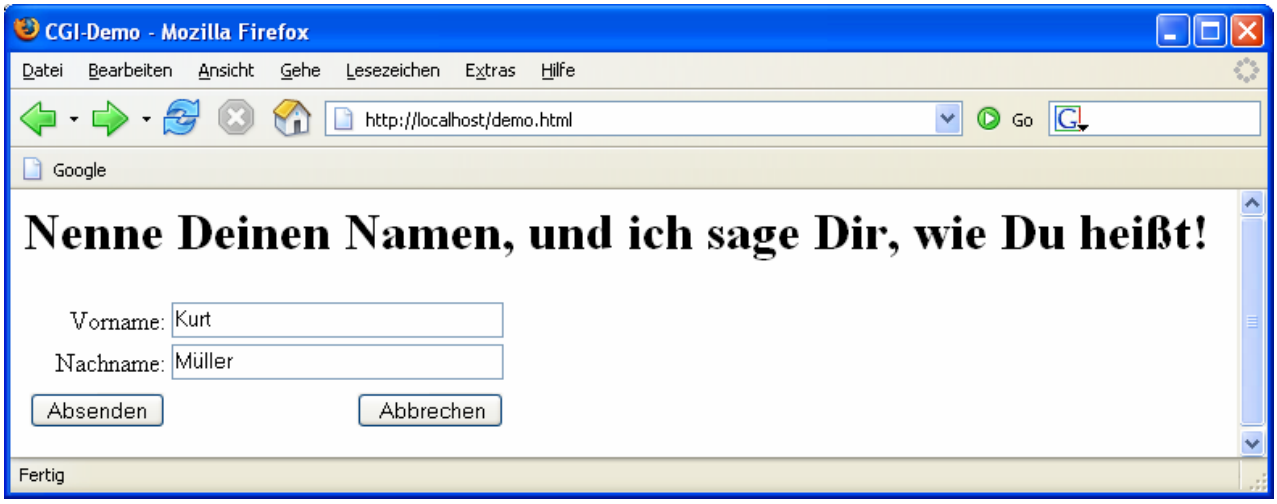
WWW-Server halten in der Regel nicht nur statische HTML-Seiten und sonstige Dateien bereit, sondern bieten auch verschiedene Technologien, um HTML-Seiten dynamisch nach Kundenwunsch zu erzeugen und an den Browser auszuliefern (z.B. mit den Ergebnissen eines Suchauftrags oder mit einer individuellen Produktkonfiguration). 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. C, PHP, Perl, Python). Traditionell läuft die Kooperation über das so genannte **Common Gateway Interface** (CGI), wobei das externe Ergänzungsprogramm bei jeder Anforderung neu gestartet und nach dem Erstellen der HTML-Antwortseite wieder beendet wird. Mittlerweile kann die Ergänzungssoftware aber auch über Module des WWW-Servers eingebunden werden und permanent aktiv bleiben (z.B. bei PHP, MS-ASP oder Cold Fusion von Macromedia). 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 (z.B. in einer ASP-Seite) statische HTML-Abschnitte mit Skript-Bestandteilen zur dynamischen Produktion individueller Abschnitte kombiniert werden.
Bei den integrierten, performanten und wartungsfreundlichen Lösungen spielt auch Java eine herausragende Rolle, und insbesondere bei anspruchsvollen Server-Dienstleistungen (z.B. Online-Banking) sind Java-Servlets bzw. Java Server Pages (siehe Abschnitt 17) häufig anzutreffen.

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.

Mit den neuerdings sehr populären **Webdiensten** (engl.: *Web Services*), die den Aufruf von Prozeduren auf entfernten Rechnern via Internet über das XML-basierte **SOAP**-Protokoll (*Simple Object Access Protocol*) erlauben, können wir uns aus Zeitgründen nicht beschäftigen.

15.2.5.1 Formularbasierte Interaktivität im WWW

Wir betrachten ein einfaches Formular mit zugehörigem CGI-Skript, um einige technische Details zu erläutern. Hinter der Webseite



steckt der folgende HTML-Code mit Formular:

```
<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="/cgi-bin/demo.pl">
<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, um die URL-Syntaxregeln einzuhalten. 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.

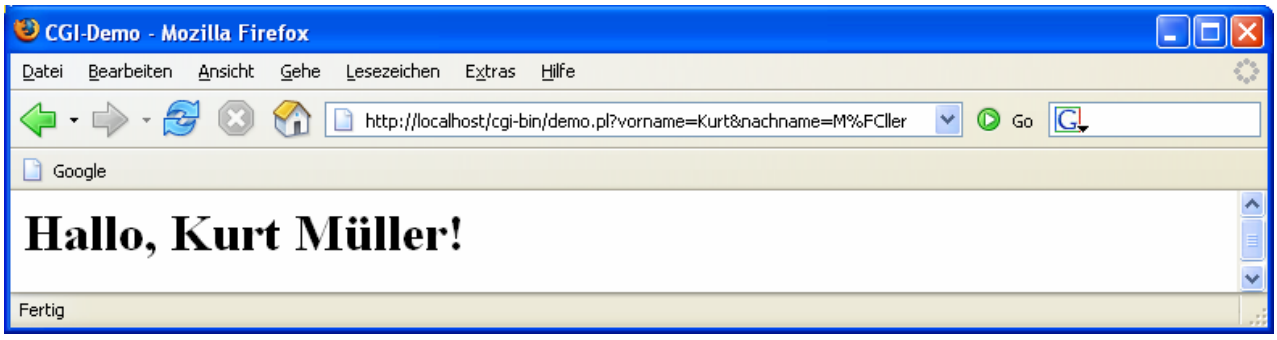
Auf gleich noch näher zu erläuternde Weise übergibt der WWW-Server die Formulardaten an das im **action**-Attribut der Formulardefinition angegebene externe Programm oder Skript. Im Beispiel handelt es sich um folgendes Perl-Skript, das wenig kreativ aus den übergebenen Namen einen Gruß formuliert:

```
#!perl
use CGI;
my $query = CGI->new();
print "Content-type: text/html\n\n";
print "<html>\n<head><title>CGI-Demo</title></head>\n<body>\n<h1>";
print 'Hallo, '.$query->param("vorname").' '.$query->param("nachname").'!';
print "</h1>\n</body>\n</html>";
```

Es schreibt Header-Felder und den eigentlichen Inhalt per **print**-Kommando an die Standardausgabe, in unserem Fall:

- Es wird nur das minimal erforderliche Header-Feld mit der Typspezifikation geschrieben.
- Der Inhaltsbereich besteht aus einer HTML-Seite.

Vom WWW-Server wird die Perl-Produktion über das HTTP-Protokoll an den Browser geschickt, z.B.:



15.2.5.2 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 hinter einem trennenden Fragezeichen am Ende des URLs zur nächsten Anforderung an den WWW-Server. Weil nach Eintreffen der Antwortseite der zugrunde liegende URL in der Adresszeile des Browsers erscheint, kann die GET-Syntax dort inspiziert werden (siehe oben).

Der WWW-Server schreibt die Name-Wert - Paare in eine Umgebungsvariable namens **QUERY_STRING** und stellt auf analoge Weise der anschließend gestarteten CGI-Software gleich noch weitere Informationen zur Verfügung, z.B.:

```
QUERY_STRING="vorname=Kurt&nachname=M%3Fller"
```

```
REMOTE_PORT="1211"
REQUEST_METHOD="GET"
REQUEST_URI="/cgi-bin/demo.pl?vorname=Kurt&nachname=M%3Fller"
SCRIPT_FILENAME="f:/programme/apache/cgi-bin/demo.pl"
SERVER_PROTOCOL="HTTP/1.1"
SERVER_SOFTWARE="Apache/1.3.27 (Win32)"
```

In obigem Perl-Skript ist der Zugriff auf die Umgebungsvariable **QUERY_STRING** nicht so ganz offensichtlich, weil das Perl-Modul **CGI.pm** verwendet (siehe z.B. Patwardhan et al. 2002, S. 376ff) und ein CGI-Objekt durch Aufruf seiner **param()**-Methode mit dem Lesen der Umgebungsvariable beauftragt wird, z.B.:

```
$query->param("vorname")
```

Dasselbe Perl-Objekt beherrscht auch den Empfang vom POST-Parametern (siehe unten), so dass zur Demonstration der beiden Parameter-Transfermethoden nur *ein* Perl-Skript benötigt wird.

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 15.2.1). Für die URL-Kodierung der Parameterwerte 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) {
      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://localhost/cgi-bin/demo.pl?vorname="+
              URLEncoder.encode(vorname,"Cp1252")+"&nachname="+
              URLEncoder.encode(nachname,"Cp1252"));
         BufferedReader br =
              new BufferedReader(new InputStreamReader(url.openStream()));
         String zeile;
         while ((zeile = br.readLine()) != null)
              System.out.println(zeile);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Weil das Programm die vom CGI-Skript gelieferte HTML-Seite nur als Text darstellt, ist sein Auftritt nicht berauschend:

```
Vorname:  Kurt
Nachname: Müller

<html>
<head><title>CGI-Demo</title></head>
<h1>Hallo, Kurt Müller!</h1>
</html>
```

In einer Übungsaufgabe sollen Sie ein GUI-Programm erstellen, das mit Hilfe eines **JEditorPane**-Objekts HTML-Code interpretieren und Browser-Qualität anzeigen kann.

Während bei vielen Browsern die maximale Länge einer GET-Parametersequenz auf 1024 Bytes beschränkt ist, kennt die nun vorzustellende POST-Methode keine Längenbegrenzung.

15.2.5.3 POST

Bei der POST-Methode, die man im **form** - Tag einer HTML-Seite durch die Angabe

```
method="post"
```

wählt, werden die Parameter (im selben Format wie bei der GET-Methode) mit Hilfe des WWW-Servers zur Standardeingabe der CGI-Software übertragen.[49] Was genau gemäß HTTP-Protokoll zu tun ist, braucht Java-Programmierer(innen) 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 15.2.2 über die **URL**-Methode **openConnection()** ein Objekt der Klasse **URLConnection** (genauer: **HttpURLConnection**) erzeugt und mit **setDoOutput(true)** darauf vorbereitet, dass Daten zum Server übertragen werden sollen. 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) {
      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://localhost/cgi-bin/demo.pl");
         URLConnection urlConn = url.openConnection();
         urlConn.setDoOutput(true);
         PrintWriter pw = new PrintWriter(urlConn.getOutputStream());
         pw.print("vorname="+URLEncoder.encode(vorname,"Cp1252")+
            "&nachname="+URLEncoder.encode(nachname,"Cp1252"));
         pw.close();
         BufferedReader br =
            new BufferedReader(new InputStreamReader(urlConn.getInputStream()));
         String zeile;
         while ((zeile = br.readLine()) != null)
              System.out.println(zeile);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Weil **PrintWriter** grundsätzlich puffern, sorgt im Beispiel ein **close()** dafür, dass die Parameterdaten auf die Reise gehen.
Weil dasselbe (für *beide* Parameterübergabemethoden geeignete) Perl-Skript angesprochen wird wie im GET-Beispiel, erhält man natürlich dieselbe Ausgabe.

---

[49] In obigem Perl-Skript ist das Lesen der Parameter aus STDIN durch die Kapselung in der **param()**-Methode eines CGI-Objektes nicht mehr zu erkennen.

15.3 Internet-Adressen ermitteln

Jeder an das Internet angeschlossen Rechner verfügt über eine **IP-Adresse** bzw. **-Nummer** (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.:

- **getLocalHost()**
  liefert ein **InetAddress**-Objekt zum lokalen Rechner
- **getByName**(**String** *Host-Name*)
  liefert ein **InetAddress**-Objekt zum Rechner mit dem angegebenen Host-Namen

Das folgende Programm stellt für den lokalen Rechner IP-Nummer 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:   136.199.198.106
Host-Name des lokalen Rechners:    rzbbg03
```

Das nächste Beispielprogramm kann die IP-Nummer zu einem Host-Namen ermitteln 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:

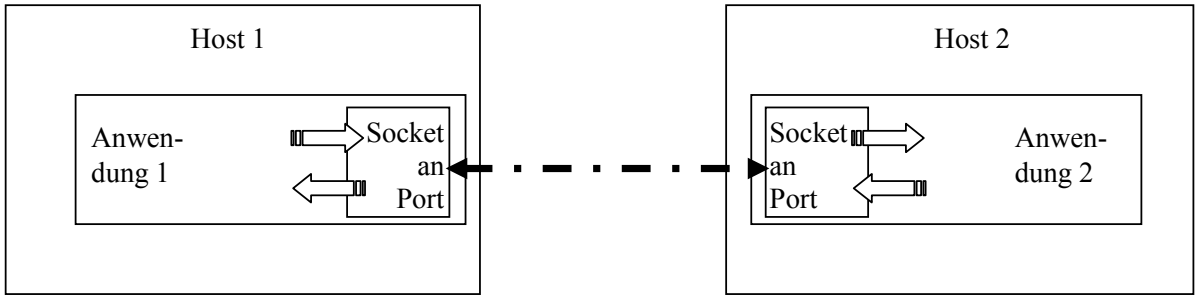
```
public void actionPerformed(ActionEvent ae) {
  if (ae.getSource() == ipBestimmen) {
    try {
      InetAddress ia = InetAddress.getByName(hostName.getText());
      ip.setText(ia.getHostAddress());
    } catch(UnknownHostException  uhe) {
      JOptionPane.showMessageDialog(null, "UnknownHostException: "+
        hostName.getText(), "Ungültiger Host-Name",
        JOptionPane.INFORMATION_MESSAGE);
    }
  } else
    System.exit(0);
}
```

Den vollständigen Quellcode finden Sie im Ordner

…\BspUeb\Netzwerkprogrammierung\DNS

15.4 Socket-Programmierung

Unsere bisherigen Beispielprogramme haben hauptsächlich WWW-Inhalte von Servern bezogen und dazu API-Klassen benutzt, die ein fest „verdrahtetes“ Anwendungsprotokoll (meist HTTP) realisieren. Im aktuellen Abschnitt bleiben wir zwar auf der Anwendungsebene, gewinnen aber 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 15.1.1) angesiedelte TCP-Protokoll schafft zwischen zwei (durch *Portnummern* identifizierten) Anwendungen, die sich meist auf verschiedenen (durch IP-Nummern 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 Klassen **Socket** und **ServerSocket** 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 Anwendungsprotokolls) Bytes in einen Ausgabestrom einspeisen bzw. aus einen Eingabestrom entnehmen. 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 dem Programmieren von **Klienten** und **Servern** und müssen dabei die jeweiligen Protokolle (der Anwendungsebene) implementieren. 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 dann einen dynamisch zugewiesenen Port benutzt.

15.4.1 TCP-Klient

Wir erstellen einen TCP-Klienten, der die aktuelle Tageszeit bei einem Server erfragt. Dazu erzeugen wir ein Objekt aus der Klasse **Socket**, das mit einem Zeit-Server Verbindung aufnimmt. Dazu muss im Konstruktor neben dem Host-Namen bzw. der IP-Adresse des Servers auch die Portnummer des Zeitansagers auftauchen. Üblicherweise lauschen solche Diesnte an Port 13. Mit **getInputStream()** besorgen wir uns beim **Socket**-Objekt seinen Eingabestrom, filtern ihn per **InputStreamReader** und puffern ihn per **BufferedReader**, damit Netzwerkzugriffe performant erfolgen:

```
import java.io.*;
import java.net.*;

class TimeClient {
  public static void main(String[] args) {
    try {
      Socket time = new Socket("uhr.uni-trier.de", 13);
      System.out.println("Verbindung hergestellt (lokaler Port: "+
        time.getLocalPort()+")");
      time.setSoTimeout(1000);
      BufferedReader br = new BufferedReader(
        new InputStreamReader(time.getInputStream()));
      System.out.print("Aktuelle Zeit: "+br.readLine());
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Sofern Netz und Server mitspielen, kann die gewünschte Uhrzeit per **readLine()** ermittelt werden, z.B.:

```
Verbindung hergestellt (lokaler Port: 1129)
Aktuelle Zeit: Thu Apr 27 21:33:14 2006
```

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, z.B.:

```
time.setSoTimeout(1000);
```

Beim Überschreiten der vereinbarten Wartezeit wird eine **SocketTimeoutException** geworfen, auf die das Programm geeignet reagieren kann.

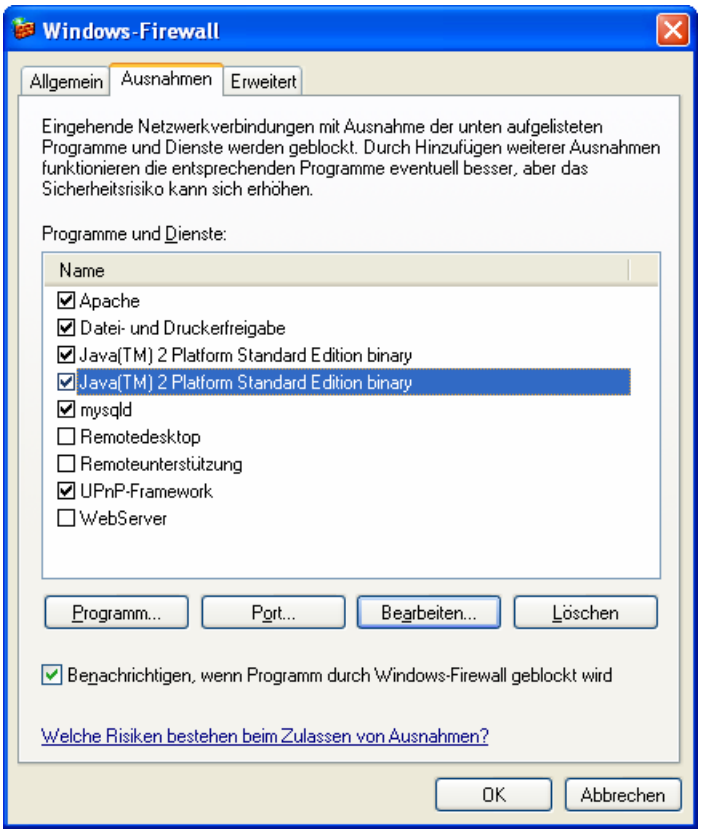
15.4.2 TCP-Server

Als Gegenstück zum eben präsentierten Zeit-Klienten wird nun ein eigener Zeit-Server entwickelt. Ein solcher Dienst kann übrigens unter Verwendung der so genannten **loopback-Adresse** 127.0.0.1 durchaus auf dem eigenen Rechner aufgesetzt werden, um Kosten und/oder Aufwand zu sparen. Allerdings ist der Dienst nur dann ansprechbar, wenn der Rechner über eine *aktive* Netzwerkverbindung (z.B. per Ethernet-LAN oder DFÜ) verfügt.

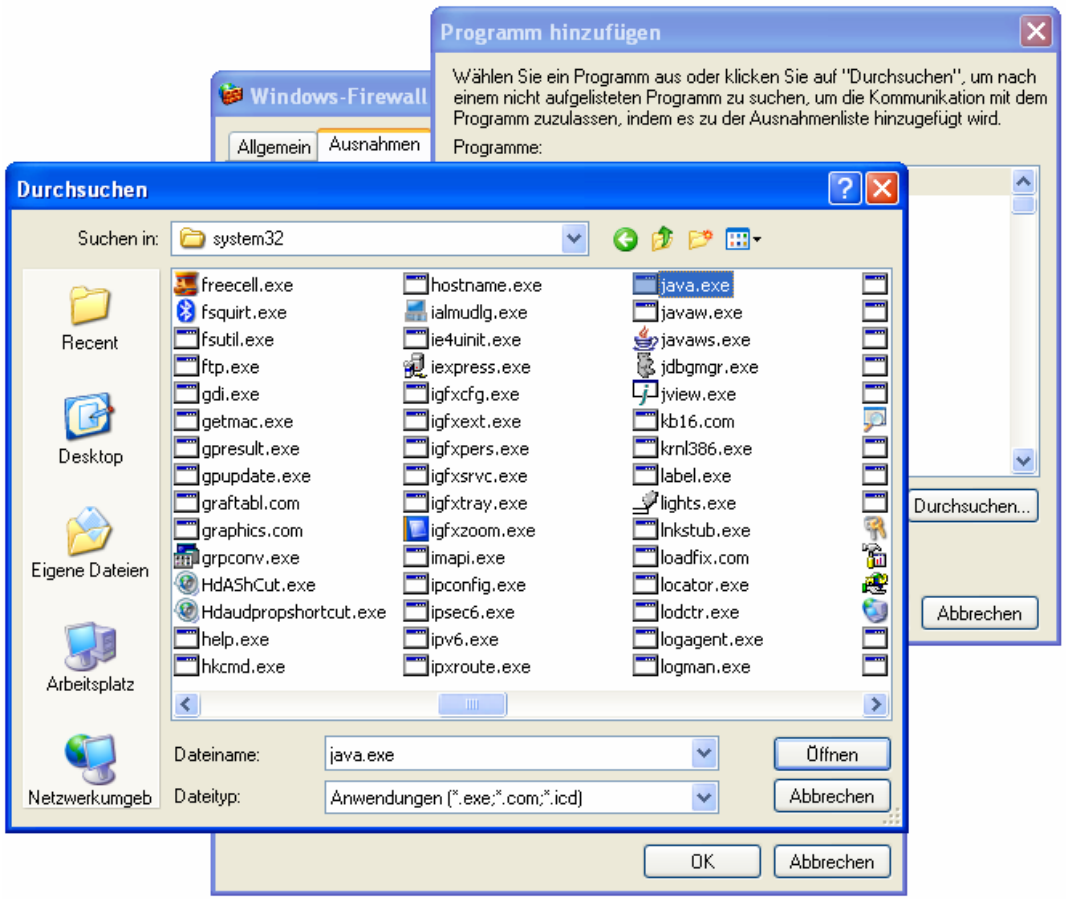
Damit unter Windows XP mit Service Pack 2 ein per **java.exe** oder **javaw.exe** gestartetes Serverprogramm an einen Port tätig werden darf, muss mit der Firewall eine entsprechende Ausnahme vereinbart werden. Man öffnet über

Systemsteuerung > Firewall

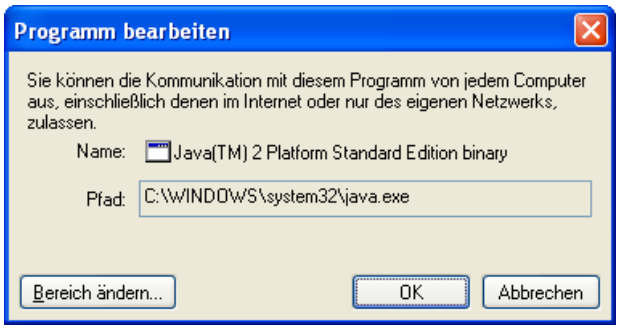
die folgende Dialogbox



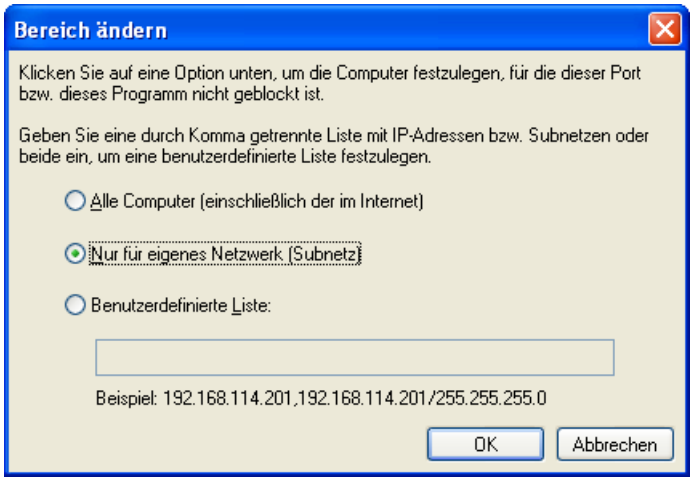
Nach einem Mausklick auf **Programm** wählt man einen JRE-Starter (**java.exe** oder **javaw.exe**):



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 Java-Anwendung!) erreichen, 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.

Nach diesen Vorbereitungen widmen wir uns wieder dem geplanten Zeit-Server und erzeugen zunächst ein Objekt aus der Klasse **ServerSocket**, das an den im Konstruktor angegebenen Port 13 gebunden wird. Für jede Klientenverbindung wird ein eigenes Objekt aus der schon bekannten **Socket**-Klasse benötigt. Mit der Methode **accept()** beauftragen wir das **ServerSocket**-Objekt, auf ein-

gehende Verbindungswünsche zu warten und ggf. zu anfragenden Klienten ein **Socket**-Objekt zu liefern. In den Ausgabestrom eines solchen Objekts wird dann die (wie in Abschnitt 15.2.2 mit Hilfe der Klassen **Date** und **DateFormat** erstellte) Zeitangabe geschrieben:

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.text.*;

class TimeServer {
   public static void main(String[] args) {
      DateFormat df = DateFormat.getDateTimeInstance();
      String zeit;
      try {
         ServerSocket timeServer = new ServerSocket(13);
         System.out.println("Zeitserver gestartet ("+df.format(new Date())+")");
         while(true) {
            Socket client = timeServer.accept();
            PrintWriter pw = new PrintWriter(client.getOutputStream());
            zeit = df.format(new Date());
            System.out.println("\n"+zeit+ " Anfrage von\n  IP-Nummer: "
               +client.getInetAddress().getHostAddress()
               +" (Port: "+client.getPort()+")");
            pw.print(zeit);
            pw.close();
            client.close();
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Während unser Klientenprogramm nach einer Anfrage beendet ist, bleibt Server permanent an Port 13 gebunden und bedient (nacheinander) beliebig viele Klienten:

```
Zeitserver gestartet (27.04.2006 23:49:29)

27.04.2006 23:51:50 Anfrage von
  IP-Nummer: 192.168.0.2 (Port: 1142)

27.04.2006 23:52:15 Anfrage von
  IP-Nummer: 192.168.0.7 (Port: 1051)

27.04.2006 23:52:23 Anfrage von
  IP-Nummer: 192.168.0.5 (Port: 1036)
```

Bei einer ernsthaften Serverprogrammierung kommt man an einer Multi-Thread-Lösung nicht vorbei, damit mehrere Klienten simultan bedient werden können. Dabei nicht für jede Anfrage zeitaufwändig ein neuer Thread gestartet werden muss, sollte sogar ein Thread-Pool zum Einsatz kommen, worauf das anschließend vorgestellte Beispielprogramm aber der Einfachheit halber verzichtet.

Wir erstellen nun einen Multi-Thread-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 als neuer Thread gestartet:

```
import java.net.*;

class EchoServer {
  public static void main(String[] args) {
    try {
      ServerSocket echoServer = new ServerSocket(9999);
      System.out.println("Echoserver gestartet");
      while(true) {
        Socket client = echoServer.accept();
        System.out.println("\nVerbindung mit:\n  IP-Nummer: "
          +client.getInetAddress().getHostAddress()
          +" (Port: "+client.getPort()+")");
        EchoServerThread est = new EchoServerThread(client);
        est.start();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

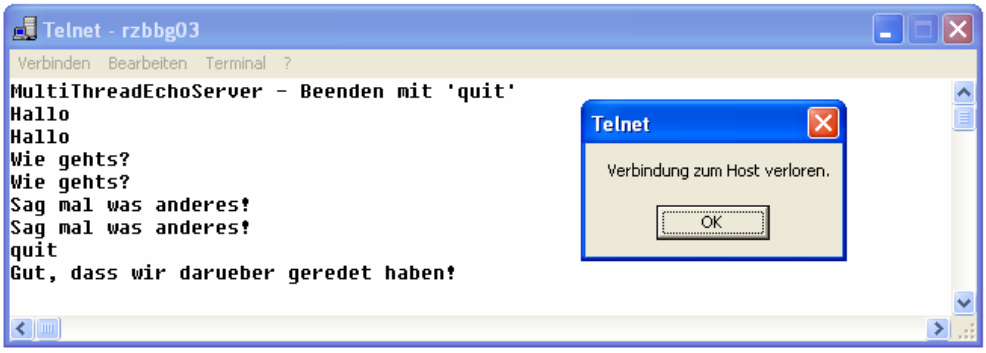
In der **run()**-Methode der Klasse `EchoServerThread` findet die Kommunikation über den Eingabe- und Ausgabestrom des **Socket**-Objektes statt. Im Konstruktor des am Ausgang angedockten **PrintWriter**s wird die **AutoFlush**-Eigenschaft auf **true** gesetzt, so dass er jede abgeschlossene Zeile automatisch aus dem Puffer abschickt.

```
import java.io.*;
import java.net.*;

class EchoServerThread extends Thread {
  Socket client;

  EchoServerThread(Socket cl) {
    client = cl;
  }

  public void run() {
    String zeile;
    try {
      BufferedReader br =new BufferedReader(
        new InputStreamReader(client.getInputStream()));
      PrintWriter pw = new PrintWriter(client.getOutputStream(), true);
      pw.println("MultiThreadEchoServer - 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);
      }
      client.close();
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

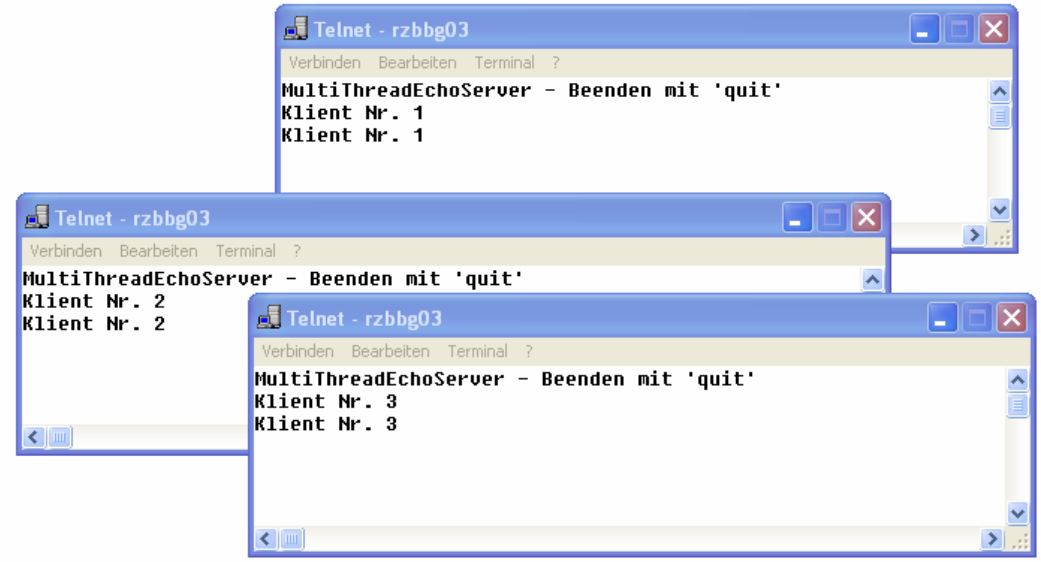
Man benötigt zum Testen des MultiThread - Echo-Servers keine spezielle Klienten-Software, sondern kann ihn mit jedem Telnet-Klienten ansprechen. In einen Konsolenfenster unter Windows taugt z.B. das folgende Kommando, um Kontakt aufzunehmen:

telnet *Host-Name* **9999**

Danach steht ein geduldig wiederholender Gesprächspartner bereit:

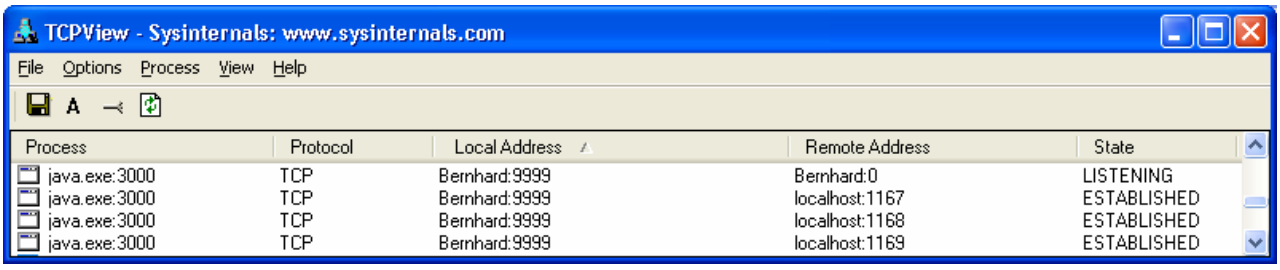


Wie die folgende Abbildung zeigt, kann der Server tatsächlich mehrere Klienten simultan versorgen kann:



Ein Telnet-Klient bewährt sich auch in anderen Situationen als bequemes Test- bzw. Debug-Werkzeug.

Nach allgemeiner (und durchaus nicht unplausibler Lehrbuchmeinung, siehe z.B. Deitel & Deitel, 2005, S. 1116) verwendet das vom **ServerSocket**-Objekt als Rückgabe der **accept()**-Methode gelieferte **Socket**-Objekt einen eigenen, neu vergebenen Port, so dass der im **ServerSocket**-Konstruktor angegebene Port weiterhin ausschließlich zum Lauschen auf neue Verbindungswünsche dient. Allerdings liefert weder die **Socket**-Methode **getLocalPort()** noch eine Analyse der TCP-Verbindungen mit dem Programm TCPView (erhältlich über den URL http://www.sysinternals.com/) einen Hinweis auf diese zusätzlichen Ports. Hier ist der Echoserver unter Verwendung des Ports 9999 mit drei Klienten verbunden und lauscht weiter auf neue Anfragen:



Es ist für die JRE sicher kein Problem, die am Port 9999 von verschiedenen Klienten eintreffenden Sendungen an der korrekten Socket zu übermitteln.

15.5 Übungsaufgaben zu Abschnitt 15

1) Erstellen Sie ein Programm, das bei Bedarf für die Sophos-Virenschutzsoftware ergänzende Virendefinitionen (erkennbar an der Dateinamenserweiterung **.ide**) vom WWW-Server des Herstellers holt und ins Sophos-Installationsverzeichnis befördert, so dass sie nach dem nächsten Start des Programms wirksam werden.
Die URL zum Virendefinitions-Aktualisierungspaket für die Sophos-Version 4.05 lautet z.B.:

http://www.sophos.com/downloads/ide/405_ides.zip

Aus Kostengründen sollte die Definitionsdatei nur dann herunter geladen werden, wenn sie sich seit dem letzten Zugriff geändert hat.
Vermutlich benötigt Ihr Programm eine Parameterdatei mit folgenden Informationen:

- Sophos-Installationsverzeichnis
- Sophos-Version
- Zeitpunkt des letzten Zugriffs

Zum Auspacken des ZIP-Archivs eignet sich die Klasse **ZipFile** im Paket **java.util.zip**.

2) Erstellen Sie ein Programm, das die webbasierte Telefondatenbank der Universität Trier durchsucht und die Ergebnisse mit Hilfe eines **JEditorPanel**-Objektes im HTML-Format anzeigt. Das zuständige CGI-Programm ist unter folgender Adresse

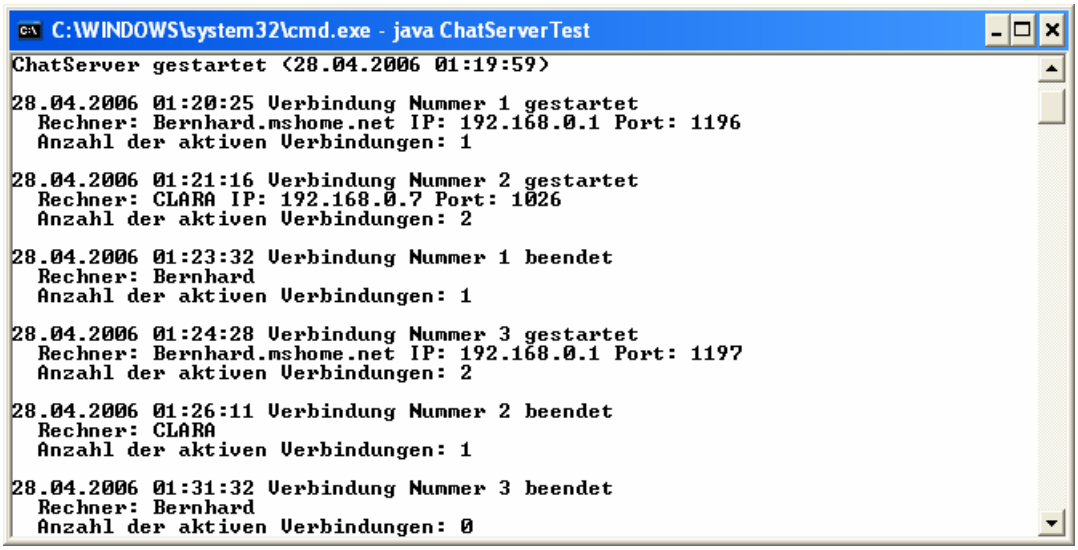
http://www.uni-trier.de/cgi-bin/telesuche

ansprechbar und kennt nur einen Parameter:

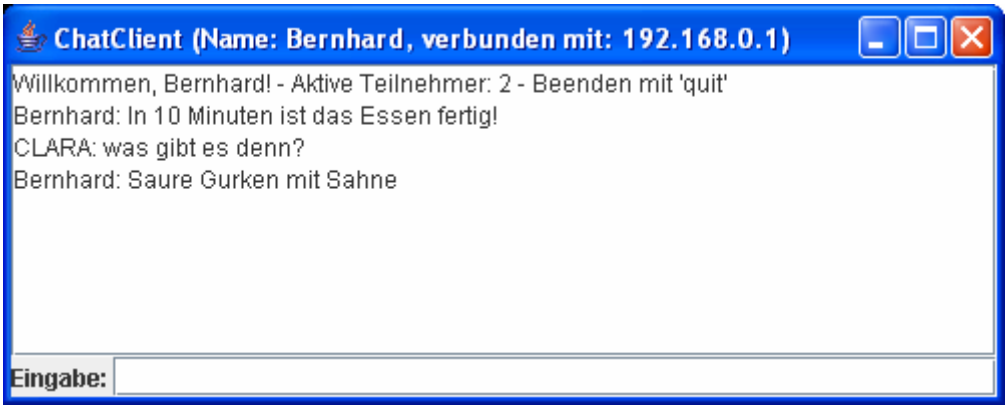
- Name: **keyword**
- Wert: Nachname der Person, deren Telefonnummer zu suchen ist
- Transfermethode: POST

Bei dieser Musterlösung fehlt vielleicht der ehrliche Hinweis darauf, dass die eigentliche Arbeit (Pflegen und Durchsuchen der Datenbank) anderen Leuten überlassen wird:

3) 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 das gesamte Kommunikationsprotokoll erscheint in einem **JTextArea**-Steuerelement.

16 Datenbankzugriff via JDBC

In diesem Abschnitt soll ein erster Eindruck zur Verwendung von Datenbanken in Java-Anwendungen vermittelt werden. Für den Begriff *Datenbank* schlägt Ebner (2000, S. 21) folgende Definition vor:

> *Eine Datenbank ist eine Sammlung von nicht-redundanten Daten, die von mehreren Anwendungen benutzt werden kann.*

Redundanz würde sich z.B. in der Datensammlung eines Versandhauses schnell einstellen, wenn man für jede Bestellung einen Datensatz anlegen und dabei auch die Adresse des Bestellers einbeziehen würde. Sobald von einem Kunden mehrere Bestellungen vorlägen, wäre seine Adresse mehrfach vorhanden. Neben dem unsinnigen Erfassungsaufwand und der Platzverschwendung hat die Redundanz einen weiteren Nachteil: die Gefahr **inkonsistenter** Daten.
Mit dem zweiten Kriterium aus obiger Definition (*Benutzbarkeit durch mehrere Anwendungen*) ist in erster Linie gemeint, dass Anwendungsprogramme nicht direkt auf die Datenbestände zugreifen sollen, sondern nur über ein spezielles **Datenbankmanagementsystem** (**DBMS**). Bekannte Beispiele für diese Softwaregattung sind: MySQL, Microsoft SQL-Server, DB2 von IBM, Oracle usw.
Bei Anwendungen, die mit komplexen Daten arbeiten müssen, hat es einige Vorteile, die Verwaltung der Daten einem DBMS zu überlassen und auf eigene Dateizugriffe zu verzichten, z.B.:

- bequeme und flexible Datenzugriffe
- Sicherheit, speziell beim Zugriff durch mehrere Benutzer bzw. Programme

Im Java-API erlaubt die **JDBC**-Bibliothek (*Java Database Connectivity*) einen einheitlichen Zugriff auf alle DBMS-Systeme mit geeignetem Treiber, wobei es mittlerweile bei der Verfügbarkeit von JDBC-Treibern kaum Probleme gibt. Im Zweifelsfall kann man sich über die folgende Webseite Gewissheit verschaffen:

http://java.sun.com/products/jdbc

16.1 Relationale Datenbanken

Heute arbeitet praktisch jedes DBMS mit der **relationalen Datenbankstruktur**, die sich gegenüber älteren Bauformen (z.B. hierarchische Datenbank, Netzwerk-Datenbank) weitgehend durchgesetzt hat und auch gegenüber den neuerdings vorgeschlagenen objektorientierten Datenbanken behauptet.

Bei einer relationalen Datenbank sind die Daten in *Tabellen* angeordnet, wobei die Spalten auch als *Felder* oder *Variablen* und die Zeilen auch als *Datensätze* (*engl: Records)* oder *Fälle* bezeichnet werden:[50]

- Jede Zeile der Tabelle enthält die zu einem Fall verfügbaren Informationen, wobei jede Tabelle Fälle eines bestimmten Typs enthält (z.B. Kunden, Artikel, Lieferanten). Welche Tabellen benötigt werden, hängt vom Anwendungsbereich ab.
- Jede Spalte (jedes Feld) enthält für alle Fälle die Werte zu einem Merkmal.
- Eine bestimmte Spalte dient als *Primärschlüssel* und enthält eindeutige Werte.

---

[50] Meist wird mit dem Begriff *Relation* eine Beziehung zwischen 2 Tabellen bezeichnet (siehe unten). Manche Autoren verwenden ihn jedoch synonym zum Begriff *Tabelle*, was durchaus mit der mathematischen Definition einer Relation als Teilmenge eines kartesischen Produktes aus mehreren Mengen harmoniert: Jede von den *m* Variablen einer Tabelle steuert die Menge ihrer potentiellen Ausprägungen bei. Ein Element des kartesischen Produktes dieser *m* Mengen ist ein *m*-Tupel mit *m* speziellen Variablenausprägungen, also eine Tabellenzeile. Eine Tabelle mit *n* Zeilen ähnelt also in der Tat einer Teilmenge des kartesischen Produkts.

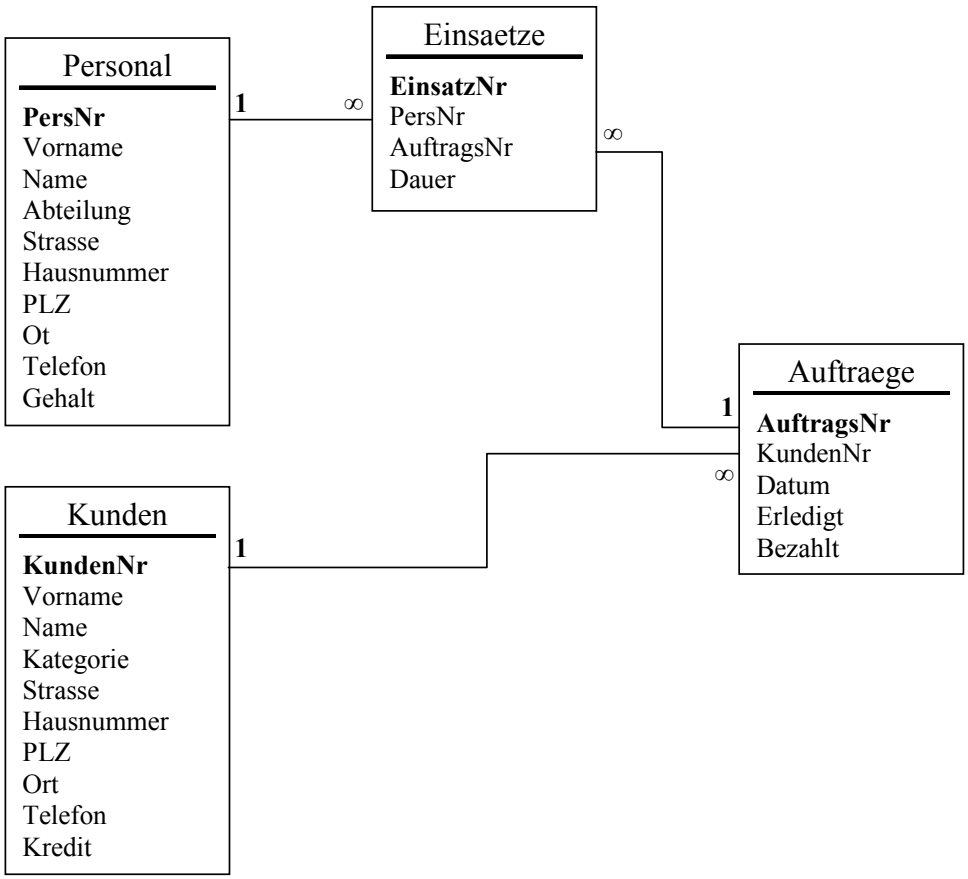
Wir betrachten in diesem Abschnitt eine aus insgesamt vier Tabellen bestehende Datenbank mit fiktiven Daten eines Handwerkbetriebs, die unter anderem eine Tabelle mit dem Personal der Firma enthält:

Tabellen der Firmendatenbank

Personal

PersNr	Vorname	Name	Abteilung	Strasse	Hausnummer	PLZ	Ort	Telefon	Gehalt
1	Ulla	Schneider	B	Heide	29	51594	Raltop	05123-12094	1900.0
2	Otto	Schmidt	A	Goethegasse	12a	52292	Drohntal	07319-78129	2490.5
3	Ludger	Müller	B	Hauptstraße	45	53277	Urdorf	04543-32212	2260.33
4	Emanuel	Fink	B	An der Ecke	177	54822	Mondorf	03423-73212	1693.0
5	Kurt	Schmidt	A	SüdStrasse	23	52292	Drohntal	07319-53487	2630.5
6	Monika	Gerber	C	Mühlenstraße	91	51594	Raltop	05123-78123	2600.0

Das zur Anzeige der Tabellen verwendete Java-Programm mit **JTable**-Steuerelement wird in Abschnitt 16.4.3 besprochen.
In der folgenden Abbildung sind alle Tabellen der Beispieldatenbank mit den zugehörigen Feldern und den Primärschlüsseln (in fetter Schrift) zu sehen:



Zwischen zwei Tabellen (z.B. **Kunden** und **Auftraege**) kann über eine gemeinsame Variable (z.B. **KundenNr**) eine **Relation** (alias: **Referenz**) hergestellt werden. Weil zu jedem Kunden potenziell mehrere Aufträge gehören, spricht man hier von einer Master-Detail – bzw. (1:∞) – Relation. Bei Betrachtung in umgekehrter Richtung wird daraus eine Detail-Master - bzw. (∞:1) – Relation.

Relationale Datenbanken werden uns als Industriestandard noch lange erhalten bleiben, denn …

- Es sind zahlreiche relationale DBMS-Produkte mit einem hohen Reifegrad und einer großen installierten Basis vorhanden.
- Der Zugriff auf relationale Datenbanken ist über die Abfragesprache SQL (siehe Abschnitt 16.2) weitgehend standardisiert.

- Relationale Datenbanken finden eine exzellente Unterstützung durch Entwicklungswerkzeuge, Analyseprogramme etc.

Allerdings ist es sicher nicht optimal, dass ein objektorientiert denkender Java-Programmierer bei der Datenverwaltung mit traditionellen Konzepten arbeiten und dabei auch noch eine zusätzliche Programmiersprache (SQL) verwenden muss. Mittlerweile ist in Java der objektorientierte Zugriff auf relationale Datenbanken über Brückenlösungen (z.B. *Hibernate*) möglich. Mit diesem *objektrelationalen Mapping*) (siehe z.B. Minhorst & Schäfer 2006) können wir uns aber in diesem Manuskript nicht befassen.

16.2 SQL

Ein DBMS interagiert mit anderen Programmen über die **Structured Query Language** (SQL). Wie der Name *SQL* nahe legt, ist die *Abfrage* von Informationen klarer Einsatzschwerpunkt, doch deckt der Sprachumfang alle Aufgaben der Datenverwaltung ab, wobei sich folgende Teilmengen unterscheiden lassen:

- DDL (**D**ata **D**efinition **L**anguage)
  Mit den DDL-Befehlen (z.B. **CREATE TABLE**, **DROP TABLE**) kann man die Struktur einer Datenbank modifizieren.
- DML (**D**ata **M**anipulation **L**anguage)
  Neben dem besonders wichtigen Befehl **SELECT** zum Abrufen, Auswählen, Suchen und Auswerten von Datensätzen gehören in diese Gruppe noch folgende Befehle:
  - **INSERT**
    Hängt einen Datensatz am Ende einer Tabelle an
  - **DELETE**
    Löscht einen Datensatz
  - **UPDATE**
    Damit **lassen** sich z.B. alle Werte in einer Tabellen-Spalte neu setzen.

16.2.1 Abfragen per SELECT-Anweisung

Das folgende Syntaxdiagramm zum **SELECT**-Befehl beschränkt sich auf die anschließend besprochenen Ausdrucksmittel:

```
SELECT  {* | Spalten}
   FROM Tabellen
   [WHERE logischer Ausdruck]
   [ORDER BY Sortierkriterium [ASC | DESC] [,Sortierkriterium [ASC | DESC] ...]]
   [GROUP BY Gruppierungsspalte [, Gruppierungsspalte ...]]
```

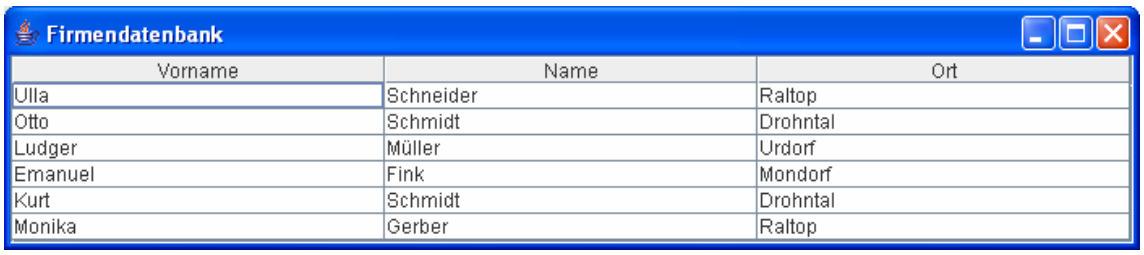
Im Unterschied zu Java ist in SQL die Groß/Klein-Schreibung irrelevant Es hat sich aber eingebürgert, die Schlüsselwörter groß zu schreiben.

16.2.1.1 Spalten einer Tabelle abrufen

Will man im **SELECT**-Befehl *alle* Spalten einer Tabelle abrufen, ist ein Stern zu setzen (*). Mehrere einzelne Spalten sind durch ein Komma zu trennen.
In der **FROM**-Klausel wird die Tabelle genannt, aus der die abgerufenen Spalten stammen sollen.
Beispiel:

```
SELECT Vorname, Name, Ort FROM Personal;
```

Ergebnis:



16.2.1.2 Fälle auswählen über die WHERE-Klausel

Mit der **WHERE**-Klausel der **SELECT**-Anweisung kann über einen logischen Ausdruck eine Teilmenge von Fällen ausgewählt werden, z.B.:

```
SELECT Vorname, Name, Ort FROM Personal WHERE Ort='Raltop';
```

Ergebnis:

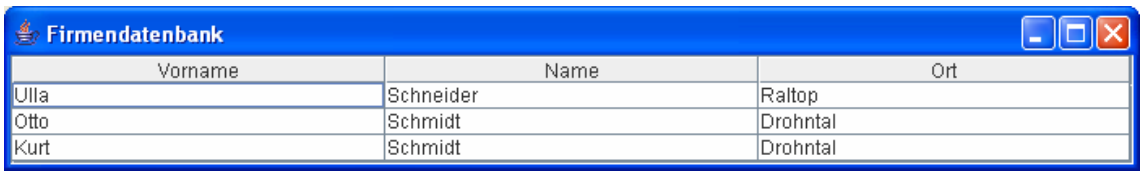
Für die Suche nach Teilzeichenfolgen bietet SQL den Vergleichsoperator **LIKE**, wobei folgende Jokerzeichen zur Verfügung stehen:

- % ersetzt ein oder mehrere beliebige Zeichen
- _ ersetzt genau ein beliebiges Zeichen

Der folgende Befehl spürt alle Firmenangehörigen in der Beispieldatenbank auf, deren Nachname mit „S“ beginnt:

```
SELECT Vorname, Name, Ort FROM Personal WHERE Name LIKE 'S%';
```

Ergebnis:



Leere Feldinhalte lassen sich mit dem Schlüsselwort **NULL** ansprechen, z.B.:

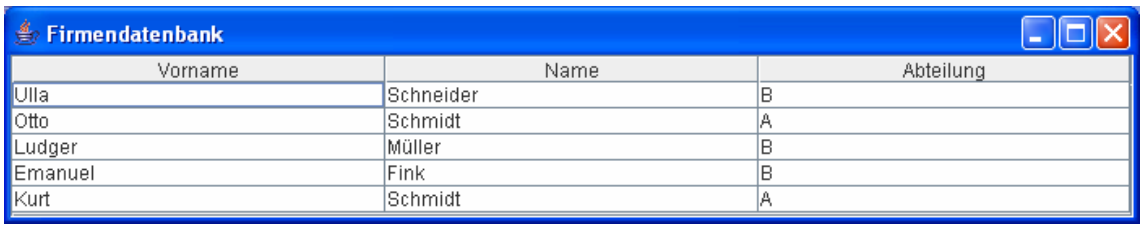
```
SELECT Vorname, Name, Ort FROM Kunden WHERE Kategorie IS NULL;
SELECT Vorname, Name, Ort FROM Kunden WHERE Kategorie IS NOT NULL;
```

Ergebnis der ersten Abfrage für die Beispieldatenbank:

Recht praktisch ist auch der Vergleichsoperator **IN**, dem eine Menge mit Trefferwerten folgt darf:

```
SELECT Vorname, Name, Abteilung FROM Personal WHERE Abteilung IN ('A', 'B');
```

Ergebnis:



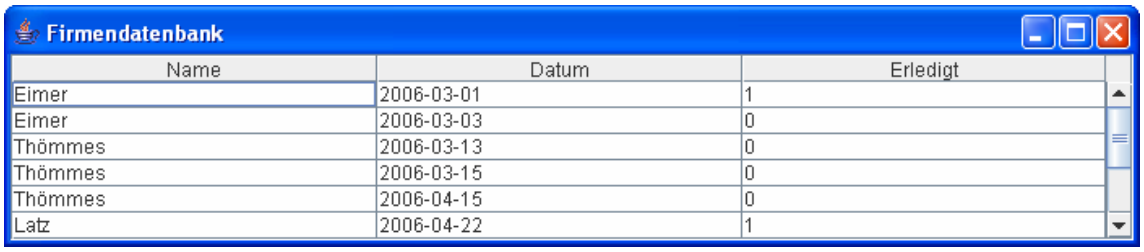
Vorname	Name	Abteilung
Ulla	Schneider	B
Otto	Schmidt	A
Ludger	Müller	B
Emanuel	Fink	B
Kurt	Schmidt	A

16.2.1.3 Daten aus mehreren Tabellen zusammenführen

Sobald die **FROM**-Klausel mehrere Tabellen enthält, muss den Namen der abgerufenen Spalten der Tabellenname vorangestellt werden, z.B.:

```
SELECT Kunden.Name, Auftraege.Datum, Auftraege.Erledigt
  FROM Kunden, Auftraege WHERE Kunden.KundenNr = Auftraege.KundenNr;
```

Ergebnis:



Name	Datum	Erledigt
Eimer	2006-03-01	1
Eimer	2006-03-03	0
Thömmes	2006-03-13	0
Thömmes	2006-03-15	0
Thömmes	2006-04-15	0
Latz	2006-04-22	1

Ohne **WHERE**-Klausel werden die beiden Spalten nach einem zwar systematischen, aber wohl nur selten sinnvollen Verfahren zusammengeführt: Jeder Fall der ersten Tabelle wird mit jedem Fall der zweiten Tabelle kombiniert. Obige **WHERE**-Klausel legt fest, dass aus einer Zeile der Tabelle **Kunden** und einer Zeile der Tabelle **Auftraege** nur dann eine Zeile der neuen Tabelle entstehen soll, wenn beide Zeilen im Feld **KundenNr** denselben Wert haben.
Die als Abfrageergebnis resultierende Tabelle enthält für jeden Auftrag eine Zeile, in der neben dem Erledigt-Vermerk auch den Namen des Kunden auftritt.
Es liegt eine Detail-Master – Relation vor, die man auch mit **INNER JOIN** konstruieren kann, z.B.:

```
SELECT Kunden.Name, Auftraege.Datum, Auftraege.Erledigt
  FROM Kunden INNER JOIN Auftraege
  ON Kunden.KundenNr = Auftraege.KundenNr;
```

Zum Glück lässt sich der bei Beteiligung mehrerer Tabellen anfallende Schreibaufwand durch Abkürzungen begrenzen:

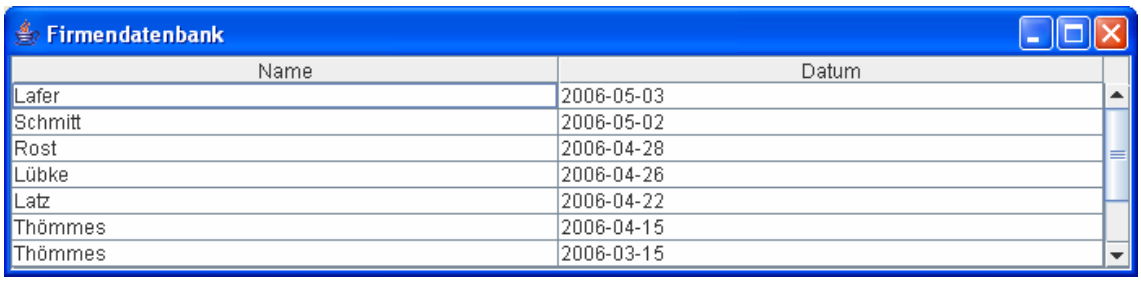
```
SELECT K.Name, A.Datum, A.Erledigt FROM Kunden K, Auftraege A
  WHERE K.KundenNr = A.KundenNr;
```

16.2.1.4 Abfrageergebnis sortieren

Mit der Klausel **ORDER BY** lässt sich das Abfrageergebnis (auf oder absteigend) sortieren, wobei Spalten oder numerische Ausdrücke als Sortierkriterien in Frage kommen, z.B.:

```
SELECT K.Name, A.Datum FROM Kunden K, Auftraege A
  WHERE K.KundenNr = A.KundenNr ORDER BY A.Datum DESC;
```

Ergebnis:



16.2.1.5 Auswertungsfunktionen

Bei den Spaltendefinitionen einer **SELECT**-Anweisung stehen auch einige Auswertungsfunktionen zur Verfügung. Im folgenden Beispiel wird das mittlere Gehalt aller Mitarbeiter(innen) festgestellt:

```
SELECT AVG(Gehalt) FROM Personal;
```

Ergebnis:

Wichtige SQL-Auswertungsfunktionen:

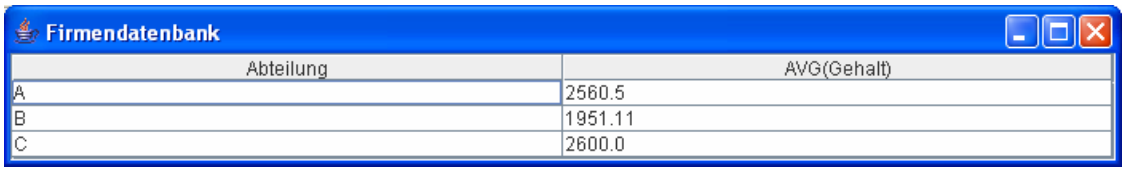
Funktion	**Beschreibung**
COUNT()	Zählt die Zeilen
SUM()	Summiert über eine Spalte
AVG()	Mittelt über eine Spalte
MIN()	Ermittelt das Minimum einer Spalte
MAX()	Ermittelt das Maximum einer Spalte

16.2.1.6 Daten aggregieren

Über die Klausel **GROUP BY** kann man Untergruppen bilden, für die sich dann obige Funktionen auswerten lassen. In unserem Beispiel kann man etwa über die Variable `Abteilung` aggregieren, um jeweils das mittlere Gehalt festzustellen:

```
SELECT Abteilung, AVG(Gehalt) FROM Personal GROUP BY Abteilung;
```

Ergebnis:



16.2.2 Sonstige SQL-Anweisungen

16.2.2.1 CREATE DATABASE

Eine neue Datenbank erzeugt man mit **CREATE DATABASE**, z.B.:

```
CREATE DATABASE IF NOT EXISTS Firma;
USE Firma;
```

Hier wird die Datenbank `Firma` erzeugt. Mit der Klausel **IF NOT EXISTS** verhindert man eine Fehlermeldung, falls bereits eine Datenbank mit diesem Namen existiert.

Aufgrund des **USE**-Kommandos weiß das DBMS, dass sich spätere Befehle auf die Datenbank `Firma` beziehen, so dass den Namen von Tabellen kein Datenbankname vorangestellt werden muss.

16.2.2.2 CREATE TABLE

Mit **CREATE TABLE** erzeugt und definiert man eine neue Datenbanktabelle, z.B.:

```
CREATE TABLE Kunden (
  KundenNr INT NOT NULL AUTO_INCREMENT,
  Vorname VARCHAR (25) NOT NULL,
  Name VARCHAR (50) NOT NULL,
  Kategorie INT (2),
  Strasse VARCHAR (40) NOT NULL,
  Hausnummer VARCHAR (5) NOT NULL,
  PLZ CHAR (5) NOT NULL,
  Ort VARCHAR (40) NOT NULL,
  Telefon VARCHAR(20) NOT NULL,
  Kredit DOUBLE NOT NULL,
  PRIMARY KEY (KundenNr)
) TYPE=INNODB;
```

Hier wird die Tabelle `Kunden` mit diversen Spalten unterschiedlichen Datentyps angelegt (vgl. MySQL AB, 2006).
Mit dem Attribut **NOT NULL** wird für eine Spalte festgelegt, dass fehlende Werte verboten sind.
Mit der **PRIMARY**-Klausel wählt man eine Spalte als Primärschlüssel aus. Im Beispiel wird bei der betroffenen Spalte `KundenNr` mit dem Attribut **AUTO_INCREMENT** dafür gesorgt, dass neue Zeilen automatisch fortlaufende Werte erhalten.
Mit dem Typ **INNODB** bietet MySQL für eine Datenbanktabelle Transaktionssicherheit, hohe Performanz und enorme Speicherkapazität (im Terabyte-Bereich, abhängig vom Betriebssystem)

Mit der **INDEX**-Klausel kann eine Spalte als Index gewählt werden, was z.B. erforderlich ist für Spalten, die bei einer Referenz (siehe oben) mitwirken sollen. In der `Einsaetze`-Tabelle der Beispieldatenbank kann ein Mitarbeiter ebenso wie ein Auftrag an mehreren Zeilen beteiligt sein. Folglich werden für die `Einsaetze`-Spalten `PersNr` und `AuftragsNr` Referenzen zur jeweils zugehörigen Spalte in der `Personal`- bzw. `Auftraege`-Tabelle vereinbart:

```
CREATE TABLE Einsaetze (
  EinsatzNr INT NOT NULL AUTO_INCREMENT,
  PersNr INT NOT NULL,
  AuftragsNr INT NOT NULL,
  Dauer DOUBLE NOT NULL,
  PRIMARY KEY (EinsatzNr),
  INDEX (PersNr),
    FOREIGN KEY (PersNr) REFERENCES Personal (PersNr),
  INDEX (AuftragsNr),
    FOREIGN KEY (AuftragsNr) REFERENCES Auftraege (AuftragsNr)
) TYPE=INNODB;
```

Aufgrund dieser Vereinbarung kann z.B. kein Mitarbeiter, der bei einem Einsatz beteiligt ist, aus der `Personal`-Tabelle gelöscht werden oder eine neuen `PersNr`-Wert erhalten.

16.2.2.3 DROP

Per **DROP** lassen sich einzelne Tabellen oder komplette Datenbanken entfernen, z.B.:

- `DROP DATABASE IF EXISTS Firma;`
  Hier wird die komplette `Datenbank` `Firma` mit allen zugehörigen Dateien gelöscht. Mit der Klausel **IF EXISTS** wird eine Fehlermeldung bei nicht existenter Datenbank verhindert.
- `DROP TABLE IF EXISTS Personal;`
  Hier wird die Tabelle `Personal` der Datenbank `Firma` gelöscht. Mit der Klausel **IF EXISTS** wird eine Fehlermeldung bei nicht existenter Tabelle verhindert.

16.2.2.4 INSERT

Mit dem **INSERT**-Kommando trägt man eine neue Zeile in eine Tabelle ein, z.B.:

```
INSERT INTO Personal (Vorname, Name, Abteilung, Strasse, Hausnummer, PLZ,
  Ort,Telefon, Gehalt) VALUES ('Ulla', 'Schneider', 'B', 'Heide', '29',
  '51594', 'Raltop', '05123-12094', 1900.00);
```

Beim Einfügen neuer Tabellenzeilen erhalten Spalten ohne explizite Versorgung automatisch einen Voreinstellungswert (z.B. leere Zeichenfolge). Soll ein Wert explizit fehlen, ist das Schlüsselwort **NULL** anzugeben, was bei Spalten mit dem Attribut **NOT NULL** allerdings verboten ist.

16.2.2.5 DELETE

Mit dem **DELETE**-Kommando löscht man Zeilen aus einer Tabelle, z.B.:

```
DELETE FROM Einsaetze WHERE PersNr = 1;
```

Aus der Tabelle `Einsaetze` werden alle Zeilen mit der `PersNr` 1 entfernt.
Anschließend könnte übrigens der betroffene Mitarbeiter trotz der oben beschriebenen Referenz auch aus der `Personal`-Tabelle gelöscht werden:

```
DELETE FROM Personal WHERE PersNr = 1;
```

16.2.2.6 UPDATE

Mit dem **UPDATE**-Kommando verändert man die Daten in einer Tabelle, z.B.:

```
UPDATE Personal SET Gehalt = 1800.50 WHERE PersNr = 4;
```

Hier wird das Gehalt eines Mitarbeiters erhöht.

16.3 MySQL unter MS Windows installieren und einrichten

Wir werden in diesem Manuskript die Open Source - Datenbank MySQL in der Version 5.0 einsetzen, die für Windows, Linux, MacOS, diverse UNIX-Varianten und Novell Netware über folgende Webseite verfügbar ist:

http://dev.mysql.com/downloads/

Als gedruckte Information zur Ergänzung der MySQL-Dokumentation (MySQL AB, 2006) eignet sich z.B. RRZN (2004).

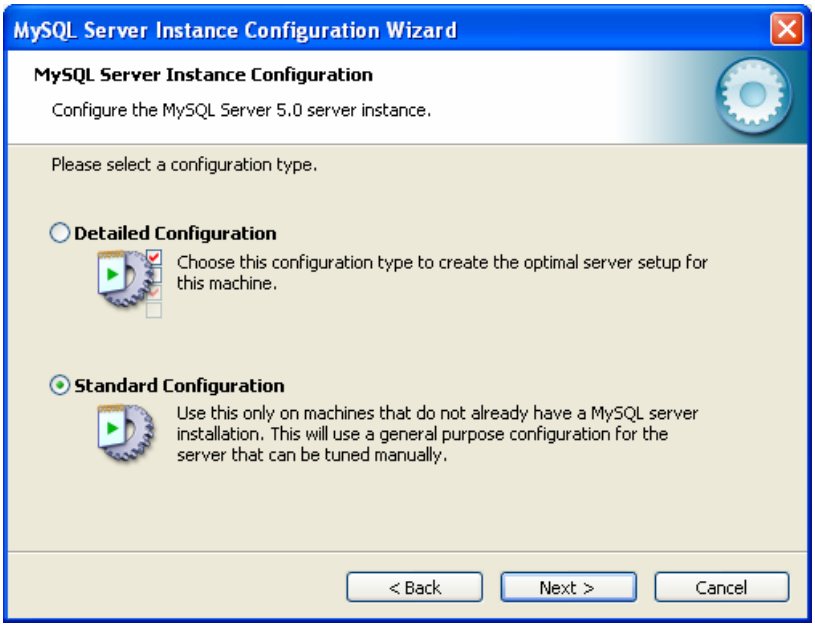
16.3.1 Installation

Unter MS-Windows landet die via **Setup.exe** gestartete MySQL-Installation per Voreinstellung im Ordner

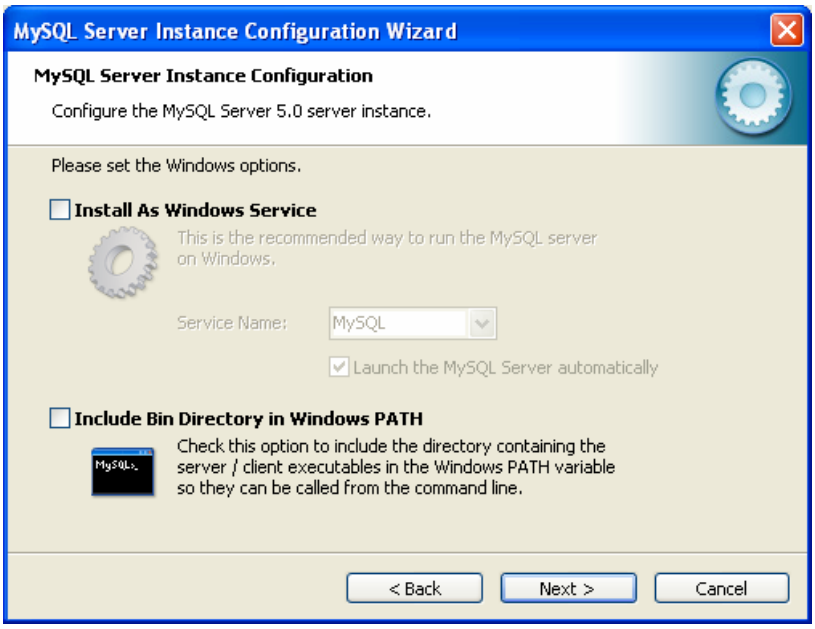
C:\Programme\MySQL\MySQL Server 5.0

Man sollte von der Möglichkeit Gebrauch machen, den MySQL-Server unmittelbar nach der Installation zu konfigurieren:

Wir beschränken uns auf die Standardkonfiguration:



Soll der MySQL-Server nur gelegentlich laufen, ist eine Installation als Windows-Dienst *nicht* erforderlich:



Es entsteht die Konfigurationsdatei

C:\Programme\MySQL\MySQL Server 5.0\my.ini

Über das Programm

C:\Programme\MySQL\MySQL Server 5.0\bin\MySQLInstanceConfig.exe

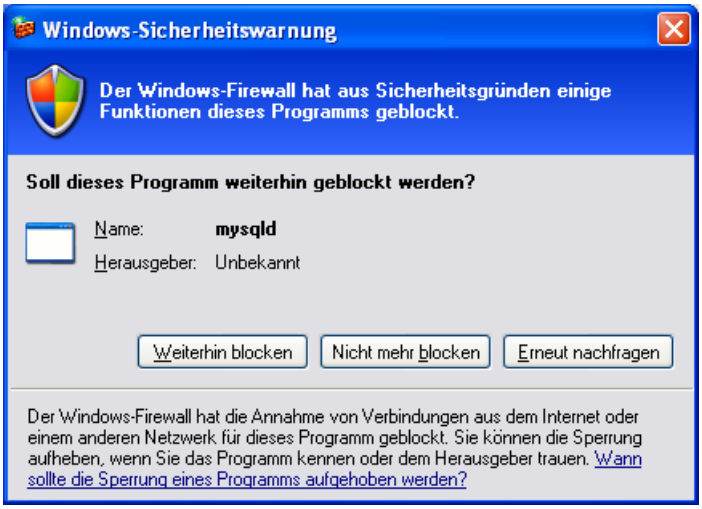
kann der Konfigurationsassistent später erneut gestartet werden.

16.3.2 Ausnahme für die Windows-Firewall

Um den MySQL-Server manuell in Betrieb zu nehmen, startet man das Programm

C:\Programme\MySQL\MySQL Server 5.0\bin\mysqld.exe

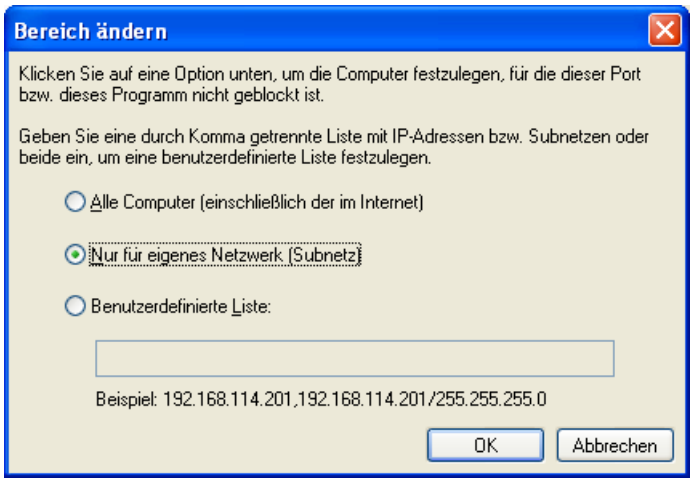
Beim ersten Start erkundigt sich Windows XP mit SP2, ob MySQL als Server fungieren darf:



Nach einem Mausklick auf **Nicht mehr blocken** wird eine Firewall-Ausnahme angelegt, die nach

Systemsteuerung > Windows-Firewall

nachjustiert werden sollte. Es bietet sich an, die Liste der zugriffsberechtigten Systeme einzuschränken, z.B.:



16.3.3 Passwort für den Benutzer root vereinbaren

Weil der höchstprivilegierte Benutzer **root** noch kein Passwort besitzt, kann jeder lokale Benutzer den MySQL-Server mit dem folgenden Kommando stoppen:

```
C:\Programme\MySQL\MySQL Server 5.0\bin>mysqladmin -u root shutdown
```

Zwar darf sich der Benutzer **root** nur am lokalen Rechner beim MySQL-Server anmelden, doch sollte er trotzdem ein Passwort erhalten.[51] Falls Sie den Server eben gestoppt haben, müssen Sie nun reaktivieren

```
C:\Programme\MySQL\MySQL Server 5.0\bin>mysqld
```

Starten Sie in einem Konsolenfenster den MySQL-Monitor als Benutzer **root**:

```
C:\Programme\MySQL\MySQL Server 5.0\bin>mysql -u root
```

Der Monitor antwortet:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 5.0.21-community

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

Tragen Sie mit dem SQL-Kommando **UPDATE** ein **root**-Passwort ein, z.B.:

```
mysql> UPDATE mysql.user SET Password=PASSWORD('zTg#d_8K') WHERE User='root';
```

MySQL antwortet:

```
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Gerade haben Sie erfahren, dass zur Verwaltung der MySQL-Benutzer die vordefinierte Datenbank **mysql** mit der Tabelle **user** dient.
Sorgen Sie mit **FLUSH PRIVILEGES** dafür, dass der Server das Passwort ohne Neustart sofort kennt:

```
mysql> FLUSH PRIVILEGES;
```

MySQL antwortet:

```
Query OK, 0 rows affected (0.00 sec)
```

Beenden Sie den Monitor:

```
mysql> exit
```

---

[51] Mit dem Konfigurationsassistenten von MySQL 5.0 lässt sich ein **root**-Passwort nur dann vereinbaren, wenn zuvor der MySQL-Start als Windows-Dienst vereinbart worden ist.

MySQL antwortet:

```
Bye
```

Wenn Sie nun versuchen, den Server mit dem Kommando

```
C:\Programme\MySQL\MySQL Server 5.0\bin>mysqladmin -u root –p shutdown
```

zu stoppen, wird das eben vereinbarte Passwort verlangt.

16.3.4 SQL-Server starten und beenden

Mittlerweile war schon mehrfach zu sehen, wie der MySQL-Server manuell gestartet

```
C:\Programme\MySQL\MySQL Server 5.0\bin>mysqld
```

bzw. beendet wird:

```
C:\Programme\MySQL\MySQL Server 5.0\bin>mysqladmin -u root –p shutdown
```

16.3.5 Benutzer anlegen oder ändern

Nun legen wir einen neuen Benutzer an mit der Berechtigung, nach Anmeldung von einem beliebigen Rechner in der Domäne **uni-trier.de** für beliebige Datenbanken die SQL-Kommandos **CREATE**, **SELECT**, **INSERT**, **DELETE**, **UPDATE**, **DROP** auszuführen. Starten Sie dazu bei aktivem MySQL-Server in einem Konsolenfenster den MySQL-Monitor als Benutzer **root**:

```
C:\Programme\MySQL\MySQL Server 5.0\bin>mysql -u root -p
```

Schicken Sie anschließend ein **GRANT**-Kommando nach folgendem Muster ab:

```
mysql> GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP
    -> ON *.*
    -> TO 'dummy'@'%.uni-trier.de'
    -> IDENTIFIED BY 'rH7%g$_c';
```

Soll der neue Benutzer berechtigt sein, sich von einem beliebigen Rechner aus beim Server anmelden, lässt man in der **TO**-Klausel die Rechnerangaben weg, z.B.:

```
    -> TO 'dummy'
```

Mit **FLUSH PRIVILEGES** können Sie dafür sorgen, dass der Server den neuen Benutzer sofort kennt, ohne neu gestartet werden zu müssen:

```
mysql> FLUSH PRIVILEGES;
mysql> exit
```

Mit dem **GRANT**-Befehl lassen sich auch die Rechte eines vorhandenen Benutzers ändern.

Über die aktuellen Rechte eines Benutzers informiert das Kommando **SHOW GRANTS**, z.B.:

```
mysql> SHOW GRANTS FOR dummy;
```

16.3.6 Benutzer löschen

Starten Sie dazu bei aktivem MySQL-Server in einem Konsolenfenster den MySQL-Monitor als Benutzer **root**:

```
C:\Programme\MySQL\MySQL Server 5.0\bin>mysql -u root -p
```

Schicken Sie anschließend ein **DELETE**-Kommando nach folgendem Muster ab:

```
mysql> DELETE FROM user WHERE user="dummy";
```

Falls dem Benutzer nur die Anmeldeberechtigung von einem bestimmten Rechner entzogen werden soll, ist folgende Variante zu benutzen:

```
mysql> DELETE FROM user WHERE user="dummy" AND host="delta.uni-trier.de";
```

Mit **FLUSH PRIVILEGES** können Sie dafür sorgen, dass der Server den neuen Benutzer sofort kennt, ohne neu gestartet werden zu müssen:

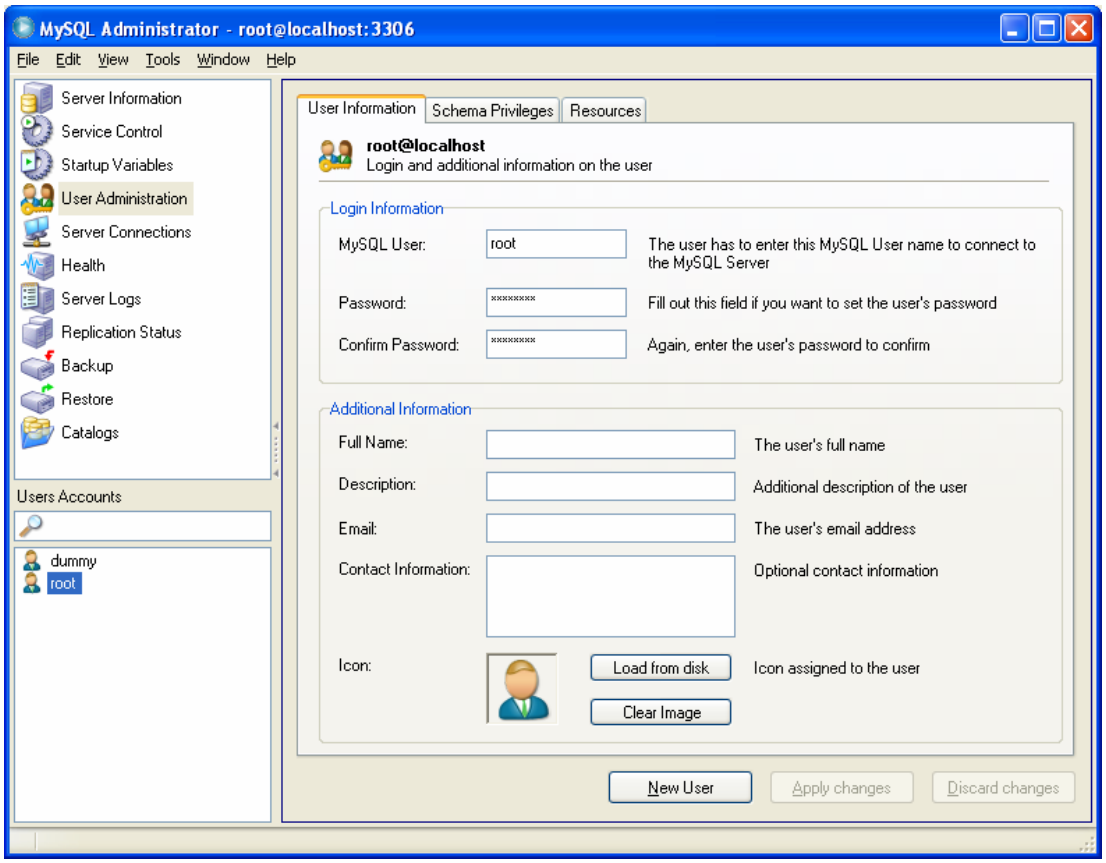
```
mysql> FLUSH PRIVILEGES;
mysql> exit
```

16.3.7 Graphische Konfiguration per MySQL Administrator

Auf der Webseite

http://dev.mysql.com/downloads/administrator/

ist ein graphisches Konfigurationswerkzeug verfügbar, das z.B. die oben beschriebene Benutzerverwaltung erleichtern kann:



Allerdings lassen sich diverse Verwaltungsarbeiten rationeller per Syntax im Stapelbetrieb ausführen.

16.3.8 Datenbank im Stapelbetrieb anlegen

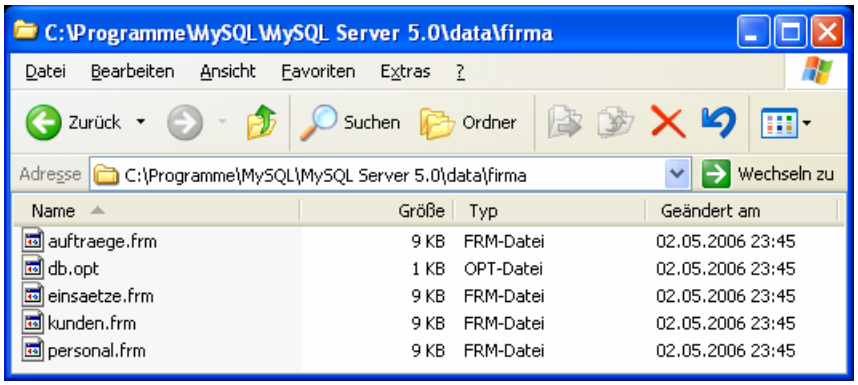
Die ca. 120 SQL-Kommandos zum Erstellen der in Abschnitt 16.1 beschriebenen Datenbank interaktiv am MySQL-Kommandoprompt einzugeben, ist nicht ratsam. Wir legen eine Textdatei mit den Kommandos an (siehe **...\BspUeb\JDBC\Firma.sql**):

```
DROP DATABASE IF EXISTS Firma;
CREATE DATABASE IF NOT EXISTS Firma;
USE Firma;
DROP TABLE IF EXISTS Personal;
DROP TABLE IF EXISTS Kunden;
DROP TABLE IF EXISTS Auftraege;
DROP TABLE IF EXISTS Einsaetze;

CREATE TABLE Personal (
  PersNr INT NOT NULL AUTO_INCREMENT,
  Vorname VARCHAR (25) NOT NULL,
  Name VARCHAR (50) NOT NULL,
  Abteilung CHAR (1) NOT NULL,
  Strasse VARCHAR (40) NOT NULL,
  Hausnummer VARCHAR (5) NOT NULL,
  PLZ CHAR (5) NOT NULL,
  Ort VARCHAR (40) NOT NULL,
  Telefon VARCHAR(20) NOT NULL,
  Gehalt DOUBLE NOT NULL,
  PRIMARY KEY (PersNr)
) TYPE=INNODB;
  . . .
  . . .
INSERT INTO Einsaetze (PersNr, AuftragsNr, Dauer) VALUES (6, 9, 6);
INSERT INTO Einsaetze (PersNr, AuftragsNr, Dauer) VALUES (6, 10,2);
```

und lassen diese bei aktivem Server im Stapelbetrieb vom MySQL-Klienten ausführen:

```
C:\Programme\MySQL\MySQL Server 5.0\bin>mysql -u root –p < Firma.sql
```

Nach der erfolgreichen Ausführung des Kommandostapels findet sich **…\mysql\data** ein Ordner mit der Beschreibung der neuen Datenbank:

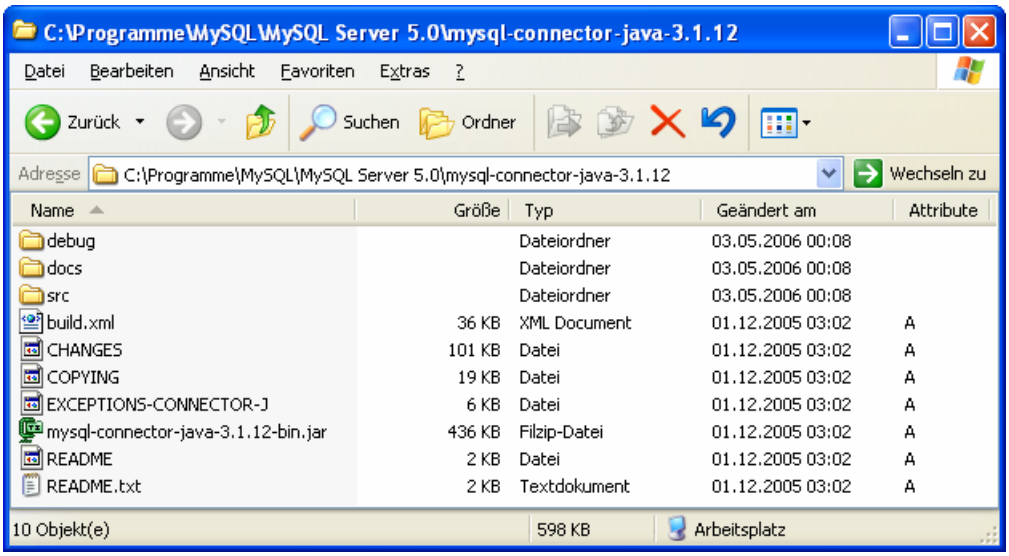


16.3.9 JDBC-Treiber installieren

Den aktuellen JDBC-Treiber zu MySQL findet man unter

http://dev.mysql.com/downloads/connector/j/

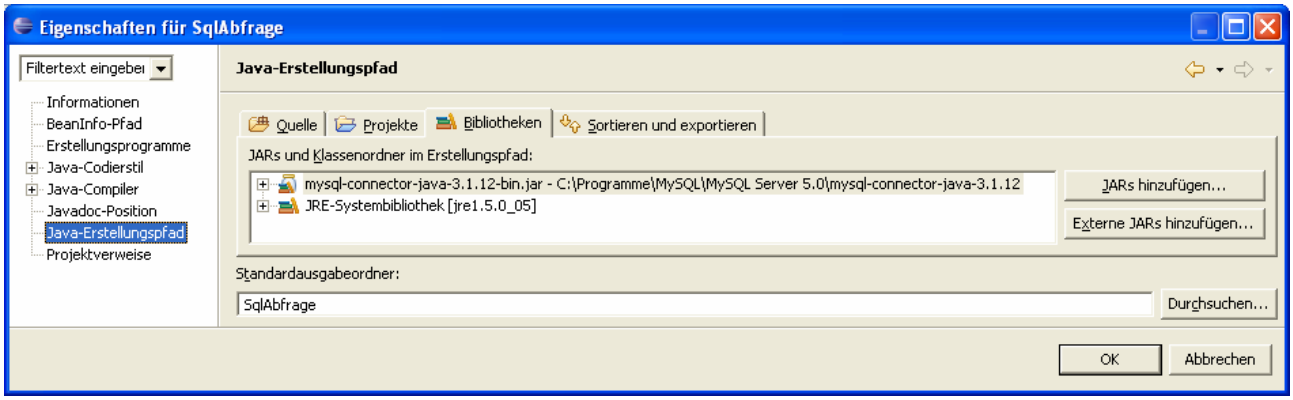
Es handelt sich um einen so genannten **Typ-4** – Treiber, der komplett in Java entwickelt ist und das datenbankspezifische Netzwerkprotokoll beherrscht. Dies erlaubt eine direkte und schnelle Verbindung zwischen Java-Programm und MySQL-Server.
Zu MySQL 5.0 passt die Version 3.1.12 des JDBC-Treibers, die zum Einsatz unter MS-Windows in der Archivdatei **mysql-connector-java-3.1.12.zip** ausgeliefert wird. Diese kann z.B. im MySQL-Installationsordner ausgepackt werden:



Wesentlich ist, dass die JAR-Datei **mysql-connector-java-3.1.12-bin.jar** mit den Klassen des Treibers bei der Programmausführung von der Java-Runtime gefunden wird. Man kann das Archiv in die CLASSPATH-Definition aufnehmen, in den Unterordner **…\lib\ext** der JRE-Installation kopieren oder beim Programmstart via **classpath**-Option bekannt machen, z.B.:

```
java -cp "C:\JDBC\mysql-connector-java-3.1.12-bin.jar;." SqlAbfrage
```

Beim Einsatz in der Entwicklungsumgebung Eclipse 3.x bietet es sich an, die JAR-Datei über den Dialog mit den Projekteigenschaften als zusätzliche Bibliothek anzugeben (Schalter **Externe JARs hinzufügen**):



16.4 SQL-Datenbanken in Java verwenden

Das folgende Programm stellt eine Verbindung zum MySQL-Server auf dem lokalen Rechner unter Verwendung des Kontos `dummy` her, und verschafft sich per **SELECT**-Anweisung aus der Tabelle `Personal` der Datenbank `Firma` die Daten der Spalten `Vorname`, `Name` und `Gehalt`:

```
import java.sql.*;

class SqlAbfrage {
   static final String DRIVER = "com.mysql.jdbc.Driver";
   static final String URL = "jdbc:mysql://localhost/firma";
   static final String USERID = "dummy";
   static final String PASSWD = "rH7%g$_c";

   public static void main(String args[]) {
      Connection verbindung = null;
      Statement abfrage = null;
      try {
         Class.forName(DRIVER);
         verbindung = DriverManager.getConnection(URL, USERID, PASSWD);
         abfrage = verbindung.createStatement();
         ResultSet daten = abfrage.executeQuery(
            "SELECT Vorname, Name, Gehalt FROM Personal");
         ResultSetMetaData metaDaten = daten.getMetaData();
         System.out.printf("%-25s %-30s  %10s\n", metaDaten.getColumnName(1),
            metaDaten.getColumnName(2), metaDaten.getColumnName(3));
         while (daten.next())
            System.out.printf("%-25s %-30s  %10.2f\n",
                 daten.getString(1), daten.getString(2), daten.getDouble(3));
      } catch (Exception e) {
         e.printStackTrace();
      } finally {
         try {
            abfrage.close();
            verbindung.close();
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
   }
}
```

Ausgabe:

```
Vorname                   Name                           Gehalt
Ulla                      Schneider                    1900,00
Otto                      Schmidt                      2490,50
Ludger                    Müller                       2260,33
Emanuel                   Fink                         1693,00
Kurt                      Schmidt                      2630,50
Monika                    Gerber                       2600,00
```

In den nächsten Abschnitten werden die wesentlichen Bestandteile des Programms erläutert.

16.4.1 Verbindung zur Datenbank herstellen

Den Aufbau der Datenbankverbindung, die im Programm durch ein Objekt der Klasse **Connection** vertreten ist, erledigt die statische Methode **getConnection()** der Klasse **DriverManager**. Sie benötigt den URL der Datenbank sowie die Kontoinformationen eines zum Zugriff berechtigten Benutzers, z.B.:

```
Connection verbindung = null;
  . . .
verbindung = DriverManager.getConnection(URL, USERID, PASSWD);
```

Ein JDBC-Datenbank-URL ist teilweise abhängig vom konkreten DBMS-Anbieter. Im Beispiel wird die Datenbank `firma` angesprochen, die ein MySQL-Server auf dem lokalen Rechner am Standardport 3306 anbietet:

```
jdbc:mysql://localhost/firma
```

Damit der **DriverManager** den JDBC-Treiber, der das geforderte Protokoll **jdbc:mysql** beherrscht, auch findet, muss dieser registriert werden, was z.B. mit der statischen Methode **Class.forName()** geschehen kann:

```
Class.forName("com.mysql.jdbc.Driver");
```

Wie dafür gesorgt werden kann, dass die JRE das Paket mit den Treiber-Klassen findet, wurde schon in Abschnitt 16.3.9 erläutert.

Statt die Angaben für den Datenbankzugriff (Treiber, URL, Konto) im Quellcode unterzubringen, sollte man sie bei ernsthaften Anwendungen der größeren Flexibilität halber mit Hilfe der Klasse **Properties** aus einer Eigenschaftsdatei lesen (siehe z.B. Hostmann & Cornell 2002, S. 279f).

16.4.2 SQL-Kommando ausführen

Um ein SQL-Kommando ausführen zu können, benötigen wir ein Objekt vom Typ **Statement**, das wir von **Connection**-Objekt per **createStatement()** erstellen lassen:

```
Statement abfrage = null;
  . . .
abfrage = verbindung.createStatement();
```

Abhängig vom konkreten SQL-Kommando sind unterschiedliche **Statement**-Methoden zu verwenden, denen jeweils ein **String** übergeben wird:

- **executeUpdate()**
  Diese Methode ist zuständig für Datenbankmodifikationen über die SQL-Kommandos **UPDATE**, **INSERT**, **DELETE**, **CREATE TABLE**, **DROP TABLE** etc. Als Rückgabe erhält man die Anzahl der vom Kommando tangierten Zeilen.
- **executeQuery()**
  Diese Methode ist zuständig für Datenbankabfragen per **SELECT**-Befehl. Als Rückgabe erhält man ein Objekt vom Typ **ResultSet**.

Im Beispiel wird eine Abfrage durchgeführt:

```
ResultSet daten = abfrage.executeQuery(
  "SELECT Vorname, Name, Gehalt FROM Personal");
```

Das **ResultSet**-Objekt enthält eine Ergebnistabelle und verwaltet einen Zeiger auf die aktuelle Zeile, der per **next()**-Methode inkrementiert werden kann, z.B.:

```
while (daten.next())
  System.out.printf("%-25s %-30s  %10.2f\n",
    daten.getString(1), daten.getString(2), daten.getDouble(3));
```

Für die aktuelle Zeile kann man über datentypspezifische Zugriffsmethoden die Werte der Spalten ermitteln, wobei unbedingt die bei **1** beginnende Indizierung zu beachten ist. Bei Hostmann & Cornell (2002, S. 285) findet sich eine Liste mit den wichtigsten SQL-Datentypen und ihren Java-Entsprechungen.

Beim **ResultSet** ist auch ein **ResultSetMetaData**–Objekt erhältlich:

```
ResultSetMetaData metaDaten = daten.getMetaData();
```

Mit seiner **getColumnName()**-Methode informiert dieses Objekt z.B. über die Namen der Spalten im **ResultSet** (wiederum indiziert ab 1):

```
System.out.printf("%-25s %-30s  %10s\n", metaDaten.getColumnName(1),
```

```
    metaDaten.getColumnName(2), metaDaten.getColumnName(3));
```

Es wird empfohlen, für **Statement**- und **Connection**-Objekte möglichst früh die **close()**-Methode aufzurufen, um damit verbundene Datenbank- und JDBC-Ressourcen freizugeben. Im Beispiel wird diese Routine eingeübt, obwohl die **close()**-Aufrufe unmittelbar vor dem Programmende (mit automatischer Freigabe aller Ressourcen) überflüssig sind.

16.4.3 Anzeige und Modifikation von Tabellen per JTable-Komponente

Das bereits in Abschnitt 16.2 zu bewundernde Frontend für die Beispieldatenbank

PersNr	Vorname	Name	Abteilung	Strasse	Hausnummer	PLZ	Ort	Telefon	Gehalt
1	Ulla	Schneider	B	Heide	29	51594	Raltop	05123-12094	1900.0
2	Otto	Schmidt	A	Goethegasse	12b	52292	Drohntal	07319-78129	3422.50
3	Ludger	Müller	F	Hauptstraße	45	53277	Urdorf	04543-32212	2260.33
4	Emanuel	Fink	B	An der Ecke	177	54822	Mondorf	03423-73212	1800.0
5	Kurt	Schmidt	A	Südstrasse	23	52292	Drohntal	07319-53487	2630.5
6	Monika	Gerber	C	Mühlenstraße	91	51594	Raltop	05123-78123	2500.0

erlaubt nicht nur eine Anzeige, sondern auch eine Veränderung der Tabellen.

Hier dient ein Objekt der Klasse **JTable** aus dem Paket **javax.swing** als Ansicht (*View*) für die von einem *Model*-Objekt verwaltete Tabelle:

```
private JTable tabelle;
   . . .
tabMod = new ResultSetTableModel(DRIVER, URL, USERID, PASSWD, TABLES[0]);
tabelle = new JTable(tabMod);
```

Während das **JTable**-Objekt mit seiner beachtlichen Funktionsvielfalt sofort einsatzfähig ist, benötigen wir für das Model-Objekt eine eigene Klasse, welche das Interface **TableModel** erfüllt. Hier sind etliche Methode gefordert, die ein **JTable**-Objekt z.B. benutzt, um …

- sich die anzuzeigenden Daten zu beschaffen
- die vom Benutzer an Tabellenzellen vorgenommenen Änderungen an das Model-Objekt zu melden.

Um nur die für uns relevanten **TableModel**-Methoden implementieren zu müssen, leiten wir unsere Model-Klasse `ResultSetTableModel` von **AbstractTableModel** ab:

```
public class ResultSetTableModel extends AbstractTableModel {
   private Connection verbindung;
   private boolean verbunden;
   private Statement selectAnweisung;
   private ResultSet abfrageErgebnis;
   private ResultSetMetaData metaData;
   private int nZeilen;

   public ResultSetTableModel(String treiber, String url, String userid,
      String passwd, String tabelle) throws SQLException, ClassNotFoundException {
      Class.forName(treiber);
      verbindung = DriverManager.getConnection(url, userid, passwd);
      verbunden = true;
      selectAnweisung = verbindung.createStatement(
         ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
      neueAbfrage(tabelle);
   }
   . . .
   . . .
}
```

Unser Model-Objekt dient als Verbindung zwischen dem **JTable**-Objekt im Vordergrund und einer Datenbank im Hintergrund. Wie im obigen Beispiel wird ein **Statement**-Objekt erstellt, das mit seiner **executeQuery()**–Methode **ResultSet**-Objekte beschafft. Weil aber mehr Flexibilität bei der Datenbankbearbeitung via **ResultSet** möglich sein soll, kommen bei der Konstruktion des **Statement**-Objekts zwei neue Parameter zum Einsatz:

- Im **createStatement()**-Parameter **resultSetType** wird festgelegt, welche Bewegungsmöglichkeiten und Dynamik die **ResultSet**-Objekte bieten sollen, wobei der geforderte **int**-Wert mit Hilfe von **ResultSet**-Konstanten geliefert werden sollte:
  - **ResultSet.TYPE_FORWARD_ONLY**
    Der Zeilenzeiger kann nur vorwärts bewegt werden.
  - **ResultSet.TYPE_SCROLL_INSENSITIVE**
    Der Zeilenzeiger kann vorwärts und rückwärts bewegt werden, jedoch reagiert das **ResultSet** dabei *nicht* auf zwischenzeitlich von anderen Benutzern vorgenommene Änderungen der Datenbank.
  - **ResultSet.TYPE_SCROLL_SENSITIVE**
    Der Zeilenzeiger kann vorwärts und rückwärts bewegt werden, und das **ResultSet** berücksichtigt die zwischenzeitlich von anderen Benutzern vorgenommenen Änderungen der Datenbank.
- Im **createStatement()**-Parameter **resultSetConcurrency** wird festgelegt, ob **ResultSet**-Objekte auch eine *Änderung* der Datenbank erlauben sollen, wobei der geforderte **int**-Wert mit Hilfe von **ResultSet**-Konstanten geliefert werden sollte:
  - **ResultSet.CONCUR_READ_ONLY**
    Nur lesender Zugriff
  - **ResultSet.CONCUR_UPDATABLE**
    Datenbank-Updates möglich

Im Beispiel sorgen wir dafür, dass die **ResultSet**-Objekte beliebige Vertikalbewegungen ohne Berücksichtigung von eventuellen zwischenzeitlichen Datenbankveränderungen erlauben und eine Modifikation der Datenbank unterstützen.

Nach dieser wichtigen Erweiterung der Datenbankzugriffstechnik kommen nun die **TableModel**-Methoden zur Sprache, die unsere Klasse `ResultSetTableModel` implementieren muss, damit ihre Objekte als Model zu einem **JTable**-View agieren können:

```
public int getRowCount() {
  if (!verbunden)
    return -1;
  else
    return nZeilen;
}

public int getColumnCount() {
  if (!verbunden)
    return 0;
  try {
    return metaData.getColumnCount();
  } catch (SQLException e) {
    return 0;
  }
}
```

```
public Object getValueAt(int row, int col) {
  if (!verbunden)
    return null;
  try {
    abfrageErgebnis.absolute(row+1);
     return abfrageErgebnis.getObject(col+1);
  } catch (SQLException e) {
    return null;
  }
}

public void setValueAt(Object obj, int row, int col) {
  if (!verbunden)
    return;
  try {
    abfrageErgebnis.absolute(row+1);
    abfrageErgebnis.updateObject(col+1, obj);
    abfrageErgebnis.updateRow();
  } catch (SQLException e) {
    e.printStackTrace();
    return;
  }
}

public boolean isCellEditable(int row, int col) {
  return true;
}
```

Über die Methoden

- **getRowCout()**
- **getColumnCount()**
- **getValueAt()**

beschafft sich ein **JTable**-Objekt die darzustellenden Daten. Wenn bei den Spaltennamen anstelle von Buchstaben informative Bezeichnungen gewünscht sind, muss auch die Methode **getColumnName()** implementiert werden, wobei sich in unserem Fall ein Objekt der Klasse **ResultSetMetaData** nützlich macht:

```
public String getColumnName(int col) {
  if (!verbunden)
    return "";
  try {
    return metaData.getColumnName(col+1);
  } catch (SQLException e) {
    return "";
  }
}
```

Zur Änderung der Datenbank via JTable-Objekt dienen die folgenden Methoden:

- **isCellEditable()**
  Wir erlauben der Einfachheit halber bei allen Zellen eine Änderung, wohl wissend, dass der SQL-Server bei Referenzen zwischen Datenbanktabellen manche Änderungen ablehnen wird. Hier zeigt sich die mangelhafte Einbindung der relationalen Datenbanktechnik in das objektorientierte Programmdesign.
- **setValueAt()**
  An diese Methode übergibt das **JTable**-Objekt einen vom Benutzer geänderten Zelleninhalt. Damit die Änderungen an der aktuellen **ResultSet**-Zeile auch in die angebundene Datenbank übertragen werden, ist noch ein Aufruf der **ResultSet**-Methode **updateRow()** fällig.

Ein Update-fähiges **ResutSet**-Objekt bietet außerdem die Methoden **insertRow()** und **deleteRow()**.

Die Klasse `ResultSetTableModel` erstellt zu einer beliebigen relationalen Datenbank ein **TableModel**–Objekt und ist damit wiederverwendbar. Um auch die folgende Klasse zur Realisation der Benutzeroberfläche Recycling-fähig zu machen, müssen nur die Konstanten durch Informationen aus einer Parameterdatei ersetzt werden:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class DbFrontend extends JFrame {
   static final String TITEL = "Tabellen der Firmendatenbank";
   static final String DRIVER = "com.mysql.jdbc.Driver";
   static final String URL = "jdbc:mysql://localhost/firma";
   static final String USERID= "dummy";
   static final String PASSWD= "dummy";
   static final String[] TABLES = {"Personal", "Kunden", "Auftraege", "Einsaetze"};

   private ResultSetTableModel tabMod;
   private JComboBox liste;
   private JTable tabelle;

   public DbFrontend() {
      super(TITEL);
      try {
         liste = new JComboBox(TABLES);
         liste.setEditable(false);
         liste.addItemListener(
               new ItemListener() {
                  public void itemStateChanged(ItemEvent e) {
                     JComboBox cb = (JComboBox) e.getSource();
                     tabMod.neueAbfrage(cb.getSelectedItem().toString());
                  }
               });

         tabMod = new ResultSetTableModel(DRIVER, URL, USERID, PASSWD, TABLES[0]);
         tabelle = new JTable(tabMod);

         add(liste, BorderLayout.NORTH );
         add(new JScrollPane(tabelle), BorderLayout.CENTER);

         setSize(700, 200);
         setVisible( true );
         setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
         addWindowListener(new Ex());
      } catch (Exception e) {
         JOptionPane.showMessageDialog(this, e);
         System.exit(1);
      }
   }

   public static void main(String args[]) {
      new DbFrontend();
   }

   private class Ex extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
         tabMod.trennen();
         System.exit(0);
      }
   }
}
```

Den vollständigen Quellcode zu den Klassen `ResultSetTableModel` und `DbFrontend` finden Sie im Ordner

…\BspUeb\JDBC\DNS\ResultTable

16.5 Übungsaufgaben zu Abschnitt 16

1) Installieren Sie MySQL 5.0 und den zugehörigen JDBC-Treiber gemäß Beschreibung in Abschnitt 16.3. Starten Sie den MySQL-Server, und erzeugen Sie die Beispieldatenbank `Firma` mit dem SQL-Skript in Abschnitt 16.3.8.
Erstellen Sie ein Java-Programm, das per **UPDATE**–Kommando das Gehalt von Emanuel Fink auf 1800 € erhöht.

17 Serverseitige Web-Anwendungen

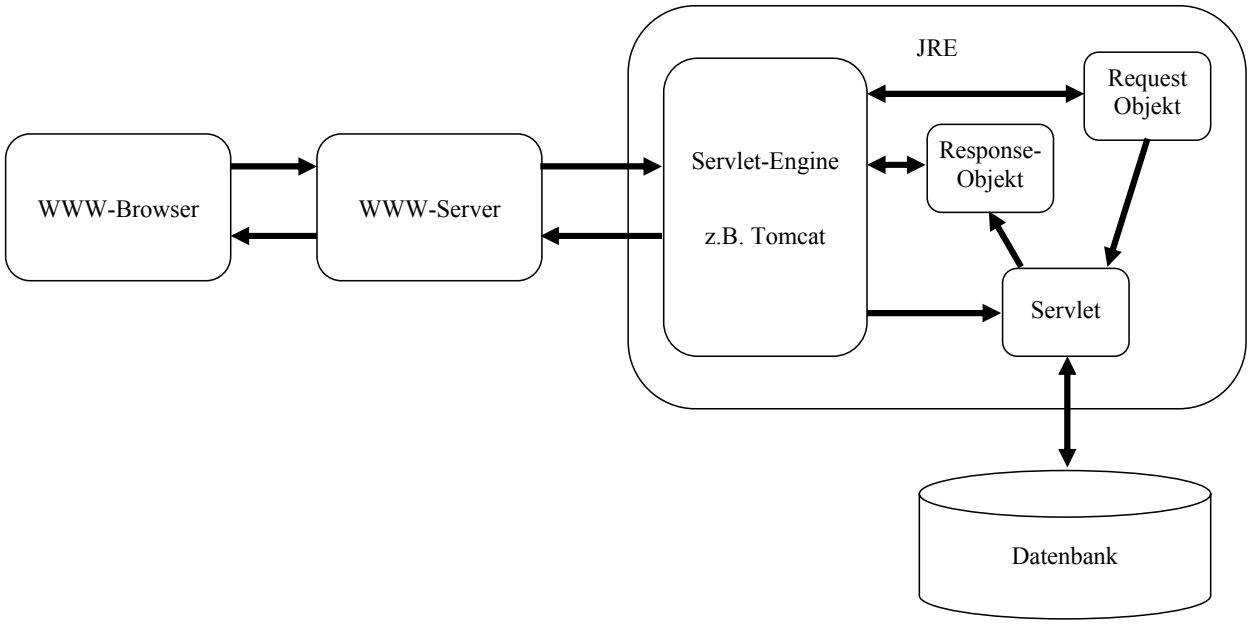
Dieser Abschnitt bietet eine Einführung in serverseitige Web-Anwendungen mit Java und behandelt die Servlets sowie die darauf aufbauenden Java Server Pages (JSP). Hier erfahren (Web)server eine Funktionserweiterung durch Java-Software, wobei die gesamte Mächtigkeit und die Sicherheitsmerkmale der Java-Bibliothek zur Verfügung stehen.

Servlets und JSPs bilden zusammen mit den *Java Server Faces* (JSF) die Web-Schicht der als *Java 2 Enterprise Edition* (J2EE) bezeichneten Java-Technologie für komplexe und verteilte Anwendungen (siehe Ball et al 2006). Wir verlassen also erstmals den Bereich der *Java2 Standard Edition* (J2SE), wobei aber das erworbene Wissen uneingeschränkt gültig bleibt. Weitere J2EE-Bestandteile, die in diesem Manuskript nicht behandelt werden, sind z.B. die *Enterprise Java Beans* (EJB) und die bereits angesprochenen Web Services.

J2EE-Software läuft im Rahmen einer Serverumgebung ab, die je nach J2EE-Anwendungsbreite als *Servlet- und JSP-Container* (z.B. Tomcat) oder als *Applikations-Server* (z.B. WebLogic von BEA, WebSphere von IBM) bezeichnet wird.

Bei einem Servlet handelt es sich um eine Java-Klasse, die in einem bestimmten Framework agieren soll und daher bestimmte Interfaces aus den J2EE-Paketen **java.servlet** bzw. **java.servlet.http** zu erfüllen hat. Es sind also Methoden (z.B. **getPost()**) zu implementieren, die vom J2EE-Server (meist im Auftrag eines WWW-Servers) aufgerufen werden. Jede JSP wird vom J2EE-Server in ein Servlet transformiert.

In der folgenden Abbildung ist die Kooperation zwischen einem WWW-Browser (Klientenanwendung), einem WWW-Server, einer Servlet-Engine und einem Servlet mit Datenbankanbindung vereinfacht dargestellt:



17.1 Tomcat 5.5 unter MS Windows installieren und einrichten

Wir werden die von Sun Microsystems im *Java Community Process*gemeinsam mit der *Apache Software Foundation* unter dem Namen *Tomcat* komplett in Java entwickelte Referenz-Implementation eines Servlet/JSP - Containers einsetzen. Aufgrund der integrierten Webserver-

Funktionalität kann Tomcat als eigenständige Serverlösung oder als Servlet/JSP – Engine für einen vorhandenen Webserver (z.B. Apache, IIS) agieren.
Im Lernumfeld ist die integrierte Lösung angenehm, für belastete Produktionssysteme eignet sich der integrierte Webserver jedoch nicht. Wie man Tomcat und einen separaten Webserver zusammen bringt, ist der Dokumentation zu entnehmen.

Tomcat ist über folgende Webseite frei verfügbar:

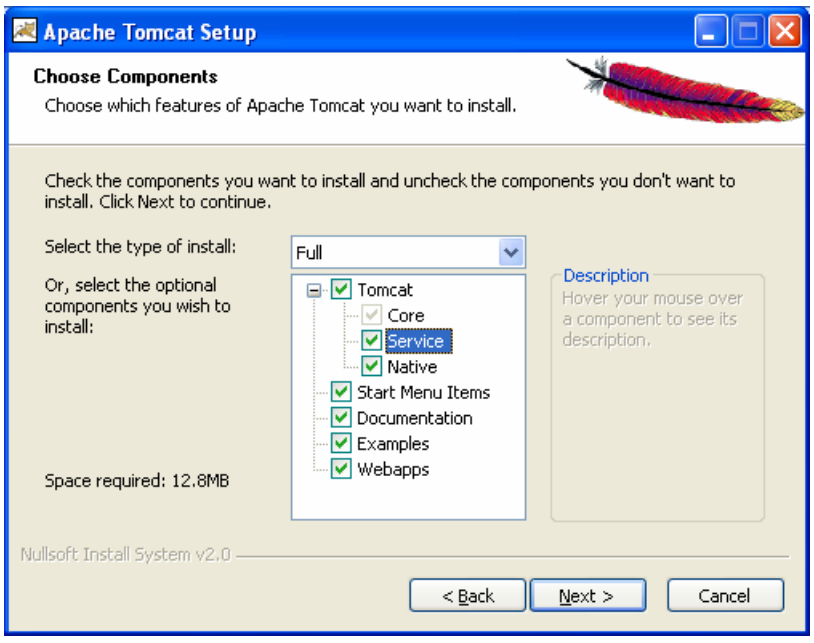
http://tomcat.apache.org/

Windows-Benutzer haben bei der aktuellen Version 5.5.17 die Wahl zwischen einem ZIP-Archiv und einem Installationsprogramm.

17.1.1 Installation

Unter MS-Windows kann die Installation via **apache-tomcat-5.5.17.exe** gestartet werden:

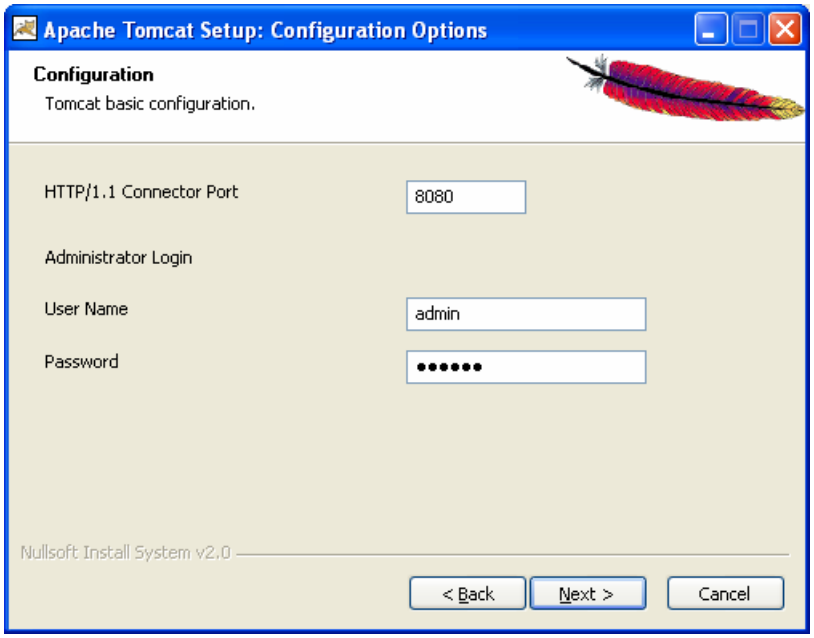
Bei einer Installation zu Lernzwecken sollte man die Zusatzoptionen **Examples** und **Webapps** wählen, beim geplanten Einsatz als Produktionsserver sind der Start als Dienst und die Einbindung nativer Software sinnvoll:



Per Voreinstellung landet die Installation im Ordner:

C:\Programme\Apache Software Foundation\Tomcat 5.5

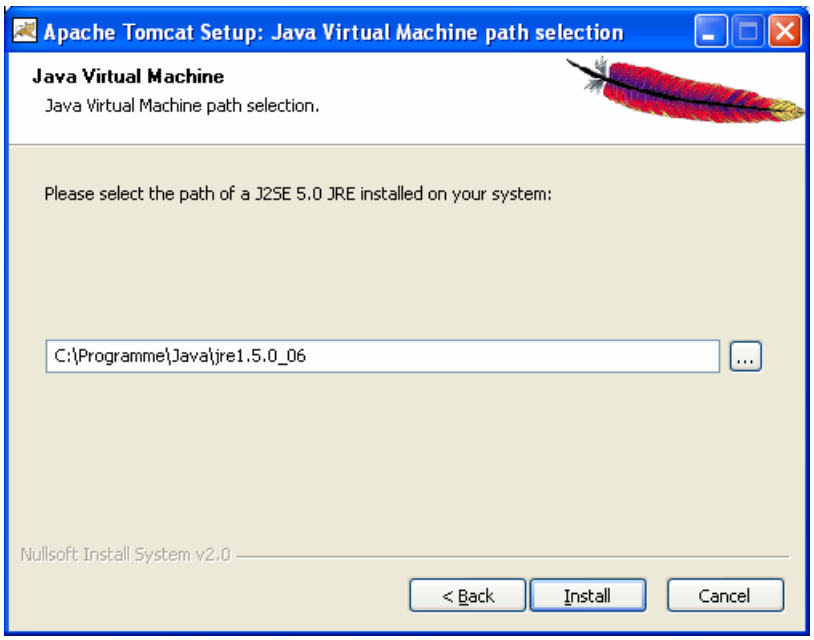
Im Konfigurationsdialog wählt man den Port, an dem der Server lauschen soll (Voreinstellung 8080) und die Anmeldedaten des Tomcat-Administrators:



Man findet die Benutzerdaten später im Klartexte (samt Passwort!) in der Datei:

C:\Programme\Apache Software Foundation\Tomcat 5.5\conf\tomcat-users.xml

Abschließend erkundigt sich das Installationsprogramm nach dem JRE-Standort:

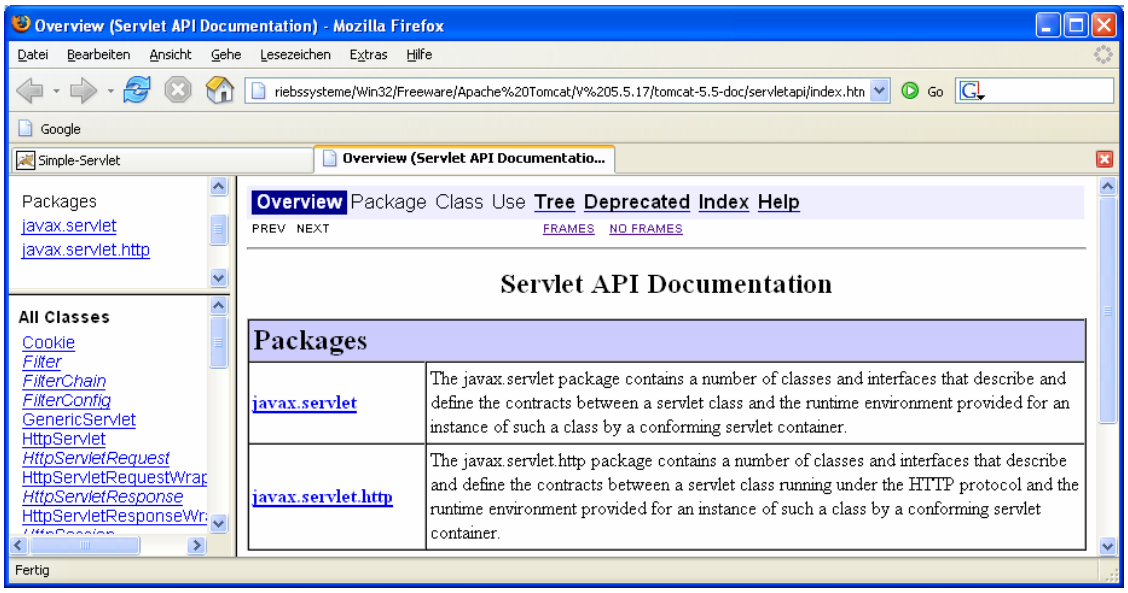


Weil Tomcat 5.5. einen eigenen Compiler (zum Übersetzen der aus JSPs erstellten Servlets) mitbringt, ist kein JDK erforderlich.

Im Ordner

C:\Programme\Apache Software Foundation\Tomcat 5.5\webapps\tomcat-docs

landet bei der Installation ein umfangreiches Dokumentationspaket, zugänglich über die HTML-Startseite **index.html**. Hier finden sich u.a. die unverzichtbaren Javadoc-HTML-Seiten zum Servlet- und zum JSP-API:

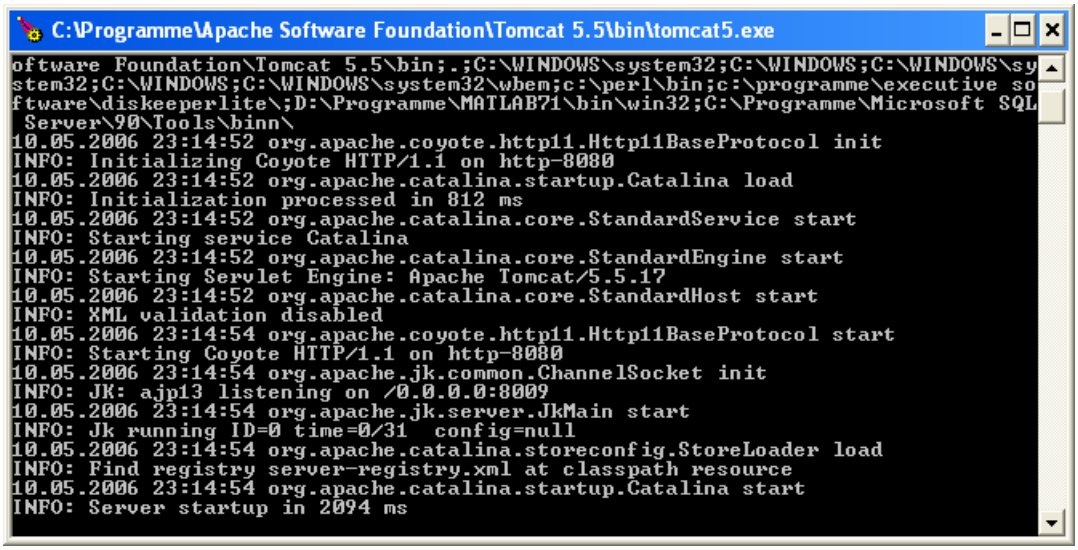


17.1.2 Tomcat-Server manuell starten und beenden

Unter MS-Windows dient zum manuellen Starten des Tomcat-Servers das Programm

C:\Programme\Apache Software Foundation\Tomcat 5.5\bin\tomcat5.exe

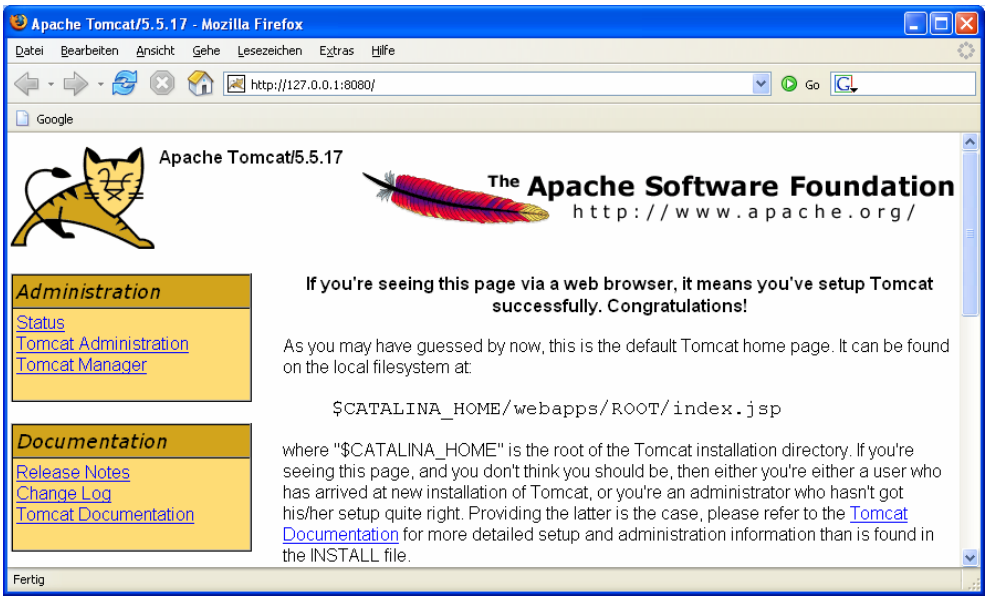
Es erscheint ein Konsolenfenster, das Statusmeldungen anzeigt und den Tastenbefehl **Strg+C** zum Beenden des Servers entgegen nimmt:



Einen ersten Funktionstest ermöglicht die Browser-Anfrage mit dem URL

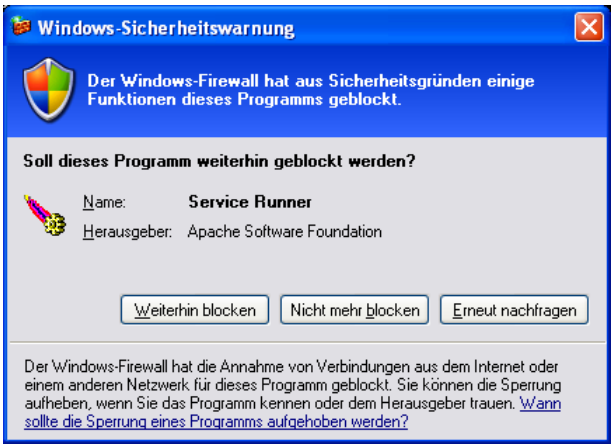
http://127.0.0.1:8080/

in der Adresszeile:



17.1.3 Ausnahme für die Windows-Firewall

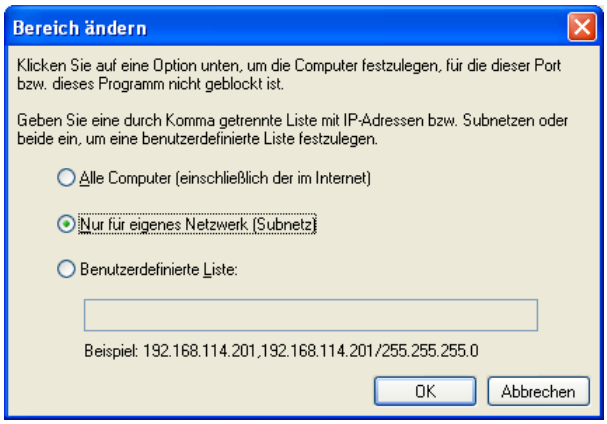
Beim ersten Tomcat-Start erkundigt sich Windows XP mit SP2, ob das Programm als Server fungieren darf:



Nach einem Mausklick auf **Nicht mehr blocken** wird eine Firewall-Ausnahme angelegt, die nach

Systemsteuerung > Windows-Firewall

nachjustiert werden sollte. Es bietet sich an, die Liste der zugriffsberechtigten Systeme einzuschränken, z.B.:



17.1.4 Bibliothek mit dem Servlet-API einrichten

Die zum Erstellen von Servlets erforderlichen Klassen und Interfaces befindet sich im Archiv:

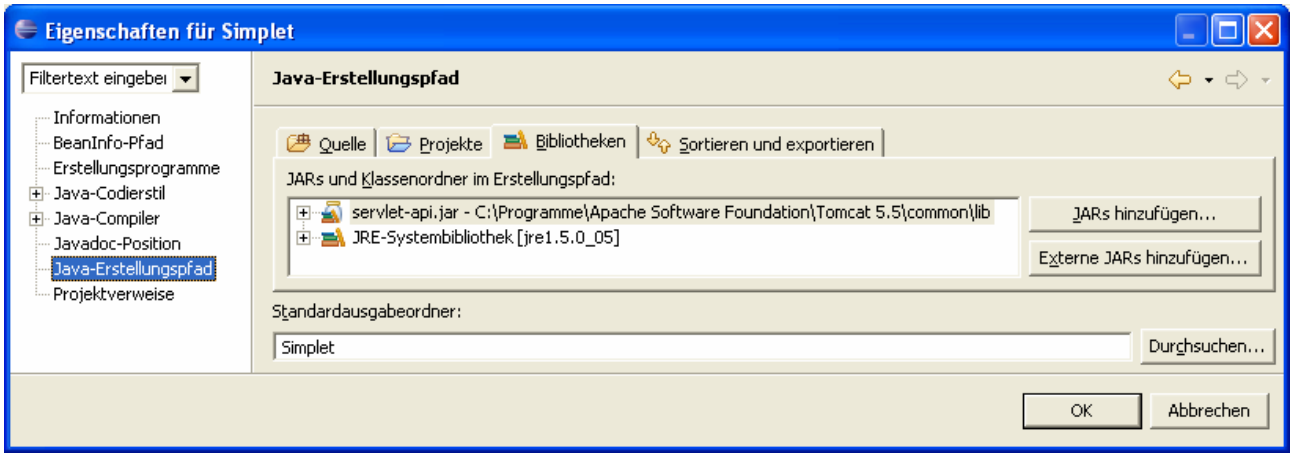
…\Apache Software Foundation\Tomcat 5.5\common\lib\servlet-api.jar

Damit es vom Compiler und von der Java-Runtime gefunden wird, kann man das Archiv in die CLASSPATH-Definition aufnehmen oder in den Unterordner

…\lib\ext

der JRE-Installation kopieren.

Beim Einsatz in der Entwicklungsumgebung Eclipse 3.x bietet es sich an, die JAR-Datei über den Dialog mit den Projekteigenschaften als zusätzliche Bibliothek anzugeben (Schalter **Externe JARs hinzufügen**):



17.2 Servlet-Framework

Ein Servlet hat das gleichnamige Interface aus dem Paket **javax.servlet** zu implementieren, was im WWW-Umfeld meist durch Verwendung das Basisklasse **HttpServlet** aus dem Paket **javax.servlet.http** geschieht. Wir beschäftigen uns ausschließlich mit Servlets, die für eine Kommunikation via HTTP-Protokoll konstruiert sind.

An ein Servlet gerichtete HTTP-Anforderungen werden vom Webserver angenommen und an die Servlet-Engine weitergeleitet. Diese erzeugt bei Bedarf (bei der ersten Anforderung) ein Servlet-Objekt und ruft zunächst dessen **init()**-Methode auf. Beim Servlet-Design kann man diese Methode überschreiben, um Initialisierungen vorzunehmen, z.B. Datenbankverbindungen aufzubauen.

Anschließend erhält das Servlet seinen Auftrag über einen Aufruf seiner **service()**-Methode. In der Klasse **HttpServlet** ist die **service()**-Methode geeignet implementiert und ruft in Abhängigkeit vom HTTP-Anforderungstyp die Methode **doGet()** oder die Methode **doPost()** auf. Es werden auf jeden Fall via Parameter übergeben:

- ein **HttpServletRequest**-Objekt zur Beschreibung der Anforderung
  Über seine **getParameter()**-Methode ermittelt man die vom Browser abgeschickten Parameterwerte.
- ein **HttpServletResponse**-Objekt zum Transport der Antwort
  Es bietet dem Servlet über seine Methode **getWriter()** ein **PrintWriter**-Objekt zum Versandt der HTML-Ergebnisseite.

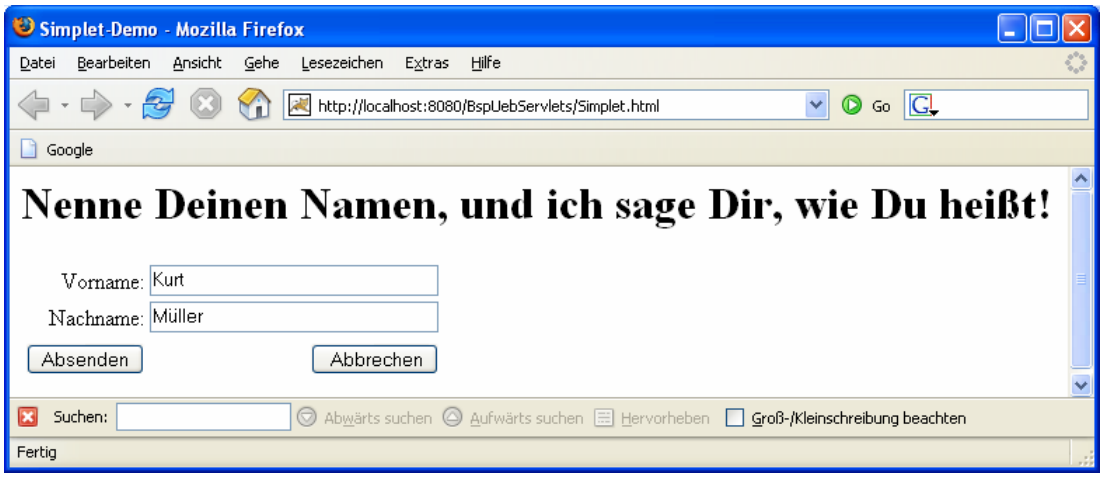
Im folgenden Beispiel

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Simplet extends HttpServlet {
   protected void doPost(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {
      response.setContentType("text/html" );
      request.setCharacterEncoding("UTF-8");
      response.setCharacterEncoding("UTF-8");
      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Servlet mit Parameterübergabe per Post</title>");
      out.println("</head>");
      out.println("<body>");
      out.println( "<h1>Hallo, "+request.getParameter("vorname")+" "+
         request.getParameter("nachname")+"!</h1>");
      out.println("</body>");
      out.println("</html>");
      out.close();
   }
}
```

wird durch Anweisungen an das Request- und das Response-Objekt (nach entsprechend schlechten Erfahrungen) dafür gesorgt, dass die UTF-8 - Kodierung zum Einsatz kommt.
In der zugehörigen HTML-Seite

```
<html>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<head>
<title>Simplet-Demo</title>
</head>
<h1>Nenne Deinen Namen, und ich sage Dir, wie Du hei&szlig;t!</h1>
<form method="post" action="/BspUebServlets/hallo">
<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>
```

wird die UTF-8 – Kodierung per Meta-Tag erzwungen.
Im Vergleich zum Abschnitt 15.2.5 über CGI-basierte Interaktivität ist an folgendem Bildschirmphoto nur die Adresszeile des Browsers neu, wo der Tomcat-Server an Port 8080 angesprochen wird:



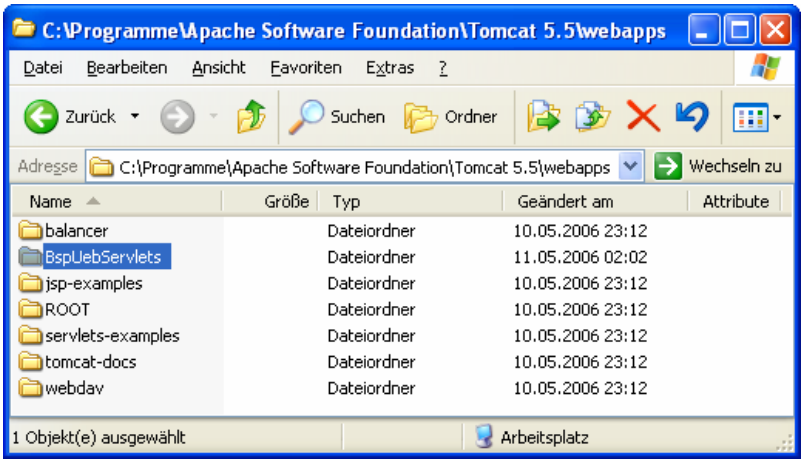
Warum die folgende **action**-Angabe im **form**-Tag der HTML-Seite

```
action="/BspUebServlets/hallo"
```

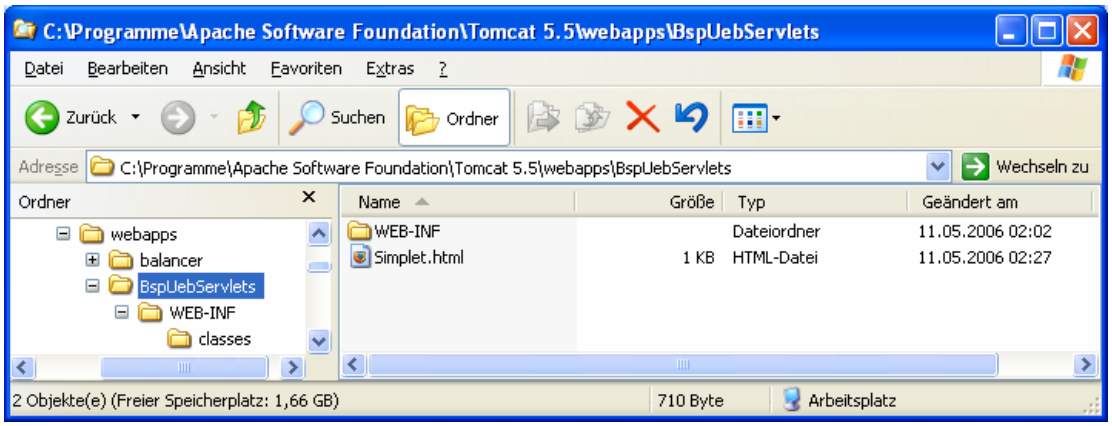
das oben vorgestellte Servlet zum Einsatz bringt, erfahren Sie gleich.

17.3 Servlets installieren

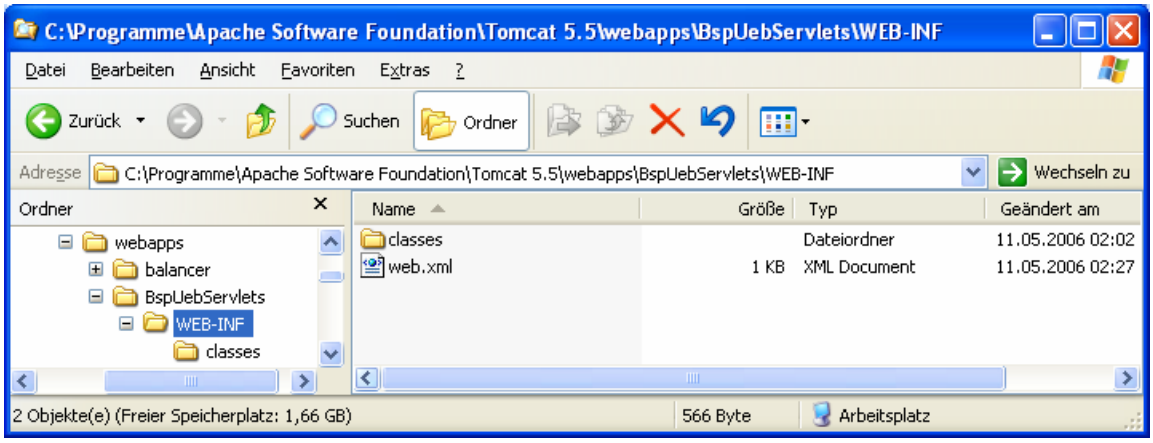
Ein Servlet wird wie jede andere Klasse in eine Bytecode-Datei übersetzt. Um im Rahmen eines Servlet-Containers nutzbar zu werden, muss diese Bytecode-Datei als Bestandteil einer Web-Applikation zusammen mit begleitenden Dateien im **webapps**-Verzeichnis des Servers untergebracht werden.
Für jede Web-Applikation, die potentiell aus mehreren Servlets bestehen kann, wird ein Ordner angelegt, im Beispiel soll er **BspUebServlets** heißen:



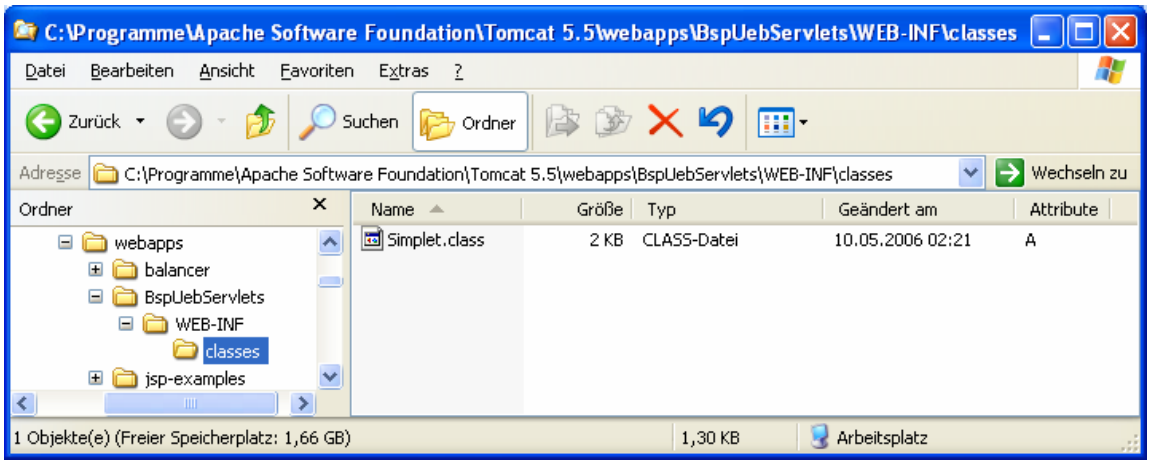
Auf der Wurzelebene des Applikationsordners (*context root* genannt) können z.B. HTML-Dateien untergebracht werden, die den Kunden das Dienstleistungsangebot erschließen:



Essentieller Bestandteil der Wurzelebene ist der Ordner **WEB-INF** mit der Datei **web.xml** zur Beschreibung der Web-Applikation:



Ebenfalls im **WEB-INF** enthalten sind die Ordner **classes** für die Bytecode-Dateien der Servlets sowie ggf. der Ordner **lib** mit Bibliothekspaketen:



In der Datei **web.xml** erwartet die Servlet-Engine alle erforderlichen Informationen, um die Anforderungen der Klienten an die zuständigen Servlets weiterleiten zu können, z.B.:

```
<web-app>

   <display-name>
      URT-Java-Kurs Servlet-Beispiele
   </display-name>

   <description>
      Servlet-Beispiele zum Java-Kurs des URT
   </description>

   <servlet>
      <servlet-name>Intro</servlet-name>
      <description>
         Ein einfaches Servlet mit Parameteruebergabe per Post
      </description>
      <servlet-class>
         Simplet
      </servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>Intro</servlet-name>
      <url-pattern>/hallo</url-pattern>
   </servlet-mapping>

</web-app>
```

Hier erhält zunächst die gesamte Web-Applikation einen Namen und eine Beschreibung.
Dann wird für jedes Servlet deklariert:

- Name
  Um im Beispiel die volle Benennungsflexibilität zu demonstrieren, verwenden wir *nicht* den Klassennamen.
- Beschreibung
- die implementierende Bytecode-Datei

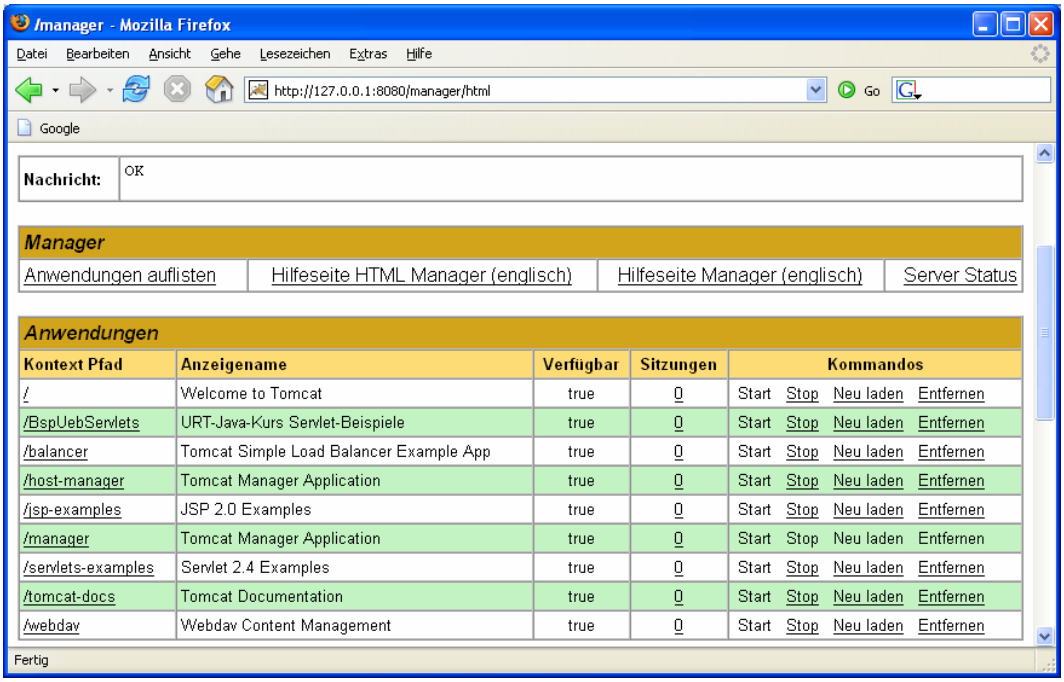
Schließlich werden URLs auf die Servlet-Namen abgebildet. Im Beispiel erhält das Servlet `Intro`, implementiert durch die Bytecode-Datei **Simplet.class**, die Zuständigkeit für den URL

```
...\BspUebServlets\hallo
```

Die Servlet-Engine versucht beim Start, alle im **webapps**-Verzeichnis angetroffenen Applikationen zu laden. Inwiefern dies gelungen ist, erfährt man beim Tomcat-Server z.B. über die **Manager-Applikation**, die per WWW-Browser über den URL

http://127.0.0.1:8080/manager/html

ansprechbar und dem Administrator vorbehalten ist (vgl. Abschnitt 17.1.1).
Im Beispiel ist zu erfahren, dass alle Applikationen verfügbar sind:



Über diese Webseite kann man z.B. Applikationen stoppen, starten, neu laden oder entfernen. Tomcat 5.5 reagiert zudem dynamisch darauf, wenn Applikationen in seinem **webapps**-Ordner auftauchen oder von dort verschwinden.

17.4 JSP

In diesem Abschnitt kann nur ein erster Eindruck von den Java Server Pages vermittelt werden. Ausführliche Informationen finden sich z.B. bei Ball et al (2006), Deitel & Deitel (2005, S. 1280ff) und Rittmeyer (2005).
Das Servlet aus Abschnitt 17.2 lässt sich durch die folgende JSP ersetzen, z.B. untergebracht in der Datei **Intro.jsp**:

```
<html>
<head>
<title>Einfache JSP mit Parameterübergabe per Post</title>
</head>
<body>
<%
   request.setCharacterEncoding("UTF-8");
   response.setCharacterEncoding("UTF-8");
%>
<h1>Hallo, <%= request.getParameter("vorname") %> <%=
request.getParameter("nachname") %>!</h1>
</body>
</html>
```

Hier erscheinen so genannte **Scripting-Tags** zwischen statischen HTML-Bestandteilen. Es sind zu unterscheiden:

- **Ausdrücke** (*Expressions*)
  Zwischen **<%=** und **%>** wird ein Java-Ausdruck geschrieben, dessen Wert auf der Antwortseite erscheinen soll, z.B.:
  ```
  <%= request.getParameter("vorname") %>
  ```
- **Scriptlets**
  Zwischen **<%** und **%>** können beliebige Java-Anweisungen geschrieben werden, sofern der JSP-Engine die benötigten Bibliotheken bekannt sind, z.B.:
  ```
  <%
    request.setCharacterEncoding("UTF-8");
    response.setCharacterEncoding("UTF-8");
  %>
  ```

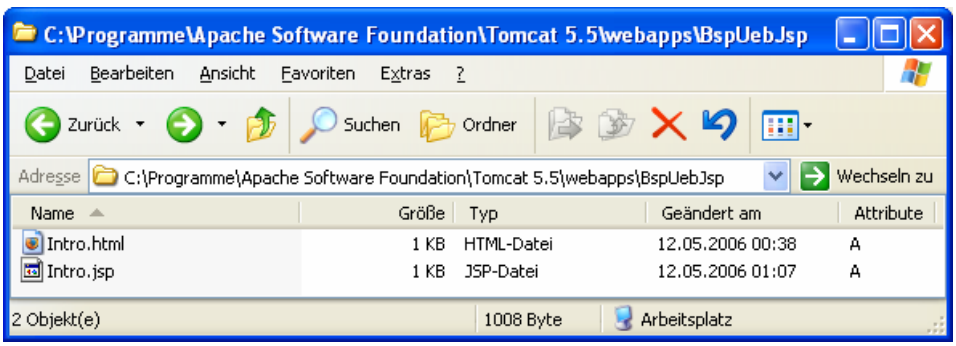
Über die **impliziten Objekte request** und **response** kann die JSP wie die **Servlet**-Methode **doPost()** auf die Anforderungen des Klienten zugreifen und Attribute der Antwortseite beeinflussen.

Mit einem leicht geänderten URL im **form**-Tag kann die zur Nutzung des Servlets entworfene HTML-Seite auch für die JSP verwendet werden:

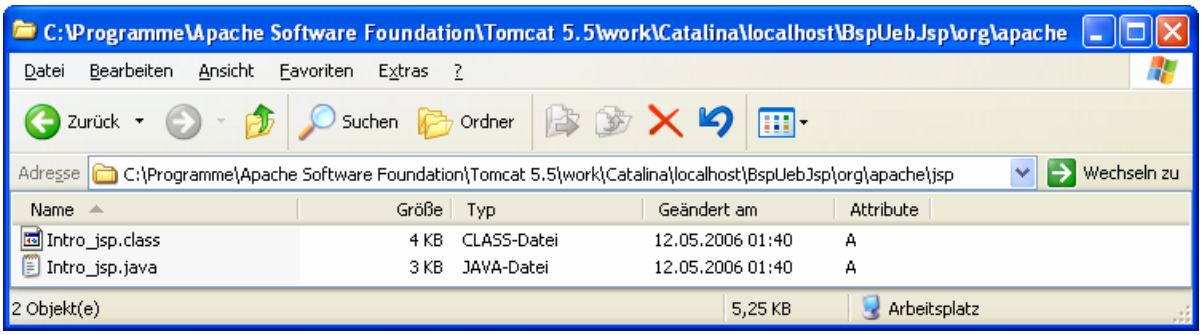
```
   . . .
<form method="post" action="Intro.jsp">
   . . .
```

Der Pfad zur JSP-Datei wird relativ angegeben und entfällt folglich, wenn sich die JSP- und die HTML-Datei im selben Ordner befinden.

Wir erstellen für die beiden Dateien im **webapps**-Verzeichnis des Servlet/JSP – Containers den Unterordner **BspUebJsp**:



Das vom JSP-Compiler **Jasper** automatisch erstellte Servlet erscheint im Ordner **work**, genauer:



Die HTML-Seite mit dem JSP-Verweis ist beim Tomcat-Server über folgenden URL aufzurufen:

http://127.0.0.1:8080/BspUebJsp/Intro.html

Abschließend soll noch demonstriert werden, dass Scriptlets nicht nur als Lückenfüller fungieren, sondern auch über die Aufnahme von statischen HTML-Bestandteilen in die Antwortseite entscheiden können. Die folgende JSP

```
<html>
<head>
<title>Kombinierte Anforderungs- Antwortseite</title>
</head>
<body>

<%
response.setCharacterEncoding("UTF-8");
request.setCharacterEncoding("UTF-8");
String vorname = request.getParameter("vorname");
String nachname = request.getParameter("nachname");
if (vorname != null  &&  nachname != null) {
%>

<h1>Hallo, <%= vorname %> <%= nachname %>!</h1>

<%
}
else {
%>

<h1>Nenne Deinen Namen, und ich sage Dir, wie Du hei&szlig;t!</h1>
<form method="post" action="Intro.jsp">
<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>

<%
}
%>

</body>
</html>
```

produziert das hinlänglich bekannte Formular, wenn beim Aufruf *keine* Werte für die Parameter `vorname` und `nachname` vorhanden sind. Anderenfalls wird freundlich gegrüßt.
Wird diese JSP in der Datei **…\webapps\BspUebJsp\Intro2.jsp** abgelegt, ist sie über folgenden URL aufrufbar:

http://127.0.0.1:8080/BspUebJsp/Intro2.jsp

Ob man besser mit Java-Syntax HTML-Seiten erstellt, also Servlets schreibt, oder in HTML-Code einzelne Java-Abschnitte einfügt, also JSPs schreibt, ist abhängig vom persönlichen Geschmack und vom Anteil der beiden Sprachen (Java, HTML) bei einem konkreten Projekt.

18 Anhang

18.1 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
[]	Feldindex
()	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.

18.2 Lösungsvorschläge zu den Übungsaufgaben

Abschnitt 2 (Werkzeuge zum Entwickeln von Java-Programmen)

Aufgabe 2

Syntaxfehler:

- Die schließende Klammer zum Rumpf der Klassendefinition fehlt.
- Die Zeichenfolge im **println()**-Aufruf muss mit dem "-Zeichen abgeschlossen werden.

Semantikfehler:

- Der Methodenname „mein“ ist falsch geschrieben.
- Die Methode **main()** muss als **public** definiert werden.

Aufgabe 3

1. falsch
2. richtig
3. richtig
4. falsch

Abschnitt 3 (Elementare Sprachelemente)

Abschnitt 3.1 (Einstieg)

Aufgabe 1

Der 1. Aufruf klappt. Der *Name* des **main()**-Parameters ist beliebig.
Der 2. Aufruf scheitert: Modifikator **static** vergessen.
Der 3. Aufruf scheitert: falscher Rückgabetyp **int**.

Aufgabe 2

Unzulässig sind:

- 4you
  Bezeichner müssen mit einem Buchstaben beginnen.
- else
  Schlüsselwörter wie **else** sind als Bezeichner verboten.

Aufgabe 3

Der erste Plus-Operator im **println()**-Parameter konvertiert sein rechtes Argument in eine Zeichenfolge, weil sein linkes Argument eine Zeichenfolge ist. Durch Klammerung muss dafür gesorgt werden, dass der *zweite* Plus-Operator zuerst ausgeführt wird und daher seine beiden Argumente addiert, bevor sein Ergebnis zum Argument für den ersten Plus-Operator wird.

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.println("3.3 + 2 = " + (3.3 + 2));` `  }` `}`	`3.3 + 2 = 5.3`

Abschnitt 3.2 (Variablen und primitive Datentypen)

Aufgabe 1

char gehört zu den integralen (ganzzahligen) Datentypen. Zeichen werden über ihre Nummer im aktuellen Zeichensatz gespeichert, das Zeichen *c* offenbar durch die Zahl 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 2

Durch das Suffix `l` im Literal `7l` wird ein **long**-Literal gekennzeichnet.

Aufgabe 3

Die Variable `i` ist nur im innersten Block gültig.

Aufgabe 4

```
class Prog {
  public static void main(String[] args) {
    System.out.println("Dies ist ein Java-Zeichenkettenliteral:\n   \"Hallo\"");
  }
}
```

Aufgabe 5

```
class Prog {
  public static void main(String[] args) {
    float PI = 3.141593f;
    double radius = 2.0;
    System.out.println("Der Flaecheninhalt betraegt: "+PI*radius*radius);
  }
}
```

Abschnitt 3.3 (Einfache Eingabe bei Konsolenanwendungen)

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 Dezimalpunkt!): ");
    double d = Simput.gdouble();
    System.out.println("d = "+ d);
  }
}
```

Abschnitt 3.4 (Operatoren und Ausdrücke)

Aufgabe 1

Ausdruck	Typ	Wert	Anmerkung
`6/4*2.0`	**double**	`2.0`	
`(int)6/4.0*3`	**double**	`4.5`	`Der Typumwandlungsoperator hat die höchste Priorität und wirkt daher auf die 6.`
`3*5+8/3%4*5`	**int**	`25`	Abarbeitung mit Zwischenergebnissen: `15+8/3%4*5` `15+2%4*5` `15+2*5` `15+10`

Aufgabe 2

`erg1 hat den Wert 0, denn:`

- `(i++ == j ? 7 : 8)` hat den Wert 8, weil $2 \neq 3$ ist.
- `8 % 2` ergibt 0.

Auch `erg2` hat den Wert 0, denn der Präinkrementoperator trifft auf die bereits vom Postinkrementoperator in der vorangehenden Zeile auf den Wert 3 erhöhte Variable `i` und setzt diese auf den Wert 4, so dass die Bedingung im Konditionaloperator erneut den Wert **false** hat.

Aufgabe 3

- `la1` hat den Wert **false**
  ^ wird zuerst ausgewertet.
- `la2` hat den Wert **true**
  Klammern erzwingen die alternative Auswertungsreihenfolge.
- `la3` hat den Wert **false**

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\Exp

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DM2Euro

Abschnitt 3.6 (Anweisungen (zur Ablaufsteuerung)

Aufgabe 1

Weil die **else**-Klausel der *zweiten* (nach oben nächstgelegenen) **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 eine *Endlosschleife* vorliegt.
Die restlichen Zeilen werden als selbständiger Anweisungsblock gedeutet, aber nie ausgeführt. Obwohl ein Programm mit dieser Konstruktion vollkommen nutzlos ist, verbraucht es reichlich CPU-Zeit und bringt den Prozessor zum Glühen.

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 $i = 52$ rechnet Java letztmals richtig.

Aufgabe 8

Lösungsvorschläge mit den beiden Algorithmus-Varianten befinden sich in den Ordnern:

...\BspUeb\Elementare Sprachelemente\GGT.Diff
...\BspUeb\Elementare Sprachelemente\GGT.Mod

Abschnitt 4 (Klassen und Objekte)

Aufgabe 1

1. richtig
2. falsch
   Die von einem Objekt ausgeführten Methoden haben stets vollen Zugriff auf die Instanzvariablen anderer Objekte derselben Klassen, sofern sie über eine Referenz auf solche Objekte besitzen.
3. richtig
4. falsch
   Lokale Variablen werden grundsätzlich nicht initialisiert, auch die lokalen Referenzvariablen nicht.

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\Life\TimeDuration

Aufgabe 3

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 Wertes („Get-Methode“)
- Verzicht auf eine **public**-Methode zum Verändern des Wertes („Set-Methode“)

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\R2Vec

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\FakulRek

Abschnitt 5 (Elementare Klassen)

Abschnitt 5.1 (Arrays (Felder))

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Arrays\Lotto

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Arrays\Eratosthenes

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Arrays\Marry

Abschnitt 5.2 (Zeichenketten)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Zeichenketten\PerZuf

Aufgabe 2

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 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Zeichenketten\StringUtil

Abschnitt 5.3 (Verpackungs-Klassen für primitive Datentypen)

Aufgabe 1

Lösungsvorschlag:

Quellcode	Ausgabe
`class Prog {` `  public static void main(String[] args) {` `  System.out.println("Min. byte-Zahl:\n " + Byte.MIN_VALUE);` `  }` `}`	`Min. byte-Zahl:` ` -128`

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 6 (Pakete)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Pakete\JAR\PackDemo

Aufgabe 2

Die beteiligten Klassen befanden sich stets im selben Verzeichnis und waren keinem Paket explizit zugeordnet, so dass sie sich in einem gemeinsamen (anonymen) Standardpaket befanden. Klassen eines Paketes haben per Voreinstellung wechselseitig volle Rechte.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Pakete\JAR\DM2Euro

Zum Erstellen des Archivs eignet sich das Kommando:

```
>jar cmf0 Manifest.txt DM2Euro.jar DM2EuroS.class Simput.class
```

Mit Hilfe der fertigen Archivdatei lässt sich die Klasse `DM2EuroS` folgendermaßen starten:

```
>java -jar DM2Euro.jar
```

Abschnitt 7 (Vererbung und Polymorphie)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Vererbung und Polymorphie\Figuren

Aufgabe 2

In der `Vater`-Klasse fehlt ein parameterfreier Konstruktor.

Aufgabe 3

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.

Abschnitt 8 (Ausnahmebehandlung)

Aufgabe 1

Lösungsvorschlag:

```
class Prog {
  public static void main(String[] args) {
    int[] vek = new int[5];
    vek = null;
    vek[0] = 1;
  }
}
```

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Exceptions\DuaLog

Abschnitt 10 (GUI-Programmierung)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\EventID

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\KeyModifiers

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\AnonClass

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\EuroKonverter

Abschnitt 11 (Ein-/Ausgabe über Datenströme)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\Editor\Mit Sichern

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\EA\BufferedOutputStream\auf die Konsole

Aufgabe 3

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 = new PrintWriter(fos);
```

Aufgabe 4

Beim Ausführen der Methode **readShort()** fordert das **DataInputStream**–Objekt den angekoppelten **InputStream** auf, zwei Bytes zu beschaffen. Schickt der Benutzer 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)

Anschließend entnimmt **readShort()** dem **InputStream** die beiden Bytes und interpretiert sie als vorzeichenbehaftete 16-Bit-Ganzzahl, was im Beispiel den dezimalen Wert 12301 ergibt.
Während der Filter korrekt arbeitet, ist die Eingabe passender Bytes via Tastatur (also **System.in**) offenbar äußerst umständlich bis unmöglich. Offenbar ist die Transformationsklasse **DataInputStream** nicht dafür eignet (und auch nicht dafür gedacht), **short**-Werte über das **InputStream**-Objekt **System.in** einzulesen. Sie wurde dafür konstruiert, die über einem **DataOutputStream** geschriebenen Bytes in Werte des ursprünglichen Datentyps zurück zu verwandeln.
Um numerische Daten von der Konsole zu lesen, sollte man die in Abschnitt 11.5 beschriebene Klasse **Scanner** verwenden.

Aufgabe 5

Bei einem **PrintStream** wird u.a. durch das Newline-Zeichen ein AutoFlush ausgelöst, beim **PrintWriter** geschieht dies jedoch nur beim Aufruf der Methoden **println()**, **printf()** und **format()**, welche die plattformspezifische Zeilenschaltung benutzt.

Abschnitt 12 (Applets)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Applets\PrimApplet

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Applets\EuroKonverter

Abschnitt 13 (Multimedia)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Multimedia\Grafik\Wuerfel

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Multimedia\Sound\Hintergrundmusik

Abschnitt 15 (Netzwerkprogrammierung)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Netzwerk\Internet-Ressourcen nutzen\Sophos-Update

Ein Beispiel für die Ausgabe bei einem Aktualisierungslauf mit **SoUp.class**:

```
Sophos-IDE-Checker 0.9
Bernhard Baltes-Goetz, URT


Es wurden folgende Parameter aus der Datei SoUp.txt gelesen:
  Sophos-Inst.verz.:    E:\Weg\Sophos\
  Sophos-Version:       4.05
  IDEs zuletzt akt.:    10.04.2006 10:27:14


Die Verbindung mit dem Sophos-Server wird hergestellt.


Informationen zu http://www.sophos.com/downloads/ide/405_ides.zip:
  Groesse in Bytes:     89361
  Letzte Aenderung:     27.04.2006 15:02:48


Die Datei
  http://www.sophos.com/downloads/ide/405_ides.zip
wird herunter geladen in das Verzeichnis
  E:\Weg\Sophos\

Die IDE-Dateien werden ausgepackt:
  E:\Weg\Sophos\mytob-hq.ide
  E:\Weg\Sophos\agentbiu.ide
  E:\Weg\Sophos\rbot-ddf.ide
   .  .  .
   .  .  .
  E:\Weg\Sophos\torpi-ap.ide
  E:\Weg\Sophos\tileb-ej.ide

Es wurden 68 Dateien ausgepackt.

Um die neuen Definitionen zu aktivieren, muss der Rechner neu gestartet werden!
```

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Netzwerk\Internet-Ressourcen nutzen\CGI\Telesuche

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Netzwerk\TCP-Programmierung\Chat

Abschnitt 16 (Datenbankzugriff via JDBC)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\JDBC\SqlUpdate

18.3 Java-Entwicklung mit dem JCreator LE

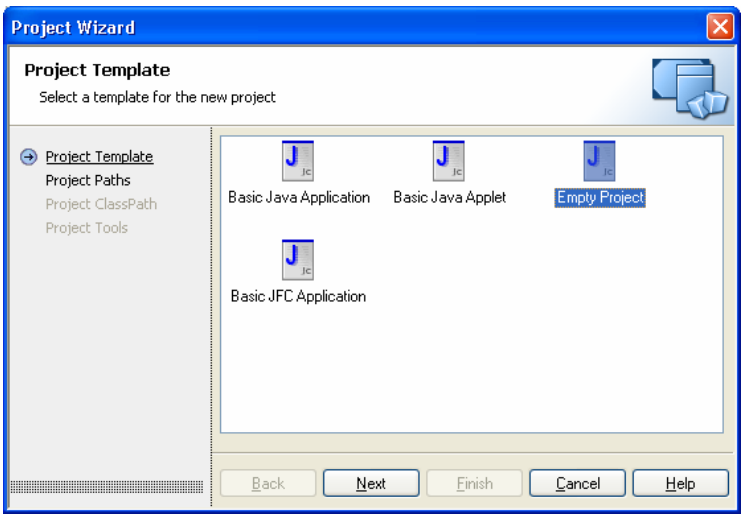
Während die Entwicklungsumgebung Eclipse einen PC mit relativ zeitgemäßer Hardware erwartet, gibt sich die ebenfalls kostenlose Alternative **JCreator LE 3.x** der Firma Xinox Software sehr bescheiden: Ein PC mit 64 MB Arbeitsspeicher und ca. 200 MHz Taktfrequenz sollte genügen. Das schlanke, unproblematisch zu bedienende und sehr flink agierende Programm ist über folgende Webseite zu beziehen:

http://www.jcreator.com/

Im Vergleich zu Eclipse fehlen zwar einige Funktionen (z.B. automatische Syntaxvervollständigung, visueller Editor), doch steht ein bequemer Texteditor mit spezieller Unterstützung für Java-Programmierer (z.B. durch Syntaxhervorhebung, intelligentes Einrücken) zur Verfügung. JCreator LE setzt Suns Java 2 SDK voraus, das daher zuerst installiert werden sollte.

18.3.1 Neues Projekt anlegen

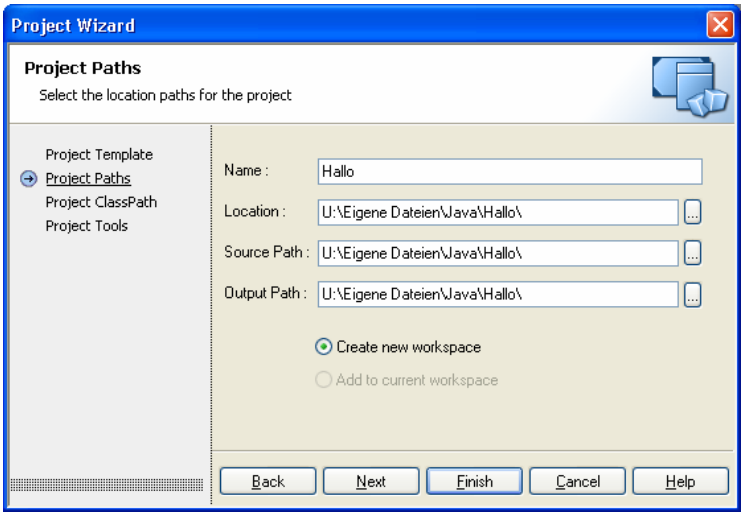
Um das `Hallo`-Programm per JCreator LE zu erstellen, legt man nach **File > New > Project** ein neues Projekt an:



Von den angebotenen Projekttypen ist für uns **Empty Project** am besten geeignet:

- Wir wollen kein Applet, sondern eine Applikation erstellen.
- Die Option **Basic Java Application** liefert den Quellcode für eine elementare GUI-Anwendung, woran wir im Moment noch nicht interessiert sind.

In der nächsten Dialogbox erhält das Projekt einen Namen:



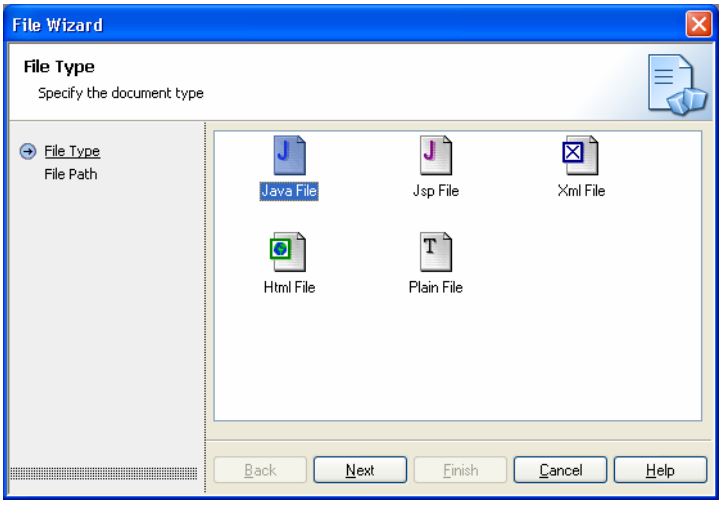
In dem unter **Location** genannten Ordner landen neben der Quellcode- und der Bytecodedatei auch die vom JCreator zur Projektverwaltung verwendeten Dateien:

- **jcp**-Datei (**J**Creator **P**roject).
  Enthält Informationen über ein Projekt (z.B. eine Anwendung oder ein Paket). Das Projekt kann per Doppelklick auf die **jcp**-Datei geöffnet werden. Es ist allerdings sinnvoller, ein JCreator - Projekt über den zugehörigen Arbeitsbereich zu öffnen:
- **jcw**-Datei (**J**Creator **W**orkspace).
  Jedes Projekt gehört zu einem Arbeitsbereich, dessen Eigenschaften in einer **jcw**-Datei verwaltet werden. Grundsätzlich kann ein Arbeitsbereich mehrere Projekte enthalten. Im Kurs lohnt sich diese zusätzliche Verwaltungsebene jedoch nicht, und Sie können die Begriff *Projekt* und *Arbeitsbereich* folglich als äquivalent betrachten.
  Um die Arbeit an einem vorhandenen Projekt fortzusetzen, öffnet man am besten den zugehörigen Arbeitsbereich, z.B. per Doppelklick auf seine **jcw**-Datei.

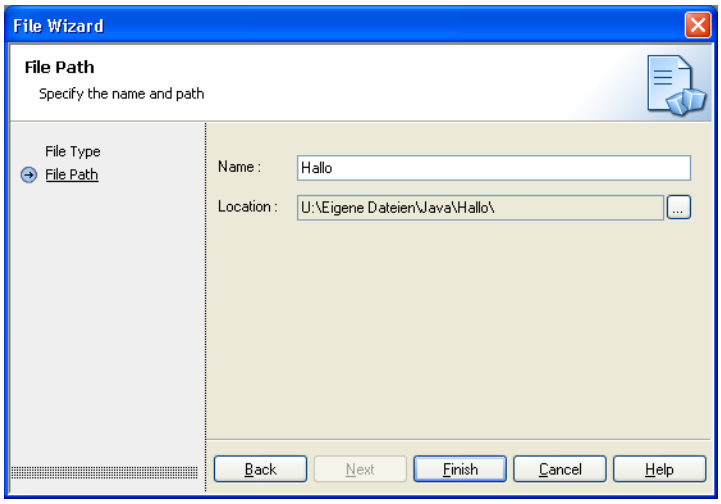
Für die Kursbeispiele ist beim Anlegen eines neuen Projektes die Option **Create new workspace** stets gegenüber der eventuell verfügbaren Alternative **Add to current workspace** zu bevorzugen.

18.3.2 Neue Klasse anlegen

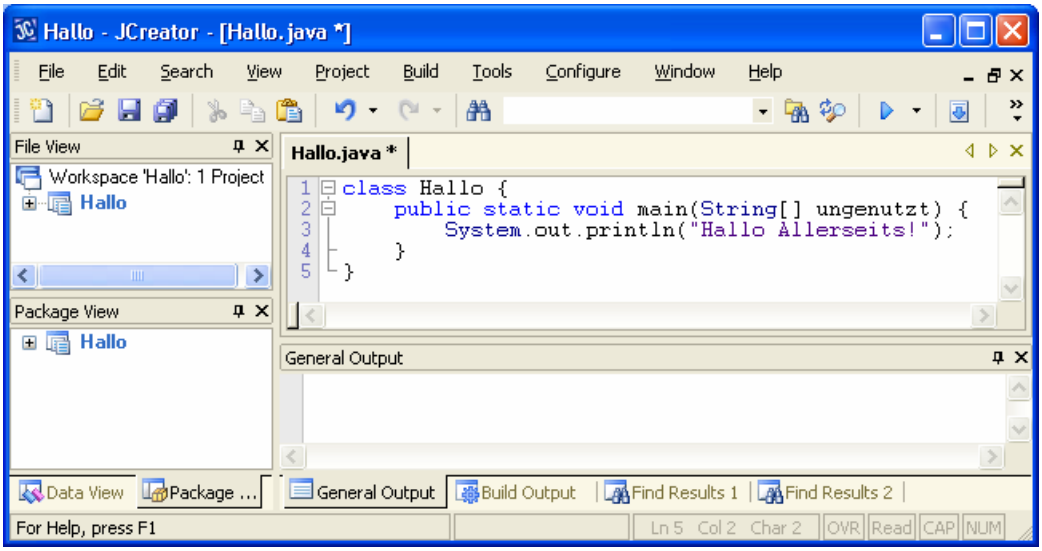
Wir verlassen den **Project Wizard** mit **Finish** und starten mit **File > New > File** den **File Wizard**, um eine leere Quellcodedatei anzulegen:



In der nächsten Dialogbox erhält die Datei einen Namen, der mit dem geplanten Klassennamen übereinstimmen sollte:



Nach dem Quittieren mit **Finish** der Dialogbox steht ein Editorfenster zur Verfügung, um den Quellcode der `Hallo`-Klasse einzutragen:



18.3.3 Editor und elementare Werkzeuge

Welche Vorteile der JCreator-Editor im Vergleich zu einem einfachen Editor (wie z.B. Notepad) bietet, wird bereits beim Eintippen des `Hallo`-Quellcodes deutlich:

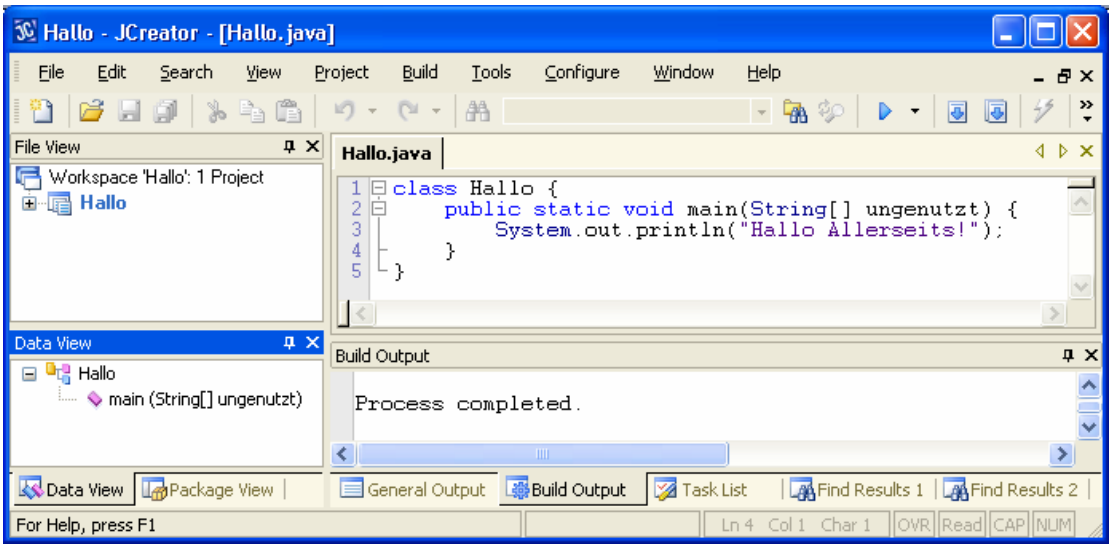
- Verschiedene Syntaxbestandteile werden farblich hervorgehoben, fest definierte Schlüsselwörter z.B. in Blau. Nach kurzer Eingewöhnungszeit werden Sie an der fehlenden Blaufärbung sofort erkennen, dass Ihnen beim Eintippen eines Schlüsselwortes (z.B. **class**) ein Fehler unterlaufen sein muss.
- Befindet sich die Schreibmarke auf oder neben einer Klammer, so wird diese *samt Gegenstück* farblich markiert. Damit sind z.B. fehlende oder überzählige Blockklammern leichter auszumachen.
- Beim Einfügen neuer Zeilen wird die Einrückebene meist korrekt gewählt.

Neben dem Editorbereich, der auch mit *mehreren* Dateien umgehen kann, bietet der Arbeitsplatz noch weitere Fenster:

- **Dateiansicht** (bei Bedarf mit **View > File View** zu öffnen)
- **Klassenansicht** (bei Bedarf mit **View > Data View** zu öffnen)
  Hier erscheinen die Eigenschaften und Methoden der im aktuellen Editorfenster definierten Klassen.

- **Paketansicht** (bei Bedarf mit **View > Package View** zu öffnen)
  Funktional verwandte Klassen fasst man zu Paketen zusammen, die auch eine wichtige Rolle bei der Verwaltung von Zugriffsrechten spielen. In unserem Beispiel wird die Klasse `Hallo` dem automatisch definierten Paket **Default** zugerechnet.
- **Compiler-Ausgaben** (bei Bedarf mit **View > Other Windows > Build** zu öffnen)
  Hier erscheinen u.a. die Fehlermeldungen des Compilers. Per Doppelklick auf eine Compiler-Fehlermeldung markiert man im Editor die betroffene Anweisung.

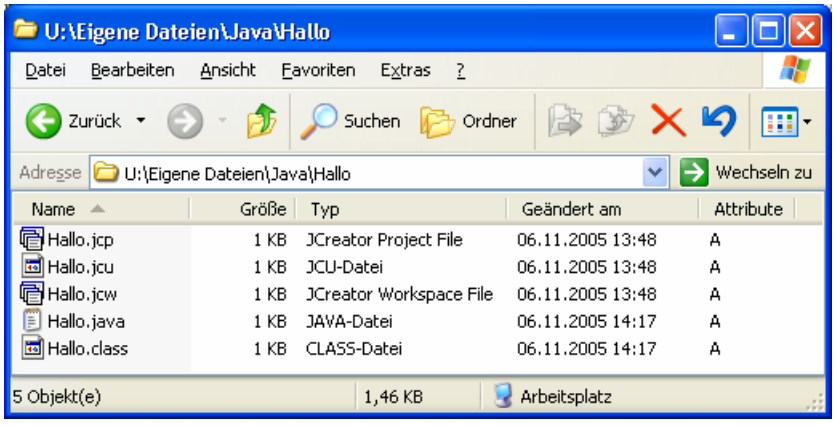
Nach dem Übersetzen der `Hallo`-Klasse zeigt sich folgendes Bild:



Zum Kompilieren und Ausführen von Programmen bietet der JCreator u.a. folgende Kommandos:

- **Build > Compile File** oder übersetzt die Quelle im aktiven Editorfenster.
- **Build > Compile Project** oder oder **F7** übersetzt *alle* Quellen des Projektes.
- **Build > Execute Project** oder oder **F5** führt das Projekt aus.

Nach einem Compiler-Lauf finden sich im `Hallo`-Projektordner folgende Dateien:



Wenn es sich beim aktuellen Projekt um ein Applet handelt, startet nach

Build > Execute Project oder oder **F5**

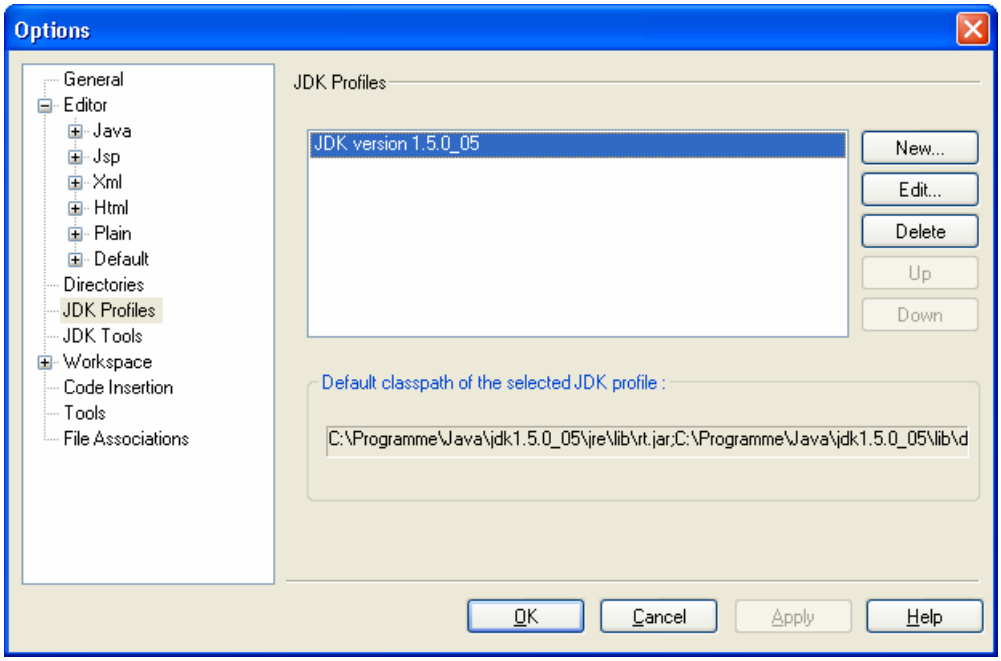
automatisch das JDK-Hilfsprogramm **appletviewer**, das die Rolle eines Browsers beim Starten und Ausführen von Java-Applets übernommen kann.

18.3.4 Klassenpfade

Beim JCreator 3.x LE ist die CLASSPATH-Umgebungsvariable des Betriebssystems mit den dort (z.B. für unsere Bequemlichkeitsklasse `Simput`) vereinbarten Klassenpfaden unwirksam, weil die Entwicklungsumgebung beim Start des Compilers bzw. –Interpreters grundsätzlich die Kommandozeilenoption „-classpath“ benutzt (vgl. Abschnitt 2.2.4). Man kann aber leicht dafür sorgen, dass bei der Kommandozeilenoption alle benötigten Pfade einbezogen werden. Dazu ist nach

Configure > Options > JDK Profiles

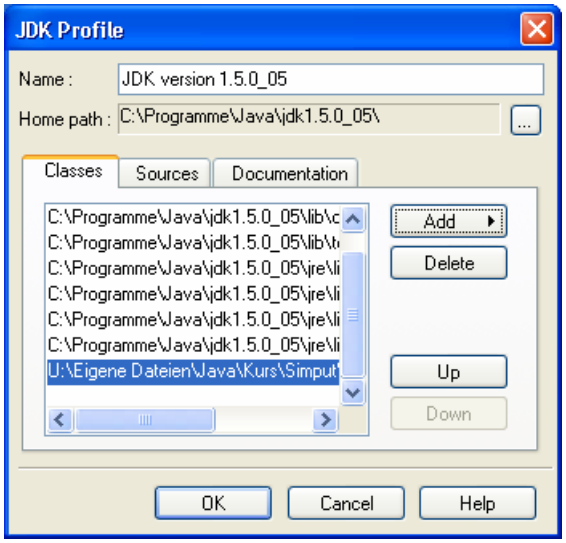
das verwendete JDK-Profil zu erweitern:



Markieren Sie in der **Options**-Dialogbox das gewünschte Profil, und fordern Sie mit **Edit** die Dialogbox **JDK Profile** an. Bei aktivem Registerblatt **Classes** kann über

Add > Add Path

ein zusätzlicher Pfad eingetragen werden, z.B.:



Alternativ kann über

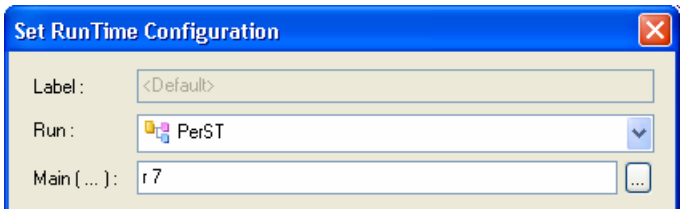
Add > Add Archive

eine **jar**-Datei in den Klassenpfad aufgenommen werden.

18.3.5 Kommandozeilenargumente

Soll ein Programm in der JCreator-Entwicklungsumgebung gestartet werden, das Kommandozeilen-Argumente benötigt, können diese folgendermaßen vereinbart werden:

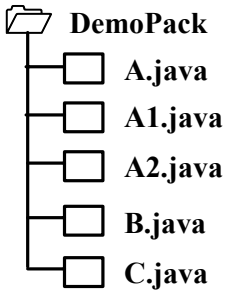
- Menübefehl **Build > Runtime Configuration**
- In der Dialogbox **RunTime Configuration** markiert man **<Default>** und klickt auf **Edit**.
- In der Dialogbox **Set RunTime Configuration** trägt man die Optionen ins Textfeld **Main(...)** ein, z.B.:



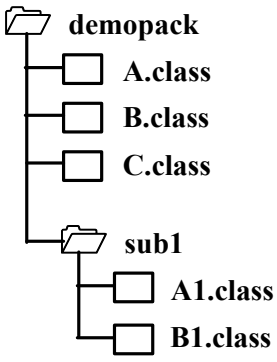
18.3.6 Paketunterstützung

Das Beispielpaket `demopack` aus Abschnitt 6.1 kann mit dem JCreator 3.x folgendermaßen erstellt werden:

- Neues Projekt beginnen, z.B. mit Projektverzeichnis `DemoPack`
- Zu jeder Klasse wird im Projektverzeichnis eine Quelldatei angelegt, wobei in der ersten Zeile eine **package**-Anweisung die Zugehörigkeit zu einem Paket oder Unterpaket festlegt (siehe oben). Im Beispiel entsteht also ein Projektverzeichnis mit folgenden Quellcode-Dateien:



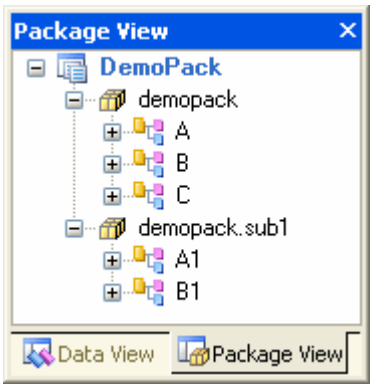
- Nach einem erfolgreichen Compiler-Lauf entsteht im Projektordner automatisch der folgende `demopack`-Dateiverzeichnisbaum mit den **class**-Dateien des Paketes:



Der JCreator bietet mit dem **Package View** eine gute Hilfe, um auch bei hierarchisch strukturierten Paketen den Überblick zu behalten. Diese Ansicht kann mit dem Menübefehl

View > PackageView

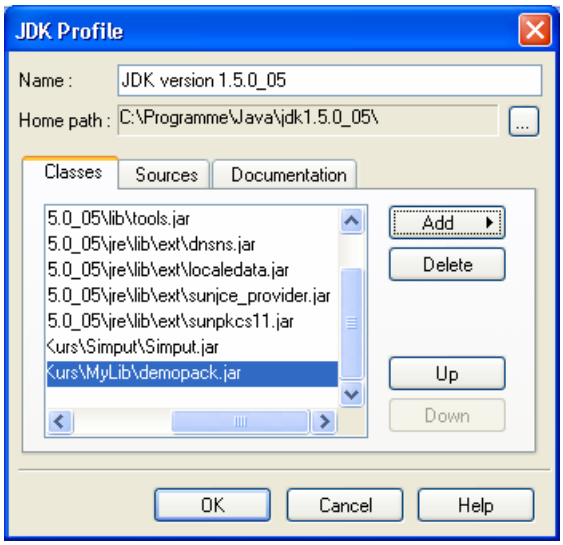
eingeschaltet werden und zeigt dann die zu einem Projekt gehörenden Pakete, z.B.:



18.3.7 Archivdateien einbinden

Um eine vorhandene JAR-Datei mit Paketen und Klassen für *alle* Projekte verfügbar machen, die ein bestimmtes JDK-Profil verwenden, geht man folgendermaßen vor:

- **Options**-Dialogbox öffnen mit **Configure > Options** und Mausklick auf **JDK Profiles**
- Ein **JDK Profile** markieren und mit **Edit** die Dialogbox **JDK Profile** öffnen.
- Bei aktivem Registerblatt **Classes** über **Add > Add Archive** eine Archivdatei wählen, z.B.:



- Alle Dialogboxen mit **OK** schließen

Literatur

Ball, J., Carson,D. B., Evans, I., Haase, K & Jendrock, E. (2006). *The Java EE 5 Tutorial*. Santa Clara,CA: Sun Microsystems.

Balzert, H. (1999). *Lehrbuch der Objektmodellierung: Analyse und Entwurf*. Heidelberg: Spektrum.

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/

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

Hostmann, 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. (2004). *Goto Java* (Version 4.0.2). Online-Dokument: http://www.javabuch.de/download.html

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.

Münz, S. (2005). *SELFHTML 8.1*. Online-Dokument: http://aktuell.de.selfhtml.org/extras/download.shtml

MySQL AB (2006). *MySQL 5.0 Reference Manual*. Bestandteil des Programnmpaketes.

Paul, M. (1999). *Datenübertragungstechnologien*. Online-Dokument: http://www.uni-trier.de/urt/user/baltes/docs/uebtech/uebtech.pdf

Patwardhan, N., Siever, E. & Spainhour, S. (2002). *Perl in a nutshell* (2nd. ed.). Beijing: O'Reilly.

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.

SUN Inc. (2005). *The Java Tutorial*. Online-Dokument: http://java.sun.com/docs/books/tutorial/

Ullenboom, C. (2004). *Java ist auch eine Insel* (4. Aufl.). Galileo Computing. Online-Dokument: http://www.galileocomputing.de/openbook/javainsel4/

Index

T