JAVA - EINFÜHRUNG

Kursunterlage

Hubert Partl

Inhaltsverzeichnis

Java Einführung - Kursunterlage

Hubert Partl
Zentraler Informatikdienst
Universität für Bodenkultur Wien
Version: Juli 2005

Vorwort

Die vorliegende Java-Einführung beschreibt nur kurz die wichtigsten Eigenschaften von Java, die für die meisten Anwendungen benötigt werden. Für darüber hinaus gehende spezielle Features und Anwendungen sowie für ausführlichere Informationen wird auf die Referenzen [Seite 176] und auf die Online-Dokumentation [Seite 8] verwiesen.

Die Java-Einführung besteht aus zwei Teilen:

- der eigentlichen **Kursunterlage** mit Beispielen und Übungsaufgaben, und
- **Musterlösungen** zu den Übungsaufgaben.

Wenn Sie Java erfolgreich lernen wollen, empfehle ich Ihnen dringend, die Musterlösungen erst dann auszudrucken und anzusehen, wenn Sie bereits *alle* Übungsbeispiele selbständig fertig programmiert haben, also erst am Ende des Kurses.

Wenn Ihnen meine Java-Einführung gefällt, sagen Sie es weiter. Wenn Sie darin Fehler entdecken oder Verbesserungsvorschläge haben, sagen Sie es bitte mir per E-Mail an hubert.partl@boku.ac.at - ich freue mich immer über solche Hilfe.

Ich danke Guido Krüger, Stefan Zeiger, Peter van der Linden, Roedy Green, Markus Reitz, Linda Radecke, Stefan Matthias Aust, Michael Seeboerger, Rainer Sawitzki und allen anderen Internet-Surfern, die mir gute Tips gegeben und Tippfehler korrigiert haben, sowie den Teilnehmern an meinen Java-Schulungen, die mir interessante Fragen gestellt haben.

Hubert Partl

Copyright

Grundlagen

1 Was ist Java?

Sie haben vielleicht schon mit Programmiersprachen wie Cobol, Fortran, Pascal, C, C++ oder Visual Basic gearbeitet oder zumindest davon gehört. Wie unterscheidet sich Java von diesen Sprachen?

1.1 Insel, Kaffee oder Programmiersprache

Java wurde von der Firma Sun entwickelt und erstmals am 23. Mai 1995 als neue, objekt-orientierte, einfache und plattformunabhängige Programmiersprache vorgestellt. Sun besitzt das Schutzrecht auf den Namen Java und stellt damit sicher, daß nur "100 % richtiges Java" diesen Namen tragen darf. Die Sprache ist aber für alle Computersysteme verfügbar (im Allgemeinen kostenlos).

Java geht auf die Sprache Oak zurück, die 1991 von Bill Joy, James Gosling und Mike Sheridan im Green-Projekt entwickelt wurde, mit dem Ziel, eine einfache und plattformunabhängige Programmiersprache zu schaffen, mit der nicht nur normale Computer wie Unix-Workstations, PCs und Apple programmiert werden können, sondern auch die in Haushalts- oder Industriegeräten eingebauten Micro-Computer, wie z.B. in Waschmaschinen und Videorekordern, Autos und Verkehrsampeln, Kreditkarten und Sicherheitssystemen und vor allem auch in TV-Settop-Boxes für "intelligente" Fernsehapparate.

Allgemein anerkannt wurde Java aber erst seit 1996 in Verbindung mit Web-Browsern und Internet-Anwendungen sowie mit der Idee eines NC (Network Computer), der im Gegensatz zum PC (Personal Computer) nicht lokal installierte, maschinenspezifische Software-Programme benötigt, sondern die Software in Form von Java-Klassen dynamisch über das Netz (Intranet) von einem zentralen Server laden kann. Diese Idee wurde später zu einem allgemeinen "Application Service Providing" (ASP) erweitert. Inzwischen wird Java aber weniger für Applets in Web-Pages, sondern mehr für selbständige Anwendungs-Programme sowie für Server-Applikationen verwendet.

Der Name wurde nicht direkt von der indonesischen Insel Java übernommen sondern von einer bei amerikanischen Programmierern populären Bezeichnung für Kaffee.

Die wichtigsten Eigenschaften von Java sind:

- plattformunabhängig
- Objekt-orientiert
- Syntax ähnlich wie bei C und C++
- umfangreiche Klassenbibliothek
- Sicherheit von Internet-Anwendungen

Bei Java-Programmen muss zwischen zwei grundsätzlichen Arten unterschieden werden: Applikationen [Seite 4] und Applets [Seite 4] .

1.2 Applikationen (application)

Java-Applikationen sind Computer-Programme mit dem vollen Funktionsumfang, wie er auch bei anderen Programmiersprachen gegeben ist. Applikationen können als lokale Programme auf dem Rechner des Benutzers laufen oder als Client-Server-Systeme über das Internet bzw. über ein Intranet oder als Server-Programme (Servlets, CGI-Programme) auf einem Web-Server.

Technisch gesehen zeichnen sich Java-Applikationen dadurch aus, dass sie eine statische Methode main enthalten.

1.3 Applets

Java-Applets werden innerhalb einer Web-Page dargestellt und unter der Kontrolle eines Web-Browsers ausgeführt. Sie werden meist über das Internet von einem Server geladen, und spezielle Sicherungen innerhalb des Web-Browsers ("Sandkasten", sandbox) sorgen dafür, dass sie keine unerwünschten Wirkungen auf den Client-Rechner haben können. So können Applets z.B. im Allgemeinen nicht auf lokale Files, Systemkomponenten oder Programme zugreifen und auch nicht auf Internet-Verbindungen außer zu dem einen Server, von dem sie geladen wurden.

Technisch gesehen zeichnen sich Java-Applets dadurch aus, dass sie Unterklassen der Klasse Applet sind.

Die meisten in dieser Kursunterlage beschriebenen Regeln gelten gleichermaßen für Applikationen und Applets. Die Spezialitäten, die nur für Applets gelten, sind in einem eigenen Kapitel über Applets [Seite 84] zusammengefasst.

1.4 JavaScript ist nicht Java

JavaScript ist eine Skript-Sprache, die in HTML eingebettet werden kann und bei manchen Web-Browsern (Netscape, Internet-Explorer) die Ausführung von bestimmten Funktionen und Aktionen innerhalb des Web-Browsers bewirkt.

Im Gegensatz zu Java ist JavaScript

- keine selbständige Programmiersprache,
- nicht von der Browser-Version unabhängig,
- nicht mit den notwendigen Sicherheitsmechanismen ausgestattet.

2 Software und Online-Dokumentation

Was brauchen Sie, um plattformunabhängige Java-Programme zu erstellen?

2.1 Java Development Kit (JDK)

Das Java Development Kit (JDK) umfasst die für die Erstellung und das Testen von Java-Applikationen und Applets notwendige Software, die Packages mit den zur Grundausstattung gehörenden Java-Klassen, und die Online-Dokumentation.

Zur Software gehören der Java-Compiler, das Java Runtime Environment (die Java Virtual Machine) für die Ausführung von Applikationen, der Appletviewer für die Ausführung von Applets, ein Java-Debugger und verschiedene Hilfsprogramme.

Die Online-Dokumentation umfasst eine Beschreibung aller Sprachelemente und aller Klassen des Application Program Interface API.

Java ist eine relativ junge Programmiersprache und daher noch immer in Entwicklung, d.h. es kommen immer wieder neue Versionen mit Ergänzungen und Verbesserungen heraus:

- JDK 1.0 (1995): die Urversion
  ab 1999 in Netscape 3.0 integriert.
- JDK 1.1 (1997): Änderungen und Ergänzungen der Klassenbibliothek (Beans-Konventionen, Event-Handlung, Text-Files, Datenbanken u.a.)
  ab 1999 in Netscape 4 und teilweise in Internet Explorer 4 integriert.
- JDK 1.2 (1998): Erweiterungen der Klassenbibliothek (Swing, Collections u.a.)
  Ab 2001 ist Java nicht mehr in die Web-Browser Netscape 6, Mozilla, Internet Explorer 5 etc. integriert, sondern die jeweils aktuelle Java-Version von Sun kann als Plug-In eingebaut werden.
  Die JDK-Versionen 1.2 bis 5.0 werden auch J2SDK = Java Plattform 2 Software Development Kit genannt.
- JDK 1.3 (2000): Fehlerkorrekturen und Performance-Verbesserungen.
- JDK 1.4 (2002): Erweiterungen der Klassenbibliothek
- JDK 5.0 (2004): Ergänzungen der Sprach-Syntax (Generics, Autoboxing u.a.) sowie Änderungen und Ergänzungen der Klassenbibliothek
- JDK 6 (geplant für 2006): weitere Ergänzungen der Sprach-Syntax und der Klassenbibliothek.

Seit Version 1.2 ist das JDK in mehrere sogenannte Editionen unterteilt:

- Java SE bzw. J2SE = Java Standard Edition (mit Graphischen User Interfaces)
- Java EE bzw. J2EE = Java Enterprise Edition (Server-seitige Erweiterungen zu J2SE)
- Java ME bzw. J2ME = Java Micro Edition (für kleine mobile Geräte)

Das JDK für ein bestimmtes System erhält man meist kostenlos (z.B. zum Download über das Internet) vom jeweiligen Hersteller, also die Solaris-Version von Sun, die HP-Version von HP, die IBM-Version von IBM. Versionen für Windows-PC, Macintosh und Linux kann man von Sun oder IBM bekommen.

2.2 Environment-Variable

Normalerweise genügt es, das Directory, in dem sich die Java-Software befindet, in die PATH-Variable einzufügen. Die anderen Variablen (CLASSPATH, JAVA_HOME) werden nur in Spezialfällen oder innerhalb der Software benötigt und müssen normalerweise *nicht* gesetzt werden. Die folgenden Hinweise sind also in den meisten Fällen **nicht notwendig** sondern *nur* in Spezialfällen:

Die Variable CLASSPATH gibt an, in welchen Directories und in welchen jar-Archiven nach Klassen und Packages gesucht werden soll, getrennt durch Doppelpunkte (Unix) bzw. Strichpunkte (Windows). Dies ist seit JDK 1.2 *nur dann* notwendig, wenn zusätzliche jar-Archive oder zusätzliche Directories (die sich auch auf Web-Servern befinden können) gebraucht werden. In diesem Fall enthält der CLASSPATH den Punkt (für das jeweils aktuelle Directory), das Start-Directory der im JDK enthaltenen Packages bzw. das Archiv-File der Klassenbibliothek. sowie die zusätzlichen jar-Archive und Directories bzw. URLs.

Wenn der CLASSPATH *nicht* gesetzt ist, wird seit JDK 1.2 nach allen benötigten Klassen automatisch zuerst im aktuellen Directory und dann in der zur verwendeten Java-Software gehörenden Klassenbibliothek gesucht. Im Normalfall soll der CLASSPATH deshalb *gar nicht* gesetzt werden.

Die Variable JAVA_HOME gibt an, in welchem Directory die Komponenten des JDK zu finden sind. Die explizite Angabe ist meistens ebenfalls *nicht* notwendig.

Damit man den Java-Compiler, das Runtime-System, den Appletviewer etc. einfach aufrufen kann, sollte das entsprechende bin-Directory in der PATH-Variablen enthalten sein.

2.3 Filenamen

Jedes Java-Source-File darf nur eine public Klasse enthalten, und der Filename muss dann der Name dieser Klasse (mit der richtigen Groß-Kleinschreibung) mit der Extension .java sein, hat also die Form

```
Xxxxx.java
```

Wenn man das Source-File auf einem PC mit MS-Word, Wordpad oder Notepad erstellt, muss man darauf achten, dass es mit dem Dateityp Text-File (nicht Word-File) erstellt wird und dass nicht automatisch eine Extension .txt angehängt wird. Eventuell muss man also beim Speichern den Filenamen mit Quotes in der Form
`"Xxxxx.java"`
schreiben, damit man nicht einen Filenamen der Form `Xxxxx.java.txt` erhält.

Der Java-Compiler erzeugt für jede Klasse ein eigenes File, das den Bytecode dieser Klasse enthält. Der Filename ist dann der Name der Klasse mit der Extension .class, hat also die Form

```
Xxxxx.class
```

Man kann auch mehrere Klassen-Files in ein komprimiertes Archiv-File [Seite 9] zusammenfassen, das dann weniger Übertragungszeit über das Internet benötigt. Solche Java-Archiv-Files haben Namen der Form

```
xxx.jar
```

2.4 Java Compiler (javac)

Der Aufruf des Java-Compilers erfolgt im einfachsten Fall in der Form

```
javac Xxxxx.java
```

Der Name des Source-Files muss *mit* der Extension .java angegeben werden. Man kann auch mehrere Source-Files angeben:

```
javac Xxxxx.java Yyyyy.java
```

Falls die Klasse andere Klassen verwendet, die neu übersetzt werden müssen, werden diese automatisch ebenfalls übersetzt, ähnlich wie bei der Unix-Utility make.

Der Compiler erzeugt für jede Klasse ein File Xxxxx.class, das den Bytecode enthält. Dieser Bytecode ist plattformunabhängig: Egal auf was für einem System der Java-Compiler aufgerufen wurde, der Bytecode kann auch auf jedem anderen Computersystem ausgeführt werden, zumindest wenn es sich um "100% pure Java" handelt, was bei manchen Microsoft-Produkten leider nicht garantiert ist.

2.5 Java Runtime System (java)

Die Ausführung des Bytecodes einer Java-Applikation erfolgt durch Aufruf der Java Virtual Machine JVM (im Java Runtime Environment JRE) in der Form

```
java Xxxxx
```

oder

```
java Xxxxx parameter parameter
```

Der Bytecode der Klasse Xxxxx muss im File Xxxxx.class oder in einem Archiv-File (zip, jar) liegen, der Klassenname muss aber *ohne* die Extension .class angegeben werden.

Ältere Versionen der JVM waren noch reine Bytecode-Interpreter und daher relativ langsam. Neuere Versionen erzeugen mit Hilfe von Just-in-Time-Compilation (JIT) und Hotspot-Technologie (Analyse des Laufzeitverhaltens) eine weitgehende Optimierung und Beschleunigung der Ausführungszeit.

2.6 Web-Browser und Appletviewer

Die Darstellung und Ausführung eines Java-Applet erfolgt durch Aufrufen der entsprechenden Web-Page (HTML-File) mit einem Java-fähigen Web-Browser wie z.B. Netscape, Internet-Explorer oder HotJava.

Für Tests und innerhalb von Java-Schulungen empfiehlt sich die Verwendung des mit dem JDK mitgelieferten Appletviewer. Dies ist ein einfacher Browser, der alle HTML-Tags außer <applet> und <param> ignoriert und nur das im HTML-File angeführte Applet ohne den Rest der Web-Page anzeigt. Der Aufruf erfolgt in der Form

```
appletviewer xxxx.html
```

oder mit einer kompletten Internet-Adresse (URL):

```
appletviewer URL
```

2.7 Online-Dokumentation (API)

Die Online-Dokumentation der Klassenbibliothek (application programming interface API) kann entweder on-line am Sun-Server http://java.sun.com/ gelesen oder zusätzlich zum JDK am eigenen Rechner installiert oder auf einem Web-Server im eigenen Bereich gespeichert und dann lokal gelesen werden. Sie liegt in der Form von Web-Pages (HTML-Files mit Hypertext-Links) vor und kann mit jedem beliebigen Web-Browser gelesen werden, z.B. mit Netscape. Man kann den jeweiligen lokalen Filenamen direkt als Aufrufparameter angeben, oder man kann sich zum File "durchklicken" und es dann in die Bookmarks eintragen und später von dort wiederum aufrufen.

Die wichtigsten Files innerhalb dieser Dokumentation sind:

- **Class Hierarchy** für die Suche nach einer bestimmten Klasse und deren Konstruktoren, Datenfeldern und Methoden.
- **All Names (Index of Fields and Methods)** für die Suche nach Datenfeldern und Methoden, wenn man nicht weiß, in welcher Klasse sie definiert sind.

In jedem dieser Files kann man mit dem "Find-in-current" Befehl des Web-Browsers (je nach Browser Ctrl-F, Strg-F, im Edit- oder Bearbeiten-Menü oder dergleichen) nach Namen oder Substrings **suchen**. Das Index-File ist sehr groß und/oder in 26 Einzel-Files je nach den Anfangsbuchstabes unterteilt, es geht daher meistens schneller, wenn man die Suche im Class-Hierarchy-File beginnt.

Wenn Sie einen **Klassennamen** anklicken, erhalten Sie die komplette Beschreibung dieser Klasse und ihrer Konstruktoren, Datenfelder und Methoden sowie eine Angabe ihrer Oberklassen. Falls Sie eine Methode in der Dokumentation einer Unterklasse nicht finden, dann sehen Sie in ihren Oberklassen nach (siehe Vererbung [Seite 47]).

Wenn Sie den Namen einer **Methode** (oder eines Konstruktors oder Datenfeldes) anklicken, erhalten Sie sofort die Beschreibung dieser Methode (bzw. des Konstruktors oder Datenfeldes). Dabei müssen Sie aber beachten, dass es mehrere gleichnamige Methoden und Konstruktoren sowohl innerhalb einer Klasse (siehe Overloading [Seite 36]) als auch in verschiedenen Klassen (siehe Overriding [Seite 47]) geben kann.

2.8 Java-Archive (jar)

Ein Java-Archiv enthält Dateien und eventuell auch ganze Directory-Strukturen (siehe Packages [Seite 54]) in dem selben komprimierten Format, das auch von PKZIP und Win-Zip verwendet wird. Sie werden mit dem Programm **jar** (java archiver) verwaltet, der Aufruf erfolgt ähnlich wie beim Unix-Programm tar (tape archiver):

Archiv-File erstellen (create):
```
jar -cvf xxx.jar *.class
```

Inhalt eines Archiv-Files anzeigen (table of contents):
```
jar -tvf xxx.jar
```

einzelne Dateien aus einem Archiv-File herausholen (extract):
```
jar -xvf xxx.jar Yyyy.class
```

Das Herausholen bzw. "Auspacken" von Archiv-Files ist meistens gar nicht nötig: Der Java-Compiler [Seite 7] und die Java Virtual Machine [Seite 7] können die Class-Files direkt aus dem Archiv-File lesen und laden. Zu diesem Zweck muss der Filename des Archiv-Files im Classpath [Seite 6] angegeben sein bzw. bei Applets im ARCHIVE-Parameter des Applet-Tag im HTML-File [Seite 85] .

2.9 Software-Tools

Von mehreren Firmen werden Editoren, GUI-Builder, Entwicklungsumgebungen (Integrated Development Environments IDE) und andere Hilfsmittel angeboten, mit denen man Java-Applikationen, Applets, Servlets etc. möglichst bequem und einfach erstellen und testen kann. Beispiele für aktuelle und historische Tools sind Eclipse, NetBeans, IntelliJ, JTogether, JBuilder, JDeveloper, Visual Age, Visual Café, Kawa, Forte, Visual J++ und viele andere.

Übersichten über die verfügbaren Software-Tools und deren Bezugsquellen finden Sie auf den in den Referenzen [Seite 176] angeführten Web-Servern, Erfahrungsberichte in den einschlägigen Usenet-Newsgruppen.

2.10 Java Beans

Unter Java Beans ("Kaffeebohnen") versteht man kleine Java-Programme (Klassen) mit genau festgelegten Konventionen für die Schnittstellen, die eine Wiederverwendung in mehreren Anwendungen (Applikationen und Applets) ermöglichen, ähnlich wie bei
Unterprogramm-Bibliotheken in anderen Programmiersprachen. Dies ist vor allem im Hinblick auf das Software-Engineering von komplexen Programmsystemen interessant. Dafür gibt es ein eigenes Beans Development Kit BDK, das man zusätzlich zum JDK installieren kann, und ein Package java.beans, das ab Version 1.1 im JDK enthalten ist,

Beans werden auch von vielen der oben erwähnten Software-Tools (IDE) unterstützt. Auch die Klassenbibliothek des JDK ist seit Version 1.2 weitgehend nach den Beans-Konventionen geschrieben, und manche Software-Firmen verkaufen spezielle Java-Beans für bestimmte Anwendungen.

Es ist empfehlenswert, auch beim Schreiben eigener Java-Programme möglichst die Konventionen von Java-Beans einzuhalten, also z.B. dass jede Klasse einen Default-Konstruktor mit leerer Parameterliste haben soll, dass die Methoden für das Setzen und Abfragen von Datenfeldern Namen der Form setXxxxx und getXxxxx bzw. isXxxxx haben sollen, oder dass alle Klassen so "selbständig" programmiert werden sollen, dass sie unabhängig davon funktionieren, wie andere Klassen programmiert wurden.

Mehr über diese Grundsätze und Empfehlungen finden Sie im Kapitel über Objekte und Klassen [Seite 31] . Für genauere Informationen über Beans und BeanInfos wird auf die Referenzen [Seite 176] verwiesen.

2.11 Vorgangsweise

Hier nochmals eine kurze Zusammenfassung der Befehle, die im Normalfall nötig sind, um ein Java-Programm zu erstellen, zu übersetzen und auszuführen:

2.11.1 Vorgangsweise bei Programmen (Applikationen)

- `notepad "Xxxxx.java"`
- `javac Xxxxx.java`
- `java Xxxxx`

2.11.2 Vorgangsweise bei Applets

- `notepad "Xxxxx.java"`
- `notepad "Xxxxx.html"`
- `javac Xxxxx.java`
- `appletviewer Xxxxx.html`

2.12 Beispiel: Einfache HelloWorld-Applikation

Es ist üblich, einen ersten Eindruck für eine neue Programmiersprache zu geben, indem man ein extrem einfaches Programm zeigt, das den freundlichen Text "Hello World!" auf die Standard-Ausgabe schreibt.

So sieht dieses Minimal-Programm als Java-Applikation aus:

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

    System.out.println("Hello World!");
  }
}
```

Dieses Java-Source-Programm muss in einem File mit dem Namen HelloWorld.java liegen. Die Übersetzung und Ausführung erfolgt dann mit

```
javac HelloWorld.java
```

```
java HelloWorld
```

Ein als Muster für objekt-orientierte Java-Programme besser geeignetes Beispiel für eine HelloWorld-Applikation finden Sie im Kapitel über Objekte und Klassen [Seite 31] .

Ein analoges Beispiel für ein minimales HelloWorld-Applet finden Sie im Kapitel über Applets [Seite 84] .

2.13 Übung: HelloWorld-Applikation

Schreiben Sie die oben angeführte HelloWorld-Applikation (oder eine Variante davon mit einem anderen Text) in ein Java-Source-File und übersetzen Sie es und führen Sie es aus.

2.14 Übung: Online-Dokumentation

Suchen Sie in der Online-Dokumentation (API) nach der Beschreibung der Methode println und des Objekts System.out.

2.15 Typische Anfänger-Fehler

Ein Java-Neuling ("Newbie") fragt: Ich habe das HelloWorld-Programm aus meinem Java-Buch abgeschrieben, aber es funktioniert nicht.

Ein erfahrener Programmierer ("Oldie") antwortet: Das ist schon richtig so, das HelloWorld-Beispiel dient dazu, dass Du die typischen Anfänger-Fehler kennen lernst und in Zukunft vermeiden kannst. Der erste Fehler war schon: Wenn Du uns nicht den genauen Wortlaut der Fehlermeldung, die Version Deiner Java-Software (JDK, IDE) und die relevanten Teile Deines Source-Programms dazu sagst, können wir den Fehler nicht sehen und Dir nicht helfen. In diesem Fall kann ich nur raten. Du hast wahrscheinlich einen der folgenden typischen Newbie-Fehler gemacht:

- Du hast das Programm nicht genau genug abgeschrieben (Tippfehler, Groß-Kleinschreibung, Sonderzeichen, Leerstellen), lies doch die Fehlermeldungen und Korrekturhinweise, die der Compiler Dir gibt.
- Du hast das Programm nicht unter dem richtigen Filenamen abgespeichert. Wenn die Klasse HelloWorld heißt, muss das File HelloWorld.java heißen, nicht helloworld.java und auch nicht HelloWorld.java.txt, im letzteren Fall versuch es mit
  `notepad "HelloWorld.java"`
- Du hast beim Compiler nicht den kompletten Filenamen mit der Extension angegeben (wieder mit der richtigen Groß-Kleinschreibung):
  `javac HelloWorld.java`

- Du hast bei der Ausführung nicht den Klassennamen ohne die Extension angegeben (wieder mit der richtigen Groß-Kleinschreibung):
  ```
  java HelloWorld
  ```
- In der Umgebungsvariable PATH ist das Directory, in dem sich die JDK-Software befindet, nicht neben den anderen Software-Directories enthalten, versuch
  ```
  set PATH=%PATH%;C:\jdk1.2\bin
  ```
  oder wie immer das auf Deinem Rechner heißen muss.
- Die Umgebungsvariable CLASSPATH ist (auf einen falschen Wert) gesetzt. Diese Variable sollte überhaupt nicht gesetzt sein, nur in seltenen Spezialfällen und dann so, dass sie sowohl die Stellen enthält, wo die Java-Klassenbibliotheken liegen, als auch den Punkt für das jeweils aktuelle Directory.
- Du hast den Compiler nicht in dem Directory bzw. Folder aufgerufen, in dem Du das Java-File gespeichert hast.
- Du hast ein Applet als Applikation aufgerufen, oder umgekehrt.
  Applikationen, die eine main-Methode enthalten, musst Du mit
  ```
  java Classname
  ```
  aufrufen.
  Applets, die ein "extends Applet" oder "extends JApplet" enthalten, musst Du innerhalb eines geeigneten HTML-Files mit
  ```
  appletviewer xxxxx.html
  ```
  oder mit Netscape oder Internet-Explorer aufrufen.

3 Syntax und Statements

- **Wie sage ich dem Computer, was er tun soll?**

Die Syntax und die Grundregeln der Sprache Java sind weitgehend identisch mit denen von C und C++. Es fehlen jedoch Sprachelemente, die leicht zu Programmfehlern führen können, wie Pointer-Arithmetik, Operator-Overloading und Goto.

3.1 Grundbegriffe der Programmierung

Für diejenigen, die noch keine Erfahrung mit Programmiersprachen haben, hier eine kurze Zusammenstellung der wichtigsten Elemente, die in Computer-Programmen vorkommen, und der wichtigsten Konzepte für die Programmierung. In den folgenden Abschnitten lernen Sie dann, wie Sie diese Elemente in der Programmiersprache Java formulieren können.

3.1.1 Programm-Elemente

Computer-Programme enthalten die folgenden Elemente:

- Variable = Datenfelder
- Konstante = Datenwerte
- Dateien (Files) = Ein- und Ausgabe von Daten
- Anweisungen (Statements) = Aktionen

Die Anweisungen werden in einer bestimmten Reihenfolge ausgeführt, dabei gibt es auch Abzweigungen (Entscheidungen, z.B. mit if oder switch) und Wiederholungen (Schleifen, z.B. mit for, while, do).

Anweisungen, die gemeinsam ausgeführt werden sollen, werden zu sogenannten Blöcken von Anweisungen zusammengefasst, die wiederum geschachtelt werden können.

Das gesamte Programm besteht meistens aus mehreren Einheiten, die als Unterprogramme (Subroutinen, Prozeduren, Funktionen, Methoden) bezeichnet werden. Die Einheit, mit der die Verarbeitung beginnt, wird als Hauptprogramm (main-Methode) bezeichnet.

3.1.2 Konzepte bei der Programmierung

Beim Design eines komplexen Programmsystems können je nach Zweckmäßigkeit einige der folgenden Konzepte eingesetzt werden:

- **strukturierte Programmierung:** Komplexe Aufgaben werden in möglichst kleine, gut überschaubare Einheiten zerlegt.
- **objekt-orientierte Programmierung:** Die Daten und die mit ihnen ausgeführten Aktionen werden zusammengefasst. Mehr darüber finden Sie im Kapitel über Klassen und Objekte.
- **top-down:** Zuerst überlegt man sich das Grundgerüst, dann die Details.
- **bottom-up:** Zuerst überlegt man sich die Einzelteile, dann erst, wie man sie zum Gesamtsystem

zusammenfügt.

- **model-view-controller** : Bei Graphischen User-Interfaces muss man jeweils die folgenden 3 Aspekte berücksichtigen:
  - die Daten (Datenmodell, model),
  - die Darstellung der Daten (Ansicht, view) und
  - die Ablaufsteuerung (Änderung der Daten, controller)

  Mehr darüber finden Sie im Kapitel über Graphische User-Interfaces.

3.2 Namenskonventionen

Es wird dringend empfohlen, die folgenden Konventionen bei der Wahl von Namen strikt einzuhalten und ausführliche, gut verständliche Namen zu verwenden:

Namen von **Klassen** und Interfaces beginnen mit einem **Großbuchstaben** und bestehen aus Groß- und Kleinbuchstaben und Ziffern. Beispiel: HelloWorld, Button2.

Namen von Konstanten beginnen mit einem Großbuchstaben und bestehen nur aus Großbuchstaben, Ziffern und Underlines. Beispiel: MAX_SPEED.

Alle anderen Namen (**Datenfelder, Methoden, Objekte,** Packages etc.) beginnen mit einem **Kleinbuchstaben** und bestehen aus Groß- und Kleinbuchstaben und Ziffern. Beispiele: openFileButton, button2, addActionListener, setSize, getSize.

Vom System intern generierte Namen können auch Dollar- oder Underline-Zeichen enthalten.

Groß- und Kleinbuchstaben haben in jedem Fall verschiedene Bedeutung (case-sensitive).

3.3 Reservierte Wörter

Die folgenden Namen sind reserviert und dürfen nicht neu deklariert werden:

abstract, boolean, break, byte, byvalue, case, cast, catch, char, class, const, continue, default, do, double, else, extends, false, final, finally, float, for, future, generic, goto, if, implements, import, inner, instanceof, int, interface, long, native, new, null, operator, outer, package, private, protected, public, rest, return, short, static, super, switch, synchronized, this, throw, throws, transient, true, try, var, void, volatile, while.

3.4 Datentypen

Die Datentypen und ihre Eigenschaften (Länge, Genauigkeit) hängen nicht vom jeweiligen Computersystem ab sondern sind in Java einheitlich festgelegt. Es wird zwischen primitiven Datentypen und Objekt-Typen (Klassen) unterschieden.

Primitive Datentypen:

- ganze Zahlen: `byte, short, int, long`
- Gleitkomma-Zahlen: `float, double`
- Logischer Typ (true oder false): `boolean`
- Zeichen (Unicode, Zahlenwert ohne Vorzeichen): `char`

Objekt-Typen:

- Zeichenketten (Unicode): `String`
- sowie alle als Klassen definierten Typen

3.5 Konstante (Literals)

Normale ganze Zahlen haben den Typ int und werden dezimal interpretiert. Werte vom Typ long kann man durch Anhängen von L erzeugen. Oktale Zahlen beginnen mit 0 (Null), hexadezimale mit 0x (Null und x).

Normale Zahlen mit Dezimalpunkt (oder mit E für den Exponent) haben den Typ double. Werte vom Typ float kann man durch Anhängen von F erzeugen.

Logische Konstanten sind die Wörter true und false.

char-Konstanten werden zwischen einfachen Apostrophen geschrieben, Beispiel: ’A’.

String-Konstanten werden zwischen Double-Quotes geschrieben, Beispiel: "Hubert Partl". Double-Quotes innerhalb des Strings müssen mit einem Backslash maskiert werden. Sonderzeichen können mit Backslash und u und dem Unicode-Wert angegeben werden, also z.B. `’\u20ac’` für das Euro-Zeichen.

Beispiele:

```
byte b;       b=123;
short i;      i=-1234;
int i;        i=-1234;
long i;       i=-1234L;
float x;      x=-123.45F;
double x;     x=-123.45;  x=1.0E-23;
boolean b;    b=true;  b=false;
char c;       c=’A’;
String s;     s="Abc";  s=null;
```

3.6 Deklarationen und Anfangswerte

Alle Datenfelder müssen mit Angabe des Datentyps deklariert werden und können dann nur zu diesem Typ passende Werte erhalten. Diese Regel hilft, Tippfehler zu erkennen.

Deklaration eines Datenfeldes:

```
typ name;
```

Deklaration von mehreren Datenfeldern des selben Typs:

```
typ name1, name2, name3;
```

Zuweisung eines Wertes:

```
name = wert;
```

Zuweisung eines Wertes mit Typumwandlung (casting):

```
name = (typ) wert;
```

Deklaration mit Zuweisung eines Anfangswertes:

```
typ name = wert;
```

Achtung! Die Deklaration gilt immer nur für den Bereich (Klasse, Methode, Block innerhalb einer Methode), innerhalb dessen die Deklaration steht. Es kann verwirrend sein, wenn man innerhalb eines Blockes den selben Namen, der bereits für eine globale Variable gewählt wurde, für eine gleichnamige lokale Variable verwendet.

Wenn man vor die Typangabe das Wort `final` setzt, handelt es sich um eine Konstante, d.h. der Wert dieses Datenfeldes kann nicht nachträglich verändert werden.

3.7 Referenzen (reference) auf Objekte oder Strings

Deklaration einer Variablen, die eine **Referenz** auf Objekte einer bestimmten Klasse enthalten kann:

```
ClassName name;
```

```
ClassName name = null;
```

Anlegen eines **Objekts** (Exemplar, Instanz) dieser Klasse und Zuweisung der Referenz auf dieses Objekt an die deklarierte Variable:

```
name = new ClassName();
```

wobei ClassName() für einen Konstruktor (constructor) der Klasse steht.

Deklaration, Anlegen und Zuweisen in einem:

```
ClassName name = new ClassName();
```

Deklaration und Anlegen von Strings [Seite 20] :

```
String s;
String s = null;
s = "Hubert Partl";
String myName = "Hubert Partl";
```

3.8 Garbage Collector

Objekte werden mit dem Operator `new` und dem Konstruktor **explizit** angelegt und dabei der Speicherplatz für das Objekt reserviert und mit den Anfangswerten belegt.

Sobald es keine Referenz mehr auf das Objekt gibt (z.B. durch Zuweisen von `null` auf die einzige Referenz-Variable), wird der vom Objekt belegte Speicherplatz durch den im Hintergrund laufenden Garbage-Collector **automatisch** zurückgegeben und für neuerliche Verwendung frei gemacht.

3.9 Felder (array)

Unter einem Feld (array) versteht man eine Menge von Datenfeldern des gleichen Typs. Felder eignen sich besonders für die Verarbeitung mit for-Schleifen (siehe unten).

Deklaration einer Variablen, die eine Referenz auf ein Feld enthalten kann:

```
typ[] name;
```

Anlegen eines Feldes von n Elementen dieses Typs und Zuweisung der Referenz auf dieses Feld an die Referenz-Variable:

```
name = new typ [n];
```

Deklaration und Anlegen in einem:

```
typ[] name = new typ [n];
```

Beispiel:

```
double[] monatsUmsatz = new double[12];
...
```

Eine Array-Referenz kann nacheinander Felder verschiedener Länge (aber nur gleichen Typs) enthalten. Beispiel:

```
double[] tagesUmsatz;
...
tagesUmsatz = new double[31];
...
tagesUmsatz = new double[28];
...
```

Zuweisen von Werten zu den Feldelementen:

```
for (int i=0; i<name.length; i++) {
  name[i] = wert;
}
```

Achtung! Die Länge des Feldes ist n, die Indizes der Feldelemente laufen aber von 0 bis n-1. Die Länge eines Feldes erhält man mit

```
name.length
```

Bitte, beachten Sie, dass es hier *nicht* length() heißt.

Deklaration, Anlegen und Zuweisung von Anfangswerten in einem:

```
int[] daysPerMonth =
  { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

legt ein Feld von 12 int-Zahlen an und belegt es mit diesen Werten.

```
String[] weekDay =
  { "So", "Mo", "Di", "Mi", "Do", "Fr", "Sa" };
```

legt ein Feld von 7 String-Referenzen an und belegt es mit diesen Strings.

3.9.1 Feld von Objekten

```
Classname[] name = new Classname [n];
for (int i=0; i<name.length; i++) {
  name[i] = new Classname();
}
```

Beispiel:

```
int maxAnzahl = 20;
int anzahl=0;
Person[] teilnehmer = new Person [maxAnzahl];
for (int i=0; i<teilnehmer.length; i++) {
  teilnehmer[i] = null;
}
...
anzahl = anzahl + 1;
teilnehmer[ anzahl-1 ] = new Person ("Hubert","Partl");
...
```

3.9.2 Zweidimensionales Feld der Größe n mal m

```
typ[][] name = new typ [n][];
for (int i=0; i<name.length; i++) {
  name[i] = new typ[m];
  for (int j=0; j<name[i].length; j++) {
    name[i][j] = wert;
  }
}
```

Die n Teilfelder können aber auch verschiedene Längen haben, die dann nicht in der i-Schleife sondern einzeln initialisiert werden. Beispiel:

```
double[][] tagesUmsatz = new double [12][];
int[] monatsLaenge = { 31,29,31,30,31,30,31,31,30,31,30,31 };
for (int monat=0; monat<tagesUmsatz.length; monat++) {
  tagesUmsatz[monat] = new double[ monatsLaenge[monat] ];
  for (int tag=0; tag<tagesUmsatz[monat].length; tag++) {
    tagesUmsatz[monat][tag] = 0.0;
  }
}
```

3.9.3 Vector, Hashtable, Collection

Als Ergänzung zu den hier beschriebenen Feldern (array) gibt es auch eine Klasse "Vector", mit der man dynamisch wachsende Speicherbereiche für beliebige Objekte anlegen kann, sowie eine Klasse "Hashtable". Ab JDK 1.2 gibt es noch mehr solche Klassen wie z.B. Collection, List, Map, Set, ArrayList, LinkedList, HashMap, TreeMap, ListModel, TableModel, TreeModel. Details finden Sie jeweils in der Online-Dokumentation (API).

3.10 Ausdrücke (expression) und Operatoren

Objekt-Operatoren: `new .`

Mathematische Operatoren: `+ - * / %`

Mathematische Zuweisungen: `++ -- = *= /= %= += -=`

Logische Operatoren: `< > <= >= == != instanceof ! && ||`

Bit-Operatoren: `<< >> >>> & |`

Bit-Zuweisungen: `<<= >>= >>>= &= |=`

String-Operatoren: `+ equals`

Achtung! Bei Referenzen auf Objekte oder Strings vergleichen die Operatoren `==` und `!=` nur, ob die Referenz die selbe ist, nicht, ob der Inhalt des Strings oder des Objekts gleich ist. Dafür muss die equals-Methode des Objekts verwendet werden. Beispiel:

```
String vorName;
...
if ( vorName.equals("Hubert") )
...
```

3.10.1 Typ-Umwandlungen (Casting)

In manchen Fällen kann der Compiler Typen automatisch umwandeln, z.B. bei der Zuweisung von int auf long oder double:

```
int i = ...;
double d = i + 1;
```

In vielen Fällen muss man dem Compiler aber mitteilen, in welchen Datentyp ein Wert umgewandelt werden soll. Dies wird als Casting bezeichnet, dazu wird der neue Typ zwischen Klammern angegeben:

```
typ1 name1 = ...;
typ2 name2 = (typ2) name1;
```

Dies ist inbesondere dann wichtig, wenn man sicherstellen will, dass eine Multiplikation mit der vollen Genauigkeit von long oder double erfolgt statt mit der geringeren Genauigkeit von int oder float, oder dass eine Division mit Bruchteilen (float oder double) statt nur ganzzahlig mit Abschneiden des Restes erfolgt. Beispiel:

```
int i = ... ;
double d = ( (double) i ) / 100.0 ;
```

Für die Umwandlung von float oder double auf int oder long gibt es die Methoden Math.floor, Math.ceil, Math.round, je nachdem, ob man abrunden, aufrunden oder kommerziell runden will.

3.11 Text-Operationen (String)

Die Länge eines Strings erhält man mit der Methode

- `length()`

Beispiel:

```
String name="Hubert Partl";
int len=name.length();
```

Für das Zusammenfügen von mehreren Strings zu einem längeren String gibt es den Operator `+`. Beispiel:

```
String name = vorName + " " + zuName;
```

Für die Abfrage auf Gleichheit gibt es die Methode `equals` (siehe oben [Seite 19]). Beispiel:

```
if (vorName.equals("Hubert") ...
```

Außerdem enthält die Klasse String zahlreiche weitere Methoden für das Erzeugen von Substrings oder von veränderten Strings (z.B. Zeichenersetzung, Umwandlung auf Groß- oder Kleinbuchstaben), sowie für die Abfrage von Zeichen und Substrings innerhalb eines Strings und für die alphabetische Reihenfolge von 2 Strings. Die Details entnehmen Sie bitte der Online-Dokumentation der Klasse String.

Für komplexere Anwendungen, bei denen ein Text-String in Einzelteile (z.B. Wörter) zerlegt werden soll, gibt es eine eigene Klasse StringTokenizer.

Für die Umwandlung von Strings, die Ziffern enthalten, in entsprechende Zahlenwerte gibt es eigene statische Methoden [Seite 36] wie z.B.

```
int    Integer.parseInt(String)
long   Long.parseLong(String)
float  Float.valueOf(String).floatValue()
double Double.valueOf(String).doubleValue()
```

Ab JDK 1.2 wird das einheitlicher, dann gibt es auch

```
float  Float.parseFloat(String)
double Double.parseDouble(String)
```

Wie Sie die dabei eventuell auftretenden Fehler abfangen müssen, ist im Kapitel über Exceptions [Seite 55] beschrieben.

Für die umgekehrte Umwandlung von Zahlen in Strings kann man die Methode

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

oder auch den folgenden Trick verwenden:

```
String s = "" + zahl;
```

3.12 Funktionen (Math)

Mathematische Funktionen können mit Hilfe der statischen Methoden in der Klasse Math aufgerufen werden. Beispiel:

```
x = Math.sqrt(y);
```

Die Details finden Sie in der Online-Dokumentation (API) [Seite 8] . Mehr über Methoden [Seite 34] und statische Methoden [Seite 36] finden Sie im Kapitel über Objekte und Klassen [Seite 31] .

3.13 Statements und Blöcke

Statements werden mit `;` (Strichpunkt, Semicolon) beendet.

Blöcke von Statements werden zwischen `{` und `}` (geschwungene Klammern, braces) eingeschlossen und können leer sein oder ein oder mehrere Statements enthalten, die jeweils mit `;` beendet werden.

3.14 Kommentare

Alles, was zwischen `/*` und `*/` oder zwischen `//` und dem Zeilenende steht, wird vom Java-Compiler ignoriert und gilt als Kommentar.

Beispiele:

```
...  // Kommentar bis Zeilenende
```

```
... /* Kommentar */ ...
```

Mit `/**` in der ersten Zeile, `*` am Beginn aller weiteren Zeilen und `*/` am Ende kann man einen Dokumentations-Kommentar schreiben, der dann mit Hilfsprogrammen extrahiert und ausgedruckt werden kann (siehe javadoc [Seite 28]).

3.15 if und else

einfache Abzweigung (Entscheidung der Form wenn - dann):

```
if ( logischer Ausdruck ) {
  Statements;
}
```

doppelte Verzweigung (wenn - dann - sonst):

```
if ( logischer Ausdruck ) {
  Statements;
}
else {
  Statements;
}
```

Beispiel:

```
    if ( guthaben >= 1000000 ) {
      System.out.println("Gratuliere, Du bist Millionaer!");
    }
```

3.16 for

Wiederholung (Schleife) mit bestimmter Anzahl:

```
for ( int name=Anfangswert ; logischer Ausdruck ; Wertänderung ) {
  Statements;
}
```

Beispiel:

```
    for ( int monat=1;  monat<=6; monat = monat + 1 ) {
      guthaben = guthaben + einzahlung;
    }
```

3.17 while und do

Wiederholung (Schleife) mit beliebiger Bedingung:

```
while ( logischer Ausdruck ) {
  Statements;
}
```

Wiederholung (Schleife) mit mindestens einem Durchlauf:

```
do {
  Statements;
} while ( logischer Ausdruck );
```

Beispiel:

```
    while ( guthaben < sparziel ) {
      guthaben = guthaben + einzahlung;
    }
```

3.18 switch und case

mehrfache Verzweigung (verschiedene Fälle):

```
switch ( ganzzahliger Ausdruck ) {
  case wert1:
    Statements;
    break;
  case wert2:
    Statements;
    break;
  ...
  default:
    Statements;
}
```

Beispiel:

```
    switch (note) {
      case 1:
        System.out.println("sehr gut");
        break;
      case 2:
        System.out.println("gut");
        break;
      ...
    }
```

Achtung! Falls die auf eine case-Angabe folgenden Statements nicht mit `break;` enden oder dort überhaupt keine Statements stehen, werden auch die bei der folgenden case-Angabe stehenden Statements ausgeführt. Beispiel:

```
    switch (firstChar) {
      case 'j':
      case 'J':
        // ja ...
        break;
      case 'n':
      case 'N':
        // nein ...
        break;
    }
```

3.19 break und continue

Mit `continue;` kann man den aktuellen Schleifendurchgang vorzeitig beenden und den nächsten Schleifendurchgang beginnen.

Mit `break;` kann man eine Schleife oder einen case-Block verlassen und zum Statement nach Ende der Schleife bzw. des switch-Blockes springen.

Mit labelname`:` kann man einen Label vor ein for-, while- oder do-Statement setzen und dann mit `continue` labelname`;` bzw. `break` labelname`;` diese (äußere) Schleife vorzeitig beenden.

Es gibt in Java *kein* goto-Statement zum Sprung an beliebige Stellen.

3.20 System.exit

Bei Erreichen des Endes der main-Methode wird das Programm automatisch beendet (außer wenn noch weitere Threads laufen, siehe die Kapitel über Graphische User-Interfaces und über Threads).

Mit `System.exit(0);` oder `System.exit(n);` kann man die Applikation vorzeitig beenden.

Bei erfolgreicher Beendigung gibt man als Exit-Code 0 an, bei einer nicht erfolgreichen Beendigung eine positive ganze Zahl als Fehler-Code, im einfachsten Fall den Wert 1.

3.21 Beispiele für einfache Programme

3.21.1 Wertzuweisungen

```
public class ProgExpr {
  public static void main (String[] args) {
    int guthaben = 1000;
    System.out.println("Guthaben   = " + guthaben);
    int einzahlung = 500;
    System.out.println("Einzahlung = " + einzahlung);
    guthaben = guthaben + einzahlung;
    System.out.println("Guthaben   = " + guthaben);
  }
}
```

3.21.2 if

```
public class ProgIf {
  public static void main (String[] args) {
    int guthaben = 2000000;
    System.out.println("Guthaben   = " + guthaben);
    if ( guthaben >= 1000000 ) {
      System.out.println("Gratuliere, Du bist Millionaer!");
    }
  }
}
```

3.21.3 for

```
public class ProgFor {
  public static void main (String[] args) {
    int guthaben = 1000;
    int einzahlung = 500;
    System.out.println("Guthaben   = " + guthaben);
    System.out.println("Einzahlung = " + einzahlung
      + " pro Monat");
    for ( int monat=1;  monat<=6; monat = monat + 1 ) {
      guthaben = guthaben + einzahlung;
      System.out.println(monat + ". Monat:");
      System.out.println("  Guthaben = " + guthaben);
    }
  }
}
```

3.21.4 while

```
public class ProgWhile {
  public static void main (String[] args) {
    int guthaben = 1000;
    int sparziel = 8000;
    int einzahlung = 600;
    System.out.println ("Guthaben   = " + guthaben);
    System.out.println ("Sparziel   = " + sparziel);
    while ( guthaben < sparziel ) {
      guthaben = guthaben + einzahlung;
      System.out.println ("neues Guthaben = " + guthaben);
    }
    System.out.println( "Sparziel erreicht.");
  }
}
```

3.21.5 switch

```
public class ProgSwitch {
  public static void main (String[] args) {
    int note = 2;

    switch (note) {
      case 1:
        System.out.println("sehr gut");
        break;
      case 2:
        System.out.println("gut");
        break;
      case 3:
        System.out.println("befriedigend");
        break;
      case 4:
        System.out.println("genuegend");
        break;
      case 5:
        System.out.println("nicht genuegend");
        break;
    }

    switch (note) {
      case 1:
        System.out.print("mit Auszeichnung ");
```

```
      case 2:
      case 3:
      case 4:
        System.out.println("bestanden");
        break;
      case 5:
        System.out.println("nicht bestanden");
        break;
    }
  }
}
```

3.21.6 Blöcke

```
public class ProgBlock {
  public static void main (String[] args) {
    int note;
    String beurteilung;
    System.out.println ("Notentabelle:");
    for ( note = 1; note <= 5; note++ ) {
      switch (note) {
        case 1:
          beurteilung = "sehr gut";
          break;
        case 2:
          beurteilung = "gut";
          break;
        case 3:
          beurteilung = "befriedigend";
          break;
        case 4:
          beurteilung = "genuegend";
          break;
        case 5:
          beurteilung = "nicht genuegend";
          break;
        default:
          beurteilung = "keine Note";
      } /* end switch */
      System.out.println (note + " = " + beurteilung);
    } /* end for */
  }
}
```

3.21.7 Ein- und Ausgabe

Hier noch ein relativ einfaches Beispiel für die zeilenweise Ein- und Ausgabe. Die darin verwendeten Sprachelemente werden allerdings erst in späteren Kapiteln erklärt (Packages [Seite 54] , Exceptions [Seite 55] , Ein-Ausgabe [Seite 112]).

```
import java.io.*;
public class ProgIO {
  public static void main (String[] args) {
    try {
      String zeile, vorName;
      int alter;
      BufferedReader infile =
        new BufferedReader ( new InputStreamReader (System.in) );
      // String lesen
      System.out.println ("Bitte gib Deinen Vornamen ein:");
```

```
      vorName = infile.readLine();
      System.out.println ("Hallo " + vorName + "!");
      // Zahl lesen
      System.out.println ("Bitte gib Dein Alter ein:");
      zeile = infile.readLine();
      alter = Integer.parseInt ( zeile.trim() );
      System.out.println ("Du bist " + alter + " Jahre alt.");
    } catch (Exception e) {
      System.out.println("falsche Eingabe - " + e);
    }
  }
}
```

3.22 Fehlersuche und Fehlerbehebung (Debugging)

Computer sind unglaublich dumme Geräte,
die unglaublich intelligente Sachen können.
Programmierer sind unglaublich intelligente Leute,
die unglaublich dumme Sachen produzieren.
("Die Presse", 30.8.1999)

Programmfehler (auf englisch bugs = Wanzen, Ungeziefer genannt) gehören zum täglichen Brot jedes Programmierers. Nur wer nichts arbeitet, macht keine Fehler. Der gute Programmierer ist nicht der, der keine Fehler macht, sondern der, der seine Fehler rasch findet und behebt.

Es gibt eigene Software-Tools, die bei der Fehlersuche helfen können. In den meisten Fällen genügt es aber, an allen wichtigen Stellen im Programm mit System.out.println oder System.err.println Informationen über das Erreichen dieser Programmstelle und über den Inhalt von wichtigen Variablen auszugeben. Damit kann man dann meistens sehr rasch finden, wo, wann und warum ein Programmfehler aufgetreten ist und was man tun muss, um ihn zu vermeiden. Beispielskizze:

```
...
System.out.println("* Beginn der Schleife");
System.out.println("n = " + n);
for (int i=0; i<=n; i++) {
  System.out.println("* i = " + i + ", n = " + n);
  ...
}
...
```

Die mit System.out.println oder System.err.println geschriebenen Test-Informationen erscheinen bei Applikationen im Ausgabe-Fenster und bei Applets in der "Java-Console" des Web-Browsers. Wenn man die Test-Ausgaben von den normalen Ausgaben einer Applikation trennen will, dann schreibt man nur die normalen Ausgaben (Programmergebnisse) auf System.out und die Test-Ausgaben auf System.err.

Wenn man die Fehler behoben hat und das Programm im Produktionsbetrieb verwenden will, muss man die Test-Ausgaben deaktivieren. Wenn man das Programm aber später verbessert oder erweitert (und das wird bei jedem brauchbaren Programm früher oder später notwendig), dann muss man sie wiederum aktivieren. Zu diesem Zweck verwendet man eine boolean Variable TEST oder DEBUG, die man in der Testversion auf true und in der Produktionsversion auf false setzt, und macht die Ausgabe der Testinformationen immer von dieser Variable abhängig. Wenn man diese Variable als static final definiert, dann kann der Compiler das Programm so optimieren, dass die Programmstücke,

die nur für das Testen dienen, beim Übersetzen mit TEST = false komplett weggelassen werden und weder Speicherplatz noch Abfragezeit kosten. Beispielskizze:

```
public class Classname {
  private static final boolean TEST = false;
  ...
  public ... methodName () {
     ...
     if (TEST) {
       System.err.println( ... test information ... );
     }
     ...
  }
}
```

Wenn man ein Paket von mehreren Klassen testen will, kann es günstiger sein, alle Definitionen in *einer* eigenen Klasse (Defined) festzulegen und dann in *allen* Klassen in der Form Defined.TEST zu verwenden. Beispielskizze:

```
public class Defined {
  public static final boolean TEST = false;
  public static final boolean XXXX = true;
  ...
}

public class Classname {
     ...
     if (Defined.TEST) {
       ...
     }
     ...
}
```

Außerdem ist es für das Verstehen und die (eventuell erst Jahre später erfolgende) Wartung der Programme wichtig, mit Kommentaren [Seite 21] innerhalb der Programme möglichst ausführliche Hinweise auf den Zweck und die Funktionsweise des Programmes und der einzelnen Programmabschnitte sowie Erklärungen zu allen programmtechnischen Tricks anzugeben.

3.23 Dokumentation von Programmen (javadoc)

Mit dem Hilfsprogramm javadoc, das Teil des JDK [Seite 5] ist, kann eine Dokumentation der Java-Programme im HTML-Format erzeugt werden, so wie die Online-Dokumentation (API) [Seite 8] der Klassenbibliothek.

Der Aufruf von
`javadoc Xxxx.java`
für einzelne Source-Programme oder
`javadoc *.java`
für alle Java-Programme im aktuellen Directory erzeugt für jedes Source-Programm ein Dokumentations-File Xxxx.html sowie (bis JDK 1.1) einen Package-Index packages.html, einen Klassen-Index tree.html und einen Namens-Index AllNames.html bzw. ab JDK 1.2 eine größere Anzahl von solchen HTML-Files, die von einem Start-File index.html ausgehen. Vorsicht, bereits existierende gleichnamige Dateien werden dabei überschrieben.

Javadoc extrahiert automatisch alle public [Seite 39] und protected Deklarationen von Klassen [Seite 32] , Datenfeldern, Konstruktoren und Methoden. Zusätzliche Informationen kann man in Form von speziellen Kommentaren hinzufügen, die jeweils in
`/** ... */`
eingeschlossen werden, *vor* der jeweiligen Deklaration stehen und einfachen HTML-Text enthalten.

Beispiel:

```
/**
* ein einfaches Hello-World-Programm.
* <p>
* Im Gegensatz zum kurzen, rein statischen "HelloWorld"
* ist dieses Programm ein Musterbeispiel
* f&uuml;r eine <b>objekt-orientierte</b> Java-Applikation.
*
* @author Hubert Partl
* @version 99.9
* @since JDK 1.0
* @see HelloWorld
*/
public class HelloDoc {

  /** der Text, der gedruckt werden soll.
  */
  public String messageText = "Hello World!";

  /** druckt den Text messageText auf System.out aus.
  *   @see #messageText
  */
  public void printText() {
    System.out.println (messageText);
  }

  /** Test des HelloDoc-Objekts.
  */
  public static void main (String[] args) {
    HelloDoc h = new HelloDoc();
    h.printText();
  }
}
```

Die von javadoc erstellte Dokumentation finden Sie im File HelloDoc.html.

3.24 Übung: einfaches Rechenbeispiel Quadratzahlen

Schreiben Sie eine einfache Applikation, in der Sie die Quadratzahlen von ein paar ganzen Zahlen berechnen und ausgeben:

1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
usw. bis
20 * 20 = 400

3.25 Übung: Rechenbeispiel Steuer

Der Bruttopreis ist mit 100 Euro inkl. 20 % USt. angegeben. Wieviel macht die Umsatzsteuer aus? Wie hoch ist der Nettopreis?

Anmerkungen: Der Steuerbetrag ist
brutto * prozentsatz / (100 + prozentsatz)
Der Nettopreis ist der Bruttopreis minus den Steuerbetrag.

3.26 Übung: einfaches Rechenbeispiel Sparbuch

Schreiben Sie eine Applikation, die für einen bestimmten Geldbetrag und einen bestimmten Zinssatz ausgibt, wie sich dieser Wert Jahr für Jahr erhöht (für 10 Jahre, jeweils Jahr und Wert).

Anmerkung: Der Wert erhöht sich jedes Jahr um den Faktor
(100.0 + zinssatz) / 100.0

Erweiterte Versionen dieses Beispiels als Applikation mit Fehlerbehandlung [Seite 58] und als Applet mit graphischem User-Interface [Seite 93] folgen in den Kapiteln über Fehlerbehandlungen [Seite 55] und Applets [Seite 84] .

4 Objekte und Klassen

In der prozeduralen Programmierung werden Daten und Prozeduren separat definiert. In der objekt-orientierten Programmierung werden die Eigenschaften und Aktionen von Objekten zusammengefasst. Welche Vorteile hat das für den Programmierer?

4.1 Objekt-Orientierung

- **Objekt = (Daten & Aktionen)**

Bei der konventionellen Programmierung erfolgt die Definition der Datenstrukturen und der mit den Daten ausgeführten Prozeduren (Aktionen) unabhängig voneinander. Das Wissen um die Bedeutung der Daten und die zwischen ihnen bestehenden Beziehungen ist nicht einfach bei den Daten sondern mehrfach in allen Programmen, die auf diese Daten zugreifen, gespeichert.

Bei der objekt-orientierten Programmierung (OOP) werden Objekte ganzheitlich beschrieben, d.h. die Festlegung der Datenstrukturen und der mit den Daten ausgeführten Aktionen erfolgt in einem.

Die wichtigsten Vorteile der objekt-orientierten Programmierung sind:

- Aufspaltung von komplexen Software-Systemen in kleine, einfache, in sich geschlossene Einzelteile,
- einfache und klar verständliche Schnittstellen zwischen den einzelnen Komponenten,
- weitgehende Vermeidung von Programmierfehlern beim Zusammenspiel zwischen den Komponenten,
- geringer Programmieraufwand durch die Wiederverwendung von Elementen (Reusability, siehe auch Vererbung [Seite 47]).

Je kleiner und einfacher die Objekte und Klassen [Seite 32] und die Schnittstellen zwischen ihnen gewählt werden (Kapselung [Seite 39] , Vererbung [Seite 47]), desto besser werden diese Ziele erreicht. Mehr darüber folgt später im Kapitel über Objekt-orientierte Analyse und Design [Seite 45] .

4.1.1 prozeduraler Programmaufbau (Beispiel COBOL)

```
DATA DIVISION.
01  DDD1.
     ... Datenblock 1 ...
01  DDD2.
     ... Datenblock 2 ...
...
PROCEDURE DIVISON.
AAA1 SECTION.
    ... Aktionen Teil 1 ...
AAA2 SECTION.
    ... Aktionen Teil 2 ...
...
```

4.1.2 prozeduraler Programmaufbau (Beispiel C)

```
struct ddd1 {
   ... Datenblock 1 ...
}
struct ddd2 {
   ... Datenblock 2 ...
}
...
aaa1() {
   ... Aktionen Teil 1 ...
}
aaa2() {
   ... Aktionen Teil 2 ...
}
...
```

4.1.3 objekt-orientierter Programmaufbau (Beispiel Java)

```
public class Ccc1 {
   ... Daten von Objekt-Typ 1 ...
   ... Aktionen von Objekt-Typ 1 ...
}

public class Ccc2 {
   ... Daten von Objekt-Typ 2 ...
   ... Aktionen von Objekt-Typ 2 ...
}

...
```

4.2 Klassen (class)

Als **Klasse** (class) bezeichnet man die Definition einer Idee, eines Konzepts, einer Art von Objekten.

Als **Objekt** (object), Exemplar oder Instanz (instance) bezeichnet man eine konkrete Ausprägung eines Objekts, also ein Stück aus der Menge der Objekte dieser Klasse.

Beispiele: Hubert Partl und Monika Kleiber sind Objekte der Klasse Mitarbeiter. Die Universität für Bodenkultur ist ein Objekt der Klasse Universität.

Je einfacher die Klassen gewählt werden, desto besser. Komplexe Objekte können aus einfacheren Objekten zusammengesetzt werden (z.B. ein Bücherregal enthält Bücher, ein Buch enthält Seiten, ein Atlas enthält Landkarten). Spezielle komplizierte Eigenschaften können auf grundlegende einfache Eigenschaften zurückgeführt werden (Vererbung [Seite 47] , z.B. ein Atlas ist eine spezielle Art von Buch).

4.2.1 Definition von Klassen - Aufbau von Java-Programmen

Java-Programme sind grundsätzlich Definitionen von **Klassen**. Sie haben typisch den folgenden Aufbau:

```
public class ClassName {
  // Definition von Datenfeldern
  // Definition von Konstruktoren
  // Definition von Methoden
}
```

Die Reihenfolge der Definitionen innerhalb der Klasse ist grundsätzlich egal, aber ich empfehle, immer eine Konvention wie z.B. die hier angeführte Reihenfolge einzuhalten.

4.2.2 Verwendung von Klassen - Anlegen von Objekten

Objekte werden in Java-Programmen mit dem Operator `new` und einem Konstruktor [Seite 37] der Klasse angelegt. Beispiel:

```
House myNewHome = new House();
```

Innerhalb der Klasse kann man das aktuelle Objekt dieser Klasse mit dem symbolischen Namen `this` ansprechen.

4.3 Beispiel: objekt-orientierte HelloWorld-Applikation

```
public class HelloText {

  public String messageText = "Hello World!";
  // or: private ...

  public void printText() {
    System.out.println (messageText);
  }

  public static void main (String[] args) {
    HelloText h = new HelloText();
    h.printText();
  }
}
```

Im Gegensatz zum statischen [Seite 10] Hello-World-Programm lässt sich das objekt-orientierte Programm leicht erweitern. Beispiel:

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

    HelloText engl = new HelloText();
    HelloText germ = new HelloText();
    germ.messageText = "Hallo, liebe Leute!";
    HelloText cat = new HelloText();
    cat.messageText = "Miau!";

    engl.printText();
    germ.printText();
    engl.printText();
    cat.printText();
    engl.printText();
  }
}
```

Hier werden drei Objekte der Klasse HelloText erzeugt, mit verschiedenen Texten, und dann erhalten diese Objekte die "Aufträge", ihren Text auszugeben ("Delegation"). Die Ausgabe sieht dann so aus:

```
Hello World!
Hallo, liebe Leute!
Hello World!
Miau!
Hello World!
```

Anmerkung: Eine weitere Verbesserung dieses HelloWorld-Programmes folgt im Kapitel Kapselung [Seite 39] .

4.4 Datenfelder (member field)

Datenfelder der Klasse werden typisch in der folgenden Form deklariert:

```
public typ name;
```

```
public typ name = anfangswert;
```

Nachdem ein Objekt dieser Klasse angelegt wurde, können seine Datenfelder in der Form

```
object.name
```

angesprochen werden. Wenn `private` statt `public` angegeben wurde, können sie jedoch von anderen Klassen nicht direkt angesprochen werden (siehe Kapselung [Seite 39]).

Datenfelder der Klasse sind eine Art von "globalen Variablen" für alle Methoden [Seite 34] der Klasse und für eventuelle innere Klassen [Seite 51] .

Innerhalb der Klasse kann man die Datenfelder einfach mit

```
name
```

ansprechen, oder in der Form

```
this.name
```

wenn eine Unterscheidung von gleichnamigen lokalen Variablen oder Parametern notwendig ist.

4.5 Methoden (method)

Methoden entsprechen den Funktionen oder Prozeduren in konventionellen Programmiersprachen. Die Deklaration erfolgt in der folgenden Form:

Methode mit Rückgabewert:

```
public typ name () {
   Statements;
   return wert;
}
```

Methode ohne Rückgabewert:

```
public void name () {
   Statements;
}
```

Methode mit Parametern:

```
public typ name (typ name, typ name, typ name) {
   Statements;
   return wert;
}
```

Mit `return;` bzw. `return wert;` innerhalb der Statements kann man die Methode vorzeitig verlassen (beenden).

Nachdem ein Objekt der Klasse angelegt wurde, können seine Methoden in einer der folgenden Formen ausgeführt werden:

```
x = object.name();
```

```
object.name();
```

```
x = object.name (a, b, c);
```

```
object.name (a, b, c);
```

Man spricht in diesem Fall vom Aufruf der Methode für das Objekt oder vom Senden einer Message an das Objekt. Wenn `private` statt `public` angegeben wurde, kann die Methode von anderen Klassen nicht direkt angesprochen werden (siehe Kapselung [Seite 39]).

Innerhalb der Klasse kann man die Methoden mit

```
name (parameterliste)
```

oder

```
this.name (parameterliste)
```

aufrufen.

Lokale Variable, die nur innerhalb einer Methode deklariert sind, können von außen nicht angesprochen werden. Bei der Deklaration von lokalen Variablen darf daher (im Gegensatz zu Datenfeldern der Klasse) weder public noch private angegeben werden.

Die Parameter werden grundsätzlich als Wert ("by value"), nicht als Name ("by reference") übergeben. **Primitive** Datentypen, die innerhalb der aufgerufenen Methode verändert werden, behalten also in der aufrufenden Methode ihren alten Wert. Bei **Objekten** und Strings wird aber nur die Referenz als Wert übergeben, nicht das Objekt selbst. Wenn das Objekt innerhalb der aufgerufenen Methode mit Hilfe dieser Referenz verändert wird, dann wirkt diese Änderung direkt auf das Objekt, also praktisch wie bei einer Übergabe als Name.

4.6 Überladen (overload)

Man kann auch mehrere Methoden definieren, die den selben Namen, aber verschiedene Parameterlisten haben. Dies wird als Überladen (overloading) von Methoden auf Grund ihrer verschiedenen "Signatures" bezeichnet. Unter der Signatur einer Methode versteht man die eindeutige Kombination von Name und Parameterliste.

Man sollte das aber nur dann tun, wenn diese Methoden ähnliche Aktionen ausführen, die sich nur durch den Typ oder die Anzahl der Parameter unterscheiden.

So sind z.B. verschiedene Varianten der Methode `println` definiert, die je nach dem Typ des Parameters dessen Wert in einen lesbaren String umwandeln und dann diesen ausdrucken.

4.7 statische Methoden (static)

Wenn man vor dem Typ einer Methode das Wort `static` angibt, dann kann diese Methode aufgerufen werden, ohne dass vorher ein Objekt der Klasse erzeugt werden muss. Der Aufruf erfolgt dann in der Form

```
ClassName.name();
```

Statische Methoden können freilich nicht auf (nicht-statische) Datenfelder oder nicht-statische Methoden der Klasse oder auf "this" zugreifen. Innerhalb von statischen Methoden können aber Objekte mit new angelegt und dann deren Datenfelder und Methoden angesprochen werden.

Die beiden wichtigsten Anwendungsfälle für statische Methoden sind

- die main-Methode [Seite 37] , mit der die Ausführung einer Java-Applikation beginnt, und
- mathematische Funktionen [Seite 21] , die nicht mit Objekten sondern mit primitiven Datentypen [Seite 14] arbeiten.

Beispielskizze für die Definition und Verwendung einer mathematischen Funktion:

```
public class Classname {
  public static int diff (int a, int b) {
    int c;
    c = a - b;
    return c;
  }
  ...
}
```

Der Aufruf erfolgt dann in der Form

```
z = Classname.diff (x, y);
```

Beispiele für statische Variable und statische Methoden sind System.out, System.exit(0), Label.CENTER, Math.sin(x) usw.

4.8 main-Methode

Jede Java-Applikation muss eine statische Methode [Seite 36] mit der folgenden Signature enthalten:

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

Diese main-Methode wird beim Starten der Applikation mit dem Befehl java [Seite 7] aufgerufen und kann ein oder mehrere Objekte der Klasse oder auch von anderen Klassen anlegen und verwenden. Der Parameter `args` enthält die beim Aufruf eventuell angegebenen Parameter, mit `args.length` erhält man deren Anzahl.

4.9 Konstruktoren (constructor)

Konstruktoren dienen zum Anlegen von Objekten der Klasse mit dem `new`-Operator. Sie legen den für das Objekt benötigten Speicherplatz an und initialisieren das Objekt, d.h. sie setzen es in einen gültigen Anfangszustand.

Laut Tov Are Jacobsen erfüllt der Konstruktor die Funktion eines Baumeisters, der ein Objekt mit den im Plan festgelegten Eigenschaften errichtet:

> When you write a class you describe the behaviour of potential objects. Much like designing a house on paper: you can't live in it unless you construct it first.
>
> And to do that you say "I want a **new** house, so I'll call the **constructor**":
>
> ```
> House myNewHome;
> myNewHome = new House();
> ```
>
> (Tov Are Jacobsen, comp.lang.java.help, 1998)

Im einfachsten Fall enthält die Definition der Klasse **keine** explizite Definition von Konstruktoren. Dann wird vom Java-Compiler ein Default-Konstruktor mit leerer Parameterliste erzeugt, der in der Form

```
ClassName object = new ClassName();
```

aufgerufen werden kann und nur den Speicherplatz anlegt und alle Datenfelder [Seite 34] auf ihre Anfangswerte setzt.

Wenn man will, dass im Konstruktor weitere Aktionen erfolgen sollen (z.B. durch Aufruf von Methoden), so muss man den Konstruktor selbst definieren. Dies erfolgt in der Form

```
public ClassName() {
  Statements;
}
```

Man kann auch einen oder mehrere Konstruktoren mit verschiedenen Parameterlisten definieren (Overloading). In diesem Fall wird vom Compiler *kein* Default-Konstruktor automatisch erzeugt: Entweder man definiert gar keinen Konstruktor oder man definiert alle explizit. Beispiel:

```
public class BookShelf {
  public Book[] book;

  public BookShelf (int maxBooks) {
     book = new Book[maxBooks];
     for (int i=0; i<book.length; i++)
       book[i]=null;
  }

  // methods for adding and removing books ...
}
```

In diesem Fall muss man im Konstruktor die Maximalanzahl angeben, es gibt keinen Default-Konstruktor ohne Parameter. Beispiele:

```
BookShelf smallShelf = new BookShelf(20);
BookShelf bigShelf   = new BookShelf(60);
```

Wenn man auch einen Default-Konstruktor haben will, was in den meisten Fällen sinnvoll ist, dann muss man ihn explizit zusätzlich angeben.

Dabei kann ein Konstruktor auch einen anderen Konstruktor mit einer entsprechenden Parameterliste aufrufen, indem man das Wort `this` für den Aufruf des Konstruktors angibt. Beispiel:

```
public class BookShelf {
  public Book[] book;

  public BookShelf() {
    this(20);
  }

  public BookShelf (int maxBooks) {
     book = new Book[maxBooks];
     for (int i=0; i<book.length; i++)
       book[i]=null;
  }

  // methods for adding and removing books ...
}
```

In diesem Fall kann man mit

```
BookShelf smallShelf = new BookShelf();
```

ein Regal mit der Standardgröße für maximal 20 Bücher anlegen und mit

```
BookShelf bigShelf = new BookShelf(60);
```

ein größeres Regal für maximal 60 Bücher.

4.10 Kapselung (Encapsulation, Data Hiding, Java Beans)

Statt der Angabe von public in den obigen Beispielen sind bei Klassen [Seite 32] , Datenfeldern [Seite 34] und Methoden [Seite 34] jeweils vier verschiedene Angaben möglich:

- **public**
  das Element kann überall angesprochen werden (öffentlich).
- **protected**
  das Element kann nur in Klassen innerhalb des selben Package oder in Subklassen (siehe Vererbung [Seite 47]) angesprochen werden, aber nicht von fremden Klassen (geschützt).
- (keine Angabe)
  das Element kann nur in Klassen innerhalb des selben Package angesprochen werden, aber nicht von fremden Klassen (default).
- **private**
  das Element kann nur innerhalb dieser Klasse angesprochen werden (privat).

	public	**protected**	default	**private**
selbe Klasse	ja	ja	ja	ja
selbes Package	ja	ja	ja	nein
Unterklasse (extends)	ja	ja	nein	nein
überall	ja	nein	nein	nein

Es wird dringend empfohlen, bei jeder Klasse genau zu überlegen, welche Datenfelder und Methoden "von außen" direkt angesprochen werden müssen und welche nur innerhalb der Klasse oder des Package benötigt werden. Alles, was nicht für die Verwendung von außen sondern nur für die Programmierung im Inneren notwendig ist, sollte im Inneren "verborgen" werden. Dies wird als Encapsulation oder Data Hiding bezeichnet und hat vor allem die folgenden Vorteile:

- einfache und klare Schnittstelle für die Verwendung dieser Klasse bzw. ihrer Objekte,
- Sicherung, dass die Datenfelder stets gültige Werte haben,
- weitgehende Unabhängigkeit der internen Programmierung dieser Klasse von der Programmierung anderer Klassen,
- weitgehende Vermeidung von Programmierfehlern beim Zusammenspiel zwischen den verschiedenen Klassen und Programmen.

Um ungültige Datenwerte in den Datenfeldern zu vermeiden, ist es empfehlenswert, alle Datenfelder als private oder protected zu definieren und eigene public Methoden für das Setzen und Abfragen der Werte vorzusehen, die schon bei der Speicherung der Daten alle Fehler verhindern.

Per Konvention ("Java-Beans") sollen diese Methoden Namen der Form setXxxx und getXxxx haben und nach folgendem Schema definiert werden:

```
public class ClassName {

  private typ xxxx = anfangswert;

  public void setXxxx (typ xxxx) {
    // Gültigkeit des Wertes kontrollieren
    this.xxxx = xxxx;
```

```
  }

  public typ getXxxx() {
    return xxxx;
  }
}
```

Bei boolean Datenfeldern soll die Methode isXxxxx statt getXxxxx genannt werden. Bei Arrays (Feldern) sollen solche Methoden sowohl für das ganze Feld als auch (mit einem int-Parameter für den Index) für die einzelnen Feldelemente vorhanden sein.

Mehr über die Programmierung der Fehlerbehandlung finden Sie im Kapitel über Exceptions [Seite 55] .

4.11 Beispiel: HelloWorld-Bean

Das HelloWorld-Programm wird nach diesen Bean-Konventionen also so abgeändert, dass das Datenfeld messageText privat deklariert wird und für den Zugriff die public Methoden setMessageText und getMessageText vorgesehen werden:

```
public class HelloBean {

  private String messageText = "Hello World!";

  public void setMessageText(String newText) {
    messageText = newText;
  }

  public String getMessageText() {
    return messageText;
  }

  public void printText() {
    System.out.println (messageText);
  }

  public static void main (String[] args) {
    HelloBean engl = new HelloBean();
    HelloBean germ = new HelloBean();
    germ.setMessageText("Hallo, liebe Leute!");
    HelloBean cat = new HelloBean();
    cat.setMessageText("Miau!");

    engl.printText();
    germ.printText();
    engl.printText();
    cat.printText();
    engl.printText();
  }
}
```

4.12 Beispiel: schlechte Datums-Klasse

Eine sehr ungünstige Möglichkeit, eine Klasse für Datum-Objekte zu definieren, bestünde darin, einfach nur die 3 int-Felder als public zu definieren und dann alles Weitere den Anwendungsprogrammierern zu überlassen:

```
public class Date1 {  // very bad, don't do this!
  public int year;
  public int month;
  public int day;
```

In der main-Methode können dann Objekte dieser Klasse angelegt und verwendet werden. Dabei kann es passieren, dass ungültige Werte erzeugt werden (frei nach Erich Kästner):

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

    Date1 today = new Date1();
    today.day = 35;
    today.month = 5;
    today.year = 1997;
```

Außerdem können verschiedene Programmierer verschiedene Ansichten darüber haben, in welcher Reihenfolge Tag und Monat angezeigt werden sollen, und damit die Benutzer verwirren:

```
    System.out.println ("Today is " +
      today.day + ". " +
      today.month + ". " +
      today.year);
    System.out.println ("Today is " +
      today.month + ". " +
      today.day + ". " +
      today.year);
```

Auch bei der Berechnung von Zeiträumen können manche Details übersehen werden, z.B. dass 1996 ein Schaltjahr war, 2021 aber keines ist:

```
    Date1 marriage = new Date1();
    marriage.day = 29;
    marriage.month = 2;
    marriage.year = 1996;
    System.out.println ("Married on " +
      marriage.day + ". " +
      marriage.month + ". " +
      marriage.year);

    Date1 silverMarriage = new Date1();
    silverMarriage.day = marriage.day;
    silverMarriage.month = marriage.month;
    silverMarriage.year = marriage.year + 25;
    System.out.println ("Silver marriage on " +
      silverMarriage.day + ". " +
      silverMarriage.month + ". " +
      silverMarriage.year);
```

Schließlich gestaltet sich auch die Programmierung von Vergleichen recht aufwändig:

```
    if ( silverMarriage.day == today.day &&
         silverMarriage.month == today.month &&
         silverMarriage.year == today.year )
      System.out.println ("Congratulations!");
  }
}
```

4.13 Beispiel: bessere Datums-Klasse

Besser im Sinne der Kapselung wäre es, die 3 Datenfelder als private zu definieren, damit sie nicht auf falsche Werte gesetzt werden können, und public Methoden und Konstruktoren vorzusehen, mit denen diese Werte gesetzt, verändert und verwendet werden:

```
public class Date2 {  // better, but incomplete
  private int year;
  private int month;
  private int day;
```

Hier drei Konstruktoren (overloading), mit denen das Datum auf einen gültigen Anfangswert gesetzt wird. Diese Konstruktoren verwenden die unten erläuterte Methode setDate2, die die Gültigkeit der Werte sicherstellt.

```
  public Date2() {
    setDate2();
}

  public Date2 (int year, int month, int day) {
    setDate2 (year, month, day);
}

  public Date2 (Date2 other) {
    setDate2 (other);
}
```

Hier nun die drei Varianten (overloading) der public Methode setDate2:

Die Variante mit leerer Parameterliste setzt das Datum auf das heutige Datum (durch Aufruf einer geeigneten Systemfunktion, hier nur skizziert, aber nicht gezeigt):

```
  public void setDate2() {
  // set to today’s date via system function
  }
```

Die zweite Variante erlaubt die Angabe des Datums mit 3 Parametern für Jahr, Monat und Tag und überprüft, ob diese Zahlen tatsächlich ein gültiges Datum darstellen (mit Berücksichtigung von eventuellen Schaltjahren, hier nicht gezeigt). Was hier auch noch fehlt, ist die Reaktion auf solche Fehler (siehe das Kapitel über Exceptions [Seite 55]).

```
  public void setDate2 (int year, int month, int day) {
    // check for valid ranges of day and month for this year
    this.day = day;
    this.month = month;
    this.year = year;
  }
```

Die dritte Variante setzt das Datum auf das gleiche wie ein bereits erstelltes Objekt des selben Typs.

Hier ist keine neuerliche Fehlerbehandlung notwendig, da dieses Objekt ohnedies nur ein gültiges Datum enthalten kann.

```
public void setDate2 (Date2 other) {
  this.day = other.day;
  this.month = other.month;
  this.year = other.year;
}
```

Für den Zugriff auf die privaten Datenfelder werden eigene public get-Methoden geschrieben:

```
public int getDay() {
   return day;
}

public int getMonth() {
   return month;
}

public int getYear() {
   return year;
}
```

Anmerkung: Bei der in Java 1.0 eingebauten Date-Klasse gibt es zwei getrennte Methoden getDate und getDay für den Tag innerhalb des Monats und den Wochentag.

Außerdem kann man auch weitere Methoden vorsehen, z.B. um den Monat als Wort statt als Nummer zu erhalten. Hier die englische Variante, bei der deutschsprachigen müßte man auch noch den Unterschied zwischen "Januar" in Deutschland und "Jänner" in Österreich berücksichtigen:

```
public String getMonthName() {
   String s = null;
   switch (month) {
     case 1: s = "January"; break;
     case 2: s = "February"; break;
     case 3: s = "March"; break;
     case 4: s = "April"; break;
     case 5: s = "May"; break;
     case 6: s = "June"; break;
     case 7: s = "July"; break;
     case 8: s = "August"; break;
     case 9: s = "September"; break;
     case 10: s = "October"; break;
     case 11: s = "November"; break;
     case 12: s = "December"; break;
   }
   return s;
}
```

Hier noch 3 Methoden zur Berechnung von Zeitintervallen in die Zukunft oder (bei negativem Argument) in die Vergangenheit, wobei die komplizierten Details von Monats- und Jahresgrenzen und Schaltjahren hier nicht gezeigt werden. Gerade weil diese Berechnung kompliziert ist, ist es wichtig, sie innerhalb der Klasse zu programmieren und es nicht jedem einzelnen Anwender zu überlassen, sich das jedesmal neu zu überlegen.

```
public void addDays (int days) {
  // compute new date with correct day, month, and year
}

public void addMonths (int months) {
  // compute new date with correct day, month, and year
}

public void addYears (int years) {
  this.year += years;
  // correct day and month, if leap year problem
}
```

Alle Klassen müssen die beiden Methoden equals (für die Abfrage auf Gleichheit von 2 Objekten) und toString (für die menschenlesbare Darstellung des Objekts als Textstring) enthalten:

```
public boolean equals (Object other) {
  if (other instanceof Date2 &&
      ( (Date2) other).day   == this.day &&
      ( (Date2) other).month == this.month &&
      ( (Date2) other).year  == this.year)
    return true;
  else
    return false;
}

public String toString() {
  String s = day + " " +
    getMonthName() + " " +
    year;
  return s;
}
```

Anmerkung: Die Angabe von this. ist hier nicht notwendig, macht aber die Bedeutung in diesem Beispiel klarer verständlich.

Die Verwendung dieser Klasse ist nun sehr viel einfacher, und gleichzeitig sind alle Fehlermöglichkeiten und Missverständnisse ausgeschlossen:

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

  Date2 today = new Date2();
  System.out.println ("Today is " + today );
```

Anmerkung: Die Angabe von today.toString() ist innerhalb der println-Methode bzw. der String-Concatenation nicht nötig, diese Methode wird bei der Typ-Umwandlung automatisch aufgerufen.

```
  Date2 yesterday = new Date2 (today);
  yesterday.addDays(-1);
  System.out.println ("Yesterday was " + yesterday );

  Date2 marriage = new Date2 (1996, 2, 29);
  System.out.println ("Married on " + marriage );

  Date2 silverMarriage = new Date2 (marriage);
  silverMarriage.addYears(25);
  System.out.println ("Silver marriage on " +
    silverMarriage );
```

```
    if ( silverMarriage.equals(today) )
      System.out.println ("Congratulations!");

  }
}
```

In Wirklichkeit ist es natürlich nicht notwendig, eine eigene Klasse für Datumsangaben zu schreiben, die ist bereits im API vorhanden und ist noch um einige Stufen komplexer als das hier gezeigte Beispiel: Die Klasse Date enthält nicht nur das Datum sondern auch die Uhrzeit, und man kann in Verbindung mit entsprechenden weiteren Klassen auch die verschiedenen Kalender, Zeitzonen, Datumsformate und Sprachen berücksichtigen (siehe das Kapitel über Datum und Uhrzeit [Seite 139]).

4.14 Übung: einfache Kurs-Klasse

Schreiben Sie eine einfache Klasse "Kurs" mit den folgenden Datenfeldern:

- Kurs-Titel (String)
- kostenlos (boolean)
- Namen der Teilnehmer (Array von Strings)
- Anzahl der angemeldeten Teilnehmer

und mit allen benötigten set- und get-Methoden, nach den Konventionen für Beans [Seite 39] , sowie einer Methode addTeilnehmer zum Anmelden eines Teilnehmers.

Fügen Sie eine main-Methode an, in der Sie diese Methoden kurz testen, indem Sie zwei oder drei Kurs-Objekte anlegen, ein paar Teilnehmer anmelden und dann die gespeicherten Informationen auf den Bildschirm ausgeben.

4.15 Objekt-orientierte Analyse und Design

Frisch geplant ist halb gewonnen!

Die Erstellung von objekt-orientierten Programmen besteht aus folgenden Schritten

1. **Objekt-orientierte Analyse (OOA)**
   = die Überlegung, aus welchen Objekten und Klassen eine Aufgabenstellung besteht, und welche Eigenschaften und Aktionen diese Objekte haben - vorerst noch unabhängig von der verwendeten Programmiersprache,
2. **Objekt-orientiertes Design (OOD)**
   das Konzept, wie das Ergebnis der Objekt-orientierten Analyse am besten in einer bestimmten Programmiersprache realisiert werden kann,
3. **Objekt-orientierte Programmierung (OOP)**
   das Schreiben der Programme, in denen die Eigenschaften und Aktionen der Objekte bzw. der Klassen von Objekten festgelegt werden.

Die Analyse erfolgt zunächst unabhängig von der verwendeten Programmiersprache und soll die logisch richtige Sicht des Problems und damit die Grundlage für das Design liefern.

Das Design, also das Konzept für die Programmierung, berücksichtigt dann die Eigenschaften der Programmiersprache (z.B. Einfach- oder Mehrfachvererbung) und die verfügbaren Klassenbibliotheken (z.B. Schnittstellen zu Graphischen User-Interfaces oder zu Datenbanken).

Um komplexe Systeme für die objekt-orientierte Programmierung in den Griff zu bekommen, müssen sie also analysiert werden, aus welchen Objekten sie bestehen und welche Beziehungen zwischen den Objekten bestehen, welche Eigenschaften die Objekte haben und welche Aktionen mit ihnen ablaufen.

Die wichtigsten Design-Regeln für das Zerlegen von komplexen Systemen in einzelne Objekte sind:

- Die Objekte sollen möglichst **einfach** sein, d.h. ein kompliziertes Objekt soll in mehrere einfache Objekte zerlegt werden.
- Die Objekte sollen möglichst **abgeschlossen** sein, d.h. jedes Objekt soll möglichst wenig über die anderen Objekte wissen müssen.

Die Analyse und das Design ergeben sich am einfachsten, indem man versucht, das System bzw. die Aufgabenstellung mit einfachen deutschen Sätzen zu beschreiben:

- Hauptwörter (Nomen) bedeuten Objekte oder primitive Datenfelder.
- Eigenschaftswörter (Adjektiv) bedeuten Datenfelder oder get-Methoden.
- Zeitwörter (Verb) bedeuten Methoden.
- Sätze der Form "A hat ein B" bedeuten Datenfelder (Eigenschaften).
- Sätze der Form "A ist ein B" bedeuten Oberklassen (Vererbung [Seite 47]).

4.16 Beispiel: Analyse Schulungen

Aufgabenstellung: Analyse und Design des Systems "Schulungsunternehmen, Schulungen, Teilnehmer und Trainer".

Lösungs-Skizze:

1. Objekte und Aktionen:
   - Ein Schulungsunternehmen bietet Schulungen an.
   - Eine Schulung findet an Schulungsterminen statt.
   - Ein Schulungsunternehmen veröffentlicht eine Liste der Schulungstermine.
   - Ein Teilnehmer meldet sich zu einem Schulungstermin an und bezahlt die Schulungsgebühr.
   - Ein Trainer hält einen Schulungstermin ab und bekommt dafür ein Honorar.
2. Eigenschaften der Objekte:
   - Eine Schulung hat Titel, Inhalt, Länge, Preis.
   - Ein Schulungstermin hat Schulung, Termin, Ort, Trainer, maximale Teilnehmerzahl, tatsächliche Teilnehmerzahl, angemeldete Teilnehmer.
   - Ein Teilnehmer ist eine Person.
   - Eine Person hat Name, Adresse, Telefon, Konto ...
   - Ein Trainer ist eine Person und hat einen Tagessatz.
   - Ein Schulungsunternehmen ist wie eine Person und hat Schulungen und Schulungstermine.

4.17 Übung: Analyse Bücherregal

Überlegen Sie, aus welchen Objekten das System "Bücherregal mit Büchern" besteht und welche Datenfelder und Methoden die Klasse Bücherregal enthalten muss.

Es geht dabei nur um die Analyse und das Design des Systems, nicht um die konkrete Programmierung. Schreiben Sie das Ergebnis daher bitte nur in der Form von Stichworten oder graphischen Skizzen auf einem Blatt Papier auf, nicht als Java-Programm.

4.18 Übung: Analyse Person und Konto

Überlegen Sie die Analyse und das Design für ein einfaches Konto-System:

In welchen Beziehungen stehen die im folgenden in alphabetischer Reihenfolge angeführten Begriffe (**Analyse** in Form von einfachen deutschen Sätzen)?

Wie können sie durch Klassen, Datenfelder und Methoden realisiert werden (**Design**, d.h. Entwurf bzw. Konzept für die Programmierung)?

- abheben
- einzahlen
- Guthaben
- Konto
- Person
- Student
- Vorname
- Zuname

Die **Programmierung** dieses Systems wird dann später in der Übung Person und Konto [Seite 53] erfolgen.

4.19 Vererbung (Inheritance, Polymorphismus, override)

Wie kann ich ein spezielles Programm auf ein anderes, bereits existierendes, allgemeineres Programm zurückführen?
Wie kann ich erreichen, dass meine Programme möglichst allgemein brauchbar und wieder-verwertbar sind (re-usability)?

Eine der wichtigsten Eigenschaften von objekt-orientierten Systemen stellt die sogenannte "Vererbung" von Oberklassen (Superklassen) auf Unterklassen (Subklassen) dar. Dies stellt eine wesentliche Vereinfachung der Programmierung dar und ist immer dann möglich, wenn eine Beziehung der Form "A ist ein B" besteht.

Beispiele: Ein Auto ist ein Fahrzeug (hier ist Fahrzeug der Oberbegriff, also die Superklasse, und Auto ist die spezielle Subklasse). Ein Bilderbuch ist ein Buch. Ein Mitarbeiter ist eine Person...

Keine Vererbung ist in den folgenden Fällen gegeben: Ein Bücherregal enthält Bücher. Ein Bilderbuch enthält Bilder. Ein Mitarbeiter liest ein Buch.

Der Vorteil der Vererbung besteht darin, dass man die selben Konzepte nicht mehrfach programmieren muss, sondern auf einfache Grundbegriffe zurückführen und wiederverwenden kann.

Dazu wird zunächst die Oberklasse, die den allgemeinen Grundbegriff möglichst einfach beschreibt, mit den für alle Fälle gemeinsam geltenden Datenfeldern und Methoden definiert.

Dann gibt man bei der Definition der Unterklasse mit `extends` an, dass es sich um einen Spezialfall der Oberklasse handelt, und braucht jetzt nur die Datenfelder und Methoden zu definieren, die zusätzlich zu denen der Oberklasse notwendig sind. Alle anderen Definitionen werden automatisch von der Oberklasse übernommen.

Beispiel:

```
public class Person {
  public String name;
  public Date geburtsDatum;
  ...
}

public class Beamter extends Person {
  public int dienstKlasse;
  public Date eintrittsDatum;
  ...
}
```

Dann enthalten Objekte vom Typ Person die beiden Felder name und geburtsDatum, und Objekte vom Typ Beamter enthalten die vier Felder name, geburtsDatum, dienstKlasse und eintrittsDatum. Das analoge gilt für die hier nicht gezeigten Methoden.

Man kann auch Methoden, die bereits in der Oberklasse definiert sind, in der Unterklasse neu definieren und damit die alte Bedeutung für Objekte des neuen Typs überschreiben (override). Dabei braucht man eventuell nur die Teile neu zu schreiben, die zur entsprechenden Methode der Oberklasse hinzukommen, und kann für den gleichbleibenden Teil die Methode der Oberklasse mit `super.name()` aufrufen.

Wenn eine Methode in der Oberklasse als `final` deklariert ist, kann sie von Unterklassen nicht überschrieben werden.

Konstruktoren werden nicht automatisch von der Oberklasse übernommen sondern müssen neu definiert werden, wenn man mehr als den Default-Konstruktor mit der leeren Parameterliste braucht. Allerdings braucht man in diesen Konstruktoren nur diejenigen Aktionen zu definieren, die gegenüber der Oberklasse neu sind, und kann vorher - als erstes Statement - mit `super();` oder `super(parameterliste);` den Konstruktor der Oberklasse aufrufen. Wenn man das nicht tut, wird vom Compiler automatisch der Aufruf des Default-Konstruktors super(); als erstes Statement hinzugefügt.

Referenzen auf ein Objekt der Unterklasse können sowohl in Variablen vom Typ der Unterklasse als auch vom Typ der Oberklasse abgespeichert werden. Das gilt auch bei der Verwendung in Parameterlisten. Erlaubt sind also z.B. die folgenden Zuweisungen:

```
Person mutter;
Person vater;
mutter = new Person();
vater = new Beamter();
```

Allerdings können in diesem Fall auch für das Objekt vater nur die Datenfelder und Methoden aufgerufen werden, die für Personen definiert sind, andernfalls erhält man einen Compiler-Fehler. Man kann aber mit instanceof abfragen, um welchen Typ (Ober- oder Unterklasse) es sich zur Laufzeit handelt, und dann mit Casting die Typumwandlung durchführen. Beispiel:

```
public void printName (Person p) {
  System.out.print(p.name);
  if (p instanceof Beamter) {
    System.out.print(", " + ((Beamter)p).dienstKlasse );
  }
  System.out.println();
}
```

Solche Konstruktionen sind aber nur sehr selten notwendig, denn nach den Prinzipien der objekt-orientierten Programmierung sollten die Oberklassen von den Unterklassen unabhängig sein, und alles, was eine Unterklasse betrifft, sollte nur in dieser Unterklasse programmiert werden, z.B. durch Überschreiben der entsprechenden Methoden.

4.20 Übung: Person und Student

Schreiben Sie 3 Klassen:

1. eine einfache Klasse "Person" mit den Datenfeldern und Methoden vorname, zuname, getVorname, setVorname, getZuname, setZuname, und einer Methode toString, die einen String liefert, der den Vor- und Zunamen anzeigt.
2. eine einfache Klasse "Student" mit den zusätzlichen Datenfeldern und Methoden uni, getUni, setUni, und einer entsprechend erweiterten Methode toString, die Vorname, Zuname und Universität anzeigt.
3. eine Klasse mit einer main-Methode, in der ein paar Objekte von Person und Student angelegt werden und deren Inhalt mit Hilfe der toString-Methode auf den Bildschirm ausgegeben wird.

4.21 Mehrfache Vererbung, Interfaces, abstrakte Klassen

Unter mehrfacher Vererbung versteht man die Möglichkeit, dass eine Klasse Unterklasse von zwei verschiedenen Oberklassen ist, also z.B. ein Dreirad ist ein Fahrzeug und ein Kinderspielzeug.

Java unterstützt *keine* mehrfache Vererbung, in extends [Seite 47] kann nur *eine* Oberklasse angegeben werden. Eine mehrfache Angabe ist aber bei Interfaces möglich.

4.21.1 Interfaces

Interfaces sind etwas Ähnliches wie Klassen, die aber nur das Konzept für die Unterklassen skizzieren, das dann von diesen "implementiert" werden muss. Interfaces enthalten nur die "Signaturen" von Methoden; statt der Definition des Methoden-Inhalts zwischen { und } enthalten sie nur einen Strichpunkt. Die Definition eines Interface folgt dem folgenden Schema:

```
public interface InterfaceName {
  public typ name1 (parameterliste) ;
  public void name2 (parameterliste) ;
}
```

Die Subklasse, die dieses Interface implementiert, muss dann *alle* darin skizzierten Methoden enthalten, mit der richtigen Signature (Typ, Name und Parameterliste) und mit einem konkreten Block von Statements, eventuell auch nur einem leeren Block. Beispiel:

```
public class ClassName implements InterfaceName {
  ...
  public typ name1 (parameterliste) {
    Statements;
  }
  public void name2 (parameterliste) {
  }
}
```

Nach dem Wort implements können auch mehrere Interfaces angeführt werden (durch Komma getrennt). Dann muss die Klasse alle in diesen Interfaces definierten Methoden implementieren. Zusätzlich kann auch noch mit extends eine Oberklasse angegeben werden. Beispiel:

```
public class SubClass extends SuperClass
                      implements Interface1, Interface2
```

4.21.2 Abstrakte Klassen

Abstrakte Klassen (abstract class) stellen eine Mischung aus Superklassen und Interfaces dar: Ein Teil der Methoden ist in der abstrakten Klasse konkret definiert wie bei Superklassen, und von einem anderen Teil ist nur die Signature der Methoden skizziert, wie bei Interfaces. Die Definition beginnt mit

```
public abstract class SuperClassName
```

und die Vererbung in der Subklasse wie gewohnt mit

```
public class SubClassName extends SuperClassName
```

Die Subklasse *muss* dann konkrete Implementierungen von allen in der Superklasse nur skizzierten Methoden enthalten und *kann* die dort konkret definierten Methoden entweder übernehmen (Vererbung) oder überschreiben (override).

4.22 Beispiel: Katzenmusik

Diese kleine Denkaufgabe soll das Verständnis für die Anwendung von Interfaces vertiefen.

Das folgende einfache Programm soll funktionieren:

```
public class Katzenmusik {
  public static void main (String[] args) {
    LautesTier tier1 = new Katze();
    LautesTier tier2 = new Hund();
    for (int i=1; i<=10; i++) {
      tier1.gibtLaut();
      tier2.gibtLaut();
    }
  }
}
```

Wie müssen LautesTier, Hund und Katze definiert sein, damit das funktioniert?

Das Wichtige an diesem Beispiel ist, dass im Programm Katzenmusik nicht die einzelnen Eigenschaften von Hunden und Katzen angesprochen werden, sondern nur die Laute, die sie von sich geben. Dadurch wird das Programm einfach und allgemein verwendbar. Wenn wir den Hund oder die Katze durch ein anderes lautes Tier wie z.B. einen Kanarienvogel oder eine Nachtigall oder Lerche ersetzen, funktioniert das Programm genauso.

Zu diesem Zweck wird ein Interface LautesTier mit der Methode gibtLaut definiert:

```
public interface LautesTier {
  public void gibtLaut() ;
}
```

Hunde und Katzen müssen (unter anderem) dieses Interface implementieren:

```
public class Hund implements LautesTier {
  public void gibtLaut() {
    System.out.println("Wau wau!");
  }
}

public class Katze implements LautesTier {
  public void gibtLaut() {
    System.out.println("Miau!");
  }
}
```

Die Klassen Hund und Katze sollten natürlich noch durch weitere Interfaces, Methoden und Datenfelder ergänzt werden, mit denen alle anderen Eigenschaften dieser Tiere beschrieben werden, aber für das Programm Katzenmusik sind die nicht relevant.

4.23 Innere Klassen (inner class)

Ab JDK 1.1 kann man Klassen auch innerhalb von anderen Klassen definieren, auch als private Klassen. Sie können dann nur innerhalb dieser äußeren Klasse verwendet werden und haben (im Gegensatz zu separat definierten Klassen) Zugriff auf alle in der äußeren Klasse definierten

Datenfelder und Methoden, auch die privaten.

4.24 Objekt-Verbindungen

Wie im Kapitel über Objekt-orientierte Analyse und Design [Seite 45] beschrieben, gibt es zwei Arten von Objekt-Verbindungen:

- in der Bedeutungsform "**A hat ein B**",
  d.h. Objekte der Klasse A haben eine Eigenschaft (**Datenfeld**) vom Typ B,
  oder
- in der Bedeutungsform "**A ist ein B**",
  d.h. die Klasse B ist ein Spezialfall (Unterklasse, **Vererbung**) der Klasse A.

4.25 Zugriff auf Objekte von anderen Klassen (Callback)

Bei Vererbung [Seite 47] hat die Unterklasse B Zugriff auf alle nicht-privaten Datenfelder und Methoden der Oberklasse A.

Bei inneren Klassen [Seite 51] hat die innere Klasse B Zugriff auf alle (auch die privaten) Datenfelder und Methoden der äußeren Klasse A.

Auf statische [Seite 36] Methoden kann man jederzeit über den Klassennamen zugreifen.

In allen **anderen** Fällen muss das Objekt B eine Referenz auf das Objekt A "haben", wenn es auf Datenfelder oder Methoden des Objektes A zugreifen soll ("Callback"). Damit hat man dann eine wechselseitige Beziehung zwischen den Objekten, zum Beispiel in der Art wie "Du bist mein Sohn, und ich bin Dein Vater".

Es gibt zwei Möglichkeiten, wie man dem Objekt B (dem "Sohn") die Referenz auf das zugehörige Objekt A (auf den "Vater") übergeben kann:

- entweder mit Hilfe einer Methode
- oder mit Hilfe eines Konstruktors.

4.25.1 Übergabe von A an B mit einer Methode:

```
public class A {
  ...
  B b = new B();
  b.setA(this); // tell b "I am your A"
  ...
}

public class B {
  private A a = null; // reference to object of class A
  public void setA (A a) {
    this.a=a;
  }
```

```
  ...
    a.xxx(); // method of object of class A
  ...
}
```

4.25.2 Übergabe von A an B mit einem Konstruktor:

```
public class A {
  ...
  B b = new B(this); // tell b "I am your A"
  ...
}

public class B {
  private A a = null; // reference to object of class A
  public B (A a) {
    this.a=a;
  }
  ...
    a.xxx(); // method of object of class A
  ...
}
```

4.26 Übung Person und Konto

Schreiben Sie eine Klasse Konto, die auch die in der Übung Person und Student [Seite 49] erstellte Klasse Person verwendet.

Ein Konto soll folgende Eigenschaften und Methoden haben:

- inhaber (Typ: Person)
  getInhaber, setInhaber (oder Konstruktor mit Parameter)
- guthaben (Typ: double)
  getGuthaben, einzahlen, abheben
- toString()

Die toString-Methode soll den Inhaber (in dem in der Klasse Person mit der dortigen toString-Methode definierten Format) und das momentane Guthaben liefern.

Überlegen Sie auch, wie Sie den Fall behandeln wollen, wenn das Konto nicht gedeckt ist, d.h. wenn versucht wird, einen Betrag abzuheben, der größer als das momentane Guthaben ist.

Zum Testen schreiben Sie eine main-Methode, in der Sie zuerst eine oder mehrere Personen anlegen, dann jeder Person ein Konto geben, und im Konto mehrere Einzahlungen und Abhebungen durchführen.

Überlegen Sie schließlich, ob und wie Sie die Klasse Konto ändern müssen, damit auch Studenten oder andere Unterklassen von Person ein Konto besitzen können, und probieren Sie das in der main-Methode aus.

4.27 Packages und import

Unter einem Package versteht man eine Menge von zusammengehörenden Klassen, ähnlich wie bei einer Programmbibliothek. Package ist im Wesentlichen identisch mit Directory: Alle Klassen, die im selben Directory liegen, gehören zu diesem Package. Unter-Directories gelten wiederum als eigene Packages.

In Java werden Packages einheitlich und plattformunabhängig mit Punkten zwischen den Directory-Namen geschrieben, egal, ob auf der jeweiligen Plattform Schrägstriche / oder Backslashes \ oder andere Zeichen als File-Separator verwendet werden.

Wenn man in einer Java-Klasse andere Klassen oder Interfaces ansprechen will, muss man diese im Allgemeinen "importieren". Dazu gibt man am Beginn des Source-Files mit import den Namen der Klasse mit ihrem Package-Namen an, oder man importiert gleich alle Klassen eines Package, indem man einen Stern * angibt:

```
import aaa.bbb.Xxxx ;

import aaa.bbb.* ;
```

Im ersten Fall wird die Klasse Xxxx aus dem Package aaa.bbb verfügbar gemacht, im zweiten Fall alle Klassen aus dem Package aaa.bbb. Bitte, vergessen Sie nicht den Strichpunkt, der das import-Statement beenden muss.

Wenn man seine eigenen Programme in Pakete aufteilen will, muss man jeweils zuerst mit package den Namen des eigenen Pakets angeben und dann mit import die anderen Pakete:

```
package pack1;
import pack2.*;
import pack3.*;
public class Classname ...
```

Der Compiler legt die .class-Files dann in entsprechende Subdirectories und der Aufruf erfolgt mit

```
java packname.Classname
```

Die Subdirectories der Packages werden immer relativ zu den Directories gesucht, die in der Environment-Variablen CLASSPATH angeführt sind.

Wenn der Punkt `.` für das jeweilige Directory in CLASSPATH enthalten ist, dann braucht man für Klassen, die im selben Directory liegen, *kein* import-Statement anzugeben, sondern alle Klassen, die im selben Directory liegen, stehen automatisch zur Verfügung.

Das zur Grundausstattung von Java gehörende Package java.lang und das "eigene" Package (eigenes Directory oder package-Angabe) werden vom Compiler automatisch gefunden und müssen nicht explizit importiert werden, wohl aber alle anderen wie z.B. java.util, java.text, java.awt, java.awt.event etc.

Selbstverständlich kann man Klassen und deren Datenfelder und Methoden nur dann ansprechen, wenn man die Erlaubnis dazu hat (siehe Kapselung [Seite 39]).

5 Fehlerbehandlung (Exceptions)

Wie kann ich mein Programm in einen "optimistischen" Teil für den idealen Programmablauf und in einen "pessimistischen" Teil für die Fehlerbehandlung aufteilen?

- **normaler Programmablauf**
- **Ausnahmen (Fehlersituationen, Exceptions)**

Java bietet eine sehr bequeme Möglichkeit, etwaige Fehlersituationen zu erkennen und dann entsprechend zu reagieren, ohne gleich das ganze Programm abzubrechen. Dabei werden die Fehler von einer aufgerufenen Methode erkannt und "geworfen" (throw) und müssen dann von der aufrufenden Methode "aufgefangen" oder "abgefangen" (catch) werden.

Es gibt mehrere Arten von Fehlerbedingungen: die entweder abgefangen werden *müssen* oder abgefangen werden *können*:

- Exception für Benutzungsfehler, die abgefangen werden *müssen*.
  Beispiele: FileNotFoundException, InterruptedException, ...
- RuntimeException für Benutzungsfehler, oder abgefangen werden *können*.
  Beispiele: ArrayIndexOutOfBoundsException, NoSuchElementException, NumberFormatException, ...
- Error für Softwarefehler, die eventuell gar nicht mehr abgefangen werden können.
- Throwable als Oberbegriff für Exception und Error.

Im Folgenden werden Exceptions beschrieben, die abgefangen werden müssen.

5.1 Fehler erkennen (throw, throws)

Wenn eine aufgerufene Methode eine Fehlersituation erkennt, dann generiert sie ein Objekt, das den aufgetretenen Fehler beschreibt, (exception) und "wirft" diese Exception mit `throw`. Dies passiert meist innerhalb eines if-Blockes irgendwo innerhalb der Methode.

Damit sofort klar ist, welche Fehlersituationen in einer Methode auftreten können und daher von den aufrufenden Methoden abgefangen werden müssen, muss man schon bei der Deklaration der Methode mit `throws` angeben, welche Exceptions in ihr eventuell geworfen werden (eventuell mehrere, durch Komma getrennt). Falls man dies vergisst, wird man vom Compiler mit einer Fehlermeldung daran erinnert, welche Exceptions man in throws angeben muss.

Beispiel:

```
public void xxxMethod (parameterliste)
  throws XxxxException {
    ...
    if (...)
      throw new XxxxException();
    ...
}
```

Man kann

- entweder eine der vordefinierten Exceptions verwenden (wie z.B. ArrayIndexOutOfBoundsException, IOException, IllegalArgumentException oder NoSuchElementException, siehe die Klassen-Hierarchie [Seite 8] der Klasse Exception)
- oder eine eigene Exception definieren, die diese spezielle Fehlersituation beschreibt.

Mit

```
public class XxxxException extends YyyyException { }
```

definiert man eine neue Exception, die den neuen Namen XxxxException und die gleichen Datenfelder und Methoden wie die bereits definierte Exception YyyyException hat. Diese Oberklasse sollte möglichst passend gewählt werden, damit die neue Exception richtig in die Hierarchie der Ober- und Unterklassen eingeordnet wird. Beispiele:

```
public class SalaryTooLowException
   extends IllegalArgumentException { }

public class StudentNotFoundException
   extends NoSuchElementException { }

public class LVANumberFormatException
   extends NumberFormatException { }
```

5.2 Fehler abfangen (try, catch)

Die aufrufende Methode hat nun zwei Möglichkeiten:

- Entweder sie "wirft" die Fehlersituation weiter an die Methode, von der sie aufgerufen wird, indem Sie die Exceptions mit `throws` anführt,
- oder sie fängt den Fehler ab und reagiert selbst. Zu diesem Zweck müssen alle Statements, die eine Fehlersituation bewirken können, in einem `try`-Block stehen, und anschließend mit einem oder mehreren `catch`-Blöcken die möglichen Fehler abgefangen werden.

Beispiel:

```
try {
  Statements;
  xxxObject.xxxMethod();
  Statements;
} catch (XxxxException e) {
  System.out.println("*** error: " + e);
}
```

Der try-Block kann ein oder mehrere Statements enthalten, und es können auch mehrere catch-Blöcke für verschiedene Fehlersituationen angegeben werden.

Wann immer eines der Statements irgendeinen Fehler "wirft", werden die restlichen Statements des try-Blocks *nicht* ausgeführt und es wird der (erste) catch-Block ausgeführt, der den aufgetretenen Fehler auffängt. Dann wird die Verarbeitung mit den Statements nach dem letzten catch-Block fortgesetzt oder der Fehler, falls er nicht abgefangen wurde, an die aufrufende Methode weitergeworfen. (Falls man einen finally-Block angibt, wird dieser jedenfalls auch noch ausgeführt.)

Wenn man mehrere catch-Blöcke angibt, ist die Reihenfolge wichtig, man muss zuerst die spezielleren Exceptions (Unterklassen) und dann die allgemeineren Exceptions (Oberklassen) angeben.

Im Catch-Block kann man die folgenden Methoden der Exception verwenden:

- `String toString()`
  für eine kurze Angabe, welcher Fehler aufgetreten ist
- `String getMessage()`
  für einen erklärenden Text
- `printStackTrace()`
  `printStackTrace(filename)`
  für das Ausdrucken der Information, welcher Fehler wo in welchem Programm aufgetreten ist.

Was man im Catch-Block ausführt, hängt von der jeweiligen Anwendung ab. Meistens wird man

- eine Fehlermeldung an den Benutzer ausgeben

und dann

- entweder den Fehler selbständig korrigieren und die Verarbeitung mit dem korrigierten Datenwert fortsetzen
- oder die Verarbeitung abbrechen.

Wenn man vergisst, dass eine bestimmte Anweisung "gefährlich" ist und eine Fehlersituation bewirken kann, dann sagt einem der Compiler meistens mit einer Fehlermeldung, dass man sie in einen try-Block einbauen soll, und gibt auch gleich an, welche Exception man im catch-Block abfangen soll. Man muss dann entweder diesen catch-Block anfügen oder - wenn man sich mit den Details nicht beschäftigen will - die Oberklasse für alle möglichen Fehler verwenden:

```
catch (Exception e) {
  System.out.println(e);
}
```

oder eventuell sogar

```
catch (Throwable e) { }
```

5.3 Mathematische Fehler

Division durch Null und andere "verbotene" mathematische Berechnungen bewirken nur bei ganzzahligen Typen wie int, long etc. eine entsprechende Exception.

Bei float- und double-Zahlen bewirken Division durch Null, Logarithmus einer negativen Zahl und ähnliche "falsche" Operationen hingegen keine Exception sondern liefern als Ergebnis die speziellen Werte "unendlich" (infinite) bzw. "unbestimmt" (keine Zahl, not a number). Für die Abfrage dieser Werte und Bedingungen gibt es eigene Konstanten und Methoden in den Klassen Float und Double.

5.4 Übung: erweitertes Rechenbeispiel Sparbuch

Schreiben Sie eine Applikation, die wie beim einfachen Sparbuch-Beispiel [Seite 30] im Kapitel Syntax und Statements [Seite 13] die Wertentwicklung eines Geldbetrages über 10 Jahre ausgibt, aber mit folgenden Erweiterungen:

- Geldbetrag und Zinssatz werden nicht in der main-Methode festgelegt sondern vom Benutzer beim Aufruf in der Parameterliste übergeben.
- Die Zahlenausgabe soll mit der Klasse DecimalFormat in ein "schönes" Format gebracht werden.

5.5 Übung: erweitertes Konto

Erweitern Sie die Klasse Konto aus dem Kapitel Objektverbindungen [Seite 53] so, dass beim Versuch, das Konto zu überziehen, also einen Betrag abzuheben, der größer ist als das derzeitige Guthaben, eine passende Exception geworfen wird.

User-Interfaces

6 Graphical User Interfaces (GUI)

In der zeilen-orientierten prozeduralen Programmierung muss der Benutzer das Programm richtig "bedienen". Wie kann ich das User-Interface so gestalten, dass das Programm dem Menschen dient?

- **Fenster, Texte und Bilder, Tastatur und Maus**
- **Ablauf vom Benutzer gesteuert, nicht vom Programm**

Bei Programmen mit graphischen Benutzungsoberflächen (GUI) kommt es auf das Zusammenspiel von 3 Bereichen an:

- **Model** = die Datenstruktur
- **View** = die graphische Darstellung (Mensch-Maschine-Interface, Renderer)
- **Controller** = die Verarbeitung der Daten (Ausführung von Aktionen)

In sehr einfachen Fällen können alle diese Aspekte innerhalb des selben Programms (der selben Klasse) behandelt werden.

Für komplexere Anwendungen wird aber empfohlen, diese 3 Bereiche getrennt zu programmieren und damit die Vorteile der objekt-orientierten Programmierung auszunützen (siehe das Kapitel über Objekte und Klassen [Seite 31]). Professionelle Software-Hersteller setzen für die Programmierung der drei Bereiche oft sogar verschiedene Personen ein, die jeweils Spezialisten für ihren Bereich sind.

6.1 Abstract Windowing Toolkit (AWT)

Das Abstract Windowing Toolkit (AWT) ist ein Package, das Klassen für die Zusammenstellung und Verwendung von graphischen Benutzungsoberflächen (GUI) enthält. Solche GUIs bestehen aus Fenstern, Menüs, Eingabefeldern, Buttons und dergleichen, und die Steuerung durch den Benutzer erfolgt meistens mit Tastatur (Keyboard) und Maus oder mit ähnlichen Hardware-Komponenten wie z.B. Trackballs, Touchscreen oder Spracheingabe.

Das Wort abstrakt (abstract) im Namen AWT deutet darauf hin, dass in diesen Klassen nur die plattformunabhängigen wesentlichen Eigenschaften der Komponenten definiert sind und für die konkrete Darstellung dann die am jeweiligen Rechner vorhandenen Systemkomponenten ("Peers") mit den auf diesem Rechner üblichen Layouts, Aussehen und Funktionalitäten ("look and feel") verwendet werden. Ein im Java-Programm mit AWT definierter Button wird also auf einem PC wie normale Windows-Buttons, auf einem Macintosh wie normale Apple-Buttons und unter Unix wie normale Motif- oder CDE-Buttons aussehen, und das Gleiche gilt für Fenster, Scrollbars, Menüs etc.

Dies hat für den Benutzer den Vorteil, dass er auf seinem Computersystem immer die selben, gewohnten GUI-Komponenten vorfindet, unabhängig von der Applikation. Es hat aber den Nachteil, dass die gleiche Applikation auf verschiedenen Computersystemen dadurch in den Details verschieden aussieht und dass im Fall von Applets keine genaue Anpassung der Größen, Farben, Schriftarten und Graphik-Elemente an die umgebende Web-Page möglich ist.

Deshalb unterstützen neuere Java-Versionen (Swing, JDK 1.2) zusätzlich zu den Peer-Komponenten auch sogenannte "leichte" (light weight) Komponenten, die komplett in Java geschrieben sind und bei denen die Layout-Details genauer definiert werden können. Damit kann man Applets und Applikationen so gestalten, dass jede Anwendung plattformübergreifend ihr eigenes Look-and-Feel hat.

Beide Methoden haben ihre Vor- und Nachteile.

Die Swing-Komponenten [Seite 80] sind Teil der sogenannten Java Foundation Classes (JFC). Sie können beim JDK 1.1 zusätzlich installiert werden und sind ab JDK 1.2 Teil des JDK. Sie werden von den meisten Web-Browsern **noch nicht** unterstützt.

Für AWT Version 1.1 müssen die Packages java.awt und java.awt.event importiert werden, für Swing zusätzlich das Package javax.swing.

6.2 AWT-Komponenten (Component)

Die folgenden Komponenten stehen in allen AWT-Versionen zur Verfügung. Sie sind Peer-Komponenten, d.h. das Aussehen und die Funktionalität entsprechen immer dem jeweiligen Rechnersystem.

6.2.1 Frame

ein Fenster mit Rahmen, das also vom Benutzer vergrößert, verkleinert, verschoben, geschlossen werden kann (hat nichts mit dem gleichnamigen HTML-Tag zu tun). Konstruktor:
`new Frame(String)`

6.2.2 Panel

ein Bereich innerhalb eines Fensters oder eines anderen Panels, ohne eigenen Rahmen. Dies dient vor allem dazu, ein Fenster oder Applet in mehrere Bereiche zu unterteilen, in denen dann weitere Komponenten mit verschiedenen Layout-Managern [Seite 64] angeordnet werden. Konstruktor:
`new Panel()`
oder `new Panel(LayoutManager)`

6.2.3 Canvas

eine Fläche, die für graphische Ein- und Ausgabe verwendet werden kann. Der Inhalt und die Größe müssen vom Programmierer definiert werden. Zu diesem Zweck wird meist eine Subklasse von Canvas definiert, in der die entsprechenden Methoden überschrieben werden, insbesondere die paint-Methode (siehe unten). Konstruktor:
`new Canvas()`

6.2.4 Label

ein einfaches Feld für die Darstellung eines einzeiligen Textes. Konstruktor:
`new Label(String)`

6.2.5 Button

ein Knopf mit Text, der vom Benutzer mit der Maus gedrückt oder angeklickt werden kann. Es wird dringend empfohlen, Buttons nur dann zu verwenden, wenn das Anklicken tatsächlich etwas bewirkt, und für Texte, die man nicht anklicken soll, stattdessen Labels zu verwenden. Außerdem wird empfohlen, Buttons immer in ihrer natürlichen Größe anzulegen (also z.B. mit dem Flow-Layout-Manager) und nicht größer werden zu lassen, damit sie vom Benutzer immer sofort als typische Buttons erkannt werden. Konstruktor:

```
new Button(String)
```

6.2.6 Checkbox

ein Schalter mit Beschriftung, der ein- oder ausgeschaltet werden kann. Checkboxes können auch in einer CheckboxGroup zusammengefasst werden, dann bewirkt das Einschalten einer Checkbox automatisch das Ausschalten der anderen (radio button). Konstruktoren:

```
new Checkbox (String, boolean)
new CheckboxGroup()
new Checkbox (String, boolean, CheckboxGroup)
```

6.2.7 Choice, List

Listen von Texten, von denen einer bzw. mehrere ausgewählt werden können. Konstruktoren:

```
new Choice()
new List(int)
```

6.2.8 TextField

ein einzeiliges Feld für Tastatur-Eingaben. Konstruktor:

```
new TextField (columns)
```

oder `new TextField (String, columns)`

6.2.9 TextArea

ein mehrzeiliges Feld für Tastatur-Eingaben. Konstruktor:

```
new TextArea (rows, columns)
```

oder `new TextArea (String, rows, columns)`

6.2.10 Dialog

ein kleines Fenster mit einem Text, das eventuell auf eine Aktion des Benutzers wartet (z.B. eine Fehlermeldung, die erst dann verschwindet, wenn der Benutzer sie anklickt). Konstruktor:

```
new Dialog (Frame, String, true)
```

6.2.11 FileDialog

ein Menü zum Auswählen einer Datei. Konstruktoren:

```
new FileDialog (Frame, String, FileDialog.LOAD)
new FileDialog (Frame, String, FileDialog.SAVE)
```

6.2.12 MenuBar, Menu, MenuItem, CheckboxMenuItem, PopupMenu

Komponenten zum Anlegen einer Menü-Leiste in einem Frame und von Haupt-Menüs, Unter-Menüs, Help-Menüs oder Popup-Menüs sowie von Menü-Items, die ausgewählt werden können. Konstruktoren:

```
new MenuBar()
new Menu(String)
new MenuItem(String)
new MenuItem (String, MenuShortcut)
new MenuShortcut(keycode)
new MenuShortcut (keycode, shift)
new CheckboxMenuItem (String, boolean)
new PopupMenu(String)
```

6.2.13 Scrollbar

ein horizontaler oder vertikaler Scrollbar. Konstruktoren:

```
new Scrollbar (Scrollbar.HORIZONTAL)
new Scrollbar (Scrollbar.VERTICAL)
new Scrollbar (orientation, value, size, min, max)
```

6.2.14 ScrollPane

ein Rahmen mit Scrollbars zu einem zu großen Panel oder Canvas. Konstruktor:

`new ScrollPane()`
oder `new ScrollPane (ScrollPane.policy)`

6.2.15 Methoden

Alle Komponenten haben die folgenden Methoden:

- setForeground, setBackground (siehe Farben [Seite 67])
- setFont (siehe Schriften [Seite 68])
- setSize, getSize, getPreferredSize, getMinimumSize (siehe Größe [Seite 68])
- addXxxxListener (siehe Event-Handling [Seite 73])
- `setVisible(true)` und `setVisible(false)` für die Sichtbarkeit
- `dispose()` für das endgültige Beenden
- paint, update und repaint für die Darstellung auf dem Bildschirm des Benutzers (siehe Graphiken [Seite 70])

Für die einzelnen Komponenten gibt es meist noch weitere, spezielle Methoden, wie z.B.

- `getText()` und `setText(String)` bei den Komponenten, die einen Text enthalten,
- `getFile()` für den im FileDialog ausgewählten Filenamen,
- `getTitle()` und `setTitle(String)` für den Text im Titlebar eines Frame,
- `setEnabled(boolean)` oder `setEditable(boolean)` für das Aktivieren und Deaktivieren eines Button oder Eingabefeldes,

und andere. Die Details finden Sie in der Online-Dokumentation (API).

Frames, Panels, Menüs und dergleichen sind Container, d.h. sie können andere Komponenten enthalten (eventuell auch mehrfach geschachtelt), und enthalten zu diesem Zweck weitere Methoden, siehe das Kapitel über Container [Seite 64] .

6.2.16 Andere Komponenten:

Falls Sie mit dem Layout oder der Funktionsweise der im AWT bzw. in Swing vorgesehenen Komponenten nicht zufrieden sind, können Sie mit Hilfe eines Canvas beliebige eigene Komponenten definieren, z.B. spezielle Buttons, die ein Bild statt der Beschriftung enthalten oder die ihr Aussehen oder ihre Beschriftung verändern, wenn sie von der Maus erreicht, gedrückt oder angeklickt werden oder wenn bestimmte andere Aktionen erfolgreich oder nicht erfolgreich abgelaufen sind.

6.3 Container

Frames, Panels, ScrollPanes, Menüs und dergleichen sind Container, d.h. sie können andere Komponenten enthalten (eventuell auch mehrfach geschachtelt), die dann in einer bestimmten Anordnung innerhalb des Containers dargestellt werden. Zu diesem Zweck enthalten Container - zusätzlich zu den oben angeführten, für alle Komponenten gültigen Methoden - die folgenden Methoden:

- `setLayout (LayoutManager)` für den Layout-Manager [Seite 64]
- `add (Component)` oder `add (Component, Zusatzangabe)` für das Hinzufügen einer Komponente in den Container
- `remove (Component)` und `removeAll()` für das Entfernen einer Komponente bzw. von allen Komponenten aus dem Container
- `validate()` für das Sichtbarmachen des neuen Zustands mit dem Layout-Manager [Seite 64]

Ein typisches Beispiel für die Anwendung dieser Methoden finden Sie weiter unten. Hier nur eine kurze Skizze:

```
Frame f = new Frame("Titlebar");
f.setLayout( new FlowLayout() );
Button b = new Button("Press me!");
f.add(b);
f.setSize(400,300);
f.setVisible(true);
```

6.4 Layout-Manager

Der Layout-Manager dient dazu, die einzelnen Komponenten innerhalb eines Containers anzuordnen und die Größe des Containers zu bestimmen. Er wird mit

```
setLayout ( new LayoutManagerName() )
```

vereinbart. Wenn man mit keinem der vordefinierten Layout-Manager zufrieden ist, kann man

```
setLayout(null)
```

angeben und muss dann alle Komponenten selbst mit setLocation, setSize, setBounds innerhalb des Containers mit absoluten Pixel-Angaben positionieren. Damit verliert man aber den Vorteil der Layout-Manager, dass die Komponenten bei Größenänderungen automatisch wieder richtig angeordnet werden.

Die wichtigsten und einfachsten Layout-Manager sind:

6.4.1 FlowLayout

```
setLayout ( new FlowLayout() )
```

Die Elemente werden zeilenweise nebeneinander angeordnet, der Reihe nach von links nach rechts; falls die Breite nicht ausreicht, dann in mehreren Zeilen. Standardmäßig werden die Zeilen zentriert, man kann aber im Konstruktor auch mit einem Parameter angeben, dass sie links- oder rechtsbündig ausgerichtet werden sollen, z.B. mit

```
setLayout ( new FlowLayout(FlowLayout.LEFT) )
```

6.4.2 BorderLayout

```
setLayout ( new BorderLayout() )
```

Der Container wird in 4 Randbereiche und einen Bereich für den restlichen Platz in der Mitte eingeteilt, kann also höchstens 5 Elemente aufnehmen. Diese Elemente sind meistens Panels, die dann ihrerseits einzelne Komponenten enthalten und diese mit geeigneten Layout-Managern anordnen.

In der add-Methode muss mit einem zusätzlichen Parameter angegeben werden, welchen der 5 Bereiche die Komponente ausfüllen soll:

- `add (Component, "North")`
  setzt die Komponente an den oberen Rand des Containers, mit minimaler Höhe und der vollen Container-Breite.
- `add (Component, "South")`
  setzt die Komponente an den unteren Rand des Containers, mit minimaler Höhe und der vollen Container-Breite.
- `add (Component, "West")`
  setzt die Komponente an den linken Rand des Containers, mit minimaler Breite und der vollen im Container zur Verfügung stehenden Höhe.
- `add (Component, "East")`
  setzt die Komponente an den rechten Rand des Containers, mit minimaler Breite und der vollen im Container zur Verfügung stehenden Höhe.
- `add (Component, "Center")`
  setzt die Komponente in den zwischen den anderen Komponenten verbleibenden Platz in der Mitte des Containers, mit der vollen dort zur Verfügung stehenden Höhe und Breite.

Man muss nicht immer alle 5 Bereiche verwenden, oft wird das Border-Layout nur dazu verwendet, 2 oder 3 Komponenten neben- oder übereinander anzuordnen.

6.4.3 GridLayout

```
setLayout ( new GridLayout (n, m) )
```

Der Container wird in n mal m gleich große Bereiche in n Zeilen und m Spalten aufgeteilt, und die Komponenten werden in der Reihenfolge der add-Aufrufe Zeile für Zeile in dieses Gitter eingefügt und alle auf die gleiche Größe vergrößert.

Damit kann man z.B. eine Zeile oder eine Spalte von gleich großen Buttons erzeugen.

Ab JDK 1.1 gibt es auch ein **GridBagLayout**, mit dem man komplexere Gitter-Strukturen definieren kann.

Ab JDK 1.2 gibt es auch ein **BoxLayout**, mit dem man verschieden große Komponenten in Zeilen, Spalten oder Gitter-Strukturen anordnen kann. Dazu muss man deren Größen jeweils mit getPreferredSize [Seite 68] angeben.

6.4.4 CardLayout

```
CardLayout cards = new CardLayout();
container.setLayout (cards);
```

Es wird jeweils nur eine der Komponenten im Container angezeigt, alle anderen Komponenten sind unsichtbar, wie beim Aufschlagen von Spielkarten oder bei einem Dia-Vortrag.

Mit `container.add (Component, String)` wird mit dem String-Parameter ein Name für die Komponente vereinbart. Die Reihenfolge der add-Aufrufe ist relevant.

Mit `cards.show(container, String)` wird die Komponente mit dem angegebenen Namen angezeigt.

Mit `cards.next(container)` wird die jeweils nächste Komponente (in der Reihenfolge der add-Aufrufe) angezeigt.

Mit `cards.first(container)` und `cards.last(container)` wird die erste bzw. letzte Komponente angezeigt.

6.4.5 Dynamische Layout-Änderungen

Wenn sich die Größe des Containers ändert, ordnet der Layout-Manager die darin enthaltenen Komponenten automatisch neu an.

Wenn sich die Größe einer Komponente ändert, kann man nach dem folgenden Schema erreichen, dass der Layout-Manager die Komponenten im Container richtig anordnet und anzeigt:

```
component.invalidate();
// change component
container.validate();
```

Dabei kann man den Container entweder direkt angeben oder mit `component.getParent()` erhalten. Das validate() im Container bewirkt auch ein validate() für die Komponente und ein doLayout() für den Container.

6.5 Übung: Layout-Manager

Erstellen Sie ein einfaches GUI (1 Frame mit 2 Buttons, 1 Label, 1 TextField) und probieren Sie 3 verschiedene Layout-Manager aus; vorerst ohne Event-Handling, Abbruch mit Ctrl-C im Befehlsfenster.

6.6 Farben (Color)

Die Vordergrundfarbe (für Texte und Linien) und Hintergrundfarbe von Komponenten können mit den folgenden Methoden festgelegt werden:

```
setForeground (Color.xxx);
setBackground (Color.xxx);
```

Diese Methoden müssen immer paarweise aufgerufen werden, d.h. man muss immer beide Farben festlegen, man kann sich nicht darauf verlassen, dass der Benutzer eine bestimmte Hintergrundfarbe wie weiß oder hellgrau eingestellt hat. Wenn man z.B. nur den Vordergrund auf blau setzt, ein Benutzer aber aus irgendwelchen Gründen für seinen Bildschirm weiße Schrift auf blauem Grund eingestellt hat, dann bekommt er blau auf blau.

Die Farbe kann auf 3 Arten angegeben werden:

- entweder mit einem der vordefinierten Color-Objekte wie z.B. `Color.black` oder `Color.white`,
- oder mit einem selbst erzeugten Objekt wie z.B. `new Color(255,151,0)` für orange, wobei die 3 Zahlen den Rot-, Grün- und Blau-Anteil zwischen 0 und 255 angeben,
- oder mit einem selbst erzeugten Objekt wie z.B. `Color.decode("#FF9900")` mit einem String-Parameter, der die Farbe wie in HTML definiert.

Die Farben von Flächen, Linien und Schriften in Graphics-Objekten [Seite 70] werden nicht mit setForeground und setBackground sondern mit setColor angegeben (siehe unten).

Nach dem folgenden Schema können Sie die Farben im Applet auf die Farben der Web-Page abstimmen: Der HTML-Angabe

```
<body bgcolor="#FFCC99" text="#006633" ...>
```

entsprechen zum Beispiel die Java-Angaben

```
Color backColor = new Color (0xFF, 0xCC, 0x99);
Color textColor = new Color (0x00, 0x66, 0x33);
setBackground (backColor);
setForeground (textColor);
```

oder analog mit den dezimalen Angaben 0, 51, 102, 153, 204, 255, oder gleich mit den String-Angaben:

```
Color backColor = Color.decode ("#FFCC99");
Color textColor = Color.decode (#006633);
setBackground (backColor);
setForeground (textColor);
```

Ab JDK 1.1 kann man mit statischen Feldern der Klasse SystemColor auf die vom Benutzer jeweils auf seinem Rechner eingestellten Standardfarben zugreifen und die Farben im Applet auf die Farben in den anderen Bildschirmfenstern abstimmen. Beispiele:

für Frame, Panel, Applet, Label:
```
setBackground (SystemColor.window);
setForeground (SystemColor.windowText);
```

für TextField, TextArea:
```
setBackground (SystemColor.text);
setForeground (SystemColor.textText);
```

für Button u.dgl.:
```
setBackground (SystemColor.control);
setForeground (SystemColor.controlText);
```

und analog für Menüs, Scrollbars und andere Komponenten sowie für spezielle Farbgebungen (highlight, inactive u.dgl.).

6.7 Schriften (Font)

Die Schriftart und Schriftgröße von Komponenten kann mit der folgenden Methode festgelegt werden:

```
Font f = new Font (Fontname, Fonttyp, Fontsize);
setFont (f);
```

Fontnamen sind "Serif", "SansSerif" und "Monospaced" (in Java 1.0: "TimesRoman", "Helvetica" und "Courier").

Fonttypen sind Font.PLAIN (normal), Font.BOLD (fett), Font.ITALIC (kursiv) oder die Summe Font.BOLD+Font.ITALIC.

Die Fontgröße wird als ganze Zahl in Points (nicht in Pixels) angegeben, eine typische Fontgröße ist 12.

6.8 Größenangaben

Bei den meisten Komponenten ergibt sich ihre richtige Größe aus ihrem Inhalt (Textstring bei einem Label oder Button, Summe der Komponenten bei einem Container) und muss daher vom Programmierer *nicht* explizit angegeben werden.

Bei einem **Fenster** (Frame oder Dialog) muss die Größe explizit angegeben werden:

- entweder mit setSize(breite, höhe);
- oder mit pack();
  Dies bedeutet, dass das Fenster mit Hilfe des Layout-Managers so klein wie möglich gemacht wird, dass gerade noch alle Komponenten darin Platz haben.

Bei einem **Zeichenobjekt** (Canvas [Seite 69]) muss die Größe ebenfalls explizit angegeben werden, und zwar mit Hilfe der folgenden Methoden, die vom Layout-Manager abgefragt werden, um den benötigten Platz festzustellen:

- getMinimumSize()
- getPreferredSize()
- getMaximumSize()

Es gibt auch weitere Spezialfälle, in denen man dem Layout-Manager mit solchen expliziten Größenangaben helfen kann, die richtige Größe zu finden. Es hängt vom jeweils verwendeten Layout-Manager ab, ob und wie er diese Größenangaben verwendet und umsetzt.

Bei allen Komponenten und Containern kann man jederzeit mit

- getSize()

abfragen, wie groß die Komponente tatsächlich ist. Diese Methode liefert ein Objekt vom Typ Dimension, das die beiden Datenfelder width und height enthält (Breite und Höhe in Pixeln).

6.9 Zeichenobjekte (Canvas)

Die Klasse Canvas beschreibt eine leere Zeichenfläche der Größe null mal null Pixels. Für konkrete Zeichenobjekte muss man deshalb jeweils eine Subklasse der Klasse Canvas definieren und in ihr die folgenden Methoden überschreiben:

- getMinimumSize etc. für die Größe [Seite 68]
  (wird vom Layout-Manager abgefragt, um die Größe und Position innerhalb des Containers festzustellen),
- paint für den graphischen Inhalt [Seite 70]
  (wird vom Window-System aufgerufen, um das Objekt am Bildschirm sichtbar zu machen).

Beispielskizze:

```
public class Xxxxx extends Canvas {

  public Dimension getMinimumSize() {
    return new Dimension (300, 200);
  }
  public Dimension getPreferredSize() {
    return getMinimumSize();
  }
  public void paint (Graphics g) {
    g.setColor (...);
    g.draw... (...);
    g.fill... (...);
  }
}
```

6.10 Graphiken (Graphics, paint)

Bei fast allen Komponenten ist das Aussehen bereits festgelegt, die paint-Methode soll daher *nicht* überschrieben und die repaint-Methode *nicht* aufgerufen werden.

Nur bei Komponenten des Typs **Canvas** muss explizit mit der paint-Methode angegeben werden, welche graphischen Elemente dargestellt werden sollen. Zu diesem Zweck wird eine Subklasse der Klasse Canvas definiert und in ihr die paint-Methode überschrieben.

Die paint-Methode arbeitet mit einem Objekt des Typs Graphics und hat den folgenden Aufbau:

```
public void paint (Graphics g) {
  g.setColor (...);
  g.draw... (...);
  g.fill... (...);
}
```

Die wichtigsten **Methoden** des Graphics-Objekts sind:

6.10.1 Farbe und Schrift:

Bevor man Linien zeichnet, Flächen füllt oder Strings ausgibt, muss man die Farbe und eventuell auch die Schriftart für die unmittelbar folgenden Graphik-Befehle festlegen.

Die Festlegung der Farbe erfolgt hier also nicht mit setForeground und setBackground sondern Schritt für Schritt mit

```
setColor ( Color );
```

Die Festlegung der Schriftart erfolgt wie gewohnt mit

```
setFont ( Font );
```

6.10.2 Linien:

```
drawLine (x1, y1, x2, y2);
```

zeichnet eine gerade Linie vom ersten Punkt zum zweiten Punkt. Wenn die beiden Punkte identisch sind, wird nur ein **Punkt** gezeichnet (1 Pixel). Die Linienbreite ist immer 1 Pixel. Breitere Linien kann man erzeugen, indem man mehrere parallele Linien, jeweils um 1 Pixel verschoben, nebeneinander zeichnet, oder mit fillPolygon (siehe unten).

Mit den folgenden Methoden werden **Linienzüge** oder **Kurven** gezeichnet. Die Parameterliste gibt meistens zuerst den linken oberen Eckpunkt und dann die Breite und Höhe der Figur bzw. des sie umschreibenden Rechtecks an (Genauer: Das Rechteck hat eine Breite von (width+1) Pixels und eine Höhe von (height+1) Pixels). Kreise erhält man als Spezialfall von Ellipsen mit gleicher Breite und Höhe.

```
drawRect (x1, y1, width, height);
```

```
drawPolygon (x[], y[], n);

drawPolyLine (x[], y[], n);

drawOval (x1, y1, width, height);

drawArc (x1, y1, width, height, startangle, angle);
```

6.10.3 Flächen:

Mit den folgenden Methoden werden Flächen farbig gefüllt. Die fill-Methoden haben stets die gleichen Parameter wie die entsprechenden draw-Methoden.

```
fillRect (x1, y1, width, height);

fillPolygon (x[], y[], n);

fillOval (x1, y1, width, height);

fillArc (x1, y1, width, height, startangle, angle);
```

6.10.4 Texte (String):

```
drawString (String, x1, y1);
```

stellt einen einzeiligen Textstring dar. Ein automatischer Zeilenumbruch [Seite 72] ist hier nicht möglich.

6.10.5 Bilder (Image)

```
drawImage (Image, x1, y1, this);
```

6.10.6 Programmtechnische Details:

Die **Koordinaten** werden in Bildpunkten (Pixels) angegeben. Der Ursprung (0,0) des Koordinatensystems liegt links oben, die x-Achse geht von links nach rechts, die y-Achse von oben nach unten (also umgekehrt wie in der Mathematik).

Winkelangaben erfolgen hier in Grad (im Gegensatz zu den mathematischen Funktionen in der Klasse Math, wo sie in Radiant erfolgen). Winkel werden von "Osten" aus gegen den Uhrzeigersinn gezählt.

Die **Reihenfolge** der Aufrufe der einzelnen Methoden für das Graphics-Objekt ist wichtig: Spätere Aufrufe überschreiben die von früheren Aufrufen eventuell bereits gesetzten Bildelemente (Pixels), d.h. die Reihenfolge der Befehle beginnt mit dem Hintergrund und endet mit dem Vordergrund.

Die paint-Methode wird in verschiedenen Situationen aufgerufen:

1. explizit beim Aufruf der Methode repaint() sowie bei setVisible(true)
2. automatisch immer dann, wenn sich die Sichtbarkeit des Fensters am Bildschirm des Benutzers ändert, also z.B. wenn der Benutzer das Fenster verschiebt, vergrößert oder verkleinert, oder wenn das Fenster ganz oder teilweise durch ein anderes Fenster verdeckt wird bzw. wieder zum Vorschein kommt.

Da man den zweiten Fall vom Programm aus nicht kontrollieren kann, ist es wichtig, dass die **gesamte** Graphik-Ausgabe innerhalb der paint-Methode definiert ist und nicht in irgendwelchen anderen Methoden, die nur explizit aufgerufen werden.

Wenn sich die dargestellte Graphik in Abhängigkeit von bestimmten Aktionen verändern soll, dann empfiehlt sich folgende Vorgangsweise: Man legt globale Datenfelder an, die die verschiedenen Situationen beschreiben und die von der paint-Methode abgefragt werden. Wenn sich das ausgegebene Bild ändern soll, dann verändert man diese Datenfelder und ruft repaint() auf, sodass die paint-Methode den neuen Zustand der Ausgabe erzeugt.

6.10.7 Java-2D und 3D (ab JDK 1.2)

Ab JDK 1.2 gibt es das sogenannte Java-2D-API, also eine Gruppe von Packages und Klassen, mit denen 2-dimensionale graphische Objekte genauer und bequemer konstruiert werden: die Klasse Graphics2DObject und eine Reihe von weiteren Klassen in den Packages java.awt.color, java.awt.geom und java.awt.image. Damit können z.B. Linien mit beliebiger Strichstärke gezeichnet sowie geometrische Figuren um beliebige Winkel gedreht werden.

Für die Zukunft wird auch an einer Java-3D-API für die Darstellung von 3-dimensionalen Objekten gearbeitet. Genauere Informationen darüber findet man auf dem Java-Web-Server (siehe die Referenzen [Seite 176]).

6.10.8 Zeichnen in einem Container

Die Klasse Container [Seite 64] ist eine Unterklasse von Component, man kann daher die Container (wie z.B. Frame oder Applet), wenn man will, wie eine einzelne Komponente behandeln, so wie einen Canvas [Seite 69] . Man kann also

- *entweder* mit `add` einzelne Komponenten zum Container hinzufügen
- *oder* den Inhalt des kompletten Container mit `paint` selbst festlegen,

aber man sollte diese beiden Möglichkeiten *nicht* gleichzeitig verwenden, weil sonst die eine die andere stört.

Wenn man Graphiken *und* Standard-Komponenten in einem Container haben will, ist es am besten, wenn man für die Graphiken eigene Canvas-Komponenten erzeugt, die die entsprechende paint-Methode enthalten, und dann alle diese Komponenten mit add zum Container hinzufügt.

Wenn man doch beides kombinieren will, z.B um mit der paint-Methode den Hintergrund des Containers hinter den Komponenten festzulegen, dann muss man in der paint-Methode nach dem Zeichnen der Hintergrund-Graphik mit
`super.paint(g)`
für das richtige Zeichnen der Komponenten sorgen.

6.11 Mehrzeilige Texte

Mit Label und TextField und mit der Graphics-Methode drawString kann jeweils immer nur *eine* Textzeile ausgegeben werden. Wenn man will, dass ein längerer Text automatisch je nach dem verfügbaren Platz in mehrere Zeilen umgebrochen wird, kann man eine TextArea ohne horizontalen Scrollbar verwenden. Beispiel:

```
TextArea t1 = new TextArea( longString, 5, 20,
       TextArea.ScrollbarS_VERTICAL_ONLY );
```

In Swing bzw. JDK 1.2 gibt es diverse Möglichkeiten für die formatierte Darstellung von Texten, von Methoden wie setLineWrap(true) und setWordWrap(false) bis zur Verwendung von HTML oder RTF. Hier nur eine kurze Beispielskizze:

```
JEditorPane pane = new JEditorPane();
String somehtml = "<html>
  <head><title>Hello</title></head>
  <body>
  <h1>Hello!</h1>
  <p>Some <b>Java-generated</b> long text
  </body>
  </html>";
pane.setContentPane("text/html");
pane.setText(somehtml);
container.add(pane);
```

6.12 Übung: Canvas einfache Verkehrsampel

Erzeugen Sie ein einfaches Bild einer Verkehrsampel, ein schwarzes Rechteck mit 3 Kreisen (rot, gelb, grün) - entweder als Unterklasse von Frame, oder besser als Unterklasse von Canvas, die Sie in einem Frame darstellen.

In dieser einfachen Version sollen alle 3 Farben gleichzeitig leuchten, ohne Event-Handling (Abbruch mit Ctrl-C im Befehlsfenster). Erweiterungen dieses Beispiels, bei der die richtigen Farben jeweils nach einander eingschaltet werden, folgen in den Kapiteln über Applets [Seite 84] . und über Threads [Seite 103] .

6.13 Event-Handling

Wie kann mein Programm auf die Aktionen des Benutzers reagieren?

Graphische User-Interfaces bieten mehrere Möglichkeiten für Eingaben und sonstige steuernde Aktivitäten des Benutzers: Er kann eine oder mehrere Tasten auf der Tastatur drücken, die Maus bewegen, eine Maustaste drücken oder loslassen oder klicken, und er kann Kombinationen davon ausführen. Das Programm kann nicht vorhersagen, was er wann tun wird, sondern muss darauf warten, welche der möglichen Aktivitäten er in welchen Komponenten des GUI durchführt, und dann jeweils darauf reagieren. Diese Reaktion auf vom Benutzer ausgelöste Ereignisse wird als Event-Handling bezeichnet.

Mit JDK 1.1 wurde eine komplett neue Version des Event-Handling eingeführt. Dazu muss das Package java.awt.event importiert werden. Die alte Version 1.0 des Event-Handling ist weiterhin im Package java.awt verfügbar und wird vom Compiler nur mit der Warnung "deprecated" versehen (kein Fehler).

Hier wird die Version 1.1 beschrieben.

Für die verschiedenen Arten von Ereignissen (Events) sind Interfaces mit Namen der Form XxxxListener vorgesehen. Zu jeder Komponente, die auf solche Events reagieren soll (z.B. Mausklicks auf einen Button oder innerhalb eines Canvas), muss mit addXxxxListener ein Listener-Objekt vereinbart werden, das die Methoden des jeweiligen Interfaces implementiert und damit die gewünschten Aktionen ausführt. Beispiel:

```
...
Button b = new Button("B");
ActionListener ali = new MyListener();
b.addActionListener(ali);
...

public class MyListener implements ActionListener {
  public void actionPerformed (ActionEvent e) {
    ...
  }
}
```

Es gibt zwei Arten von Listenern bzw. Events:

- logische Listener, die einfach die durchgeführte Aktion beschreiben (z.B. der ActionListener, der angibt, dass ein Button ausgelöst oder ein Textfeld ausgefüllt wurde), und
- physische Listener, die genau beschreiben, was der Benutzer mit welchem Eingabegerät getan hat (z.B. der MouseListener, der angibt, an welcher Position die Maustaste langsam gedrückt oder schnell geklickt wurde).

6.13.1 Listener-Interfaces

Die folgenden Interfaces stehen zur Verfügung und besitzen die folgenden Methoden, die implementiert werden müssen:

- **ActionListener**: actionPerformed
- **ItemListener**: itemStateChanged
- **TextListener**: textValueChanged
- **AdjustmentListener**: adjustmentValueChanged
- **MouseListener**: mousePressed, mouseReleased, mouseEntered, mouseExited, mouseClicked
- **MouseMotionListener**: mouseDragged, mouseMoved
- **KeyListener**: keyPressed, keyReleased, keyTyped
- **WindowListener**: windowClosing, windowClosed, windowOpened, windowIconified, windowDeiconified, windowActivated, windowDeactivated
- **ComponentListener**: componentMoved, componentResized, componentHidden, componentShown
- **ContainerListener**: componentAdded, componentRemoved
- **FocusListener**: focusGained, focusLost

6.13.2 Event-Klassen

Die Methoden haben jeweils einen Parameter vom Typ XxxxEvent (also ActionEvent, ItemEvent, MouseEvent etc., nur beim MouseMotionListener wird ebenfalls der MouseEvent verwendet), und diese Objekte haben Methoden, mit denen man abfragen kann, welcher Button angeklickt wurde, welcher Text in ein Textfeld eingegeben wurde, an welcher Position die Maustaste gedrückt oder

losgelassen wurde, welche Taste auf der Tastatur gedrückt wurde etc.

Alle Events haben die folgende Methode:

- `Object getSource()`
  für die Komponente, in der das Ereignis ausgelöst wurde.

Die wichtigsten weiteren Methoden in den einzelnen Event-Arten sind:

- im **ActionEvent**: getActionCommand() für den Namen der angeklickten Komponente (des Button)
- im **ItemEvent**: getItem() für die Komponente (das ausgewählte Item)
- im **TextEvent**: getSource() für die Komponente (das Texteingabefeld)
- im **AdjustmentEvent**: getAdjustable() für die Komponente (den Scrollbar) und getValue() für den eingestellten Wert
- im **MouseEvent**: getX() und getY() für die Position der Maus, getClickCount() für die Unterscheidung von Einfach- und Doppelklick, isShiftDown(), isCtrlDown(), isAltDown() für gleichzeitig festgehaltene Tasten
- im **KeyEvent**: getKeyChar(), getKeyCode(), getKeyText() für die gedrückte Taste, isShiftDown(), isCtrlDown(), isAltDown() für gleichzeitig festgehaltene Tasten
- im **WindowEvent**: getWindow() für die Komponente (das Fenster)
- im **ComponentEvent**: getComponent() für die Komponente
- im **ContainerEvent**: getContainer() für den Container, getChild() für die hinzugefügte oder entfernte Komponente
- im **FocusEvent**: getComponent() für die Komponente

Wenn diese Methoden "nur" ein Objekt (die Komponente, die das Ereignis ausgelöst hat) liefern, dann kann man in diesem Objekt die genaueren Informationen finden, z.B. welcher Text im Texteingabefeld steht. Für die Details wird auf die Online-Dokumentation (API) verwiesen. Beispiel:

```
String s = null;
if (e.getSource() instanceof TextField) }
  s = ( (TextField)(e.getSource()) ).getText();
}
```

6.14 Beispiel: typischer Aufbau einer einfachen GUI-Applikation

Bei sehr einfachen GUI-Applikationen kann man alles in *einer* Klasse definieren, wie hier skizziert. Bei komplexeren GUI-Applikationen ist es aber besser, die Anwendung nach den Grundsätzen der objekt-orientierten Programmierung in mehrere kleinere Klassen aufzuteilen (siehe den nachfolgenden Abschnitt [Seite 77]).

```
import java.awt.* ;
import java.awt.event.* ;

public class TestFrame extends Frame
    implements ActionListener, WindowListener {

  private Button b1;
```

```
  // ... more Components ...

  public TestFrame (String s) {
    super(s);
  }

  public void init() {
    setLayout (new FlowLayout() );
    b1 = new Button("Press me! ");
    b1.addActionListener (this);
    add(b1);
    // ... more Components ...
    addWindowListener(this);
    setSize(300,200); // or: pack();
    setVisible(true);
  }

  public static void main (String[] args) {
    TestFrame f = new TestFrame("Test");
    f.init();
  }

  public void actionPerformed (ActionEvent e) {
    System.out.println("Button was pressed.");
  }
  public void windowClosing (WindowEvent e) {
    dispose();
    System.exit(0);
  }
  public void windowClosed (WindowEvent e) { }
  public void windowOpened (WindowEvent e) { }
  public void windowIconified (WindowEvent e) { }
  public void windowDeiconified (WindowEvent e) { }
  public void windowActivated (WindowEvent e) { }
  public void windowDeactivated (WindowEvent e) { }
}
```

Anmerkungen:

Im Sinn der objekt-orientierten Programmierung erfolgen die wesentlichen Aktionen nicht direkt in der statischen main-Methode, sondern in der nicht-statischen Methode init. Der Name dieser Methode wurde so gewählt, dass eine Umwandlung der Applikation in ein Applet bei Bedarf leicht möglich ist. Der Aufruf dieser init-Methode kann entweder, wie oben, explizit in der main-Methode oder, wie im nächsten Beispiel unten, im Konstruktor erfolgen.

Bei Fenstern (Frames) soll *immer* ein WindowListener aktiv sein, der das im Fenster laufende Programm beendet, wenn der Benutzer das Fenster schließt - dies gehört zum normalen Verhalten von Fenstern in jedem System.

Weitere Beispiele für typische GUI-Komponenten und Event-Handlung finden Sie im Kapitel über Applets [Seite 84] :

- Button-Aktionen (Button, Label, ActionListener) [Seite 89]
- Maus-Aktionen (Graphics, MouseListener, MouseMotionListener) [Seite 90]
- Text-Aktionen (TextField, Label, ActionListener) [Seite 91]

Das Event-Handling ist in allen diesen Beispielen der Einfachheit halber mit public Methoden in den public Klassen realisiert. Wenn man den Zugriff auf diese Methoden von fremden Klassen aus verhindern will, kann man stattdessen sogenannte "anonyme innere Klassen" verwenden, Beispiel:

```
frame.addWindowListener (
   new WindowAdapter() {
      public void windowClosing (WindowEvent e) {
         frame.dispose();
         System.exit(0);
      }
   }
);
```

Erklärungen dazu finden Sie in den Referenzen [Seite 176] .

6.15 Beispiel: typischer Aufbau einer komplexen GUI-Applikation

Bei sehr einfachen GUI-Applikationen kann man alles in *einer* Klasse definieren (siehe oben [Seite 75]). Bei komplexeren GUI-Applikationen ist es aber besser, die Anwendung nach den Grundsätzen der objekt-orientierten Programmierung [Seite 31] in mehrere kleinere Klassen für die einzelnen Objekte aufzuteilen. In Sinne des MVC-Konzepts [Seite 60] (Model, View, Controller) sollte man zumindest die folgenden Klassen bzw. Gruppen von Klassen verwenden:

- Applikation (main-Methode)
- Datenmodell (model)
- Ansicht (view)
- Steuerung (controller)

Von den vielen Möglichkeiten, wie man das gestalten kann, wird im Folgenden eine Möglichkeit skizziert:

6.15.1 Applikation

Die Applikation legt zuerst ein Objekt des Datenmodells an und dann ein Objekt der Ansicht. Die Referenz auf das Datenobjekt wird der Ansicht im Konstruktor übergeben. Die init-Methode der Ansicht wird gleich vom Konstruktor ausgeführt:

```
public class XxxApplication {
  public static void main (String[] args] {
    XxxModel model = new XxxModel();
    String title = "Title String";
    XxxView view  = new XxxView (model, title);
  }
}
```

6.15.2 Datenmodell (Model)

Das Datenmodell enthält die Datenfelder, die get- und set-Methoden für alle Datenfelder, sowie Berechnungsmethoden, mit denen die Ergebnisse aus den Eingabewerten berechnet werden:

```
public class XxxModel {
  private type xxx;
  ...
  public void setXxx (type xxx) {
    this.xxx = xxx;
  }
  public type getXxx() {
    return xxx;
  }
  ...
}
```

6.15.3 Ansicht (View)

Die Ansicht baut in der init-Methode das graphische User-Interface auf. Das Event-Handling wird jedoch nicht von dieser Klasse sondern von einem Objekt der Steuerungsklasse oder von mehreren solchen Objekten durchgeführt. Dem Steuerungsobjekt werden die Referenzen sowohl auf das Datenobjekt als auch auf die Ansicht (this) als Parameter übergeben.

Im Konstruktor der Ansicht wird nicht nur der Title-String des Frame sondern auch die Referenz auf das Datenobjekt übergeben. Die Datenfelder können dann mit den get-Methoden des Datenmodells abgefragt und in den GUI-Komponenten graphisch dargestellt werden.

Für komplexere Anwendungen ist es eventuell besser, dass sich alle View-Klassen bei der zugehörigen Model-Klasse "registrieren". Eine Skizze dazu finden Sie weiter unten am Ende dieses Beispiels.

Außerdem wird im Konstruktor auch gleich die init-Methode aufgerufen, die den Frame komplett aufbaut.

Die Komponenten werden hier nicht als private sondern als protected oder package-friendly deklariert, damit sie vom Steuerungsobjekt angesprochen werden können. Die im Beans-Konzept verlangten set- und get-Methoden fehlen in dieser Skizze.

```
import java.awt.* ;
import java.awt.event.* ;

public class XxxView extends Frame {
  private XxxModel model;
  private XxxController controller;
  protected TextField field1;
  protected Label label1;
  ...

  public XxxView (XxxModel m, String s) {
    super(s);
    this.model = m;
    this.init();
  }

  public void init() {
    controller = new XxxController (this, model);
    setLayout ( new FlowLayout() );
    field1 = new TextField (30);
    field1.addActionListener (controller);
    add(field1);
    label1 = new Label ( model.getYyy() );
    add (label1);
```

```
    ...
    addWindowListener (controller);
    setSize (300, 200);
    setVisible (true);
  }
}
```

Wenn das GUI aus mehreren Objekten besteht, werden dafür wiederum eigene Klassen definiert. So werden z.B. für die graphische Darstellung von Datenfeldern Subklassen von Canvas (oder dergleichen) verwendet, denen ebenfalls die Referenz auf das Datenobjekt im Konstruktor übergeben wird.

6.15.4 Steuerung (Controller)

Die Steuerungsklasse implementiert die Listener-Interfaces und führt die entsprechenden Aktionen durch. Ihr werden im Konstruktor die Referenzen sowohl auf das Datenobjekt als auch auf das Ansichtobjekt übergeben. Sie kann dann je nach den Benutzer-Aktionen die Datenfelder mit den set-Methoden des "Model" setzen und die Anzeige der Daten und Ergebnisse mit den set- und repaint-Methoden der GUI-Komponenten im "View" bewirken.

```
import java.awt.* ;
import java.awt.event.* ;

public class XxxController
    implements YyyListener {
  private XxxModel model;
  private XxxView view;

  public XxxController (XxxView v, XxxModel m) {
    this.view = v;
    this.model = m;
  }

  ... // Methoden der Listener-Interfaces
}
```

Hier eine Beispielskizze, wie die Aktionen innerhalb der Methoden des Listener-Interface aussehen können:

```
  public void actionPerformed (ActionEvent e) {
    Object which = e.getSource();
    if (which == view.field1) {
      model.setXxx ( view.field1.getText() );
      view.label1.setText ( model.getYyy() );
    }
    ...
  }
```

Wenn mehrere, verschiedene Aktionen von verschiedenen GUI-Komponenten ausgelöst werden, dann können dafür jeweils eigene Steuerungsklassen definiert und entsprechend mehrere Steuerungsobjekte angelegt werden.

In dem hier skizzierten Fall muss der Controller "wissen", welche Ansichten (View-Klassen) welche Daten (Model-Klassen) darstellen. Eine bessere Lösung wäre es, dass sich alle View-Klassen bei der Model-Klasse registrieren und dann von der Model-Klasse automatisch benachrichtigt werden, wenn die Daten sich geändert haben und daher von der View-Klasse neu dargestellt werden müssen. Beispielskizze:

```
import java.util.*;
public class XxxModel extends Observable {
  ...
  // change some data in the model
  setChanged();
  notifyObservers (arg);
  ...
}

import java.util.*;
public class XxxView extends Xxx implements Observer {
  XxxModel m;
  ...
  public XxxView (XxxModel m) {
    this.m = m;
    m.addObserver(this);
    }
  ...
  public void update (Observable o, Object arg) {
    // show the new data in the view
    ...
  }
  ...
}
```

6.16 Übung: einfache GUI-Applikation

Schreiben Sie eine einfache GUI-Applikation, die ein Fenster mit folgendem Inhalt anzeigt:

- Am oberen Rand soll in der Mitte ein kurzer Text stehen.
- Am unteren Rand sollen 2 Buttons nebeneinander zu sehen sein:
  - ein Open-Button, der keine Aktion bewirkt, und
  - ein Close-Button, der die Beendigung der Applikation bewirkt.

Führen Sie diese Applikation aus und testen Sie auch, was passiert, wenn Sie das Fenster vergrößern oder verkleinern.

Anmerkung: Diese Übung stellt gleichzeitig eine Vorarbeit für die Übung im Kapitel Applets dar.

6.17 Swing-Komponenten (JComponent)

In Swing (Java Foundation Classes JFC) stehen als Alternative zu den AWT-Komponenten [Seite 61] die folgenden "light weight" Komponenten zur Verfügung. Sie verwenden keine Peers vom Betriebssystem, sondern sind komplett in Java definiert. Man kann das Aussehen und die Funktionalität (look and feel) für diese Komponenten im Java-Programm genau festlegen:

- entweder, indem man eines der vorgefertigten Designs auswählt (z.B. wie bei Motif oder wie bei Windows)
- oder, indem man alle Details selbst festlegt und damit ein produktspezifisches Look-and-Feel erzeugt.

Swing-Komponenten sollten *niemals* mit Peer-Komponenten gemischt verwendet werden, weil dann das Zeichnen der Swing-Komponenten durch die Java Virtual Machine und das Zeichnen AWT-Komponenten durch das Window-System nicht richtig koordiniert würden. .

Für Swing muss das Package javax.swing importiert werden. Dieses Package bzw. die Java Foundation Classes JFC müssen bei JDK 1.1 zusätzlich installiert werden und sind ab JDK 1.2 Teil des JDK.

Die Swing-Komponenten werden von den meisten Web-Browsern **noch nicht** unterstützt.

6.17.1 Komponenten und Datenmodelle

Die wichtigsten Swing-Kompontenten sind:

- **JFrame** mit ContentPane, für ein Fenster mit Rahmen (wie Frame)
- **JPanel, JInternalFrame, JDesktopPane, JLayeredPane, JTabbedPane, JApplet** für einen Bereich (wie Panel bzw. Applet)
- **JComponent** als Oberklasse für alle Swing-Komponenten und Container, und für eine Zeichenfläche (wie Canvas und Panel)
- **JLabel** für einen Text oder ein Icon (wie Label)
- **JButton** für einen Knopf mit Text oder Icon (wie Button)
- **JCheckbox, JRadioButton, JToggleButton** für einen Schalter (wie Checkbox)
- **JList, JComboBox** in Verbindung mit einem Vector oder einem ListModel bzw. ComboBoxModel und ListCellRenderer für Auswahllisten (wie Choice bzw. List)
- **JTextField, JPasswordField, JTextArea** in Verbindung mit String oder Document, für Text-Eingaben (wie TextField bzw. TextArea)
- **JDialog, JOptionPane** für Dialogfenster (wie Dialog)
- **JFileDialog, JFileChooser** für die Auswahl einer Datei (wie FileDialog)
- **JMenuBar, JMenu, JMenuItem, JCheckboxMenuItem, JPopupMenu, JSeparator** für Menüs (wie MenuBar, Menu etc.)
- **JScrollbar, JScrollPane** für Scrollbars (wie Scrollbar bzw. ScrollPane)
- **ImageIcon** für ein kleines Bild
- **JProgressBar** für die graphische Darstellung eines Zahlenwertes
- **JSlider** für die graphische Eingabe eines Zahlenwertes
- **JColorChooser** für die Auswahl einer Farbe
- **JToolBar** für eine Button-Leiste
- **JToolTip** bzw. setToolTipText() für eine Zusatzinformation zu jeder Komponente
- **JTable** in Verbindung mit TableModel und TableCellRenderer für die Anordnung von Komponenten in Tabellenform
- **JTree** in Verbindung mit TreeModel, TreePath, TreeNode und TreeCellRenderer oder TreeCellEditor sowie TreeSelectionModel und TreeSelectionListener für die Darstellung eines hierarchischen Baums von Elementen wie z.B. ein Directory mit Subdirectories und Files
- **JEditorPane, JTextPane** in Verbindung mit einem String oder einem InputStream oder Reader oder einem Document wie PlainDocument, DefaultStyledDocument oder HTMLDocument mit HyperlinkListener, und mit einem EditorKit wie DefaultEditorKit, HTMLEditorKit oder RTFEditorKit, für die formatierte Darstellung von Text. Die HTML-Unterstützung ist dabei nur relativ einfach, für "besseres" HTML gibt es, allerdings nicht kostenlos, eine HotJava-Bean von der Firma Sun.

6.17.2 Beispielskizze

Hier eine kurze Skizze für die Verwendung von einfachen Swing-Komponenten. Die Beschreibung der verwendeten Klassen und Methoden finden Sie in der Online-Dokumentation von Swing (im JDK ab 1.2).

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SwingTest {
  public static void main (String[] args) {

    JFrame f = new JFrame ("Swing Test");
    Container inhalt = f.getContentPane();
    inhalt.setLayout (new FlowLayout() );

    String labText =
    "<p align=\"center\"><b>Lautst&auml;rke</b><br>(volume)</p>" ;
    JLabel lab = new JLabel ("<html>" + labText + "</html>");
    inhalt.add(lab);

    JSlider slid = new JSlider (0, 100, 50);
    // slid.addChangeListener ( ... );
    slid.setToolTipText("Stellt die Lautstaerke ein.");
    inhalt.add(slid);

    ImageIcon stopIcon = new ImageIcon ("images/stop.gif");
    JButton but = new JButton ( "Stop", stopIcon );
    // but.addActionListener ( ... );
    but.setToolTipText("Stoppt das Abspielen.");
    inhalt.add(but);

    f.setDefaultCloseOperation (JFrame.DISPOSE_ON_CLOSE);
    f.setSize(400,150);
    f.setVisible(true);
  }
}
```

6.17.3 Model, View, Controller

Die Swing-Klassen unterstützen das Prinzip von Model, View und Controller [Seite 60] .

Zu den sichtbaren Komponenten (View) gibt es jeweils geeignete Datenobjekte (Model, siehe die Hinweise in der obigen Klassenliste).

Die Verbindung vom View zum Controller erfolgt wie beim AWT durch das Event-Handling [Seite 73] mit den add-Listener-Methoden.

Der Controller muss nur die Daten im Model-Objekt verändern (set-Methoden).

Die Verbindung zwischen Model und View erfolgt automatisch nach dem Konzept von Observer und Observable. Dafür muss man nur im Konstruktor des View-Objekts das entsprechende Model-Objekt angeben. Die Registrierung des View als Observer des Model (addObserver) wird vom Konstruktor erledigt, und das automatische Update der Daten-Darstellung im View (notifyObservers) wird von den set-Methoden des Model erledigt.

Wie die Darstellung der Daten im View aussehen soll, wird mit Renderern festgelegt.

Beispiel:

- Die Klasse JTree ist eine Komponente, mit der bestimmte Informationen (Knoten und Blätter) am Bildschirm in einer Baum-Struktur sichtbar gemacht werden sollen.
- Die im JTree dargestellten Informationen (Model) werden mit einem TreeModel beschrieben, das aus TreePaths und TreeNodes besteht. Für die meisten Anwendungsfälle kann man die Klassen DefaultTreeModel und DefaultMutableTreeNode verwenden.
- Die Darstellung von TreeNodes im JTree (View) erfolgt mit einem TreeCellRenderer.
- Die Steuerung der Aktionen, die der Benutzer mit dem JTree durchführt (Controller), z.B. durch Auswahl oder Doppelklick auf Elemente darin, erfolgt mit dem in diesen Klassen vorgesehenen Event-Handling [Seite 73] . Außerdem generieren die Methoden des TreeModel eigene Events, die dafür sorgen, dass sich die Darstellung im JTree ändert, wenn sich die Daten im TreeModel ändern.

Analoges gilt für andere Komponenten wie JList, JTable usw.

Für genauere Informationen über die Swing-Klassen für Komponenten, Models, Renderer, Events, LookAndFeels etc. und über die Konstruktoren und Methoden dieser Klassen wird auf die Online-Dokumentation (API) und auf die Referenzen [Seite 176] verwiesen.

7 Applets

Als Java-Applikationen bezeichnet man selbständig laufende Programme.
Als Applets bezeichnet man "unselbständige" Java-Programme, die als Teil einer Web-Page laufen.

- **Applet = Panel innerhalb einer Web-Page**
- **läuft innerhalb des Web-Browsers**

Java-Applets werden innerhalb einer Web-Page dargestellt und unter der Kontrolle eines Web-Browsers ausgeführt.

Technisch gesehen zeichnen sich Java-Applets dadurch aus, dass sie Unterklassen der Klasse Applet sind. Die Klasse Applet ist ihrerseits eine Unterklasse von Panel [Seite 61] . Applets sind also Panels, die innerhalb des Browser-Fensters (das eine Art von Frame ist) dargestellt werden.

Ausgaben auf System.out oder System.err werden bei Web-Browsern in einem eigenen Bereich, der "Java-Console", ausgegeben, der den Benutzern meist nur auf Umwegen zugänglich ist; beim Appletviewer erscheinen sie in dem Terminal-Fenster, in dem der Appletviewer aufgerufen wird. Dies eignet sich also nur für Test-Meldungen, aber nicht für die normale Interaktion mit dem Benutzer. Diese sollte ausschließlich über die graphische Benutzungsoberfläche (GUI) des Applet erfolgen.

Bei der Deklaration von Applets müssen die Packages java.applet sowie java.awt und java.awt.event importiert werden.

7.1 Sicherheit (security, sandbox, signed applets)

Applets werden meist über das Internet von einem Server geladen, und spezielle Sicherungen innerhalb des Web-Browser ("Sandkasten", sandbox) sorgen dafür, dass sie keine unerwünschten Wirkungen auf den Client-Rechner haben können.

So können Applets im Allgemeinen

- nicht auf lokale Files zugreifen,
- nicht auf Systemkomponenten oder andere Programme zugreifen,
- keine Befehle oder Programme am Client starten,
- keine Internet-Verbindungen aufbauen außer zu dem einen Server, von dem das Applet geladen wurde.

Bei signierten und zertifizierten Applets (signed Applets) kann der Benutzer, wenn er sie für vertrauenswürdig hält, auch Ausnahmen von den strengen Sicherheitsregeln zulassen. Er kann z.B. vereinbaren, dass ein bestimmtes Applet auf ein bestimmtes File oder ein bestimmtes Subdirectory seiner Harddisk zugreifen darf (nur lesend, oder lesend und schreibend), oder dass ein anderes Applet E-Mail an fremde Internet-Rechner senden darf.

Wie man die Applets bzw. ein Archiv-File, in dem sie gespeichert sind, mit einer verschlüsselten Unterschrift (Signatur) versieht, wie man diesen Schlüssel von einer vertrauenswürdigen Person oder Institution beglaubigen (zertifizieren) lässt, und wie das Applet dann den Benutzer bittet, dass dieser die Erlaubnis für die normalerweise verbotenen Aktionen erteilt, muss leider für jeden Web-Browser anders gemacht werden. Genauere Informationen über die Version von Sun (javakey) findet man in der Online-Dokumentation von Java, die Informationen für die Versionen von Netscape und Microsoft findet man auf den Web-Servern dieser Firmen.

Für die Zertifizierung von Applets innerhalb eines kleinen Benutzerkreises wie z.B. innerhalb des Intranet einer Firma braucht man keine weltweite Zertifizierung, sondern es genügt eine Gruppen- bzw. Firmen-interne Beglaubigung der Signatur durch eine persönlich bekannte vertrauenswürdige Person.

Beim Testen von Applets in einem lokalen HTML-File mit dem Appletviewer [Seite 8] sind *nicht alle* diese Einschränkungen aktiv. Zum Testen der Security-Einschränkungen empfiehlt sich die Angabe eines URL im Appletviewer oder die Verwendung eines Web-Browsers.

7.2 HTML-Tags <applet> <object> <embed> <param>

Innerhalb der Web-Page werden Applets mit dem HTML-Tag <applet> eingefügt und aufgerufen. Dies erfolgt nach dem folgenden Schema:

```
<applet code="Xxxx.class" width="nnn" height="nnn">
<param name="xxx" value="yyy">
  Text
</applet>
```

Im **Applet**-Tag muss mit dem Parameter **code** der Name des Java-Applet angegeben werden (Class-File mit dem plattformunabhängigen Bytecode), und mit **width** und **height** die Breite und Höhe des Applet in Pixels.

In diesem Fall muss sich das Class-File im selben Directory am selben Server wie das HTML-File befinden. Wenn dies nicht der Fall ist, muss man mit einem zusätzlichen Parameter **codebase** den URL des Directory angeben, aus dem das Class-File geladen werden soll. Der URL (Uniform Resource Locator) gibt im Allgemeinen das Protokoll, die Internet-Adresse des Servers und den Namen des Directory innerhalb des Servers an. Dieses Directory ist dann auch das Package [Seite 54] , aus dem eventuell weitere Klassen geladen werden, und dieser Server ist derjenige, auf den das Applet über das Internet zugreifen darf. Beispiel:

```
<applet codebase="http://www.boku.ac.at/javaeinf/"
        code="HelloApp.class" width="300" height="100">
```

Falls das Class-File kein einzelnes File ist sondern in einem komprimierten Archiv [Seite 9] (ZIP- oder JAR-File) enthalten ist, muss man den Namen dieses Archiv-Files mit dem Parameter **archive** angeben:

```
<applet codebase="http://hostname/dirname/"
        archive="xxxxx.jar" code="Xxxxx.class"
        width="300" height="100">
```

Mit einem oder mehreren **Param**-Tags können Parameter an das Applet übergeben werden, die innerhalb des Applet mit der Methode getParameter abgefragt werden können. Ein Beispiel dafür folgt im Kapitel über Parameter [Seite 97] .

Der (abgesehen von den Param-Tags) zwischen <applet> und </applet> stehende **Text** wird von den Browser-Versionen, die Java nicht unterstützen, an Stelle des Java-Applet dargestellt. Dies kann beliebig formatierter HTML-Text sein, also z.B. auch Bilder enthalten.

In der neuen HTML-Norm HTML 4.0 ist vorgesehen, dass statt dem Tag <applet> der Tag <object> verwendet werden soll. Dies wird aber von dem meisten Browsern noch nicht unterstützt.

Ausführlichere Informationen über Web-Pages und die Hypertext Markup Language HTML finden Sie in der HTML-Einführung und in den Referenzen [Seite 176] .

7.2.1 Java Plug-in

Als Alternative zu den in die Web-Browser integrierten Java Virtual Machines, die die neueste Java-Version meist noch nicht oder nur unvollständig unterstützen, bietet die Firma Sun ein "Java Plug-in" an (auch Java Activator genannt), das von den Benutzern kostenlos in den Internet-Explorer oder in den Netscape-Navigator unter MS-Windows oder Sun Solaris oder HP-UX eingebaut werden kann und dann die jeweils neueste Version des JDK voll unterstützt. Bei Netscape Version 6, iPlanet und Mozilla ist dies sogar der Standard.

Dazu genügt allerdings nicht die Installation der entsprechenden Software am Client, sondern es muss auch das HTML-File am Web-Server verändert werden: Je nach der Browser-Version muss der Aufruf des Applet in diesem Fall nicht mit <applet> sondern mit <object> oder <embed> erfolgen. Wenn man als Autor einer Web-Page erreichen will, dass die Web-Page mit allen Web-Browsern funktioniert, muss man dafür eine geeignete Kombination der HTML-Befehle <object>, <embed> und <applet> angeben. Genauere Informationen darüber finden Sie on-line an der Adresse
http://java.sun.com/products/plugin/

7.3 Beispiel: einfaches HelloWorld Applet

Um einen ersten Eindruck zu geben, wie ein Applet aussieht, hier das Gegenstück zur primitiven Hello-World-Applikation aus dem ersten Teil dieser Kursunterlage. Eine Erklärung der darin vorkommenden Methoden folgt weiter unten.

7.3.1 Applet (Java-Source)

```
import java.awt.*;
import java.applet.*;

public class HelloApp extends Applet {

  private String s = "Hello World!";

  public void paint (Graphics g) {
    g.drawString (s, 25, 25);
  }
}
```

Dieses Java-Programm muss in einem File mit dem Filenamen `HelloApp.java` stehen.

7.3.2 Web-Page (HTML-File)

Ein einfaches HTML-File, das dieses Applet enthält, könnte so aussehen:

```
<html>
<head>
<title>Example</title>
</head>
<body>

<h1>Example</h1>

<p align=center>
<applet code="HelloApp.class" width="200" height="100" >
  Hello World!
</applet>
</p>

</body>
</html>
```

Für Tests mit dem Appletviewer [Seite 8] genügt die folgende Minimalversion:

```
<html>
<applet code="HelloApp.class" width="200" height="100">
</applet>
</html>
```

Wenn dieses HTML-File den Filenamen `hello.html` hat, dann erfolgen Compilation und Ausführung des Applet mit den folgenden Befehlen:

```
javac HelloApp.java

appletviewer hello.html
```

Wenn Sie sehen wollen, wie das Applet funktioniert, probieren Sie es in Ihrem Browser aus (falls er das kann).

7.4 Übung: einfaches HelloWorld Applet

Schreiben Sie das oben angeführte HelloWorld-Applet (oder eine Variante davon mit einem anderen Text) und ein entsprechendes HTML-File, übersetzen Sie das Applet und rufen es mit dem Appletviewer auf.

Falls Sie einen Java-fähigen Web-Browser zur Verfügung haben, probieren Sie auch aus, ob dieser Ihr Applet richtig verarbeiten kann.

7.5 Methoden init, start, stop

Analog zur main-Methode von Applikationen werden in Applets die folgenden Methoden automatisch ausgeführt:

- `public void init()`
- `public void start()`
- `public void stop()`

Die Methode **init** wird ausgeführt, wenn die Web-Page und das Applet geladen werden.

Die Methode **start** wird immer dann ausgeführt, wenn die Web-Page mit dem Applet im Browser sichtbar gemacht wird, also sowohl beim ersten Laden als auch bei jeder Rückkehr in die Seite (z.B. mit dem Back-Button des Browsers, wenn man inzwischen eine andere Web-Page angesehen hat).

Die Methode **stop** wird immer dann ausgeführt, wenn die Web-Page verlassen wird, also beim Anklicken einer anderen Web-Page oder beim Beenden des Web-Browsers.

Im Allgemeinen gibt man in der init-Methode alles an, was zur Vorbereitung des Applet und zum Aufbau der graphischen Benutzungsoberfläche (GUI) dient. In der start-Methode werden dann eventuelle Aktionen wie z.B. Animationen oder Datenbankverbindungen gestartet. In der stop-Methode müssen alle in der start-Methode begonnenen Aktionen beendet werden.

Da das Laden und Starten des Applet ohnehin immer eine längere Wartezeit mit sich bringt, wird empfohlen, alle während der Interaktion mit dem Benutzer benötigten weiteren Klassen sowie Bilder, Töne, Datenfiles etc. so weit wie möglich und sinnvoll schon bei der Initialisierung des Applet über das Internet zu laden (also in der init-Methode), damit die interaktive Arbeit mit dem Benutzer dann zügig ablaufen kann und nicht durch neuerliche Wartezeiten für das nachträgliche Laden solcher Files verzögert bzw. unterbrochen wird.

7.6 Größenangabe

Die Größe des Applet wird *nicht* innerhalb des Applet mit setSize, getPreferredSize oder dergleichen angegeben, sondern mit den Parametern width und height im Applet-Tag im HTML-File.

In den width- und height-Parametern von <applet> können nicht nur Pixel-Angaben sondern auch Prozent-Angaben stehen, dann kann der Benutzer die Größe des Applet verändern.

Innerhalb des Java-Programms kann man die aktuelle Größe des Applet mit der Methode getSize() abfragen.

7.7 Beenden des Applet

Bei Applikationen mit graphischen User-Interfaces ist meist ein Close- oder Exit-Button notwendig, um die Applikation zu beenden.

Bei Applets ist ein Close- oder Exit-Button im Allgemeinen *nicht* sinnvoll. Das Applet wird vom Benutzer einfach dadurch beendet, dass er die Web-Page verlässt oder den Web-Browser (mit Close oder Exit in seinem File-Menü) beendet.

In Applets soll daher auch kein `System.exit` ausgeführt werden, d.h. das Applet darf den Web-Browser nicht "abschießen".

7.8 Beispiel: typischer Aufbau eines Applet mit Button-Aktionen

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class ButtonTest extends Applet
  implements ActionListener {

  private Button b1, b2;
  private Label mess;

  public void init() {
    setLayout (new FlowLayout() );
    b1 = new Button("B1");
    b1.addActionListener (this);
    add(b1);
    b2 = new Button("B2");
    b2.addActionListener (this);
    add(b2);
    mess = new Label("Nothing was pressed.");
    add(mess);
    setVisible(true);
  }

  public void actionPerformed (ActionEvent e) {
    String s = e.getActionCommand();
    if (s.equals("B1")) {
      mess.setText("The first button was pressed.");
    }
    else
    if (s.equals("B2")) {
      mess.setText("The second button was pressed.");
    }
    this.validate();
  }
}
```

Hier das zugehörige minimale HTML-File, mit der Angabe der Größe des Applet:

```
<html>
<applet code="ButtonTest.class" width="500" height="100">
</applet>
</html>
```

Wenn Sie wissen wollen, was dieses Applet tut, probieren Sie es einfach aus (in Ihrem Browser, wenn er das kann, oder Abschreibübung oder Download).

7.9 Beispiel: typischer Aufbau eines Applet mit Maus-Aktionen

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class MouseTest extends Applet
  implements MouseListener, MouseMotionListener {

  private int mouseX, mouseY;
  private int radius, diameter = 20;
  private Color circleColor = Color.red;

  public void init() {
    Dimension full = getSize();
    mouseX = full.width/2;
    mouseY = full.height/2;
    radius=diameter/2;
    addMouseListener(this);
    addMouseMotionListener(this);
  }

  public void paint (Graphics g) {
    g.setColor (circleColor);
    g.fillOval ((mouseX-radius), (mouseY-radius),
       diameter, diameter);
  }

  public void mousePressed (MouseEvent e) {
    mouseX = e.getX();
    mouseY = e.getY();
    circleColor = Color.green;
    repaint();
  }
  public void mouseReleased (MouseEvent e) {
    mouseX = e.getX();
    mouseY = e.getY();
    circleColor = Color.red;
    repaint();
  }
  public void mouseClicked (MouseEvent e) { }
  public void mouseEntered (MouseEvent e) { }
  public void mouseExited (MouseEvent e) { }

  public void mouseDragged (MouseEvent e) {
    mouseX = e.getX();
    mouseY = e.getY();
    circleColor = Color.green;
    repaint();
  }
  public void mouseMoved (MouseEvent e) { }

}
```

Wenn Sie wissen wollen, was dieses Applet tut, probieren Sie es einfach aus (in Ihrem Browser, wenn er das kann, oder Abschreibübung oder Download).

7.10 Beispiel: typischer Aufbau eines Applet mit Text-Eingaben

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class TextTest extends Applet
  implements ActionListener {

  private TextField field1, field2;
  private Label helpText;

  public void init() {
    setLayout (new FlowLayout() );
    field1 = new TextField("", 30);
    field1.addActionListener(this);
    add(field1);
    field2 = new TextField("", 30);
    field2.addActionListener(this);
    add(field2);
    helpText = new Label ("Enter text and hit return.");
    add(helpText);
    setVisible(true);
  }

  public void actionPerformed (ActionEvent e) {
    String s = null;
    Object which = e.getSource();
    if (which==field1) {
      s = field1.getText();
      field2.setText(s);
      field1.setText("");
    }
    else
    if (which==field2) {
      s = field2.getText();
      field1.setText(s);
      field2.setText("");
    }
  }
}
```

Wenn Sie wissen wollen, was dieses Applet tut, probieren Sie es einfach aus (in Ihrem Browser, wenn er das kann, oder Abschreibübung oder Download).

7.11 Übung: einfaches Applet Thermostat

Schreiben Sie ein Applet, das folgende Daten und Steuerelemente darstellt und die folgenden Aktionen ausführt:

- Im oberen Teil soll in der Mitte die Temperatur angezeigt werden.
- Im unteren Teil sollen 2 Buttons nebeneinander zu sehen sein, die den Wert der Temperatur um 1 verringern oder erhöhen.

Als Erweiterung können Sie die Anzeige farbig gestalten, und zwar je nach dem erreichten Temperaturbereich in verschiedenen Farben.

Wenn Sie sehen wollen, wie das aussehen könnte, probieren Sie es in Ihrem Browser aus (falls er das kann).

Für das Grundgerüst dieses Applet können Sie Teile der Übung [Seite 80] aus dem Kapitel über Graphical User-Interfaces [Seite 60] verwenden.

Führen Sie dieses Applet mit einem minimalen HTML-File im Appletviewer aus und testen Sie auch, was passiert, wenn Sie das Fenster vergrößern oder verkleinern.

Falls Ihnen dieses Beispiel zu einfach ist und Sie mehr Zeit investieren wollen, können Sie auch eines oder mehrere der folgenden, aufwendigeren Beispiele ausprobieren.

7.12 Übung: Applet einfache Verkehrsampel

Schreiben Sie ein Applet, das folgende Elemente darstellt und folgende Aktionen ausführt:

- Am oberen Rand soll in der Mitte der Text "Traffic Light Exercise" stehen.
- Im mittleren Bereich soll ein einfaches rechteckiges Bild einer Verkehrsampel zu sehen sein, ein schwarzes Rechteck mit bunten Kreisen, reihum in jeweils einem der folgenden 4 Zustände:
  1. rot
  2. rot + gelb
  3. grün
  4. gelb

  (Auf grünes oder gelbes Blinklicht verzichten wir in dieser Übung, siehe die Übung im Kapitel über Threads [Seite 103])
- Am unteren Rand sollen 2 Buttons zu sehen sein:
  - ein Next-Button, der ein Weiterspringen der Verkehrsampel auf den jeweils nächsten Zustand bewirkt, und
  - ein Stop-Button, der die Ampel auf rot stellt.

Wenn Sie sehen wollen, wie das aussehen könnte, probieren Sie es in Ihrem Browser aus (falls er das kann).

Führen Sie dieses Applet mit einem minimalen HTML-File im Appletviewer aus und testen Sie auch, was passiert, wenn Sie das Fenster vergrößern oder verkleinern.

7.13 Übung: komplexes Applet Sparbuch

Erweitern Sie das einfache Rechenbeispiel Sparbuch [Seite 30] aus dem Kapitel Syntax und Statements [Seite 13] bzw. dessen Erweiterung [Seite 58] aus dem Kapitel Exceptions [Seite 55] zu einem Applet mit graphischem User-Interface.

Überlegen Sie zunächst die zu Grunde liegenden **Daten (Model):** Geldbetrag, Zinssatz, und der Wert in jedem Jahr.

Überlegen Sie dann die **graphische Darstellung (View):**

- Oben sollen Eingabefelder für Geldbetrag und Zinssatz sowie Textbereiche für Erklärungen und Fehlermeldungen vorhanden sein.
- Unten soll eine Übersicht über die Wertentwicklung in den 10 Jahren sowohl in Form einer Tabelle (Zahlenangaben) als auch in Form eines Diagramms (graphisch) erscheinen.

Überlegen Sie schließlich die **Steuerung (Controller):** Wenn der Benutzer einen neuen Geldbetrag oder Zinssatz eingibt, sollen sofort die entsprechenden Werte sichtbar werden (oder eine Fehlermeldung, wenn die Eingabe ungültig war).

Wenn Sie sehen wollen, wie das aussehen könnte, probieren Sie es in Ihrem Browser aus (falls er das kann).

Führen Sie dieses Applet mit einem minimalen HTML-File im Appletviewer aus und testen Sie auch, was passiert, wenn Sie das Fenster vergrößern oder verkleinern.

7.14 Übung: komplexes Applet: Bäume im Garten

Überlegen, entwerfen und schreiben Sie ein Applet, das folgende Elemente darstellt und folgende Aktionen ausführt:

- Am oberen Rand soll in der Mitte eine Überschrift stehen.
- Im linken Hauptbereich soll der Grundriss eines Gartens zu sehen sein, zunächst ohne Bäume, dann mit Kreisen für die gepflanzten Bäume. Mit Mausklicks oder Mausdrags kann der Benutzer hier Stellen für das geplante Pflanzen eines Baumes auswählen.
- Im rechten Hauptbereich soll ein einfaches buntes Bild des Gartens im Aufriss zu sehen sein, mit allen gepflanzten Bäumen.
- Der untere Bereich soll aus einem Textbereich und einem Buttonbereich bestehen.
- Der Textbereich soll kurze Texte (Anleitungen, Hinweise und Fehlermeldungen) für die Benutzerführung enthalten, je nach den Aktionen des Benutzers.
- Darunter sollen 3 Buttons zu sehen sein:
  - ein Okay-Button, der den Baum an der vom Benutzer ausgewählten Stelle pflanzt, wenn es eine erlaubte Stelle war.
  - ein Undo-Button, mit dem der gerade geplante bzw. der zuletzt gepflanzte Baum entfernt wird (bei mehrmaliger Verwendung bis zum Anfangszustand zurück).
  - ein Empty-Button, der den Anfangszustand (Garten ohne Bäume) wieder herstellt.

Führen Sie dieses Applet mit einem minimalen HTML-File im Appletviewer aus und testen Sie alle vorgesehenen Aktionen.

Wenn Sie sehen wollen, wie das aussehen könnte, probieren Sie es in Ihrem Browser aus (falls er das kann).

7.15 Vermeidung von Flimmern oder Flackern (buffered image)

Wenn die paint-Methode eine längere Rechnerzeit benötigt, kann beim Aufruf der repaint-Methode ein störendes Flimmern oder Flackern entstehen. Dies hat folgende Gründe:

Die repaint-Methode ruft intern die update-Methode auf. Diese löscht zunächst den bisherigen Inhalt der Komponente (clear screen), indem sie die Fläche mit der Hintergrundfarbe füllt, und ruft dann die paint-Methode auf, die die Graphik-Ausgabe bewirkt. Wenn der ganze Vorgang länger als etwa 1/20 Sekunde beträgt, wird er als Flimmern oder Flackern wahrgenommen.

Dieses störende Flimmern oder Flackern kann auf die folgende Weise vermieden werden:

1. Man macht die zeitaufwändige Berechnung des neuen Bildes nicht in der paint-Methode sondern in einer eigenen Methode in einem eigenen Image-Objekt (Pufferbild, buffered image), und definiert die paint-Methode so, dass sie nur das fertig aufbereitete Bild ausgibt.
2. Man definiert die update-Methode so, dass sie nur die paint-Methode aufruft, ohne vorheriges Clear-Screen, das in diesem Fall auch nicht nötig wäre, weil von der paint-Methode ohnehin die gesamte Fläche neu gezeichnet wird.

Dies erfolgt nach dem folgenden Schema:

In der Klasse (die im Allgemeinen eine Subklasse von Canvas oder Applet ist) werden globale Datenfelder für das Pufferbild und die darin enthaltene Graphik angelegt:

```
private Image img;
private Graphics g;
```

In der init-Methode werden die Bild- und Graphik-Objekte mit der richtigen Größe angelegt:

```
Dimension d = getSize();
img = createImage (d.width, d.height);
g = img.getGraphics();
```

Das globale Graphik-Objekt kann nun in beliebigen Methoden der Klasse bearbeitet werden, und mit repaint kann dann die Ausgabe bewirkt werden:

```
g.setColor (...);
g.draw... (...);
g.fill... (...);
repaint();
```

Die paint-Methode wird so definiert, dass sie nur das globale Pufferbild in ihren Graphik-Parameter ausgibt, und zwar in diesem Fall ohne Image-Observer (null als 4. Parameter):

```
public void paint (Graphics g) {
  if (img != null)
    g.drawImage (img, 0, 0, null);
}
```

Die update-Methode wird so definiert, dass sie nur die paint-Methode aufruft:

```
public void update (Graphics g) {
  paint(g);  // no clear
}
```

7.16 Uniform Resource Locators (URL)

Die Klasse URL im Paket java.net beschreibt Netz-Adressen (Uniform Resource Locator). Applets können im Allgemeinen nur Files und Directories auf dem Server ansprechen, von dem sie geladen wurden. Zu diesem Zweck können die folgenden Methoden verwendet werden:

- `URL getCodeBase()`
  für die Adresse bzw. das Directory, aus dem das Applet (Class-File) geladen wurde. Dies ist das Package, aus dem eventuell andere Klassen geladen werden, und dies gibt den Server an, den das Applet über das Internet ansprechen darf.
- `URL getDocumentBase()`
  für die Adresse bzw. das Directory, aus dem die Web-Page (das HTML-File), die das Applet enthält, geladen wurde. Dies kann ein anderes Directory oder ein anderer Server als bei getCodeBase sein.

Wie man ein Daten- oder Text-File vom Web-Server lesen kann, wird im Kapitel über Ein-Ausgabe [Seite 112] beschrieben. Hier folgen Hinweise zum Laden von Bildern und Tönen und dergleichen vom Web-Server.

7.17 Bilder (Image, MediaTracker)

Bilder können mit getImage vom Web-Server geladen und dann mit drawImage im Applet oder in einem Canvas innerhalb des Applet dargestellt werden. Die allgemein unterstützten Bildformate sind GIF und JPEG.

Methode im Applet-Objekt:

- `Image getImage (URL directory, URL filename)`

Methoden im Image-Objekt:

- `int getWidth(this)`
- `int getHeight(this)`

Methode im Graphics-Objekt:

- `drawImage (Image, int x1, int y1, this)`

Beispielskizze:

```
Image logo;
public void init() {
  logo = getImage( getDocumentBase(), "logo.gif");
}
public void paint (Graphics g) {
  if (logo != null)
    g.drawImage (logo, 25, 25, null);
}
```

Die Methode getImage startet nur das Laden des Bildes. Der Ladevorgang über das Internet kann längere Zeit dauern. Wenn man abwarten will, bis der Ladevorgang erfolgreich abgeschlossen ist, muss man ein Objekt der Klasse **MediaTracker** verwenden.

Dies ist zum Beispiel dann notwendig, wenn man mit getWidth und getHeight die Größe des Bildes abfragen will, denn diese Werte sind erst dann verfügbar, wenn das Bild fertig geladen wurde.

Beispiel:

```
Image img;
int imgWidth, imgHeight;

public void init() {
  MediaTracker mt = new MediaTracker(this);
  img = getImage( getDocumentBase(), "img.jpg");
  mt.addImage (img, 0);
  try {
    mt.waitForID(0);
    imgWidth = img.getWidth(this);
    imgHeight = img.getHeight(this);
    ...
  } catch (InterruptedException e) {
    System.out.println("Image not loaded.");
  }
}
```

In Applikationen kann man Bilder analog mit
`x.getToolkit().getImage(...)`
(wobei `x` irgendeine AWT-Komponenente ist) oder mit
`Toolkit.getDefaultToolkit().getImage(...)`
laden.

7.18 Töne (sound, AudioClip)

Töne (Audio-Clips) können mit getAudioClip vom Web-Server geladen und mit den Methoden play, loop und stop abgespielt werden. Bis JDK 1.1 wird nur das Audio-Clip-Format (File-Extension .au) unterstützt, ab JDK 1.2 auch andere Audio-Formate wie MIDI und WAV und auch Video-Clips (Java Media Framewortk JMF).

Methode im Applet-Objekt:

- `AudioClip getAudioClip (URL directory, URL filename)`

Methoden im AudioClip-Objekt:

- `play()`
- `loop()`
- `stop()`

Beispielskizze für ein einmaliges Abspielen beim Starten des Applet:

```
AudioClip sayHello;
public void init() {
  sayHello = getAudioClip( getDocumentBase(), "hello.au");
  sayHello.play();
}
```

Beispielskizze für eine ständige Wiederholung bis zum Verlassen des Applet:

```
AudioClip music;
public void init() {
  music = getAudioClip( getDocumentBase(), "music.au");
}
public void start() {
  music.loop();
}
public void stop() {
  music.stop();
}
```

Für das Abspielen von AudioClips "im Hintergrund" zugleich mit anderen Aktionen verwendet man eigene Threads [Seite 103] .

7.19 Parameter

Ähnlich wie die Laufzeit-Parameter beim Starten einer Applikation mit java, können mit dem <param>-Tag in der Web-Page [Seite 85] Parameter an das Applet übergeben werden. Jeder Parameter hat einen Namen und einen Wert:

```
<param name="xxx" value="yyy">
```

Innerhalb des Applet können die Parameter mit der Methode getParameter abgefragt werden. Die in HTML-Tags angegebenen Werte sind immer Strings. Die Umwandlung von Ziffernfolgen in Zahlen etc. muss innerhalb des Java-Applet erfolgen. Bei Namen in HTML-Tags wird meist nicht zwischen Groß- und Kleinbuchstaben unterschieden, es wird daher empfohlen, die Namen nur mit Kleinbuchstaben zu vereinbaren.

Beispiel:

```
<html>
<applet code="ParamTest.class" width="300" height="200">
  <param name="firstname" value="Octavian">
  <param name="age" value="17">
```

```
</applet>
</html>

import java.applet.*;
public class ParamTest extends Applet {
  String firstname = null;
  int age = 0;
  public void init() {
    try {
      firstname = getParameter ("firstname");
      String ageString = getParameter ("age");
      age = Integer.parseInt(ageString);
    } catch (Exception e) {
    }
    if (age < 18) {
     System.out.println(name + ", you are too young!");
    }
    ... // do something
  }
}
```

7.20 Web-Browser

7.20.1 AppletContext

Der Web-Browser, der das Applet geladen hat und darstellt, kann mit der Methode getAppletContext verfügbar gemacht werden. Dann kann man mit getApplets und getApplet auf die anderen, in der selben Web-Page enthaltenen Applets zugreifen, oder den Web-Browser mit der Methode showDocument bitten, eine andere Web-Page zu laden und darzustellen.

Methode im Applet-Objekt:

- `AppletContext getAppletContext()`

Methoden im AppletContext-Objekt:

- `Enumeration getApplets()`
- `Applet getApplet (String appletName)`
- `showDocument (URL)`
- `showDocument (URL, String target)`

7.20.2 Web-Pages, Mail, News

Wenn showDocument mit nur einem Parameter aufgerufen wird, wird die aktuelle Web-Page (die das Applet enthält) durch eine neue Web-Page ersetzt.

Wenn showDocument mit zwei Parametern aufgerufen wird, wird die neue Web-Page in dem angegebenen Target-Frame dargestellt. Dies eignet sich z.B. für die Darstellung von Hilfe-Texten zum Applet in einem eigenen Frame (oder Browser-Fenster) mit dem Targetnamen "help".

Wenn der URL nicht mit http: oder ftp: sondern mit mailto: oder news: beginnt, wird das Mail- bzw. News-Fenster des Web-Browsers mit der angegebenen Mail-Adresse bzw. Newsgruppe geöffnet, und der Benutzer kann eine E-Mail absenden bzw. eine Newsgruppe lesen.

In allen Fällen hängt es vom Web-Browser ab, ob und wie er das alles unterstützt.

Beispielskizze:

```
URL newPage = null;
AppletContext browser = getAppletContext();
try {
  newPage = new URL (getDocumentBase(), "xxx.html");
  browser.showDocument (newPage);
} catch (Exception e {
}
```

7.20.3 Zugriff auf andere Applets

Wenn die Web-Page mehrere Applets enthält, etwa in der Form

```
<applet code="Applet1.class" name="a1"
  width="nnn" height="nnn">
</applet>
  ...
<applet code="Applet2.class" name="a2"
  width="nnn" height="nnn">
</applet>
```

dann kann man innerhalb des Applet a1 nach folgendem Schema auf Felder xxxx und Methoden yyyy des Applets a2 zugreifen:

```
AppletContext browser = getAppletContext();
Applet2 otherApplet = (Applet2) browser.getApplet("a2");
x = otherApplet.xxxx;
otherApplet.yyyy();
otherApplet.xxxx.yyyy();
```

In dieser Skizze fehlt noch die Fehlerbehandlung, z.B. ob das andere Applet in der Web-Page gefunden wurde.

7.20.4 Zugriff auf JavaScript

Wenn ein Web-Browser sowohl Java als auch JavaScript [Seite 99] unterstützt, dann kann man auch zwischen Java-Applets und JavaScript-Scripts Daten und Informationen austauschen:

Innerhalb von JavaScript kann man public Methoden und Felder von Java-Applets in der Form
`document.appletname.feldname`
bzw.
`document.appletname.methodenaufruf`
ansprechen.

Innerhalb von Java-Applets kann man JavaScript-Funktionen aufrufen, wenn man die Klasse JSObject aus dem Package netscape.javascript (von Netscape) verwendet und im <applet>-Tag den Parameter `mayscript` angibt. Beispielskizze:

```
import netscape.javascript.*;
...
String[] somehtml = {
  "<h1>Hello!</h1>",
  "<p>Some <b>Java-generated</b> hypertext ..."
  };
JSObject w = JSObject.getWindow(this);
w.eval("w1 = window.open (’’,’windowname’,’’)");
w.call("w1.document.write", somehtml);
w.eval("w1.document.close()");
```

Achtung! JavaScript wird nicht [Seite 99] von allen Web-Browsern unterstützt, und viele Benutzer haben JavaScript wegen des damit verbundenen Sicherheitsrisikos [Seite 99] in ihren Browser-Optionen ausgeschaltet. JavaScript eignet sich daher *nicht* für allgemein verwendbare Applets über das Internet sondern *nur* für Spezialfälle innerhalb eines Intranet.

7.21 Übung: Zwei Applets

Schreiben Sie eine Web-Page mit zwei Applets:

- ein Resultat-Applet, das einen Text anzeigt, und
- ein Steuerungs-Applet, in dem der Benutzer einen Text eingibt, der dann im anderen Applet angezeigt wird.

7.22 Applet-Dokumentation

Um den Benutzern die richtige Verwendung eines Applet zu erleichtern, sollten die folgenden Hilfsmittel verwendet werden:

- spezielle Kommentare, aus denen mit Javadoc [Seite 28] die Dokumentation des Java-Programms erstellt wird, und
- die Methoden getAppletInfo() und getParameterInfo() im Applet, deren Ergebnisse von manchen Web-Browsern (z.B. dem Appletviewer) beim Anklicken von "Eigenschaften" oder "Info" angezeigt werden.

Beispiel:

```
import java.awt.*;
import java.applet.*;

/**
* an example for an Applet with javadoc documentation,
* Applet info, und parameter info.
* @since JDK 1.0
*/
public class AppletDoc extends Applet {

  String text;
  int fontSize;
  Font f;

  public void init () {
```

```
    text = getParameter("text");
    if (text == null)
      text = "Hello World!";
    try {
      fontSize = Integer.parseInt (
        getParameter("fontsize").trim() );
    } catch (Exception e) {
        fontSize = 12;
    }
    f = new Font ("Helvetica", Font.BOLD, fontSize);
  }

  public void paint (Graphics g) {
    g.setFont (f);
    g.drawString (text, 25, 50);
  }

  public String getAppletInfo() {
    String s =
      "AppletDoc = Hello World Applet with parameters";
    return s;
  }

  public String[][] getParameterInfo() {
    String[][] s = {
      { "text",     "String",  "Text to be displayed" },
      { "fontsize", "Integer", "Font size in points" }
    };
    return s;
  }
}
```

7.23 Doppelnutzung als Applet und Applikation

Man kann ein Java-Programm auch so schreiben, dass es sowohl als Applet innerhalb einer Web-Page als auch als lokale Applikation ausgeführt werden kann. Zu diesem Zweck muss das Programm

- eine Subklasse von Applet sein und
- eine main-Methode enthalten.

Die main-Methode wird nur bei Aufruf als Applikation ausgeführt und im Applet ignoriert. Sie muss ein Frame (als Ersatz für das Fenster des Web-Browsers) erzeugen und in diesem ein Objekt des Applet einfügen und dann die init- und start-Methoden des Applet aufrufen, die den Inhalt des Applet darstellen.

Außerdem sollte ein WindowListener implementiert werden, der beim Schließen des Frame die Applikation beendet (analog zum Schließen des Web-Browsers).

Falls das Applet eine stop-Methode enthält, muss diese von der Applikation vor deren Beendigung aufgerufen werden.

Beispiel:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class AppApp extends Applet
    implements WindowListener {

  public void init() {
    setLayout( new FlowLayout() );
    Label hello = new Label("Hello World!");
    add(hello);
    // add more content ...
    setVisible(true);
  }

  public static void main (String[] args) {
    AppApp a = new AppApp();
    Frame f = new Frame("AppApp");

    f.setLayout( new BorderLayout() );
    a.init();
    f.add (a, "Center");
    f.setSize(300,200);
    f.addWindowListener( a );
    f.setVisible(true);
    a.start();
  }

  public void windowClosing (WindowEvent e) {
    this.stop();
    System.exit(0);
  }
  public void windowClosed (WindowEvent e) { }
  public void windowOpened (WindowEvent e) { }
  public void windowIconified (WindowEvent e) { }
  public void windowDeiconified (WindowEvent e) { }
  public void windowActivated (WindowEvent e) { }
  public void windowDeactivated (WindowEvent e) { }
}
```

8 Threads

Napoleon konnte angeblich mehrere Arbeiten gleichzeitig erledigen. Kann das mein Computer auch?

- **Threads = Programmfäden**
- **mehrere Aktionen parallel**

8.1 Multithreading, Thread, Runnable

Threads ("Fäden") sind Programmteile, die parallel, also unabhängig voneinander, mehr oder weniger gleichzeitig ablaufen.

Es gibt Threads, die von der Java Virtual Machine automatisch erzeugt werden, z.B. für das Event-Handling [Seite 73] und den Garbage-Collector [Seite 17] . Für die Aufspaltung eigener Programme in mehrere Threads dienen Objekte der folgenden beiden Klassen:

- die Klasse Thread [Seite 103] für die Steuerung von Threads (starten und beenden),
- das Interface Runnable [Seite 103] , mit dem das innerhalb des Thread laufende Programm implementiert wird.

Das Programm, das innerhalb des Thread [Seite 103] läuft, muss das Interface Runnable implementieren und eine Methode mit der Signature

- `public void run()`

enthalten, die dann innerhalb des Thread ausgeführt wird.

Beispiel:

```
public class MyProgram implements Runnable {
  public void run() {
    System.out.println("Hello World!");
  }
}
```

8.2 Starten eines Thread (start)

Konstruktor:

- `new Thread (Runnable)`

Der Parameter gibt an, welches Programm innerhalb des Thread laufen soll (siehe oben).

Das Starten des Thread erfolgt mit der Methode

- `start()`

Damit wird die run-Methode des Runnable-Objekts gestartet.

Beispielskizze:

```
Thread t;
Runnable theProgram = new MyProgram();
t = new Thread (theProgram);
t.start();
```

Der Zusammenhang zwischen **Runnable**-Objekt und **Thread**-Objekt ist ähnlich wie zwischen einem **Auto** und seinem **Lenker**:

- Das Runnable-Objekt entspricht dem Auto.
- das Thread-Objekt entspricht dem Lenker, der das Auto steuert.
- Jedes Auto muss von einer Person gelenkt werden, und jede Person kann nur ein Auto lenken.
- Der Konstruktor des Thread-Objekts entspricht der Übergabe des Autoschlüssels an den Lenker: Damit wird festgelegt, welches Auto er lenken soll.
- Die start-Methode entspricht dem Starten mit dem Startschlüssel, die run-Methode entspricht dem Laufen des Motors: Der Lenker dreht den Startschlüssel, und das bewirkt, dass der Motor zu laufen beginnt.

8.3 Beenden oder Abbrechen eines Thread

Der Thread läuft selbständig (also unabhängig von dem Thread, der ihn gestartet hat, siehe auch unten) und endet automatisch, wenn das Ende der run-Methode erreicht wird.

Falls die Aktionen des Thread eine längere Ausführungszeit benötigen oder in einer Schleife wiederholt werden, muss man eine Möglichkeit vorsehen, den Thread "von außen" zu beenden oder abzubrechen.

Dazu verwendet man am besten ein boolean Datenfeld, das von außen gesetzt werden kann und innerhalb des Runnable-Objekts abgefragt wird.

Beispielskizze:

```
Thread t;
MyProgram myProg = new MyProgram();
t = new Thread (myProg);
t.start();
...
myProg.setRunFlag(false);

public class MyProgram implements Runnable {
  private volatile boolean runFlag;
  public void setRunFlag (boolean runFlag) {
    this.runFlag = runFlag;
  }
  public boolean getRunFlag() {
    return this.runFlag;
```

```
  }
  public void run() {
    runFlag=true;
    while (runFlag) {
      // do something
    }
  }
}
```

Es gibt auch Methoden stop(), interrupt(), suspend() und resume() für das Beenden, Abbrechen oder Unterbrechen eines Thread. Die Verwendung dieser Methoden wird jedoch *nicht* empfohlen ("deprecated" ab JDK 1.2), weil es dabei entweder zu Deadlock-Situationen oder zu unvollständig ausgeführten Aktionen und damit zu ungültigen Objekt- oder Datenzuständen kommen kann.

Beispiel: Wenn ein Thread, der mehrere zusammengehörige Daten in eine oder mehrere Files oder Datenbanken schreiben will, mit einer dieser Methoden abgebrochen wird, so kann es passieren, dass nur ein Teil der neuen Daten abgespeichert wird und damit ein ungültiger, inkonsistenter Datenzustand entsteht.

8.4 Unterbrechungen (sleep)

Man kann nicht davon ausgehen, dass die Rechenzeit "gerecht" auf alle parallel laufenden Threads aufgeteilt wird. Es kann durchaus passieren, dass ein Thread alle Betriebsmittel für sich verbraucht und alle anderen Threads dadurch aufgehalten werden und untätig warten müssen. Dies kann auch durch das Setzen von Prioriäten mit setPriority() oder yield() nicht völlig ausgeschlossen werden.

Deshalb muss man bei parallel laufenden Threads immer dafür sorgen, dass jeder von ihnen Pausen einlegt, in denen die anderen Threads eine Chance bekommen, ihrerseits zu laufen. Dies gilt auch für das Programm, das den Thread mit start() gestartet hat und nun will, dass dieser auch tatsächlich läuft.

Die wichtigste Warte-Methode ist die statische Methode

- `Thread.sleep (long)`
  für eine Pause mit einer bestimmten Zeitdauer (in Millisekunden)

Beim Aufruf von sleep muss die Fehlersituation InterruptedException abgefangen werden, für den Fall, dass der Thread während der Wartezeit gestoppt oder abgebrochen wird.

Besonders wichtig ist die sleep-Methode, wenn die run-Methode eine Schleife oder Endlos-Schleife ist, die bestimmte Aktionen immer wieder wiederholt (so lange, bis der Thread beendet wird). Beispiel:

```
public class MyProgram implements Runnable {
  private volatile boolean runFlag;
  public void setRunFlag (boolean runFlag) {
    this.runFlag = runFlag;
  }
  public boolean getRunFlag() {
    return this.runFlag;
  }
  public void run() {
    runFlag=true;
    while (runFlag) {
```

```
      // do something
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) { }
    }
  }
}
```

8.5 Synchronisierung (join, wait, notify, synchronized)

Wenn ein Thread die Ergebnisse verwenden will, die ein anderer Thread produziert, muss der eine Thread auf den anderen warten und der andere dem einen mitteilen, wann er mit seiner Aufgabe fertig ist. Dazu dienen die folgenden Methoden:

Methode im Thread-Objekt:

- `join()`
  für das Abwarten, bis der gestartete Thread fertig gelaufen ist (d.h. sein Ende erreicht hat)

Methoden in jedem Objekt (jeder Subklasse von Object):

- `wait()`
  für das Abwarten, bis der gestartete Thread ein notify() für dieses Objekt sendet
- `wait(long timeout)`
  ebenso, aber mit einem Timeout (maximale Wartezeit im Millisekunden)
- `notify()`
  für das Beenden eines wait()-Zustandes für dieses Objekt, wenn ein oder mehrere Threads mit wait() darauf warten; sonst ohne Bedeutung
- `notifyAll()`
  für das Beenden von allen eventuell für dieses Objekt wartenden wait()-Zuständen

Dabei muss darauf geachtet werden, dass keine Deadlock-Situation auftritt (Prozess A wartet, dass Prozess B ein notify() sendet, und Prozess B wartet, dass Prozess A ein notify() sendet).

Wenn zwei oder mehrere parallel laufende Threads das selbe Objekt ansprechen, kann es zu Problemen kommen, wenn beide gleichzeitig das Objekt verändern wollen und dabei auf vom anderen Thread halb ausgeführte Zwischenzustände zugreifen (concurrent update problem). In diesem Fall muss man *alle* "gefährlichen" Aktionen jeweils in einen synchronized-Block der folgenden Form einfügen:

```
synchronized (Object) {
  Statements;
}
```

wobei für Object meistens `this` anzugeben ist, oder die jeweilige Methode insgesamt als synchronized deklarieren:

```
public synchronized typ name (parameter) {
  Statements;
}
```

Dies bewirkt, dass der mit synchronized markierte Statement-Block (bzw. die mit synchronized markierte Methode) nur dann ausgeführt wird, wenn kein anderer, ebenfalls mit synchronized markierter Statement-Block für dieses Objekt läuft; andernfalls wartet der neue Statement-Block, bis der alte Statement-Block fertig ist und sich das Objekt daher wieder in einem gültigen Zustand befindet, und führt dann seine Aktionen ebenfalls komplett und ungestört aus.

Die Methoden wait() und notify() sind nur innerhalb von synchronized Blocks oder Methoden erlaubt.

Ab JDK 1.3 kann man auch sogenannte "Piped Streams" zur Kommunikation von Daten und von Wartezuständen zwischen zwei Threads verwenden (Klassen PipedWriter und PipedReader im Paket java.io): Jedes Lesen aus dem PipedReader in dem einen Thread wartet, bis vom anderen Thread entsprechende Daten in den zugehörigen PipedWriter geschrieben werden.

8.6 Beispiel: einfacher HelloWorld-Thread

Wir erweitern unser Hello-World-Programm [Seite 40] nun so, dass es das Runnable-Interface implementiert:

```
public class HelloLoop implements Runnable {

  private String messageText = "Hello World!";

  public void setMessageText(String newText) {
    messageText = newText;
  }

  public String getMessageText() {
    return messageText;
  }

  public void printText() {
    System.out.println (messageText);
  }

  public void run() {
    for (int i=1; i<=5; i=i+1) {
      printText();
      try { Thread.sleep(50); }
      catch (InterruptedException e) {}
    }
  }

}
```

Zunächst schreiben wir eine main-Methode, die direkt die run-Methoden aufruft:

```
public class HelloRun {
  public static void main (String[] args) {
    HelloLoop engl = new HelloLoop();
    HelloLoop germ = new HelloLoop();
    germ.setMessageText("Hallo, liebe Leute!");
    HelloLoop cat = new HelloLoop();
    cat.setMessageText("Miau!");

    engl.run();
```

```
    germ.run();
    cat.run();
  }
}
```

Dieses Programm läuft als ein einziger Thread, in dem die Methodenaufrufe nach einander ausgeführt werden, und bewirkt die folgenden Ausgabe:

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hallo, liebe Leute!
Hallo, liebe Leute!
Hallo, liebe Leute!
Hallo, liebe Leute!
Hallo, liebe Leute!
Miau!
Miau!
Miau!
Miau!
Miau!
```

Nun schreiben wir eine main-Methode, die drei parallel laufende Threads erzeugt:

```
public class HelloThreads {
  public static void main (String[] args) {
    HelloLoop engl = new HelloLoop();
    HelloLoop germ = new HelloLoop();
    germ.setMessageText("Hallo, liebe Leute!");
    HelloLoop cat = new HelloLoop();
    cat.setMessageText("Miau!");

    Thread te = new Thread (engl);
    Thread tg = new Thread (germ);
    Thread tc = new Thread (cat);
    te.start();
    tg.start();
    tc.start();
  }
}
```

Dieses Programm bewirkt läuft im Multi-Threading und bewirkt eine Ausgabe wie zum Beispiel:

```
Hello World!
Hallo, liebe Leute!
Miau!
Hello World!
Miau!
Hallo, liebe Leute!
Hello World!
Miau!
Hallo, liebe Leute!
Hello World!
Miau!
Hallo, liebe Leute!
Hello World!
Hallo, liebe Leute!
Miau!
```

wobei sich je nach der Prozessorbelastung auch andere Reihenfolgen ergeben können.

8.7 Beispiel: eine einfache Animation

```
import java.awt.* ;
import java.applet.* ;

public class Anima extends Applet
  implements Runnable {

  private volatile boolean runFlag;
  private int x, y, height, width;
  private Image img;
  private Graphics g;
  private Thread t = null;
  private Color nightColor = new Color (0, 0, 102);
  private Color moonColor  = new Color (204, 204, 255);

  public void init() {
    Dimension d = getSize();
    width = d.width;
    height = d.height;
    img = createImage (width, height);
    g=img.getGraphics();
    x = width/2;
    y = height/2;
  }

  public void start() {
    if (t == null)
      {
      t = new Thread (this);
      t.start();
      }
  }
  public void stop() {
    if (t != null)
      {
      runFlag=false; // or: t.stop();
      t=null;
      }
  }

  public void run () {
    runFlag=true;
    while (runFlag) {
      g.setColor (nightColor);
      g.fillRect(0,0,width,height);
      g.setColor (moonColor);
      g.fillArc(x, y-25, 50, 50, 270, 180);
      repaint();
      x = x + 2;
      if (x > width+50 ) {
        x = -50;
      }
      try { Thread.sleep(100); }
      catch (InterruptedException e) {}
    }
```

```
  }

  public void update (Graphics g) {
    paint(g);
  }

  public void paint (Graphics g) {
    if (img != null) {
      g.drawImage(img,0,0,null);
    }
  }

}
```

Wenn Sie wissen wollen, was dieses Applet tut, probieren Sie es einfach aus (in Ihrem Browser, wenn er das kann, oder Abschreibübung oder Download).

8.8 Übung: einfaches Applet Blinklicht

Schreiben Sie ein einfaches Applet, das eine gelb blinkende Verkehrsampel zeigt (schwarzes Rechteck mit oder ohne gelbem Kreis).

Wenn Sie sehen wollen, wie das aussehen könnte, probieren Sie es in Ihrem Browser aus (falls er das kann).

8.9 Übung: erweitertes Applet Verkehrsampel mit Blinklicht

Erweitern Sie das Applet Verkehrsampel (siehe die Übung im Kapitel Applets [Seite 92]) in folgender Weise:

- Fügen Sie einen weiteren Button "Blink" hinzu.
- Wenn dieser Blink-Button angeklickt wird, dann soll die Verkehrsampel gelb blinken (wie in der vorhergehenden einfachen Übung).
- Das Blinken soll enden und die Verkehrsampel wieder die normalen Farben zeigen, wenn der Blink-Button nochmals angeklickt wird, oder wenn der Next- oder Stop-Button angeklickt wird, oder wenn die Web-Page verlassen oder das Applet oder der Web-Browser beendet wird.

Wenn Sie sehen wollen, wie das aussehen könnte, probieren Sie es in Ihrem Browser aus (falls er das kann).

System-Interfaces

- Ein-Ausgabe (IO) [Seite 112]
- Networking (Web-Server, Sockets) [Seite 130]
- System-Funktionen und Utilities [Seite 138]
- Datenbanken [Seite 152]

9 Ein-Ausgabe (IO)

Wir haben bereits gelernt, wie die Ein- und Ausgabe in Graphischen User-Interfaces programmiert wird. Nun wollen wir uns auch damit beschäftigen, wie wir Daten von Dateien einlesen und in Dateien speichern können.

- **Ja, das Schreiben und das Lesen ...**

Das Lesen und Schreiben von lokalen Dateien (Files) ist im Allgemeinen nur für Applikationen, aber nicht für Applets erlaubt. Applets können im Allgemeinen nur Files auf dem Server ansprechen, von dem sie geladen wurden (siehe die Kapitel über das Lesen von einem Web-Server [Seite 95] und über Networking [Seite 130]).

Für die Verwendung der Ein-Ausgabe-Klassen muss das Package java.io importiert werden.

Alle Ein-Ausgabe-Operationen können bestimmte Fehlersituationen auslösen (IOException, SecurityException, FileNotFoundException u.a.), die mit try und catch abgefangen werden sollen (siehe das Kapitel über Exceptions [Seite 55]).

9.1 Dateien (Files) und Directories

Die folgenden Datenströme ("Systemdateien") stehen innerhalb der System-Klasse statisch zur Verfügung und müssen nicht explizit deklariert werden:

- `PrintStream System.out` für die Standard-Ausgabe
- `PrintStream System.err` für die Fehlerausgabe
- `InputStream System.in` für die Standard-Eingabe

Andere Files kann man im einfachsten Fall ansprechen, indem man den Filenamen im Konstruktor eines Datenstromes [Seite 114] angibt (siehe unten).

Für kompliziertere Aktionen mit Dateien und Directories können File-Objekte mit den folgenden Konstruktoren angelegt und mit den folgenden Methoden verwendet werden:

9.1.1 Deklaration eines Files:

```
File f = new File (filename);

File f = new File (directory, filename);
```

Der Filename ist als String anzugeben. Er kann auch den kompletten Pfad enthalten, also mit Schrägstrichen oder Backslashes oder Doppelpunkten, je nach Betriebssystem, die Applikation ist dann aber nicht plattformunabhängig. Statt diesen File-Separator explizit im String anzugeben, sollte man deshalb immer den am jeweiligen System gültigen Separator mit dem statischen Datenfeld
`File.separator`
ansprechen und mit + in den String einbauen.

Die Version des Konstruktors mit zwei Parametern ist dann sinnvoll, wenn man mehrere Files in einem Directory ansprechen will, oder wenn das Directory innerhalb des Programms bestimmt oder vom Benutzer ausgewählt wird. Für den Directory-Namen kann entweder ein String (wie oben) oder ein File-Objekt, das ein Directory beschreibt, angegeben werden.

Das Öffnen oder Anlegen des Files erfolgt erst dann, wenn das File-Objekt im Konstruktor eines Datenstromes [Seite 114] (InputStream, OutputStream, Reader, Writer, siehe unten) verwendet wird.

9.1.2 Deklaration eines Directory:

Directories werden wie Files deklariert und "geöffnet":

```
File dir = new File (dirname);

File dir = new File (parentdir, dirname);
```

Mit den Methoden isFile() und isDirectory() kann abgefragt werden, ob es sich um ein File oder um ein Directory handelt. Das Anlegen eines Directory erfolgt mit der Methode mkdir().

9.1.3 Methoden

Die wichtigsten Methoden für File-Objekte (Dateien und Directories) sind:

- `String getName()`
  für den File- bzw. Directory-Namen ohne Pfad
- `String getPath()` und `getAbsolutePath()`
  für den File- bzw. Directory-Namen mit Pfad
- `String getParent()`
  für das Parent-Directory
- `boolean exists(), boolean canWrite(), boolean canRead()`
  für die Existenz und die Permissions
- `boolean isFile(), boolean isDirectory()`
  für die Art (File oder Directory)
- `long length()`
  für die Länge in Bytes
- `long lastModified()`
  ist eine system-interne Zahl, die das Alter des File-Inhalts angibt (nur für relative Vergleiche brauchbar, nicht für Datums-Angaben)
- `boolean mkdir()`
  für das Anlegen des Directory (liefert true, wenn erfolgreich durchgeführt, sonst false)
- `String[] list()`
  für eine Liste von allen in einem Directory enthaltenen File- und Directory-Namen. Wenn man diese Files oder Subdirectories ansprechen will, muss man mit Konstruktoren wie `new File(name[i])` File-Objekte dafür erzeugen.
- `static File[] listRoots()`
  für eine Liste von allen Filesystem-Anfängen ("Laufwerken" unter MS-Windows).
- `boolean renameTo (new File(filename))`
  für die Änderung des File- oder Directory-Namens (liefert true, wenn erfolgreich durchgeführt, sonst false)
- `boolean delete()`
  für das Löschen des Files oder Directory (liefert true, wenn erfolgreich durchgeführt, sonst false)

9.2 Datenströme (stream)

Im Sinne der Grundsätze der objekt-orientierten [Seite 31] Programmierung gibt es in Java verschiedene Klassen und Objekte für Datenströme, die je nach der Anwendung entsprechend kombiniert werden müssen. Dabei sind die Spezialfälle jeweils Unterklassen der einfachen Fälle (Vererbung [Seite 47]), alle Ein-Ausgabe-Operationen können also direkt am Objekt der Unterklasse durchgeführt werden.

- Einfache, grundlegende Stream-Klassen definieren auf einer niedrigen Ebene die Ein- und Ausgabe in Dateien (Files) oder auch in anderen Datenströmen wie z.B. Pipes, Byte-Arrays oder Netzverbindungen (Sockets [Seite 133] , URLs [Seite 95]).
- Spezielle, darauf aufbauende Stream-Klassen definieren zusätzliche Eigenschaften wie z.B. Pufferung, zeilenorientierte Text-Ausgabe, automatische Zeilennummerierung und dergleichen.

Es gibt in Java zwei verschiedene Gruppen von Datenströmen:

- InputStream [Seite 114] , OutputStream [Seite 116] und RandomAccessFile [Seite 121] für byte-orientierte **Datenfiles**,
- Reader [Seite 121] und Writer [Seite 121] für zeichen- und zeilen-orientierte **Textfiles**.

9.3 InputStream (Eingabe)

InputStream ist die Oberklasse für das Lesen von Datenströmen, also für die Byte-orientierte Eingabe (siehe auch die Alternative Reader [Seite 121] für die Zeichen- und Zeilen-orientierte Eingabe).

Es werden zwei Arten von InputStream-Klassen verwendet:

- Klassen, die angeben, **von wo** gelesen werden soll:
  FileInputStream, URL, Socket, ServletRequest, Process, ByteArrayInputStream, ...
- Klassen, die angeben, **wie** gelesen werden soll:
  BufferedInputStream, DataInputStream, ObjectInputStream, ...

Bei den Klassen der ersten Gruppe wird im Konstruktor angegeben, wo der Datenstrom herkommt (File, Internet-Adresse, Speicherbereich).

Bei den Klassen der zweiten Gruppe wird im Konstruktor irgendein anderer InputStream angegeben. Diese Klassen und die von ihnen unterstützten Funktionalitäten können daher beliebig kombiniert werden.

9.3.1 InputStream - Methoden

Die wichtigsten Methoden für InputStreams sind:

- `int read()`
  liefert ein Byte, oder den Wert -1, wenn das Ende des Datenstroms (End-of-File) erreicht wurde.
  Beispiel:
  ```
  int b = infile.read();
  ```

- `int read (byte[])`
  `int read (byte[], offset, length)`
  liest so viele Zeichen wie möglich in das Byte-Array bzw. in den angegebenen Teilbereich des Byte-Array und liefert die Anzahl der gelesenen Zeichen (eventuell 0), oder den Wert -1, wenn das Ende des Datenstroms (End-of-File) erreicht wurde. Beispiel:
  ```
  byte[] b = new byte[1024];
  int n = infile.read (b[]);
  ```
- `skip(long)`
  überspringt die angegebene Anzahl von Bytes im Eingabestrom.
- `close()`
  schließt den Eingabestrom bzw. das File.

9.3.2 Byte-weises Lesen

Ein typisches Beispiel für das Byte-weise Lesen eines Daten-Files hat folgenden Aufbau:

```
int b;
try {
  BufferedInputStream in = new BufferedInputStream (
    new FileInputStream ("filename.dat") );
  while( (b = in.read()) != -1 ) {
    // do something ...
  }
  in.close();
} catch (Exception e) {
    System.out.println("error " + e);
}
```

Wenn die Daten von einer anderen Quelle kommen, muss nur der Parameter im Konstruktor von BufferedInputStream entsprechend geändert werden, der Rest des Programms bleibt unverändert (siehe Lesen von einem Web-Server [Seite 95]).

9.3.3 FileInputStream

FileInputStream ist die grundlegende Klasse für das Lesen von Dateien (siehe auch die Alternative FileReader [Seite 121]).

Das Öffnen des Lese-Stroms bzw. des Files erfolgt mit einem Konstruktor der Form

```
FileInputStream infile = new FileInputStream (filename);
```

Der Filename kann entweder als String oder als File-Objekt angegeben werden (siehe oben).

9.3.4 BufferedInputStream

Um die Eingabe effizienter und schneller zu machen, soll nicht jedes Byte einzeln von der Hardware gelesen werden, sondern aus einem Pufferbereich. Daher sollte fast immer ein BufferedInputStream über den einfachen FileInputStream gelegt werden.

Konstruktoren:

- `new BufferedInputStream (InputStream)`
  für einen Puffer mit Standardgröße
- `new BufferedInputStream (InputStream, int)`

für einen Puffer mit der angegebenen Größe

Beispiel:

```
BufferedInputStream in = new BufferedInputStream
  ( new FileInputStream (filename) );
```

9.4 Lesen einer Datei von einem Web-Server (URL)

Mit den Klassen URL (uniform resource locator) und HttpURLConnection im Paket java.net kann eine beliebige Datei von einem Web-Server oder FTP-Server gelesen werden (bei Applikationen von einem beliebigen Server, bei einem Applet im Allgemeinen nur von dem Web-Server, von dem es geladen wurde). Mit der Methode openStream() erhält man einen InputStream, der dann wie ein FileInputStream verwendet werden kann.

Beispiel:

```
import java.io.*;
import java.net.*;
...
BufferedInputStream in;
URL url;
try {
  url = new URL( "http://www.xxx.com/yyy" );
  in = new BufferedInputStream ( url.openStream() );
  while( (b = in.read()) != -1 ) {
    // do something ...
  }
  in.close();
} catch (Exception e) {
    System.out.println("error " + e);
}
```

Dies eignet sich bis JDK 1.1 primär für das Lesen von Textfiles (siehe InputStreamReader [Seite 121]) und Datenfiles (siehe DataInputStream). Ab JDK 1.2 gibt es auch eigene Klassen für die Interpretation von HTML-Files (JEditorPane, HTMLDocument, HTMLReader).

Komplexere Möglichkeiten für Ein-Ausgabe-Operationen über das Internet sind im Kapitel über Networking [Seite 130] beschrieben.

9.5 OutputStream (Ausgabe)

OutputStream ist die Oberklasse für das Schreiben von Datenströmen, also für die Byte-orientierte Ausgabe (siehe auch die Alternative Writer [Seite 121] für die Zeichen- und Zeilen-orientierte Ausgabe).

Es werden zwei Arten von OutputStream-Klassen verwendet:

- Klassen, die angeben, **wohin** geschrieben werden soll:
  FileOutputStream, HttpURLConnection, Socket, ServletResponse, Process,
  ByteArrayOutputStream, ...

- Klassen, die angeben, **wie** geschrieben werden soll:
  BufferedOutputStream, DataOutputStream, ObjectOutputStream, ...

Bei den Klassen der ersten Gruppe wird im Konstruktor angegeben, wo der Datenstrom hingeht (File, Internet-Adresse, Speicherbereich).

Bei den Klassen der zweiten Gruppe wird im Konstruktor irgendein anderer OutputStream angegeben. Diese Klassen und die von ihnen unterstützten Funktionalitäten können daher beliebig kombiniert werden.

9.5.1 OutputStream - Methoden

Die wichtigsten Methoden für OutputStreams sind:

- `write(int)`
  schreibt ein Byte mit den angegeben Wert
- `write (byte[])`
  `write (byte[], offset, length)`
  schreibt alle im Byte-Array bzw. im angegebenen Teilbereich des Byte-Array enthaltenen Bytes
- `flush()`
  sorgt dafür, dass alle eventuell noch im Puffer wartenden Bytes tatsächlich in die Datei geschrieben werden (sollte immer vor dem Close bzw. vor der Beendigung oder einem Abbruch des Programms aufgerufen werden)
- `close()`
  schließt den Ausgabestrom bzw. das File.

9.5.2 Byte-weises Schreiben

Ein typisches Beispiel für das Byte-weise Schreiben eines Daten-Files hat folgenden Aufbau:

```
try {
  BufferedOutputStream out = new BufferedOutputStream (
    new FileOutputStream ("filename.dat") );
  ...
  out.write(...);
  ...
  out.flush();
  out.close();
} catch (Exception e) {
    System.out.println("error " + e);
}
```

9.5.3 FileOutputStream

FileOutputStream ist die grundlegende Klasse für das Schreiben von Dateien (siehe auch die Alternative FileWriter [Seite 121]).

Das Öffnen des Ausgabe-Stroms bzw. des Files erfolgt mit Konstruktoren der Form

- `new FileOutputStream (filename);`
- `new FileOutputStream (filename, false);`
- `new FileOutputStream (filename, true);`

Der Filename kann entweder als String oder als File-Objekt angegeben werden (siehe oben). In den ersten beiden Fällen wird die Datei neu geschrieben oder überschrieben, im dritten Fall (Appendmode true) werden die Informationen an das Ende einer bereits existierenden Datei hinzugefügt.

In JDK 1.0 ist die Angabe des Appendmode nicht möglich, dort muss für das Hinzufügen an eine bereits bestehende Datei stattdessen die Klasse RandomAccessFile [Seite 121] verwendet werden (siehe unten).

9.5.4 BufferedOutputStream

Um die Ausgabe effizienter und schneller zu machen, soll nicht jedes Byte einzeln auf die Hardware geschrieben werden, sondern aus einem Pufferbereich. Daher sollte fast immer ein BufferedOutputStream über den einfachen FileOutputStream gelegt werden.

Konstruktoren:

- `new BufferedOutputStream (OutputStream)`
  für einen Puffer mit Standardgröße
- `new BufferedOutputStream (OutputStream, int)`
  für einen Puffer mit der angegebenen Größe

Beispiel:

```
BufferedOutputStream out = new BufferedOutputStream
  ( new FileOutputStream (filename) );
```

9.5.5 PrintStream

Die Klasse PrintStream enthält die Methoden print und println, mit denen Datenfelder und Objekte jeweils in die für Menschen lesbare Textdarstellung umgewandelt werden. Sie dient seit JDK 1.1 nur mehr für die Text-Ausgabe auf die System-Console mit **System.out** und **System.err**. In allen anderen Fällen soll man für die zeichen- und zeilenorientierte Ausgabe lieber die Klasse PrintWriter [Seite 121] verwenden (siehe unten)

9.5.6 ByteArrayOutputStream und ByteArrayInputStream

Die Klasse ByteArrayOutputStream dient dazu, Bytes oder Daten mit write-Befehlen in einen Speicherbereich statt in ein File zu schreiben.

Die Klasse ByteArrayInputStream dient dazu, mit read-Befehlen die Bytes oder Daten aus einem mit ByteArrayOutputStream beschriebenen Speicherbereich zu lesen, also in seine Einzelteile zu zerlegen.

9.6 Data- und Object-Streams (Serialisierung)

Wenn man nicht einzelne Bytes sondern ganze Datenfelder oder Objekte lesen und schreiben will, kann man dafür die Klassen Data- und Object- Input- und Output-Stream verwenden. Dies entspricht dem Begriff **"binäre Datenfiles"** in anderen Programmiersprachen.

Im Gegensatz zu anderen Programmiersprachen ist bei Java die Bit- und Byte-weise Speicherung von Datenfeldern und Objekten nicht systemabhängig sondern für alle Java-Programme einheitlich festgelegt. Man kann also ein solches binäres Datenfile, das mit Java auf einem Computer erstellt wurde, mit Java auf jedem beliebigen anderen Computer lesen und verarbeiten.

9.6.1 DataOutputStream

Der DataOutputStream erlaubt es, bestimmte primitive Datentypen direkt in ihrer (plattformunabhängigen) Java-internen Darstellung zu schreiben ("binäres Datenfile", kann dann mit DataInputStream gelesen werden).

Konstruktor:

- `new DataOutputStream (OutputStream)`

Beispiel:

```
DataOutputStream out = new DataOutputStream
  ( new BufferedOutputStream
    ( new FileOutputStream (filename) ) );
```

Methoden:

- `writeBoolean(boolan)`
- `writeByte(int)`
- `writeBytes(String)`
- `writeInt(int)`
- `writeLong(long)`
- `writeFloat(float(`
- `writeDouble(double)`
- und so weiter

9.6.2 DataInputStream

Der DataInputStream erlaubt es, bestimmte primitive Datentypen direkt einzulesen, d.h. die Daten müssen in der plattformunabhängigen Java-internen Darstellung im Input-Stream stehen ("binäres Datenfile", z.B. mit DataOutputStream geschrieben).

Konstruktor:

- `new DataInputStream (InputStream)`

Beispiel:

```
DataInputStream in = new DataInputStream
  ( new BufferedInputStream
    ( new FileInputStream (filename) ) );
```

Methoden:

- `boolean readBoolean()`
- `byte readByte()`
- `int readInt()`
- `long readLong()`

- `float readFloat()`
- `double readDouble()`
- und so weiter

Es gibt hier auch eine String-Methode readLine(), es wird aber empfohlen, für die Verarbeitung von Textzeilen lieber die Klasse BufferedReader [Seite 121] zu verwenden (siehe unten).

9.6.3 ObjectOutputStream (Serialisierung)

Der ObjectOutputStream hat die gleiche Funktionalität wie der DataOutputStream, erlaubt es aber zusätzlich auch, ganze Objekte in der Java-internen Darstellung zu schreiben (kann dann mit ObjectInputStream gelesen werden). Es ist üblich, solche Filenamen mit der Extension ".ser" zu versehen.

Dazu muss das Objekt das Interface **Serializable** implementieren, das angibt, dass sich das Objekt "serialisieren", also als eine Folge von Bits und Bytes speichern lässt, die alle seine Datenfelder enthalten. Das Interface Serializable ist ein sogenanntes Marker-Interface, d.h. man muss keine Methoden implementieren, sondern es kommt nur darauf an, ob das Interface geerbt wird oder nicht. Fast alle Objekte sind serialisierbar.

Konstruktor:

- `new ObjectOutputStream (OutputStream)`

Methoden:

- wie beim DataOutputStream, und zusätzlich
- `writeObject(Object)`

Anmerkung: Object ist die Superklasse für alle Klassen, der Parameter von writeObject kann also ein Objekt einer beliebigen Klasse sein. Beispiel:

```
Date d;
...
out.writeObject (d);
```

9.6.4 ObjectInputStream

Der ObjectInputStream hat die gleiche Funktionalität wie der DataInputStream, erlaubt es aber zusätzlich auch, ganze Objekte zu lesen, die im Input-Stream in der Java-internen Darstellung stehen (Serialisierung, z.B. mit ObjectOutputStream geschrieben, Filenamen der Form xxxx.ser).

Konstruktor:

- `new ObjectInputStream (InputStream)`

Methoden:

- wie beim DataInputStream, und zusätzlich
- `Object readObject()`
  für beliebige Objekte

Anmerkung: Object ist die Superklasse für alle Klassen. Das Ergebnis von readObject() muss mit "Casting" auf den Typ des tatsächlich gelesenen Objekts umgewandelt werden. Beispiel:

```
Date d;
...
d = (Date) in.readObject();
```

9.7 RandomAccessFile (Ein- und Ausgabe)

Random-Access-Files erlauben es, Daten nicht nur in einem Datenstrom vom Anfang bis zum Ende zu lesen oder zu schreiben, sondern an bestimmte Stellen eines Files zu "springen" und dort direkt zu lesen und zu schreiben. Dabei ist auch ein Abwechseln zwischen Lese- und Schreiboperationen möglich.

Konstruktor:

- `new RandomAccessFile (filename, mode)`

Der Filename kann als String oder als File-Objekt angegeben werden.

Der Mode ist ein String, der entweder nur "r" oder "rw" enthält. Im ersten Fall kann der File-Inhalt nur gelesen werden, im zweiten Fall (auch) geschrieben bzw. verändert werden.

Methoden:

Es stehen alle für DataInputStream und für DataOutputStream definierten Methoden zur Verfügung, also insbesondere read, write, close, flush, und außerdem die folgenden Methoden:

- `long getFilePointer()`
  liefert die aktuelle Position innerhalb des Files
- `long length()`
  liefert die Länge des Files
- `seek (long)`
  setzt den FilePointer auf eine bestimmte Stelle für das nächste read oder write

Wenn man Daten an das Ende einer bereits existierenden Datei hinzufügen will, kann man diese Datei als RandomAccessFile mit mode "rw" öffnen und mit
`file.seek( file.length() );`
an das File-Ende positionieren, bevor man den write-Befehl ausführt. Ab JDK 1.1 kann man dies aber einfacher mit dem Appendmode-Parameter im Konstruktor des FileOutputStream [Seite 116] bzw. FileWriter [Seite 121] erreichen.

9.8 Reader und Writer (Text-Files)

Java als plattformunabhängige Sprache verwendet den Zeichencode "Unicode" (16 bit pro Zeichen) für die Verarbeitung von allen weltweit verwendeten Schriftzeichen. Die Reader- und Writer-Klassen unterstützen die Umwandlung zwischen diesem 16 bit Unicode und dem auf dem jeweiligen Rechnersystem verwendeten Zeichencode für Textfiles (meist 8 bit). Dies kann der in

englisch-sprachigen und westeuropäischen Ländern verwendete Zeichencode Iso-Latin-1 (ISO-8859-1) oder ein anderer Zeichencode sein, z.B. für osteuropäische lateinische Schriften (Iso-Latin-2) oder für kyrillische, griechische, arabische, hebräische oder ostasiatische Schriften, oder die Unicode-Kodierung mit UTF-8, oder auch ein Computer-spezifischer Code wie z.B. für MS-DOS, MS-Windows oder Apple Macintosh.

Für die Verarbeitung von Textfiles sollten daher ab JDK 1.1 Reader und Writer statt Input-Streams und Output-Streams verwendet werden. Diese Klassen enthalten nicht nur Methoden für das Lesen und Schreiben von einzelnen Zeichen (read, write) sondern auch zusätzliche Methoden für das Lesen und Schreiben von ganzen Zeilen (readLine, newLine) sowie für die Darstellung von Zahlen und anderen Datentypen als menschenlesbare Texte (print, println).

9.9 Reader

Reader ist die Oberklasse für das Lesen von Texten, also für die zeichen- und zeilenorientierte Eingabe.

Es werden zwei Arten von Reader verwendet:

- Klassen, die angeben, **von wo** gelesen werden soll:
  FileReader, InputStreamReader, StringReader, ...
- Klassen, die angeben, **wie** gelesen werden soll:
  BufferedReader, LineNumberReader, ...

Bei den Klassen der ersten Gruppe wird im Konstruktor angegeben, wo der Datenstrom herkommt (File, InputStream, Speicherbereich).

Bei den Klassen der zweiten Gruppe wird im Konstruktor irgendein anderer Reader angegeben. Diese Klassen und die von ihnen unterstützten Funktionalaitä:ten können daher beliebig kombiniert werden.

9.9.1 Reader - Methoden

Die wichtigsten Methoden bei den Reader-Klassen sind:

- `int read()`
  liefert ein Zeichen, oder den Wert -1, wenn das Ende des Datenstroms (End-of-File) erreicht wurde.
- `int read (char[])`
  `int read (char[], offset, length)`
  liest so viele Zeichen wie möglich in das Zeichen-Array bzw. in den angegebenen Teilbereich des Zeichen-Array und liefert die Anzahl der gelesenen Zeichen (eventuell 0), oder den Wert -1, wenn das Ende des Datenstroms (End-of-File) erreicht wurde.
- `String readLine()`
  (in BufferedReader)
  liefert eine Zeile, oder den Wert null, wenn das Ende des Datenstroms (End-of-File) erreicht wurde.
- `skip(long)`
  überspringt die angegebene Anzahl von Zeichen im Eingabestrom.
- `close()`

schließt den Eingabestrom bzw. das File.

9.9.2 zeichenweises Lesen

Ein typisches Beispiel für das zeichenweise Lesen eines Text-Files hat folgenden Aufbau:

```
int ch;
try {
  BufferedReader in = new BufferedReader (
    new FileReader ("filename.txt") );
  while( (ch = in.read()) != -1 ) {
    // do something ...
  }
  in.close();
} catch (Exception e) {
    System.out.println("error " + e);
}
```

9.9.3 zeilenweises Lesen

Ein typisches Beispiel für das zeilenweise Lesen eines Text-Files hat folgenden Aufbau:

```
String thisLine;
try {
  BufferedReader in = new BufferedReader (
    new FileReader ("filename.txt") );
  while( (thisLine = in.readLine()) != null ) {
    // do something ...
  }
  in.close();
} catch (Exception e) {
    System.out.println("error " + e);
}
```

9.9.4 FileReader

Der FileReader ist die grundlegende Klasse zum text- und zeilen-orientierten Lesen von Dateien (Files). Aus Effizienzgründen sollte er innerhalb eines BufferedReader verwendet werden (siehe unten).

Konstruktor:

- `new FileReader (filename)`

Der Filename ist als String oder als File-Objekt anzugeben.

Das Encoding kann hier nicht angegeben werden, es wird immer das lokale Encoding des Systems angenommen. Falls Sie das Encoding (z.B. "8859_1") explizit angeben wollen, müssen Sie statt dem FileReader eine Kombination von InputStreamReader und FileInputStream verwenden (siehe unten).

9.9.5 BufferedReader

Um die Eingabeeffizienter und schneller zu machen, soll nicht jedes Byte einzeln von der Hardware gelesen werden, sondern aus einem Pufferbereich. Daher sollte fast immer ein BufferedReader über den einfachen FileReader oder InputStreamReader gelegt werden.

Konstruktoren:

- `new BufferedReader (Reader)`
  für einen Puffer mit Standardgröße
- `new BufferedReader (Reader, int)`
  für einen Puffer mit der angegebenen Größe

Beispiel:

```
BufferedReader infile = new BufferedReader
  (new FileReader (infileName) );
```

9.9.6 InputStreamReader

Die Klasse InputStreamReader dient dazu, einen byte-orientierten InputStream zum Lesen von Texten und Textzeilen zu verwenden. Dies ist für Spezialfälle notwendig, die nicht mit dem FileReader abgedeckt werden können. Aus Effizienzgründen sollte sie innerhalb eines BufferedReader verwendet werden.

Konstruktor:

- `new InputStreamReader (InputStream)`
- `new InputStreamReader (InputStream, String)`

Der String im zweiten Fall gibt den Zeichencode an, z.B. "8859_1". Im ersten Fall wird der am jeweiligen Rechnersystem "übliche" Zeichencode verwendet. Beim Lesen von Dateien über das Internet soll immer der Code der Datei explizit angegeben werden, damit man keine bösen Überraschungen auf Grund von lokalen Spezialfällen am Client erlebt.

Beispiele:

```
BufferedReader stdin = new BufferedReader
  (new InputStreamReader (System.in) );

BufferedReader infile = new BufferedReader
  (new InputStreamReader
    (new FileInputStream("message.txt"), "8859_1" ) )

String fileUrl = "ftp://servername/dirname/filename";
BufferedReader infile = new BufferedReader
  (new InputStreamReader
    ( ( new URL(fileUrl) ).openStream(), "8859_1" ) )
```

9.10 Writer

Writer ist die Oberklasse für das Schreiben von Texten, also für die zeichen- und zeilenorientierte Ausgabe.

Es werden zwei Arten von Writer verwendet:

- Klassen, die angeben, **wohin** geschrieben werden soll:
  FileWriter, OutputStreamWriter, StringWriter, ServletResponse, ...
- Klassen, die angeben, **wie** geschrieben werden soll:
  BufferedWriter, PrintWriter, ...

Bei den Klassen der ersten Gruppe wird im Konstruktor angegeben, wo der Datenstrom herkommt (File, InputStream, Speicherbereich).

Bei den Klassen der zweiten Gruppe wird im Konstruktor irgendein anderer Writer angegeben. Diese Klassen und die von ihnen unterstützten Funktionalaitä:ten können daher beliebig kombiniert werden.

9.10.1 Writer - Methoden

Die wichtigsten Methoden bei den Writer-Klassen sind:

- `write(int)`
  schreibt ein Zeichen mit dem angegeben Wert
- `write (char[])`
  `write (char[], offset, length)`
  schreibt alle im Zeichen-Array bzw. im angegebenen Teilbereich des Zeichen-Array enthaltenen Zeichen
- `write(String)`
  schreibt die im angebenen String enthaltene Zeichenkette
- `newLine()`
  schreibt ein Zeilenende-Zeichen
- `flush()`
  sorgt dafür, dass alle eventuell noch im Puffer wartenden Zeichen tatsächlich in die Datei geschrieben werden (sollte immer vor dem Close bzw. vor der Beendigung oder einem Abbruch des Programms aufgerufen werden)
- `close()`
  schließt den Ausgabestrom bzw. das File.

Anmerkung: Um plattformunabhängige Dateien zu erzeugen, sollten Zeilenenden stets mit der Methode newLine() oder println() geschrieben werden und nicht mit "\n" oder "\r\n" innerhalb von Strings.

9.10.2 zeichen- und zeilenweises Schreiben

Ein typisches Beispiel für das Schreiben eines Text-Files hat folgenden Aufbau:

```
try {
  BufferedWriter out = new BufferedWriter (
    new FileWriter ("filename.txt") );
  ...
  out.write(...);
  out.newLine();
  ...
  out.flush();
  out.close();
} catch (Exception e) {
    System.out.println("error " + e);
}
```

oder mit Verwendung von PrintWriter (siehe unten).

9.10.3 FileWriter

Der FileWriter ist die grundlegende Klasse zum text- und zeilen-orientierten Schreiben von Dateien (Files). Aus Effizienzgründen sollte er innerhalb eines BufferedWriter verwendet werden (siehe unten).

Konstruktoren:

- `new FileWriter (filename)`
- `new FileWriter (filename, false)`
- `new FileWriter (filename, true)`

Der Filename ist als String oder als File-Objekt anzugeben. In den ersten beiden Fällen wird die Datei neu geschrieben oder überschrieben, im dritten Fall (Appendmode true) werden die Informationen an das Ende einer bereits existierenden Datei hinzugefügt.

Das Encoding kann hier nicht angegeben werden, es wird immer das lokale Encoding des Systems angenommen. Falls Sie das Encoding (z.B. "8859_1") explizit angeben wollen, müssen Sie statt dem FileWriter eine Kombination von OutputStreamWriter und FileOutputStream verwenden (siehe unten).

9.10.4 BufferedWriter

Um die Ausgabe effizienter und schneller zu machen, soll nicht jedes Byte einzeln auf die Hardware geschrieben werden, sondern aus einem Pufferbereich. Daher sollte fast immer ein BufferedWriter über den einfachen FileWriter oder OutputStreamWriter gelegt werden.

Konstruktoren:

- `new BufferedWriter (Writer)`
  für einen Puffer mit Standardgröße
- `new BufferedWriter (Writer, int)`
  für einen Puffer mit der angegebenen Größe

Beispiel:

```
BufferedWriter outfile = new BufferedWriter
  (new FileWriter (outfileName) );
```

9.10.5 OutputStreamWriter

Die Klasse OutputStreamWriter dient (analaog zum InputSreamReader) dazu, einen byte-orientierten OututStream zum Schreiben von Texten und Textzeilen zu verwenden. Dies ist für Spezialfälle notwendig, die nicht mit dem FileWriter abgedeckt werden können. Aus Effizienzgründen sollte sie innerhalb eines BufferedWriter verwendet werden.

Konstruktor:

- `new OutputSteamWriter (OutputStream)`
- `new OutputSteamWriter (OutputStream, String)`

Der String im zweiten Fall gibt den Zeichencode an, z.B. "8859_1". Im ersten Fall wird der am jeweiligen Rechnersystem "übliche" Zeichencode verwendet.

Beispiele:

```
BufferedWriter stdout = new BufferedWriter
  (new OutputStreamWriter (System.out) );

BufferedWriter outfile = new BufferedWriter
  (new OutputStreamWriter
    (new FileOutputStream("message.txt"),
      "8859_1" ) )
```

9.10.6 PrintWriter

Der PrintWriter ist eine spezielle Klasse zum text- und zeilen-orientierten Schreiben von Datenströmen oder Files. Sie enthält zusätzliche Methoden für die Umwandlung von Zahlen und anderen Objekten in menschenlesbare Texte (print, println, siehe unten).

Konstruktoren:

- `new PrintWriter (Writer)`
- `new PrintWriter (OutputStream)`

Das Encoding kann hier nicht angegeben werden, es wird immer das lokale Encoding des Systems angenommen. Der PrintWriter hat **zusätzlich** zu den Writer-Methoden auch die folgenden **Methoden**:

- `print(String)`
  für die Ausgabe eines Textstrings
- `print(boolean)`
  `print(char)`
  `print(int)`
  `print(long)`
  `print(float)`
  `print(double)`
  und so weiter für eine menschenlesbare Darstellung des jeweiligen Wertes in Zeichen
- `print(Object)`
  für eine menschenlesbare Darstellung des Objekts unter Verwendung der Methode toString()
- `println()`
  für die Ausgabe eines Zeilenende-Zeichens
- `println(String)`
  `println(Object)`
  `println(boolean)`
  und so weiter
  für die Ausgabe des Strings bzw. Wertes bzw. Objekts und anschließend eines Zeilenende-Zeichens ("print line")

Wenn Sie angeben wollen, mit wie vielen Nachkommastellen Float- und Double-Zahlen ausgegeben werden sollen, können Sie die Klassen DecimalFormat oder NumberFormat aus dem Package java.text verwenden. Beispiel:

```
double x = ...;
DecimalFormat dec = new DecimalFormat ("#,###,##0.00");
System.out.println("x = " + dec.format(x));
```

Ein typisches **Beispiel** für das Schreiben eines Text-Files mit PrintWriter hat folgenden Aufbau:

```
try {
  PrintWriter out = new PrintWriter (
    new BufferedWriter (
      new FileWriter ("filename.txt") ) );
  ...
  out.print(...);
  out.println(...);
  ...
  // out.flush(); not needed with auto-flush PrintWriter
  out.close();
} catch (Exception e) {
    System.out.println("error " + e);
}
```

9.10.7 StringWriter und StringReader

Die Klasse StringWriter dient dazu, einen langen String mit mehreren write-Befehlen zusammen zu setzen. Die write-Befehle schreiben also - ähnlich wie bei ByteArrayOutputStream (siehe oben) - in einen Speicherbereich statt in ein File.

Der mit StringWriter erzeugte lange String kann dann entweder mit den Methoden der Klasse String oder mit StringTokenizer oder mit StringReader verarbeitet, also in seine Einzelteile zerlegt werden.

9.11 Übung: zeilenweises Ausdrucken eines Files

Schreiben Sie eine einfache Applikation, die ein Text-File (ihr eigenes Java-Source-File) Zeile für Zeile liest und auf die Standard-Ausgabe ausgibt.

Dieses Programm kann dann als Muster für kompliziertere Programme verwendet werden.

9.12 Übung: zeichenweises Kopieren eines Files

Schreiben Sie eine einfache Applikation, die ein Text-File (ihr eigenes Java-Source-File) Byte für Byte oder Zeichen für Zeichen auf ein neues File ("test.out") kopiert.

Diese beiden Programmvarianten können dann als Muster für kompliziertere Programme verwendet werden.

9.13 Übung: Lesen eines Files über das Internet

Schreiben Sie eine einfache Applikation, die ein kurzes Text-File von einem Web-Server oder FTP-Server liest und Zeile für Zeile auf die Standard-Ausgabe ausgibt.

10 Networking

Java unterstützt standardmäßig nicht nur das Lesen und Schreiben von lokalen Dateien (siehe oben [Seite 112]) sondern auch die Kommunikation über Computer-Netze mit der Technik des Internet-Protokolls TCP/IP.

Die wichtigsten Möglichkeiten, Pakete und Klassen sind:

- **Zugriff auf Web-Server (URL):**
  Paket java.net
  Klassen URL, HttpURLConnection, ...
- **Java im Web-Server (Servlet):**
  Pakete javax.servlet, javax.servlet.http
  Klassen GenericServlet, HttpServlet, ...
- **Client-Server-Systeme (Socket):**
  Paket java.net
  Klassen ServerSocket, Socket

Für die Details wird auf die Online-Dokumentation verwiesen. Beispiele für die Verwendung der Klasse URL finden Sie oben [Seite 116] . Weitere Hinweise zu einigen dieser Klassen finden Sie nachfolgend.

10.1 Java im Web-Server (CGI, Servlets)

Web-Server können nicht nur fertige Files liefern sondern auch Programme ausführen. Dazu dient die Schnittstelle Common Gateway Interface (CGI). Die **CGI-Programme** können, eventuell in Abhängigkeit von Benutzer-Eingaben, bestimmte Aktionen ausführen und die Ergebnisse über das Hypertext Transfer Protocol HTTP an den Web-Browser senden.

CGI-Programme können im HTML-File oder Applet entweder über ein Hypertext-Link aufgerufen werden (nur Ausgabe an den Client) oder über ein Formular oder GUI (Eingabe vom Client an das CGI-Programm, Ausgabe an den Client).

CGI-Programme können in jeder beliebigen Programmier- oder Script-Sprache geschrieben werden, auch in Java. In diesem Fall besteht das CGI-Programm aus einem Shell-Script (Batch-Datei), in dem die Java Virtual Machine aufgerufen wird, die den Bytecode der Java-Applikation interpretiert, etwa in einer der folgenden Formen:

```
java Classname
java Classname Parameter
java -Dvariable=wert Classname
```

Dies bedeutet, dass bei jedem Aufruf des CGI-Programms die Java Virtual Machine neu gestartet werden muss, was eventuell zu längeren Wartezeiten führen kann.

Diesen Nachteil kann man vermeiden, wenn man einen Web-Server verwendet, der die Java Virtual Machine integriert enthält und Java-Programme sofort direkt aufrufen kann (z.B. die neueren Versionen von Apache, Netscape Enterprise Server und vielen anderen).

Diese Java-Programme werden als **Servlets** bezeichnet. Der Name "Servlet" ist analog zu "Applet" gebildet: So wie Applets von einer Java Virtual Machine innerhalb des Web-Browsers ausgeführt werden, so werden Servlets von einer Java Virtual Machine innerhalb des Web-Servers ausgeführt.

Dafür gibt es die Packages javax.servlet und javax.servlet.http sowie ein Java Servlet Development Kit JSDK mit einem ServletRunner zum Testen von Servlets, bevor sie im echten Web-Server eingebaut werden.

Die wichtigsten Methoden von Servlets sind:

- Wenn der Web-Server startet und die Servlets initialisiert, wird die `init`-Methode ausgeführt.
- Bei jedem Client-Request ("User-Click") wird die `service`-Methode oder die von dort je nach der Request-Methode aufgerufene Methode `doGet`, `doPost` etc. ausgeführt. Diese Methoden haben 2 Parameter:
  - Mit dem Parameter `ServletRequest` können die vom Client mitgesendeten Informationen abgefragt werden.
  - Mit dem Parameter `ServletResponse` muss die Antwort an den Client gesendet werden.
- Mit der Methode `getServletInfo` kann man das Servlet dokumentieren.

Servlets können wie Java-Applikationen auch auf lokale Files, Programme und Systemfunktionen am Web-Server zugreifen.

Für ausführlichere Informationen wird auf die Referenzen [Seite 176] und die Literatur verwiesen. Hier nur eine kurze Skizze für den typischen Aufbau eines Servlet:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ... extends HttpServlet {

  public void init (ServletConfig config)
      throws ServletException {
    super.init( config );
    ...
  }

  public void service
      (HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {

    String xxx = req.getParameter("xxx");
    ...

    resp.setContentType("text/html");
    PrintWriter out =
      new PrintWriter( resp.getOutputStream() );
    out.println (" ... ");
    ...
    out.println (" ... ");
    out.flush();
    out.close();
```

```
  }

  public String getServletInfo() {
    return "...";
  }
}
```

10.2 Internet-Protokoll, Server und Clients

Java unterstützt die Kommunikation über das weltweite Internet und über interne Intranets und Extranets mit den Internet-Protokollen TCP und UDP. Dazu muss das Package java.net importiert werden.

10.2.1 Grundbegriffe

Die programmtechnischen Mechanismen für Netzverbindungen werden **Sockets** (vom englischen Wort für Steckdose) genannt.

Für die Adressierung werden Hostname und Portnummer verwendet.

Der **Hostname** ist eine weltweit bzw. netzweit eindeutige Bezeichnung des Rechners (Name oder Nummer).

Die **Portnummer** gibt an, welches Programm auf diesem Rechner die über das Netz übertragenen Informationen verarbeiten soll. Portnummern unter 1024 haben eine vordefinierte Bedeutung und können nur mit Supervisor-Privilegien (root unter Unix) verwendet werden. Portnummern über 1024 sind "frei". Für Java-Anwendungen, die von gewöhnlichen Benutzern geschrieben werden, kommen also meist nur Portnummern über 1024 in Frage, und man muss sicherstellen, dass nicht jemand anderer auf dem selben Rechner die selbe Portnummer schon für andere Zwecke verwendet.

Server sind die Rechner, die ein bestimmtes Service bieten und damit die Kunden (Clients) "bedienen".

Clients ("Kunden") sind die Benutzer, die das Service des Servers in Anspruch nehmen wollen, bzw. die von ihnen dafür benützten Rechner.

10.2.2 Vorgangsweise

Der **Server** "horcht" (listen) mit Hilfe einer **ServerSocket** auf eine Portnummer, d.h. er wartet darauf, dass ein Client etwas von ihm will ("einen Request sendet"). In diesem Fall baut er, meist in einem eigenen Thread, eine Verbindung (connection) mit dem Client über eine **Socket** auf, liest eventuell vom Client kommende Befehle und Dateneingaben und sendet jedenfalls Meldungen und Ergebnisse an den Client.

Clients bauen über eine **Socket** eine Verbindung (connection) zum Server auf, senden eventuell Befehle oder Daten an den Server und lesen jedenfalls alle vom Server kommenden Informationen.

10.3 Sockets

10.3.1 ServerSocket (listen, accept)

Die Klasse ServerSocket dient dazu, auf Client-Requests zu warten (listen) und bei Bedarf eine Verbindung (connection, Socket) zum Client aufzubauen.

Konstruktor:

- `new ServerSocket (int portnumber)`
- `new ServerSocket (int portnumber, int max)`

Mit portnumber gibt man an, auf welche Portnummer die ServerSocket "horcht". Mit max kann man angeben, wie viele Verbindungen maximal gleichzeitig aktiv sein dürfen (default 50).

Methoden:

- `Socket accept()`
  wartet auf eine Verbindungsaufnahme (request) von einem Client und baut dann eine Verbindung (connection, Socket) zum Client mit der entsprechenden Portnummer auf.
- `close()`
  beendet die ServerSocket.
- `InetAddress getInetAddress()`
  liefert den Hostnamen des Servers
- `int getLocalPort()`
  liefert die Portnummer
- `synchronized setSoTimeout(int)`
  setzt eine Beschränkung für die maximale Wartezeit des accept(), in Millisekunden, oder 0 für unbeschränkt.
- `synchronized int getSoTimeout()`
  liefert den Timeout-Wert.

Die Konstruktoren und Methoden können Fehlersituationen werfen, die Unterklassen von IOException sind, z.B. wenn bereits ein anderes Programm diese Portnummer verwendet.

10.3.2 Socket (connection)

Die Klasse Socket dient für die Verbindung (connection) zwischen Client und Server.

Konstruktor:

- `new Socket (String hostname, int portnumber)`

baut eine Verbindung zum angegebenen Rechner (z.B. Server) unter der angegebenen Portnummer auf.

Methoden:

- `InputStream getInputStream()`
  öffnet einen Datenstrom zum Empfangen (Lesen) von Informationen über die Verbindung
- `OutputStream getOutputStream()`
  öffnet einen Datenstrom zum Senden (Schreiben) von Informationen über die Verbindung

- `synchronized close()`
  beendet die Verbindung (connection).
- `InetAddress getLocalAddress()`
  liefert den eigenen Hostnamen (z.B. des Client)
- `InetAddress getInetAddress()`
  liefert den Hostnamen des anderen Rechners (z.B. des Servers)
- `int getLocalPort()`
  liefert die Portnummer
- `synchronized setSoTimeout(int)`
  setzt eine Beschränkung für die maximale Wartezeit beim Lesen im InputStream, in Millisekunden, oder 0 für unbeschränkt.
- `synchronized int getSoTimeout()`
  liefert den Timeout-Wert.

Die Konstruktoren und Methoden können Unterklassen von IOException oder eine InterruptedException werfen, z.B. wenn der Rechnername ungültig ist oder der Server keine Verbindung unter dieser Portnummer akzeptiert oder wenn beim Lesen ein Timeout passiert.

Die Datenströme können genauso wie "normale" Datenströme [Seite 114] (siehe oben) zum Lesen bzw. Schreiben verwendet werden und zu diesem Zweck mit weiteren Datenströmen wie BufferedInputStream, BufferedOutputStream, DataInputStream, InputStreamReader, OutputStreamWriter, BufferedReader, BufferedWriter, PrintWriter kombiniert werden.

Lese-Befehle (read, readLine) warten dann jeweils, bis entsprechende Daten von der Gegenstelle gesendet werden und über die Verbindung ankommen.

Um eine effiziente Übertragung über das Netz zu erreichen, wird die Verwendung von Pufferung empfohlen, dann darf aber beim Schreiben die Methode flush() nicht vergessen werden.

Für die Verarbeitung von "Befehlen", die aus mehreren Wörtern oder Feldern bestehen, kann die Klasse StringTokenizer oder StreamTokenizer verwendet werden (siehe die Online-Dokumentation).

10.4 Beispiel: typischer Aufbau eines Servers

10.4.1 ServerSocket

Das Hauptprogramm des Servers, das auf die Portnummer "horcht", hat folgenden Aufbau:

```
import java.net.* ;

public class ServerMain {
  public static void main (String[] args) {

    ServerSocket server = null;
    Socket s = null;
    ServerCon doIt;
    Thread t;
    int port = ...;

    try {
      server = new ServerSocket (port);
      while(true) {
```

```
        s = server.accept();
        doIt = new ServerCon (s);
        t = new Thread (doIt);
        t.start();
      }
    } catch (Exception e) {
        try { server.close(); } catch (Exception e2) {}
        System.out.println("ServerMain " +e);
        System.exit(1);
    }
  }
}
```

10.4.2 Socket

Das für jede Verbindung aufgerufene Server-Programm ist ein Thread mit folgendem Aufbau:

```
import java.net.* ;
import java.io.* ;

public class ServerCon implements Runnable {

  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;

  public ServerCon (Socket s) {
    socket = s;
  }

  public void run() {
    try {
      out = new PrintWriter
        (new BufferedWriter
          (new OutputStreamWriter
            (socket.getOutputStream() ) ) );
      in = new BufferedReader
        (new InputStreamReader
          (socket.getInputStream() ) );
      ...
      ... in.readLine() ...
      ...
      out.println(...);
      out.flush();
      ...
      in.close();
      out.close();
      socket.close();
    } catch (Exception e) {
        System.out.println("ServerCon " + e);
    }
  }
}
```

10.5 Beispiel: typischer Aufbau eines Client

```
import java.net.* ;
import java.io.* ;
public class ClientCon {
  public static void main (String[] args) {
    int port = ...;
    String host = "...";
    Socket socket;
    BufferedReader in;
    PrintWriter out;
    try {
      socket = new Socket (host, port);
      out = new PrintWriter
        (new BufferedWriter
          (new OutputStreamWriter
            (socket.getOutputStream() ) ) );
      in = new BufferedReader
        (new InputStreamReader
          (socket.getInputStream() ) );
      ...
      out.println(...);
      out.flush();
      ...
      ... in.readLine() ...
      ...
      out.close();
      in.close();
      socket.close();
    } catch (Exception e) {
        System.out.println ("ClientCon " + e);
    }
  }
}
```

10.6 Übung: einfache Client-Server-Applikation

Schreiben Sie einen **Server** (als Applikation), der auf Verbindungsaufnahmen wartet und an jeden Client zunächst einen Willkommensgruß und dann - 6 mal (also ca. 1 Minute lang) in Abständen von ungefähr 10 Sekunden - jeweils zwei Textzeilen liefert, die einen kurzen Text und Datum und Uhrzeit enthalten, und schließlich die Verbindung zum Client schließt.

Für die Datums- und Zeitangabe verwenden Sie das Standard-Format der aktuellen Uhrzeit, das Sie mit

```
( new Date() ).toString()
```

erhalten, auch wenn dabei nicht die richtige Zeitzone verwendet wird. (Mehr über die Problematik von Datums- und Zeitangaben finden Sie im Kapitel über Datum und Uhrzeit [Seite 139]).

Schreiben Sie einen **Client** (als Applikation), der mit diesem Server Verbindung aufnimmt und alle Informationen, die vom Server gesendet werden, Zeile für Zeile auf die Standard-Ausgabe ausgibt.

Testen Sie dieses System. Die Beendigung des Servers (und auch des Clients, falls er Probleme macht) erfolgt durch Abbruch des Programmes mittels Ctrl-C.

Für diese Übung müssen Sie die Hostnamen der verwendeten Rechner (innerhalb des lokalen Netzes) kennen und sich auf eine Portnummer einigen.

11 System-Funktionen und Utilities

Einer der großen Vorteile von Java ist die umfangreiche Klassenbibliothek, die für eine Unmenge von Anwendungsbereichen bereits fertige Programme und Klassen enthält, die einfach und bequem verwendet werden können.

Ein paar typische Beispiele für besonders interessante Klassen sind:

- **Paket java.lang**
  Klassen Object, Class, System, Runtime, Process,
  String, StringBuffer, Math, Integer, Double ...
- **Paket java.util**
  Date, GregorianCalendar ...
  StringTokenizer,
  Collection, Interator,
  List, ArrayList, Vector, LinkedList, Comparable, Comparator,
  Map, HashMap, Hashtable, Tree Map, Set, HashSet, TreeSet ...
- **Paket java.text**
  SimpleDateFormat, DecimalFormat, ...
- **Paket java.awt**
  Toolkit, PrintJob ...
- **Pakete javax.swing, javax.swing.xxxx**
  ListModel, TableModel, TreeModel ...

Für die Details wird auf die Online-Dokumentation verwiesen. Zu einigen dieser Klassen finden Sie nachfolgend ein paar Hinweise.

11.1 Sammlungen, Listen und Tabellen (Collection)

Seit JDK 1.2 gibt es mehrere Interfaces und Klasen, die eine flexiblere und objekt-orientierte Alternative zu Arrays [Seite 17] darstellen:

- Interface Collection
  enthält die allgemein verwendbaren Methoden
  - Interface List
    für **Listen**, bei denen die Elemente eine bestimmte Reihenfolge haben (Index)
    - Klassen ArrayList, Vector, LinkedList
  - Interface Map
    für **Zuordnungen**, bei denen die Elemente einen bestimmten Schlüsselwert haben (Key and Value)
    - Klassen HashMap, Hashtable, TreeMap
  - Interface Set
    für **Mengen**, bei denen jedes Elemente nur einmal oder gar nicht enthalten sein kann
    - Klassen HashSet, TreeSet

Jedes Interface wird von mehreren Klassen implementiert, die intern verschieden arbeiten. Bei manchen Klasen ist der Zugriff optimiert und das Einfügen oder Löschen von Elementen weniger effizient, bei anderen Klassen ist es umgekehrt. Deshalb wird empfohlen, in Datenfeldern und Methodenparametern immer nur den Interface-Namen zu deklarieren, damit man die konkrete Klasse je nach der Performance ändern kann. Beispiel:

```
List personen = new ArrayList();
```

Für den **Zugriff** auf alle Elemente einer Collection gibt es die Interfaces Iterator und Enumeration.

Für das **Sortieren** von Listen gibt es die Interfaces Comparable und Comparator sowie die Hilfsklassen Collections und Arrays.

Für die Details wird auf die Online-Dokumentation und die Referenzen [Seite 176] verwiesen. Hier nur ein einfaches **Beispiel** für die Anwendung dieser Interfaces und Klassen, eine Liste von Personen:

```
List pers = new ArrayList();
pers.add ( new Person ("Schmidt", "Anna") );
pers.add ( new Student ("Schmidt", "Barbara", "TU Wien") );
pers.add ( new Student ("Fischer", "Georg", "BOKU") );
pers.add ( new Person ("Schlosser", "Wilhelm") );
Collections.sort( pers );
Iterator i = pers.iterator();
while (i.hasNext()) {
   System.out.println( i.next() );
}
```

Zu diesem Zweck muss die Klasse Person das Interface Comparable implementieren und die folgenden Methoden enthalten:

```
public boolean equals (Object other) {
   return ( other instanceof Person ) &&
   ( this.vorname .equals ( ((Person)other).vorname ) ) &&
   ( this.zuname .equals ( ((Person)other).zuname ) );
}

public int compareTo (Object other)
           throws ClassCastException {
   Person otherP = (Person)other; // ClassCastException
   if ( this.zuname .equals( otherP.zuname ) ) {
      return this.vorname .compareTo ( otherP.vorname );
   } else {
      return this.zuname .compareTo ( otherP.zuname );
   }
}
```

11.2 Datum und Uhrzeit (Date)

Für die Verarbeitung von Datums- und Zeitangaben muss das Package java.util importiert werden, für die DateFormat-Klassen zusätzlich das Package java.text.

Im JDK ab Version 1.1 [Seite 143] muss für die "richtige", für alle Länder der Welt brauchbare Verarbeitung von Datums- und Zeitangaben eine Kombination von mehreren Klassen für Datum und Uhrzeit, Datums-Berechnungen (Kalender), Zeitzonen, Sprachen und Ausgabeformate verwendet

werden. Ein kurzes Beispiel folgt unten.

Die Verwendung ist dementsprechend kompliziert, außerdem enthalten die meisten Implementierungen auch verschiedene Fehler (Bugs).

Für einfache Anwendungen können die alten Funktionen der Date-Klasse Version 1.0 [Seite 140] verwendet werden (siehe anschließend).

11.3 Date Version 1.0

Die hier beschriebenen Funktionen der Klasse Date werden vom Compiler mit der Warnung "deprecated" versehen, sind aber für einfache Anwendungen ausreichend und wesentlich einfacher zu programmieren als die neuen Funktionen von Version 1.1 [Seite 143] (siehe unten).

11.3.1 Konstruktoren

- `new Date()`
  setzt Datum und Uhrzeit auf heute und die aktuelle Zeit.
- `new Date (year, month, day)`
  setzt das Datum auf den angegebenen Tag (siehe unten) und die Uhrzeit auf 0 Uhr.
- `new Date(year, month, day, hour, minutes)`
  setzt Datum und Uhrzeit auf die angegebene Zeit (siehe unten).
- `new Date(year, month, day, hour, minutes, seconds)`
  setzt Datum und Uhrzeit auf die angegebene Zeit (siehe unten).
- `new Date(long)`
  setzt Datum und Uhrzeit auf die angegebene Zeit im Millisekunden (siehe unten).

Jahre müssen relativ zum Jahr 1900 angegeben werden, also
-8 bedeutet das Jahr 1892,
98 bedeutet das Jahr 1998,
101 bedeutet das Jahr 2001,
1990 bedeutet das Jahr 3890.

Monate müssen im Bereich 0 bis 11 angegeben werden, also
0 für Jänner,
1 für Februar, usw. bis
11 für Dezember.
Dies ist für die Verwendung als Array-Index günstig, führt aber sonst leicht zu Irrtümern. Es wird daher empfohlen, statt der Zahlenangaben lieber die in der Klasse Calendar (Version 1.1) definierten symbolischen Namen zu verwenden:
`Calendar.JANUARY` für Jänner,
`Calendar.FEBRUARY` für Februar,
`Calendar.MARCH` für März, usw. bis
`Calendar.DECEMBER` für Dezember.

Tage innerhalb der Monate werden wie gewohnt im Bereich 1 bis 31 angegeben.

Wochentage werden im Bereich 0 bis 6 angegeben, dafür sollten aber lieber die symbolischen Namen
`Calendar.SUNDAY` für Sonntag,
`Calendar.MONDAY` für Montag, usw. bis
`Calendar.SATURDAY` für Samstag
verwendet werden.

Stunden werden im Bereich 0 bis 23 angegeben, **Minuten** und **Sekunden** im Bereich 0 bis 59.

Für alle bisher genannten Zahlen wird der Typ int verwendet.

Für die Berechnung von Zeitdifferenzen wird eine Größe in **Millisekunden** verwendet, deren Nullpunkt am 1. Jänner 1970 um 0 Uhr liegt. Dafür wird der Typ long verwendet.

11.3.2 Methoden

- `int getTime()`
  liefert den Zeitpunkt in Millisekunden (siehe oben).
- `int getYear()`
  liefert das Jahr (relativ zu 1900, siehe oben).
- `int getMonth()`
  liefert den Monat (0-11, siehe oben).
- `int getDate()`
  liefert den Tag innerhalb des Monats (1-31).
- `int getDay()`
  liefert den Wochentag (0-6, siehe oben).
- `int getHours()`
  liefert die Stunde (0-23).
- `int getMinutes()`
  liefert die Minuten (0-59).
- `int getSeconds()`
  liefert die Sekunden (0-59, bei Schaltsekunden auch 60).
- `setTime(long)`
  setzt Datum und Uhrzeit auf den angegebenen Zeitpunkt in Millisekunden (siehe oben).
- `setYear(int)`
  `setMonth(int)`
  `setDate(int)`
  `setHours(int)`
  `setMinutes(int)`
  `setSeconds(int)`
  ändert jeweils nur das angegebene Feld (siehe oben).
- `boolean after(Date)`
  liefert true, wenn der Zeitpunkt später als der angegebene liegt.
- `boolean before(Date)`
  liefert true, wenn der Zeitpunkt früher als der angegebene liegt.
- `boolean equals(Date)`
  liefert true, wenn die beiden Zeiten gleich sind.
- `String toString(Date)`
  liefert einen TextString, der Datum, Uhrzeit und Zeitzone in dem auf Unix-Rechnern üblichen Format angibt.
- `String toGMTString(Date)`

liefert einen TextString, der Datum und Uhrzeit in dem am Internet (z.B. im Date-Header von Electronic Mail) üblichen Format angibt.

- `String toLocaleString(Date)`
  liefert einen TextString, der Datum, Uhrzeit und Zeitzone in einem auf dem lokalen System, in der lokalen Sprache und Zeitzone üblichen Format angibt.

Außerdem gibt es eine statische Methode

- `long Date.parse (String)`

mit der ein Datums-String in einem genormten Format (z.B. Unix- oder Internet-Format, siehe oben) in den entsprechenden Zeitpunkt in Millisekunden umgewandelt wird.

Wenn man mit den vorgefertigten Ausgabeformaten nicht zufrieden ist, kann man eine eigene Version einer DateFormat-Klasse schreiben, die die gewünschte Datums-Darstellung liefert. Damit die Verwendung zur Klasse DateFormat von Version 1.1 kompatibel ist, sollte diese Methode die folgende Signature haben:

- `public String format (Date d)`

11.3.3 Beispiel für ein einfaches, selbst geschriebenes DateFormat:

```
import java.util.*;
public class MyDateFormat { // Date Version 1.0
  public String format (Date d) {
    String s;
    // like SimpleDateFormat("d. M. yyyy")
    s = d.getDate() + ". " +
      (d.getMonth()+1) + ". " +
      (d.getYear()+1900);
    return s;
  }
}
```

11.3.4 Beispiel für die einfache Verarbeitung von Datums-Angaben in Version 1.0:

```
import java.util.*;

public class Date10 {  // Date Version 1.0
  public static void main (String[] args) {
    MyDateFormat df = new MyDateFormat();

    Date today = new Date();
    long todayInt = today.getTime();
    int currentYear = today.getYear() + 1900;
    System.out.println ("Today is " +
      df.format(today) );
    System.out.println ("The current year is " +
      currentYear );

    long yesterdayInt = todayInt - 24*60*60*1000L;
    Date yesterday = new Date (yesterdayInt);
    System.out.println ("Yesterday was " +
      df.format(yesterday) );

    long next30Int = todayInt + 30*24*60*60*1000L;
    Date next30 = new Date (next30Int);
```

```
    System.out.println ("30 days from today is " +
      df.format(next30) );

    Date marriage = new Date (90, Calendar.FEBRUARY, 7);
    long marriageInt = marriage.getTime();
    int marriageYear = marriage.getYear() + 1900;
    System.out.println ("Married on " +
      df.format(marriage) );
    System.out.println ("Married for " +
      (currentYear-marriageYear) + " years." );
    System.out.println ("Married for " +
      (todayInt-marriageInt)/(24*60*60*1000L) +
      " days." );

    Date silver= new Date (marriageInt);
      silver.setYear( marriage.getYear()+25 );
      // works for all dates except Feb 29th
    System.out.println ("Silver marriage on " +
      df.format(silver) );

    if ( silver.getYear() == today.getYear() &&
         silver.getMonth() == today.getMonth() &&
         silver.getDate() == today.getDate() )
    System.out.println ("Congratulations!");

  }
}
```

11.4 Date und Calendar Version 1.1

11.4.1 Klassen

Im JDK 1.1 soll für die "richtige", für alle Länder der Welt brauchbare Verarbeitung von Datums- und Zeitangaben eine Kombination der folgenden Klassen verwendet werden:

- **Calendar** bzw. **GregorianCalendar** für Datums- und Zeit-Berechnungen (mit Fehlerkontrollen)
- **DateFormat** bzw. **SimpleDateFormat** für die Darstellung von Datum und Uhrzeit als Textstring
- **TimeZone** bzw. **SimpleTimeZone** für die Zeitzone
- **Locale** für das Land oder die Sprache
- **Date** für Zeitpunkte

Die Klasse Date soll in diesem Fall *nur* für die Speicherung eines Zeitpunkts verwendet werden. Von den oben für Version 1.0 [Seite 140] angeführten get- und set-Methoden sollen nur diejenigen verwendet werden, die den Zeitpunkt in Millisekunden angeben. Für die Angabe von Jahr, Monat, Tag und Uhrzeit sollen stattdessen die "besseren" Methoden des Kalenders verwendet werden.

Die richtige Kombination und Verwendung dieser Klassen ist einigermaßen kompliziert. Außerdem enthalten manche Implementierungen ein paar Fehler (Bugs) oder unterstützen nicht alle Länder und Sprachen. Deshalb greifen viele Java-Benutzer auf die alte Version 1.0 oder auf selbst geschriebene Klassen wie z.B. BigDate zurück.

Hier wird nur ein kurzes Beispiel für typische Anwendungen gegeben. Für die Details wird auf die Online-Dokumentation und auf die Fragen und Antworten in den einschlägigen Usenet-Newsgruppen verwiesen (siehe Referenzen [Seite 176]).

11.4.2 Beispiel für Datums- und Zeit-Angaben und -Berechungen in Version 1.1:

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

public class Date11 {  // Date Version 1.1
  public static void main (String[] args) {

    DateFormat df = new SimpleDateFormat ("d. MMMM yyyy",
      Locale.GERMANY);
    DateFormat tf = new SimpleDateFormat ("HH.mm",
      Locale.GERMANY);

    SimpleTimeZone mez = new SimpleTimeZone( +1*60*60*1000,
      "CET");
    mez.setStartRule (Calendar.MARCH, -1, Calendar.SUNDAY,
      2*60*60*1000);
    mez.setEndRule (Calendar.OCTOBER, -1, Calendar.SUNDAY,
      2*60*60*1000);

    Calendar cal = GregorianCalendar.getInstance (mez);
    cal.setLenient(false); // do not allow bad values

    Date today = new Date();
    System.out.println ("Heute ist der " +
      df.format(today) );
    System.out.println ("Es ist " +
      tf.format(today) + " Uhr");

    cal.setTime(today);
    int currentYear = cal.get(Calendar.YEAR);
    System.out.println ("Wir haben das Jahr " +
      currentYear );

    cal.add (Calendar.DATE, -1);
    Date yesterday = cal.getTime();
    System.out.println ("Gestern war der " +
      df.format(yesterday) );

    try {
      cal.set (1997, Calendar.MAY, 35);
      cal.setTime(cal.getTime()); // to avoid a bug
      Date bad = cal.getTime();
      System.out.println ("Bad date was set to " +
        df.format(bad) );
    } catch (Exception e) {
      System.out.println ("Invalid date was detected "
        + e);
    }

    cal.set (1996, Calendar.FEBRUARY, 29);
    cal.setTime(cal.getTime()); // to avoid a bug
    Date marriage = cal.getTime();
    System.out.println ("geheiratet am " +
      df.format(marriage) );
```

```
    long todayInt = today.getTime();
    long marriageInt = marriage.getTime();
    long diff = todayInt - marriageInt;
    System.out.println ("seit " +
      diff/(24*60*60*1000L) +
      " Tagen verheiratet" );
    int marriageYear = cal.get(Calendar.YEAR);
    System.out.println ("seit " +
      (currentYear-marriageYear) +
      " Jahren verheiratet" );

    cal.setTime(marriage);
    /* bypass leap year error in add YEAR method: */
    if ( (cal.get(Calendar.MONTH) == Calendar.FEBRUARY) &&
         (cal.get(Calendar.DATE) == 29) ) {
      cal.add (Calendar.DATE, 1);
    }
    /* end of leap year error bypass */
    cal.add (Calendar.YEAR, 25);
    Date silverMarriage = cal.getTime();
    System.out.println ("Silberne Hochzeit am " +
      df.format(silverMarriage) );

    String todayDay = df.format(today);
    String silverDay = df.format(silverMarriage);
    // compare only day, not time:
    if ( silverDay.equals(todayDay) )
      System.out.println ("Herzlichen Glueckwunsch!");
  }
}
```

11.5 Zeitmessung

Für die Berechnung von Laufzeiten kann man entweder Date-Objekte (siehe oben) oder die folgende statische Methode verwenden:

- `static long System.currentTimeMillis()`
  liefert den aktuellen Zeitpunkt in Millisekunden.

Beispiel:

```
public static void main (String[] args) {
  long startTime = System.currentTimeMillis();
  ... // do something
  long endTime = System.currentTimeMillis();
  long runTime = endTime - startTime;
  float runSeconds = ( (float)runTime ) / 1000.F;
  System.out.println ("Run time was " + runSeconds + " seconds." );
}
```

11.6 Ausdrucken (PrintJob, PrinterJob)

11.6.1 PrintJob

Das AWT Version 1.1 enthält auch eine Klasse PrintJob, mit der ein Printout erzeugt und mit Hilfe des systemspezifischen Printer-Selection-Dialogs auf einem Drucker ausgedruckt werden kann.

Dazu dienen die folgenden Methoden in den verschiedenen Klassen:

- `Toolkit getToolkit()`
  liefert bei einem Frame das Toolkit, das für die Darstellung des Frame und der in ihm enthaltenen Komponenten auf dem System zuständig ist.
- `PrintJob getPrintJob(Frame, String, Properties)`
  erzeugt bei einem Toolkit einen PrintJob für das Frame, mit dem angegebenen Title-String und eventuellen Druckereigenschaften (oder null).
- `Graphics getGraphics()`
  liefert bei einem PrintJob das Graphics-Objekt, das einen Printout enthalten kann. Für jede neue Seite soll ein neues Graphics-Element angelegt werden. Dieses Graphics-Element kann mit beliebigen Graphik-Operationen gefüllt werden. Meistens verwendet man aber nur die folgenden beiden Methoden:
- `print (Graphics g)`
  bei einer AWT-Komponente stellt einen Printout nur dieser Komponente in das Graphics-Objekt.
- `printAll (Graphics g)`
  bei einem AWT-Container wie z.B. Frame oder Applet stellt einen Printout dieses Containers mit seinem gesamten Inhalt in das Graphics-Objekt.
- `dispose()`
  für das Graphics-Objekt setzt das fertig erstellte Graphics-Objekt (eine Seite des Printout) in den Printout.
- `end()`
  für den PrintJob beendet den Printout. Nun wird er vom Print-Spooling des Systems an den Drucker gesendet, und danach kann der Drucker wieder für andere Jobs verwendet werden.

Beispielskizze:

```
Frame f = new Frame ("Test");
f.setLayout(...);
f.add(...);
...
Toolkit t = f.getToolkit();
PrintJob pJob = t.getPrintJob (f, "Test", null);
Graphics g = pJob.getGraphics();
f.printAll(g); // or g.drawxxx ...
g.dispose();
pJob.end();
```

Wenn man ein Applet ausdrucken will, muss man mit getParent() das zugehörige Frame bestimmen und dann dieses in getPrintJob angeben. Außerdem muss der Benutzer dem Applet mit dem SecurityManager die Erlaubnis zum Drucken am Client-Rechner geben.

11.6.2 PrinterJob

Ab JDK 1.2 gibt es eine neue Klasse PrinterJob mit den Interfaces Printable und Pageable im Package java.awt.print.

Dazu dienen unter anderem die folgenden Klassen und Methoden:

- `PrinterJob.getPrinterJob()`
  liefert ein PrinterJob-Objekt.
- `setPrintable (Printable)`
  gibt an, welches Objekt gedruckt werden soll. Dieses Objekt muss das Interface Printable implementieren, also eine print-Methode enthalten.
- Die `print`-Methode im PrinterJob bewirkt, dass die print-Methode des Printable-Objekts aufgerufen und das Ergebnis an den Drucker gesendet wird (oder die PrintException geworfen wird).
- Die `print`-Methode des Printable-Objekts hat die folgenden Parameter:
  - das `Graphics`-Objekt, das wie in der paint-Methode (oder durch Aufruf der print-Methode oder der printAll-Methode) mit der graphischen Information gefüllt werden muss, eventuell mit Verschiebung des Eckpunktes mittels `translate`,
  - ein `PageFormat`-Objekt, und
  - die Seitennummer (`pageIndex`).

  und muss einen der folgenden beiden Werte zurückgeben:
  - `PrinterJob.PAGE_EXISTS`
    wenn eine Seite erzeugt wurde,
  - `PrinterJob.NO_SUCH_PAGE`
    wenn keine Seite erzeugt wurde.

Beispielskizze:

```
public class Xxxx extends Frame
  implements ActionListener, Printable {
  ...
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == printButton) {
      PrinterJob printJob = PrinterJob.getPrinterJob();
      printJob.setPrintable(this);
      try {
       printJob.print();
      } catch (PrintException ex) {
      }
    }
 }
 public int print(Graphics g, PageFormat pageFormat, int pageIndex) {
   if (pageIndex == 0) {
      g.translate(100, 100);
      paint(g); // or: this.printAll(g); or: g.drawxxx ...
      return Printable.PAGE_EXISTS;
   }
   return Printable.NO_SUCH_PAGE;
 }
}
```

Außerdem können mit PrinterJob auch Objekte, die das Interface `Pageable` implementieren, ausgedruckt werden, das sind Dokumente, die aus mehreren Seiten (Page) bestehen, mit Seiteninhalt, Kopf- und Fußzeilen und Seitennummern. Die Klasse `Book` ist eine Klasse, die dieses Interface implementiert.

11.6.3 Sonstige Möglichkeiten

Weitere Alternativen zur Verwendung von PrintJob sind die Ausgabe auf das Pseudo-File mit den Filenamen "lpt1" oder die Verwendung der Print-Screen-Funktion durch den Benutzer, um den aktuellen Fensterinhalt (z.B. eine Web-Page mit einem Applet mit Benutzer-Eingaben) auszudrucken. In manchen Fällen kann es auch günstig sein, die Information, die ausgedruckt werden soll, als HTML-File zu erzeugen, das dann vom Benutzer in seinem Web-Browser angezeigt und ausgedruckt werden kann.

11.7 Ausführung von Programmen (Runtime, exec)

Mit der Klasse Runtime und deren Methode exec kann man innerhalb von Applikationen andere Programme (Hauptprogramme, Prozesse) starten. Dies hat den Vorteil, dass man alle Funktionen des Betriebssystems ausnützen kann, und den Nachteil, dass die Java-Applikation dadurch nicht mehr Plattform- oder Rechner-unabhängig ist

Applets können im Allgemeinen *keine* anderen Programme starten.

Für die Verwendung von Runtime und Process muss das Package java.util importiert werden, im Fall von Ein-Ausgabe-Operationen auch java.io.

Die für diesen Zweck wichtigsten Methoden der Klasse **Runtime** sind:

- `static Runtime Runtime.getRuntime()`
  liefert das Runtime-Objekt der Java Virtual Machine
- `Process exec(String)`
  erzeugt einen Prozess, der den im String enthaltenen Befehl (Befehlswort und eventuell Parameter) ausführt.
- `Process exec(String[])`
  erzeugt einen Prozess, der den im String-Array enthaltenen Befehl (Befehlswort und Parameter) ausführt.

Die exec-Methode kann die Fehlersituation IOException werfen, wenn es den Befehl (das Programm) nicht gibt.

Die wichtigsten Methoden der Klasse **Process** sind:

- `waitFor()`
  wartet, bis der Prozess fertig gelaufen ist.
- `destroy()`
  bricht den Prozess ab.
- `int exitValue()`
  liefert den Return-Code (Exit-Status) des Prozesses (meist 0 für okay, ungleich 0 für Fehler)
- `InputStream getInputStream()`

liefert einen InputStream zum Lesen der vom Prozess geschriebenen Standard-Ausgabe.

- `InputStream getErrorStream()`
  liefert einen InputStream zum Lesen der vom Prozess geschriebenen Fehler-Ausgabe.
- `OutputStream getOutputStream()`
  liefert einen OutputStream zum Schreiben der vom Prozess gelesenen Standard-Eingabe.

Die waitFor-Methode kann die Fehlersituation InterruptedException werfen, wenn das Programm während des Wartens abgebrochen wird.

Beispiel (Unix):

```
try {
  Process p = Runtime.getRuntime().exec
    ("/usr/local/bin/elm");
  p.waitFor();
} catch (Exception e) {
    System.err.println("elm error " +e);
}
```

Das analoge Beispiel für einen PC enthält

```
Process p = Runtime.getRuntime().exec
  ("c:\\public\\pegasus\\pmail.exe");
```

Befehlsnamen müssen im Allgemeinen mit dem Pfad angegeben werden.

Auf PCs muss man beachten, dass man als Befehlsname nur "echte" exe- oder com-Programme angeben kann. Die meisten DOS-Befehle wie DIR oder COPY sind aber nicht eigene Programm-Files sondern werden von der "Shell" COMMAND.COM (unter DOS, Windows 3 und Windows 95) bzw. CMD.EXE (unter Windows NT) ausgeführt. Beispiel:

```
Process p = Runtime.getRuntime().exec
  ("command.com /c dir");
```

Wenn man unter Unix eine Eingabe- oder Ausgabe-Umleitung oder Pipe für den Befehl angeben will, muss man analog zuerst eine Unix-Shell aufrufen und dieser Shell dann den kompletten Befehl (einschließlich der Umleitung) als einen einzigen String-Parameter angeben. Beispiel:

```
Process p = Runtime.getRuntime().exec (new String[] {
  "/usr/bin/sh", "-c", "/usr/bin/ls > ls.out" } );
```

Mit Hilfe der Methode getOutputStream kann man Eingaben vom Java-Programm an den Prozess senden. Mit Hilfe der Methoden getInputStream und getErrorStream kann man die vom Prozess erzeugte Ausgabe im Java-Programm verarbeiten. Beispiel:

```
try {
  String cmd = "/usr/bin/ls -l /opt/java/bin";
  String line = null;
  Process p = Runtime.getRuntime().exec(cmd);
  BufferedReader lsOut = new BufferedReader
    (new InputStreamReader
      (p.getInputStream() ) );
  while( ( line=lsOut.readLine() ) != null) {
    System.out.println(line);
```

```
  }
} catch (Exception e) {
    System.err.println("ls error " +e);
}
```

Wenn man allerdings zwei oder alle drei dieser Eingabe- und Ausgabe-Ströme lesen bzw. schreiben will, muss man das in getrennten Threads [Seite 103] tun, weil sonst ein auf Eingabe wartendes Read die anderen blockiert.

11.8 Verwendung von Unterprogrammen (native methods, JNI)

Man kann innerhalb von Java-Applikationen auch Unterprogramme aufrufen, die in einer anderen Programmiersprache geschrieben sind, insbesondere in den Programmiersprachen C und C++. Solche Unterprogramme werden als "eingeborene" (native) Methoden bezeichnet, das entsprechende Interface als Java Native Interface (JNI). Der Vorteil liegt darin, dass man sämtliche von dieser Programmiersprache unterstützten Funktionen und Unterprogramm-Bibliotheken verwenden kann. Der Nachteil liegt darin, dass die Java-Applikation dann im Allgemeinen nicht mehr Plattform- oder auch nur Rechner-unabhängig ist.

Der Vorgang läuft in folgenden Schritten ab:

Zunächst wird eine **Java-Klasse** geschrieben, die folgende Elemente enthält:

- die Deklaration der native Methode. Diese Deklaration enthält nur die Signature, mit der zusätzlichen Angabe "native", und dann nur einen Strichpunkt, keinen Statement-Block. Dieser wird später in der Programmiersprache C oder C++ geschrieben.
- einen statischen Block (static), der die zugehörige Laufzeit-Bibliothek lädt.

Beispiel:

```
public class ClassName {
  public native void name() ;
  static {
    System.loadLibrary ("libname");
  }
  ...
}
```

Diese Klasse kann wie jede normale Klasse verwendet werden, d.h. man kann Objekte dieser Klasse mit `new` anlegen und ihre Methoden für dieses Objekt aufrufen.

Als nächstes werden mit dem Hilfsprogramm `javah` Header-Files und ein sogenanntes Stub-File erstellt. Diese Files enthalten die entsprechenden Deklarationen in C-Syntax, die dann vom C-Programm verwendet werden. Es gibt auch Umwandlungs-Programme für die Umwandlung von Java-Strings in C-Strings und umgekehrt.

Unter Verwendung dieser Hilfsmittel wird nun das **C-Programm** geschrieben und übersetzt und in einer Library (Bibliotheks-File) für dynamisches Laden zur Laufzeit gespeichert. Diese Library muss in die Environment-Variable LD_LIBRARY_PATH hinzugefügt werden.

Schließlich werden die **Java-Klassen** mit dem Java-Compiler `javac` übersetzt und mit dem Befehl `java` ausgeführt. Dabei wird das C-Programm automatisch aus der vorher erstellten Library dazugeladen.

Für die Details wird auf die Online-Dokumentation und auf die einschlägige Literatur verwiesen.

12 Datenbanken

Java eignet sich besonders gut für Graphische User-Interfaces und für Internet-Anwendungen. Datenbanksysteme eignen sich besonders gut für die Speicherung von komplexen und umfangreichen Datenmengen. Wie kann ich diese beiden Technologien "verheiraten"?

Zu den wichtigsten Anwendungsgebieten von Java zählen User-Interfaces zu Datenbanksystemen.

Das Java-Programm kann dabei ein Applet [Seite 84] , eine Applikation [Seite 5] oder ein Servlet [Seite 130] sein und kann am selben Rechner wie die Datenbank laufen oder auch auf einem anderen Rechner und von dort über das Internet oder ein Intranet auf die Datenbank zugreifen (siehe Datenbank-Anwendungen über das Internet [Seite 165]).

Die "Java Database Connectivity" (JDBC) [Seite 157] ist im Sinne der Plattformunabhängigkeit von Java so aufgebaut, dass das Java-Programm von der Hard- und Software des Datenbanksystems unabhängig ist und somit für alle Datenbanksysteme (MS-Access, Oracle etc.) funktioniert.

Mit den im JDBC enthaltenen Java-Klassen (Package java.sql) kann man Daten in der Datenbank so bequem speichern und abfragen wie beim Lesen und Schreiben von Dateien oder Sockets. Auf diese Weise kann man die Vorteile von Java, die vor allem bei der Gestaltung von (graphischen und plattformunabhängigen) User-Interfaces und von Netz-Verbindungen liegen, mit der Mächtigkeit von Datenbanksystemen verbinden.

12.1 Relationale Datenbanken

Relationale Datenbanken bestehen aus **Tabellen** (Relationen). Die Tabellen entsprechen in etwa den *Klassen* in der Objektorientierten Programmierung. Beispiele: Eine Personaldatenbank enthält Tabellen für Mitarbeiter, Abteilungen, Projekte. Eine Literaturdatenbank enthält Tabellen für Bücher, Zeitschriften, Autoren, Verlage.

Diese Tabellen können in Beziehungen zueinander stehen (daher der Name "Relation"). Beispiele: Ein Mitarbeiter gehört zu einer Abteilung und eventuell zu einem oder mehreren Projekten. Jede Abteilung und jedes Projekt wird von einem Mitarbeiter geleitet. Ein Buch ist in einem Verlag erschienen und hat einen oder mehrere Autoren.

Beispielskizze für eine Tabelle **"Mitarbeiter"**:

Abteilung	*Vorname*	*Zuname*	*Geburtsjahr*	*Gehalt*
EDV-Zentrum	Hans	Fleißig	1972	2400.00
EDV-Zentrum	Grete	Tüchtig	1949	3200.00
Personalstelle	Peter	Gscheitl	1968	1600.00

Jede **Zeile** der Tabelle (row, Tupel, Record) enthält die Eigenschaften eines Elementes dieser Menge, entspricht also einem *Objekt*. In den obigen Beispielen also jeweils ein bestimmter Mitarbeiter, eine Abteilung, ein Projekt, ein Buch, ein Autor, ein Verlag. Jede Zeile muss eindeutig sein, d.h. verschiedene Mitarbeiter müssen sich durch mindestens ein Datenfeld (eine Eigenschaft) unterscheiden.

Jede **Spalte** der Tabelle (column, field, entity) enthält die gleichen Eigenschaften der verschiedenen Objekte, entspricht also einem *Datenfeld*. Beispiele: Vorname, Zuname, Geburtsjahr und Abteilung eines Mitarbeiters, oder Titel, Umfang, Verlag und Erscheinungsjahr eines Buches.

Ein Datenfeld (z.B. die Sozialversicherungsnummer eines Mitarbeiters oder die ISBN eines Buches) oder eine Gruppe von Datenfeldern (z.B. Vorname, Zuname und Geburtsdatum einer Person) muss eindeutig sein, sie ist dann der Schlüssel (key) zum Zugriff auf die Zeilen (Records) in dieser Tabelle. Eventuell muss man dafür eigene Schlüsselfelder einrichten, z.B. eine eindeutige Projektnummer, falls es mehrere Projekte mit dem gleichen Namen gibt.

Die Beziehungen zwischen den Tabellen können auch durch weitere Tabellen (Relationen) dargestellt werden, z.B. eine Buch-Autor-Relation, wobei sowohl ein bestimmtes Buch als auch ein bestimmter Autor eventuell in mehreren Zeilen dieser Tabelle vorkommen kann, denn ein Buch kann mehrere Autoren haben und ein Autor kann mehrere Bücher geschrieben haben. Das Gleiche gilt für die Relation Mitarbeiter-Projekt.

Das Konzept (Design) einer Datenbank ist eine komplexe Aufgabe, es geht dabei darum, alle relevanten Daten (Elemente, Entities) und alle Beziehungen zwischen ihnen festzustellen und dann die logische Struktur der Datenbank entsprechend festzulegen. Die logisch richtige Aufteilung der Daten in die einzelnen Tabellen (Relationen) wird als Normalisierung der Datenbank bezeichnet, es gibt dafür verschiedene Regeln und Grundsätze, ein Beispiel ist die sogenannte dritte Normalform.

Fast alle Datenbanksysteme seit Ende der 70er- oder Beginn der 80er-Jahre sind relationale Datenbanksysteme und haben damit die in den 60er- und 70er-Jahren verwendeten, bloß hierarchischen Datenbanksysteme abgelöst.

Eine zukunftweisende Weiterentwicklung der Relationalen Datenbanken sind die "Objektrelationalen Datenbanken", bei denen - vereinfacht gesprochen - nicht nur primitive Datentypen sondern auch komplexe Objekte als Datenfelder möglich sind, ähnlich wie in der objektorientierten Programmierung.

12.1.1 Datenschutz und Datensicherheit

Datenbanken enthalten meist umfangreiche und wichtige, wertvolle Informationen. Daher muss bei Datenbanksystemen besonderer Wert auf den Datenschutz und die Datensicherheit gelegt werden.

Die Datenbank-Benutzer werden in 2 Gruppen aufgeteilt: den Datenbankadministrator und die Datenbankanwender.

Der **Datenbankadministrator** legt mit der Data Definition Language (DDL) die logischen Struktur der Datenbank fest und ist für den Datenschutz, die Vergabe der Berechtigungen an die Datenbankanwender und für die Datensicherung verantwortlich.

Die **Datenbankanwender** können mit einer Data Manipulation Language (DML) oder Query Language die Daten in der Datenbank speichern, abfragen oder verändern.

Mit der Hilfe von Usernames und Passwörtern werden die einzelnen Benutzer identifiziert, und der Datenbankadministrator kann und muss sehr detailliert festlegen, welcher Benutzer welche Aktionen (lesen, hinzufügen, löschen, verändern) mit welchen Teilen der Datenbank (Tabellen, Spalten, Zeilen) ausführen darf.

12.1.2 Datenkonsistenz

Die in einer Datenbank gespeicherten Informationen stehen meistens in Beziehungen zueinander, bestimmte Informationen hängen in eventuell recht komplexer Weise von anderen, ebenfalls in der Datenbank gespeicherten Informationen ab. Es muss sichergestellt werden, dass die gesamte Datenbank immer in einem gültigen Zustand ist, dass also niemals ungültige oder einander widersprechende Informationen darin gespeichert werden.

Dies wird mittels **Transaktionen** erreicht: Unter einer Transaktion versteht man eine Folge von logisch zusammengehörenden Aktionen, die nur entweder alle vollständig oder überhaupt nicht ausgeführt werden dürfen.

Beispiel: Wenn zwei Mitarbeiter den Arbeitsplatz tauschen, muss sowohl beim Mitarbeiter 1 der Arbeitsplatz von A auf B als auch beim Mitarbeiter 2 der Arbeitsplatz von B auf A geändert werden. Würde bei einer dieser beiden Aktionen ein Fehler auftreten, die andere aber trotzdem ausgeführt werden, dann hätten wir plötzlich 2 Mitarbeiter auf dem einen Arbeitsplatz und gar keinen auf dem anderen.

Ein anderes Beispiel: Wenn in der Datenbank nicht nur die Gehälter der einzelnen Mitarbeiter sondern auch die Summe aller Personalkosten (innerhalb des Budgets) gespeichert ist, dann müssen bei jeder Gehaltserhöhung beide Felder um den gleichen Betrag erhöht werden, sonst stimmen Budgetplanung und Gehaltsauszahlung nicht überein.

Um solche Inkonsistenzen zu vermeiden, müssen zusammengehörende Aktionen jeweils zu einer **Transaktion** zusammengefasst werden:

- Wenn alle Aktionen erfolgreich abgelaufen sind, wird die Transaktion beendet (**commit**) und die Datenbank ist in einem neuen gültigen Zustand.
- Falls während der Transaktion irgendein Fehler auftritt, werden alle seit Beginn der Transaktion ausgeführten unvollständigen Änderungen rückgängig gemacht (**rollback**), und die Datenbank ist wieder in dem selben alten, aber gültigen Zustand wie vor Beginn der versuchten Transaktion.

12.2 Structured Query Language (SQL)

SQL hat sich seit den 80er-Jahren als die von allen Datenbanksystemen (wenn auch eventuell mit kleinen Unterschieden) unterstützte Abfragesprache durchgesetzt, und die Version SQL2 ist seit 1992 auch offiziell genormt.

SQL umfasst alle 3 Bereiche der Datenbankbenutzung:

- die Definition der Datenbankstruktur,
- die Speicherung, Löschung und Veränderung von Daten und
- die Abfrage von Daten.

Für eine komplette Beschreibung von SQL wird auf die Fachliteratur verwiesen, hier sollen nur ein paar typische Beispiele gezeigt werden.

12.2.1 Datentypen

SQL kennt unter anderem die folgenden Datentypen:

- `CHAR (n)` = ein Textstring mit einer Länge von genau n Zeichen (ähnlich wie String, wird mit Leerstellen aufgefüllt)
- `VARCHAR (n)` = ein Textstring mit einer variablen Länge von maximal n Zeichen (n < 255)
- `LONGVARCHAR (n)` = ein Textstring mit einer variablen Länge von maximal n Zeichen (n > 254)
- `DECIMAL` oder `NUMERIC` = eine als String von Ziffern gespeicherte Zahl (ähnlich wie BigInteger)
- `INTEGER` = eine ganze Zahl (4 Bytes, wie int)
- `SMALLINT` = eine ganze Zahl (2 Bytes, wie short)
- `BIT` = wahr oder falsch (wie boolean)
- `REAL` oder `FLOAT` = eine Fließkommazahl (wie float)
- `DOUBLE` = eine doppelt genaue Fließkommazahl (wie double)
- `DATE` = ein Datum (Tag)
- `TIME` = eine Uhrzeit
- `TIMESTAMP` = ein Zeitpunkt (Datum und Uhrzeit, ähnlich wie Java-Date)

12.2.2 Definition der Datenbankstruktur (DDL)

CREATE = Einrichten einer neuen Tabelle.

Beispiel:

```
CREATE TABLE Employees (
  INT      EmployeeNumber ,
  CHAR(30) FirstName ,
  CHAR(30) LastName ,
  INT      BirthYear ,
  FLOAT    Salary
)
```

definiert eine Tabelle "Employees" (Mitarbeiter) mit den angegebenen Datenfeldern. Die Erlaubnis dazu hat meistens nur der Datenbankadministrator.

ALTER = Ändern einer Tabellendefinition.

DROP = Löschen einer Tabellendefinition.

12.2.3 Änderungen an den Daten (Updates)

INSERT = Speichern eines Records in einer Tabelle.

Beispiel:

```
INSERT INTO Employees
  (EmployeeNumber, FirstName, LastName, BirthYear, Salary)
  VALUES ( 4710, 'Hans', 'Fleißig', 1972, 2400.0 )
INSERT INTO Employees
  (EmployeeNumber, FirstName, LastName, BirthYear, Salary)
  VALUES ( 4711, 'Grete', 'Tüchtig', 1949, 3200.0 )
```

speichert zwei Mitarbeiter-Records mit den angegebenen Daten.

UPDATE = Verändern von Datenfeldern in einem oder mehreren Records.

Beispiel:

```
UPDATE Employees
  SET Salary = 3600.0 WHERE LastName = 'Tüchtig'
```

setzt das Gehalt bei allen Mitarbeitern, die Tüchtig heißen, auf 3600.

DELETE = Löschen eines oder mehrerer Records

Beispiel:

```
DELETE FROM Employees WHERE EmployeeNumber = 4710
```

löscht den Mitarbeiter mit der Nummer 4710 aus der Datenbank.

12.2.4 Abfrage von Daten

SELECT = Abfragen von gespeicherten Daten.

Beispiele:

```
SELECT * FROM Employees
```

liefert alle Datenfelder von allen Records der Tabelle Employees.

```
SELECT LastName, FirstName, Salary FROM Employees
```

liefert die angegebenen Datenfelder von allen Records der Tabelle Employees.

```
SELECT LastName, Salary
  FROM Employees WHERE BirthYear <= 1970
  ORDER BY Salary DESC
```

liefert den Zunamen und das Gehalt von allen Mitarbeitern, die 1970 oder früher geboren sind, sortiert nach dem Gehalt in der umgekehrten Reihenfolge (höchstes zuerst).

```
SELECT * FROM Employees
  WHERE LastName = 'Fleißig' AND FirstName LIKE 'H%'
```

liefert alle Daten derjenigen Mitarbeiter, deren Zuname Fleißig ist und deren Vorname mit H beginnt.

12.3 Datenbank-Zugriffe in Java (JDBC)

Mit Hilfe der "Java Database Connectivity" (JDBC) kann man innerhalb von Java-Programmen auf Datenbanken zugreifen und Daten abfragen, speichern oder verändern, wenn das Datenbanksystem die "Standard Query Language" SQL verwendet, was bei allen wesentlichen Datenbanksystemen seit den 80er-Jahren der Fall ist.

Im Java-Programm werden nur die logischen Eigenschaften der Datenbank und der Daten angesprochen (also nur die Namen und Typen der Relationen und Datenfelder), und die Datenbank-Operationen werden in der genormten Abfragesprache SQL [Seite 154] formuliert (Version SQL 2 Entry Level).

Das Java-Programm ist somit von der Hard- und Software des Datenbanksystems unabhängig. Erreicht wird dies durch einen sogenannten "Treiber" (Driver), der zur Laufzeit die Verbindung zwischen dem Java-Programm und dem Datenbanksystem herstellt - ähnlich wie ein Drucker-Treiber die Verbindung zwischen einem Textverarbeitungsprogramm und dem Drucker oder zwischen einem Graphikprogramm und dem Plotter herstellt. Falls die Datenbank auf ein anderes System umgestellt ist, braucht nur der Driver ausgewechselt werden, und die Java-Programme können ansonsten unverändert weiter verwendet werden.

Der JDBC-Driver kann auch über das Internet bzw. Intranet vom Java-Client direkt auf den Datenbank-Server zugreifen, ohne dass man eine eigene Server-Applikation (CGI oder Servlet) schreiben muss.

Das JDBC ist bereits im Java Development Kit JDK enthalten (ab 1.1), und zwar im Package java.sql. Mit dem JDK 1.2 kam eine neue Version JDBC 2.0 heraus, die eine Reihe von zusätzlichen Features enthält. Im Folgenden werden aber nur die (wichtigen) Features beschrieben, die sowohl in JDBC 1.x als auch in JDBC 2.0 enthalten sind.

Die Software für die Server-Seite und die JDBC-Driver auf der Client-Seite müssen vom jeweiligen Software-Hersteller des Datenbanksystems erworben werden (also z.B. von der Firma Oracle). Dafür gibt es dann eigene Packages wie z.B. sun.jdbc oder com.firma.produkt. Das JDK enthält auch eine sogenannte JDBC-ODBC-Brücke für den Zugriff auf ODBC-Datenbanken (Open Database Connectivity, z.B. MS-Access).

12.3.1 Die wichtigsten Programmelemente

- `import java.sql.*;`
  macht das Package verfügbar.
- `Connection con = DriverManager.getConnection (db, user, password);`
  baut die Verbindung zu einer Datenbank auf.
- `Statement stmt = con.createStatement();`
  oder `PreparedStatement ...`
  gibt an, in welcher Form die SQL-Befehle zur Datenbank gesendet werden.
- `ResultSet rs = stmt.executeQuery (sql);`
  oder `int n = stmt.executeUpdate (sql);`
  führt einen SQL-Befehl aus (im ersten Fall eine Abfrage, im zweiten Fall eine Datenveränderung).

12.3.2 Beispielskizze für eine Datenbank-Abfrage (select)

Hier eine Skizze für den Aufbau einer typischen Datenbank-Anwendung mit einer Abfrage von Daten. Mehr Informationen über die darin vorkommenden Klassen und Methoden finden Sie in den nachfolgenden Abschnitten und in der Online-Dokumentation.

```
import java.sql.*;
...
try {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  String username="admin";
  String password="geheim";
  Connection con = DriverManager.getConnection
    ("jdbc:odbc://hostname/databasename", username, password);
  con.setReadOnly(true);
  Statement stmt = con.createStatement();

  ResultSet rs = stmt.executeQuery
    ("SELECT LastName, Salary, Age, Sex FROM Employees");
  System.out.println("List of all employees:");
  while (rs.next()) {
    System.out.print(" name=" + rs.getString(1) );
    System.out.print(" salary=" + rs.getDouble(2) );
    System.out.print(" age=" + rs.getInt(3) );
    if ( rs.getBoolean(4) ) System.out.print(" sex=M");
    else                    System.out.print(" sex=F");
    System.out.println();
  }

  rs.close();
  stmt.close();
  con.close();
} catch ...
```

12.3.3 Beispielskizze für einzelne Updates der Datenbank (insert, update, delete)

```
import java.sql.*;
...
try {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  String username="admin";
  String password="geheim";
  Connection con = DriverManager.getConnection
    ("jdbc:odbc://hostname/databasename", username, password);
  Statement stmt = con.createStatement();
  int rowCount = stmt.executeUpdate
    ("UPDATE Employees " +
     "SET Salary = 5000.0 WHERE LastName = 'Partl' ");
  System.out.println(
     rowCount + "Gehaltserhoehungen durchgefuehrt.");
  stmt.close();
  con.close();
} catch ...
```

oder bei mehreren Änderungen analog mit:

```
 int rowCount = 0;
 for (int i=0; i<goodPerson.length; i++) {
   rowCount = rowCount + stmt.executeUpdate
    ("UPDATE Employees SET Salary = " + goodPerson[i].getSalary() +
     " WHERE LastName = '" + goodPerson[i].getName() + "' ");
  }
  System.out.println(
    rowCount + "Gehaltserhoehungen durchgefuehrt.");
```

12.3.4 Beispielskizze für umfangreiche Updates der Datenbank (PreparedStatement)

```
import java.sql.*;
...
try {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  String username="admin";
  String password="geheim";
  Connection con = DriverManager.getConnection
    ("jdbc:odbc://hostname/databasename", username, password);
  con.setAutoCommit(false);

  PreparedStatement ps = con.prepareStatement
    ("UPDATE Employees SET Salary = ? WHERE LastName = ? ");
  for (int i=0; i<goodPerson.length; i++) {
    ps.setDouble (1, goodPerson[i].getSalary() );
    ps.setString (2, goodPerson[i].getName()   );
    ps.executeUpdate();
  }
  ps.close();
  con.commit();

  con.close();
} catch ...
```

12.4 Driver

Vor dem Aufbau einer Verbindung [Seite 160] zur Datenbank muss der JDBC-Driver [Seite 157] in das Java-Programm geladen werden,

- entweder mit der statischen Methode
  `Class.forName("Drivername");`
- oder mit dem Konstruktor
  `Class.forName("Drivername").newInstance();`
- oder mit der statischen Methode
  `DriverManager.registerDriver( new Drivername() );`

Typische Drivernamen sind zum Beispiel:
`sun.jdbc.odbc.JdbcOdbcDriver`
`oracle.jdbc.driver.OracleDriver`

12.5 Connection

Die Klasse Connection dient zur Verbindung mit einer Datenbank. Erzeugt wird diese Verbindung vom JDBC DriverManager [Seite 157] in der Form

```
Connection con =
   DriverManager.getConnection (URL, username, password);
```

Der URL hat eine der folgenden Formen:

- `jdbc:protocol:databasename`
  für eine lokale Datenbank mit JDBC-Driver
- `jdbc:protocol://hostname:port/databasename`
  für eine Datenbank auf einem anderen Rechner, mit JDBC-Driver
- `jdbc:odbc:datasourcename`
  für eine lokale ODBC-Datenbank, mit JDBC-ODBC-Driver
- `jdbc:odbc://hostname/datasourcename`
  für eine ODBC-Datenbank auf einem anderen Rechner, mit JDBC-ODBC-Driver

Der Username und das Passwort sind als String-Parameter anzugeben. Aus Sicherheitsgründen empfiehlt es sich, das Passwort nicht fix im Programm anzugeben sondern vom Benutzer zur Laufzeit eingeben zu lassen, am besten in einem TextField mit setEchoCharacter('*') oder in Swing mit einem JPasswordField.

Die wichtigsten Methoden der Klasse Connection sind:

- `createStatement();`
  erzeugt ein Statement [Seite 161] für die Ausführung von SQL-Befehlen.
- `prepareStatement (String sql);`
  erzeugt ein Prepared Statement [Seite 163] für die optimierte Ausführung von SQL-Befehlen.
- `prepareCall (String call);`
  erzeugt ein Callable Statement [Seite 164] für die optimierte Ausführung einer in der Datenbank gespeicherten Stored Procedure.
- `DatabaseMetaData getMetaData()`
  liefert die DatabaseMetaData [Seite 165] (DDL-Informationen) der Datenbank.
- `setReadOnly(true);`
  es werden nur Datenbank-Abfragen durchgeführt, der Datenbank-Inhalt wird nicht verändert, es sind daher keine Transaktionen notwendig und der Zugriff kann optimiert werden.
- `setReadOnly(false);`
  alle SQL-Statements sind möglich, also sowohl Abfragen als auch Updates (Default).
- `setAutoCommit(true);`
  jedes SQL-Statement wird als eine eigene Transaktion ausgeführt, commit() und rollback() sind nicht notwendig (Default).
- `setAutoCommit(false);`
  mehrere SQL-Statements müssen mit commit() und rollback() zu Transaktionen zusammengefasst werden.
- `commit();`
  beendet eine Transaktion erfolgreich (alle Statements wurden ausgeführt).
- `rollback();`
  beendet eine Transaktion im Fehlerfall (alle Änderungen seit Beginn der Transaktion werden rückgängig gemacht).

- `close();`
  beendet die Verbindung mit der Datenbank.

Beispielskizze:

```
String username="admin";
String password="geheim";
Connection con = DriverManager.getConnection
  ("jdbc:odbc://hostname/databasename", username, password);
Statement stmt = con.createStatement();
con.setReadOnly(true);
...
stmt.close();
con.close();
```

Innerhalb einer Connection können mehrere Statements geöffnet werden, eventuell auch mehrere gleichzeitig.

12.6 Statement

Die Klasse Statement dient zur Ausführung eines SQL-Statements oder von mehreren SQL-Statements nacheinander. Erzeugt wird dieses Statement von der Connection [Seite 160] in der Form

```
Statement stmt = con.createStatement();
```

Die wichtigsten Methoden der Klasse Statement sind:

- `ResultSet res = executeQuery( String sql );`
  führt ein SQL-Statement aus, das ein ResultSet [Seite 162] als Ergebnis liefert, also z.B. ein SELECT-Statement.
- `int rowCount = executeUpdate( String sql );`
  führt ein SQL-Statement aus, das kein ResultSet liefert, also z.B. ein INSERT, UPDATE oder DELETE-Statement. Der Rückgabewert ist die Anahl der Records, für die der Befehl durchgeführt wurde, oder 0.
- `boolean isResultSet = execute( String sql );`
  führt ein SQL-Statement aus, das mehrere Ergebnisse (ResultSets oder UpdateCounts) liefern kann, die man dann mit den Methoden getResultSet, getUpdateCount und getMoreResults abarbeiten muss. Dies trifft nur in seltenen Spezialfällen zu.
- `setQueryTimeout ( int seconds );`
  setzt ein Zeitlimit für die Durchführung von Datenbankabfragen (in Sekunden).
- `close();`
  beendet das Statement.

Beispielskizze für eine Abfrage:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery
  ("SELECT LastName, Salary, Age, Sex FROM Employees");
...
rs.close();
stmt.close();
```

Beispielskizze für ein Update:

```
Statement stmt = con.createStatement();
int rowCount = stmt.executeUpdate
  ("UPDATE Employees" +
   "SET Salary = 5000.0 WHERE LastName = 'Partl' ");
...
stmt.close();
```

Innerhalb eines Statements darf zu jedem Zeitpunkt immer nur höchstens 1 ResultSet offen sein.

12.7 ResultSet

Ein ResultSet ist das Ergebnis einer SQL-Abfrage [Seite 161] , im Allgemeinen das Ergebnis eines SELECT-Statements [Seite 154] . Es enthält einen oder mehrere Records von Datenfeldern und bietet Methoden, um diese Datenfelder ins Java-Programm hereinzuholen.

Die wichtigsten Methoden der Klasse ResultSet sind:

- `boolean next()`
  setzt den "Cursor" auf den ersten bzw. nächsten Record innerhalb des ResultSet. Dessen Datenfelder können dann mit den getXXX-Methoden angesprochen werden.
- `boolean getBoolean( String name )`
  `int getInt ( String name )`
  `float getFloat ( String name )`
  `double getDouble ( String name )`
  `String getString ( String name )`
  usw. liefert den Wert des Datenfeldes mit dem angegebenen Namen (case-insensitive), wenn der Name in der Datenbank eindeutig ist.
- `boolean getBoolean( int n )`
  `int getInt ( int n )`
  `float getFloat ( int n )`
  `double getDouble ( int n )`
  `String getString ( int n )`
  usw. liefert den Wert des n-ten Datenfeldes im Ergebnis (von 1, nicht von 0 an gezählt); wenn man im SELECT-Befehl Feldnamen angegeben hat, dann in der angegebenen Reihenfolge; wenn man im SELECT-Befehl * angegeben hat, dann in der in der Datenbank definierten Reihenfolge.
- `java.sql.Date getDate ( String name )`
  `Time getTime ( String name )`
  `Timestamp getTimestamp ( String name )`
  `java.sql.Date getDate ( int n )`
  usw. liefert analog ein Datums- bzw. Zeit-Objekt. Dabei ist zu beachten, dass die Klasse java.sql.Date verschieden von java.util.Date ist, es gibt in diesen Klassen aber Methoden, um das eine Datum in das andere umzuwandeln (am einfachsten mit getDate und setDate).
- `boolean wasNull()`
  gibt an, ob das zuletzt gelesene Datenfeld leer war (d.h. den SQL-Nullwert enthielt).
- `close()`
  beendet die Abfrage bzw. schließt das ResultSet.
- `ResultSetMetaData getMetaData()`
  liefert die ResultSetMetaData [Seite 165] (DDL-Informationen) zu diesem ResultSet.

Beispielskizze:

```
ResultSet rs = stmt.executeQuery
  ("SELECT LastName, Salary, Age, Sex FROM Employees");
System.out.println("List of all employees:");
  while (rs.next()) {
    System.out.print(" name=" + rs.getString(1) );
    System.out.print(" salary=" + rs.getDouble(2) );
    System.out.print(" age=" + rs.getInt(3) );
    if ( rs.getBoolean(4) ) System.out.print(" sex=M");
    else                    System.out.print(" sex=F");
    System.out.println();
  }
rs.close();
```

Anmerkungen:

Innerhalb eines Statements darf zu jedem Zeitpunkt immer nur höchstens 1 ResultSet offen sein. Wenn man mehrere ResultSets gleichzeitig braucht, muss man dafür mehrere Statements innerhalb der Connection öffnen (sofern das Datenbanksystem das erlaubt).

Die Zugriffe auf die Felder sollen in der Reihenfolge erfolgen, wie sie von der Datenbank geliefert werden, also in der Reihenfolge, in der sie im SELECT-Befehl bzw. bei * im Record stehen. Bei JDBC 1.x können die Records auch nur in der Reihenfolge, in der sie von der Datenbank geliefert werden, verarbeitet werden (mit next).

Ab JDBC 2.0 enthält die Klasse ResultSet zahlreiche weitere Methoden, die auch ein mehrmaliges Lesen der Records (z.B. mit previous) und auch Datenänderungen in einzelnen Records (z.B. mit updateString, updateInt) erlauben.

Ansonsten kann man ein mehrmaliges Abarbeiten der Resultate erreichen, indem man sie in einer Liste oder einem Vector zwischenspeichert. Beispielskizze:

```
ResultSet rs = stmt.getResultSet();
ResultSetMetaData md = rs.getMetaData();
int numberOfColumns =  md.getColumnCount();
int numberOfRows = 0;
Vector rows = new Vector();
while (rs.next()) {
  numberOfRows++;
  Vector newRow = new Vector();
  for (int i=1; i<=numberOfColumns; i++) {
    newRow.addElement(rs.getString(i));
  }
  rows.addElement (newRow);
}
rs.close();
```

12.8 PreparedStatement

Wenn man viele Updates nacheinander ausführt und dazu jedesmal mit executeUpdate einen kompletten INSERT- oder UPDATE-SQL-Befehl an die Datenbank sendet, muss jedesmal wieder der SQL-Befehl interpretiert und dann ausgeführt werden. Bei einer großen Anzahl von ähnlichen Updates kann dies sehr viel unnötige Rechenzeit in Anspruch nehmen.

Um den Update-Vorgang zu beschleunigen, kann man in diesem Fall mit der Connection-Methode prepareStatement ein Muster für den SQL-Befehl an das Datenbanksystem senden, in dem die variablen Datenfelder mit Fragezeichen gekennzeichnet sind, und dann mit den Methoden der Klasse PreparedStatement nur mehr die Daten in diese vorbereiteten SQL-Statements "einfüllen".

Beispielskizze:

```
con.setAutoCommit(false);
PreparedStatement ps = con.prepareStatement
  ("UPDATE Employees SET Salary = ? WHERE LastName = ? ");
for (int i=0; i<goodPerson.length; i++) {
  ps.setFloat  (1, newSalary[i]  );
  ps.setString (2, goodPerson[i] );
  ps.executeUpdate();
}
con.commit();
con.close();
```

12.9 Stored Procedure und CallableStatement

Eine effiziente Alternative zu SQL-Befehlen ("ad hoc queries") sind sogenannte "stored procedures", das sind Abfragen oder Update-Aktionen, die in der Datenbank bereits fertig ausführbar gespeichert sind.

In diesem Fall kann man mit der Conection-Methode prepareCall ein Muster für den Aufruf der Stored Procedure an das Datenbanksystem senden, in dem die variablen Datenfelder mit Fragezeichen gekennzeichnet sind, und dann mit den Methoden der Klasse CallabaleStatement nur mehr die Parameterwerte in den Aufruf "einfüllen" und das Ergebnis "herausholen".

Beispielskizze für eine Stored Procedure mit einfachem Rückgabewert:

```
CallableStatement cs = con.prepareCall
  ("{ ? = call PROCNAME(?,?) }");
cs.registerOutParameter(1, Types.DOUBLE);
cs.setDouble (2, 0.00  );
cs.setString (3, "..." );
cs.execute();
double result = cs.getDouble(1);
cs.close();
```

Beispielskizze für eine Stored Procedure mit ResultSet (Spezialfall Oracle):

```
CallableStatement cs = con.prepareCall
  ("{ ? = call PROCNAME(?,?) }");
cs.registerOutParameter(1, OracleTypes.CURSOR);
cs.setDouble (2, 0.00  );
cs.setString (3, "..." );
cs.execute();
ResultSet rs = ( (OracleCallableStatement) cs).getCursor(1);
while (rs.next()) {
  ... // get values from result set ...
}
rs.close();
cs.close();
```

12.10 DatabaseMetaData und ResultSetMetaData

Mit den Klassen DatabaseMetaData und ResultSetMetaData kann man Informationen über die Datenbank bzw. das ResultSet erhalten, also z.B. welche Tabellen (Relationen) definiert sind, wie die Datenfelder heißen und welchen Typ sie haben, und dergleichen.

Den Zugriff auf die DatabaseMetaData erhält man mit der Methode getMetaData in der Connection. Damit kann man dann alle möglichen Eigenschaften des Datenbanksystems, des JDBC-Drivers, der Datenbank und der Connection abfragen (siehe die Online-Dokumentation).

Den Zugriff auf die ResultSetMetaData erhält man mit der Methode getMetaData im ResultSet. Ein paar typische Methoden der Klasse ResultSetMetaData sind:

- `int getColumnCount()`
  Anzahl der Datenfelder
- `String getColumnName ( int n )`
  welchen Namen das n-te Datenfeld hat
- `int getColumnType ( int n )`
  welchen der SQL-Datentypen das n-te Datenfeld hat (siehe die statischen Konstanten in der Klasse Types)
- `boolean isSearchable ( int n )`
  ob das n-te Datenfeld ein Suchfeld ist, das in der WHERE-Klausel angegeben werden darf
- `int getColumnDisplaySize ( int n )`
  wie viele Zeichen man maximal für die Anzeige des n-ten Datenfeldes braucht
- `String getColumnLabel ( int n )`
  welche Bezeichnung man für das n-te Datenfeld angeben soll (eventuell verständlicher als der Datenfeldname)
- `String getSchemaName ( int n )`
  in welchem Datebank-Schema (DDL) das n-te Datenfeld definiert ist
- `String getTableName ( int n )`
  zu welcher Tabelle (Relation) das n-te Datenfeld gehört

12.11 Datenbank-Anwendungen über das Internet

Wie kann das Konzept bzw. die "Architektur" einer Java-Datenbank-Anwendung aussehen, wenn die Benutzer über das Internet oder ein Intranet von ihren Client-Rechnern aus auf die Datenbank zugreifen sollen?

Ein solches Client-Server-System kann z.B. aus folgenden Komponenten zusammengesetzt werden:

- einer Java-Applikation (oder Servlet) auf dem Server, die auf die Datenbank zugreift, und
- einem HTML-Formular oder einem Java-Applet, das innerhalb einer Web-Page auf den Clients läuft und das User-Interface realisiert.

Die Applikation auf dem Server (bzw. das Servlet) greift auf das Datenbanksystem und damit auf die Daten zu. Es erhält vom Client Abfragen oder Daten, führt die entsprechenden Datenbank-Abfragen oder Daten-Eingaben durch und liefert die Ergebnisse an den Client.

Das Applet stellt das User-Interface dar. Es sendet die Abfragen oder Dateneingaben des Benutzers an den Server und gibt die von dort erhaltenen Daten oder Meldungen an den Benutzer aus.

Das Applet kann von den Benutzern als Teil einer Web-Page mit dem Web-Browser über das Internet geladen werden. Der Benutzer braucht für den Datenbankzugriff also keine andere Software als nur seinen Java-fähigen Web-Browser.

Die Kommunikation zwischen Applet und Server kann je nach der verfügbaren Software auf verschiedene Arten erfolgen:

- Am Server arbeitet ein Servlet innerhalb des Web-Servers, die Kommunikation mit dem Applet erfolgt über HTTP.
- Am Server arbeitet eine Java-Applikation als CGI-Programm innerhalb des Web-Servers, die Kommunikation mit dem Applet erfolgt über HTTP.
- Am Server arbeitet eine selbständige Java-Applikation, die Kommunikation mit dem Applet erfolgt über Sockets.

Die Kommunikation zwischen Server-Applikation (bzw. Servlet) und Datenbank kann

- über JDBC
- oder über das JNI (native Interface) mit Hilfe von Unterprogrammen in C oder einer anderen Programmiersprache

erfolgen. Dabei können das Server-Programm und das Datenbanksystem

- am selben Rechner laufen (Datenbank-Server mit Web-Interface)
- oder auf zwei verschiedenen Rechnern laufen, die über das Internet oder Intranet verbunden sind (Applikations-Server und Datenbank-Server).

Eine weitere Möglichkeit wäre es, dass das Applet selbst mit JDBC über das Internet direkt auf die Datenbank zugreift, mit einem geeigneten JDBC-Driver, der in diesem Fall allerdings auf jedem Client installiert sein oder mit dem Applet mitgeladen werden muss.

Außerdem wäre es auch möglich, statt eines Applet eine Java-Applikation am Client zu installieren, die dann wiederum entweder mit Umweg über ein Server-Programm oder direkt mit einem JDBC-Driver auf die Datenbank zugreift.

Fortgeschrittene Java-Programmierer können auch RMI (Remote Method Invocation) oder EJB (Enterprise Java Beans) oder CORBA (Common Object Request Broker Architecture) verwenden.

Bei wichtigen oder sensiblen Daten muss jeweils auf die Datensicherheit und auf den Datenschutz geachtet werden (Passwort-Schutz, Verschlüsselung der Daten bei der Übertragung).

12.12 Übung: eine kleine Datenbank

Diese Übung besteht aus mehreren Schritten und kann nur dann ausgeführt werden, wenn Sie ein Datenbanksystem und den zugehörigen JDBC-Driver auf Ihrem Rechner installiert haben und wenn Sie wissen, wie man mit diesem Datenbanksystem Datenbanken definiert und verwendet. Wenn dies nicht der Fall ist, können Sie "nur" das Konzept dieser Übungsbeispiele überlegen und die Java-Programme nur schreiben und compilieren, aber nicht ausführen.

Für diese Übung werden die Datenbank und die Java-Applikationen am selben Rechner angelegt und ausgeführt, also ohne Internet-Verbindung.

12.12.1 Vorbereitung

Legen Sie - zum Beispiel mit MS-Access - eine Datenbank an, die eine Tabelle "Mitarbeiter" mit den folgenden Datenfeldern enthält: Abteilung (Text, String), Vorname (Text, String), Zuname (Text, String), Geburtsjahr (Zahl, int), Gehalt (Zahl, double).

Füllen Sie diese Tabelle mit ein paar Datenrecords, etwa wie im Beispiel im Kapitel über relationale Datenbanksysteme [Seite 152] .

Registrieren Sie diese Datenbank auf Ihrem Rechner, zum Beispiel mit dem ODBC-Manager der MS-Windows Systemsteuerung als DSN (Datenquellenname). Dann können Sie den einfachen JDBC-ODBC-Driver verwenden, der im JDK enthalten ist (aber manchmal unerklärliche Fehler liefert).

Bei der Verwendung von anderen Datenbanksystemen müssen Sie sicherstellen, dass ein entsprechender JDBC-Driver verfügbar ist, und die Datenbank dementsprechend anlegen und registrieren.

12.12.2 Einfache Übrungsbeispiele

12.12.3 1. Liste der Geburtsjahre

Schreiben Sie eine einfache Java-Applikation, die eine Liste aller Mitarbeiter mit Vorname und Geburtsjahr auf den Bildschirm ausgibt.

12.12.4 2. Berechnung des Durchschnittsgehalts

Schreiben Sie eine einfache Java-Applikation, die das Durchschnittsgehalt der Mitarbeiter berechnet und auf den Bildschirm ausgibt. Zu diesem Zweck fragen Sie das Gehalt von allen Mitarbeitern ab, bilden die Summe, zählen die Anzahl und dividieren schließlich die Summe durch die Anzahl.

12.12.5 Komplizierte Übrungsbeispiele

Wenn Sie wollen, können Sie das einfache Beispiel auch durch die folgenderen, etwas aufwändigeren Aufgaben ergänzen.

12.12.6 3. Speichern

Schreiben Sie eine einfache Java-Applikation, die ein paar (mindestens 3, höchstens 10) Mitarbeiter-Records in dieser Datenbank speichert.

12.12.7 4. Liste (Abfrage)

Schreiben Sie eine einfache Java-Applikation, die eine Liste aller Mitarbeiter mit Vorname, Zuname und Alter (in Jahren) ausgibt.

12.12.8 5. Berechnungen (Abfrage)

Schreiben Sie eine einfache Java-Applikation, die die Anzahl der Mitarbeiter, das Durchschnittsalter und die Gehaltssumme (Summe der Monatsgehälter) ausgibt.

12.12.9 6. Einfache GUI-Applikation (Abfrage)

Schreiben Sie eine Java-GUI-Applikation, bei der der Benutzer in einem TextField einen Zunamen eingeben kann und dann entweder alle Daten über diesen Mitarbeiter am Bildschirm aufgelistet erhält oder eine Fehlermeldung, dass es keinen Mitarbeiter mit diesem Namen gibt. Das GUI soll auch einen Exit-Button haben, mit dem man die Applikation beenden kann.

Username und Passwort geben Sie der Einfachheit halber direkt im Programm an, auch wenn man das bei echten Datenbanken aus Sicherheitsgründen nicht tun sollte.

12.12.10 7. Aufwändigere GUI-Applikation (Updates)

Schreiben Sie eine Java-GUI-Applikation, bei der der Benutzer zunächst mit 2 Textfeldern seinen Username und sein Passwort eingibt, um mit der Datenbank verbunden zu werden, und dann in einem neuen Fenster mit 4 Textfeldern alle Daten für einen zusätzlichen Mitarbeiter eingeben und mit einem Store-Button in der Datenbank speichern kann. Jedes dieser Fenster soll auch einen Exit-Button haben, mit dem man die Applikation beenden kann.

Fügen Sie mit dieser Applikation ein paar (mindestens 2, höchstens 5) weitere Mitarbeiter in die Datenbank ein und führen Sie dann die Übungsprogramme 4, 5 und 6 neuerlich aus, um zu testen, ob alle Mitarbeiter richtig gespeichert wurden.

Wenn Sie sehr viel mehr Zeit investieren wollen, können Sie auch überlegen, wie ein graphisches User-Interface aufgebaut sein müsste, um damit nicht nur die Speicherung von neuen Records sondern auch Abfragen von gespeicherten Records, Datenänderungen an einzelnen Records und Löschungen von einzelnen Records durchführen zu können.

12.12.11 8. Löschen

Schreiben Sie eine einfache Java-Applikation, die alle Mitarbeiter-Records aus der Datenbank löscht.

Führen Sie dieses Programm aus und testen Sie dann mit Programm 1 oder 4, ob tatsächlich keine Mitarbeiter mehr gespeichert sind.

13 Glossar

Wo ist Java?

Abstract Windowing Toolkit (AWT)
ein Paket von Java-Klassen für Graphische User-Interfaces, die zur Laufzeit vom lokal installierten System sichtbar gemacht werden. Der Name bedeutet übersetzt abstrakter Werkzeugsatz für die Arbeit mit Fenstern.

Applet
ein Java-Programm, das innerhalb eines Web-Browsers abläuft und innerhalb einer Web-Page dargestellt wird; außerdem die Oberklasse für die Programmierung von solchen Applets.

Applikation
ein Programm, das selbständig auf einem Computer abläuft, im Fall von Java-Programmen innerhalb der Java Virtual Machine,

Binärprogramm
die von der Computer-Hardware mit Hilfe des Betriebssystems ausführbare Version eines Programms, wird mit Hilfe des Compilers aus dem für Menschen lesbaren Source-Programm erzeugt. Im Fall von Java-Programmen enthält das Binärprogramm den Bytecode für die Java Virtual Machine.

boolean
Datentyp für logische Werte (true oder false), benannt nach dem Wissenschaftler Boole.

byte
Datentyp für ein Byte; das ist die kleinste Einheit, die bei der Ein- und Ausgabe übertragen wird (8 Bits, entspricht einer Zahl zwischen 0 und 255 oder zwischen -127 und 128).

Bytecode
das vom Java-Compiler erstellte Binärprogramm, das auf allen Computern ausgeführt werden kann, auf denen eine Java Virtual Machine installiert ist.

char
Datentyp für einzelne Zeichen, vom englischen Wort *character* für Zeichen.

Common Gateway Interface (CGI)
die Festlegung, wie auf einem Web-Server laufende Programme die vom Client gewünschten Informationen an den Client senden.

Client
ein Computer oder Programm, das die Informationen oder Dienste bekommen möchte, die auf einem Server angeboten werden. Der Name kommt vom englischen Wort für Kunde.

Compiler (Übersetzer)
eine Software, die das für Menschen lesbare Source-Programm in das maschinenlesbare Binärprogramm übersetzt.

Component
die Oberklasse für alle Klassen, die Komponenten von Graphischen User-Interfaces beschreiben.

Container
die Oberklasse für alle GUI-Komponenten, die weitere Komponenten (*Components*) enthalten können.

Date
eine Klasse für Zeitpunkte (Datum und Uhrzeit).

Datei
siehe File.

Datenbank
: eine Möglichkeit zur Speicherung von großen oder komplexen Datenmengen.

Datenbanksystem
: eine Software zur Verwaltung und Benutzung von Datenbanken.

Datenfeld
: eine Eigenschaft eines Objekts, die einen bestimmten Wert enthalten kann, oder eine lokale Variable innerhalb einer Methode, die zur Speicherung eines Zwischenergebnisses dient.

Debugging
: Entfernen von Programmfehlern, vom englischen Wort für die Vernichtung von Ungeziefer.

double
: Datentyp für Kommazahlen mit normaler Genauigkeit (doppelt so genau wie *float*), vom englischen Wort für doppelt.

Exception
: eine Ausnahme vom normalen Programmablauf; außerdem eine Klasse, die (mit zahlreichen Unterklassen) solche Ausnahmen beschreibt.

Event
: eine Aktion des Benutzers in einem Graphischen User-Interface; außerdem eine Klasse, die (mit mehreren Unterklassen) solche Ereignisse beschreibt.

equals
: die Methode, die Objekte auf gleichen Inhalt untersucht.

Feld (array)
: eine Menge von gleichartigen Datenfeldern.

File
: eine Klasse für die Verarbeitung von Dateien (*Files*) und Verzeichnissen (*Directories*).

float
: Datentyp für Kommazahlen mit geringer Genauigkeit (halb so genau wie *double*); vom englischen Wort *floating point* für Fließkomma.

Frame
: eine Klasse für Bildschirmfenster in Graphischen User-Interfaces.

Garbage-Collector
: ein Programmteil, der den von Objekten nicht mehr benötigten Speicherbereich für neue Objekte frei macht. Der Name kommt vom englischen Wort für Müllsammler.

Graphisches User-Interface (GUI)
: die Möglichkeit, ein Programm mit Tastatur und Maus (oder ähnlichen Geräten) zu bedienen und Informationen nicht nur zeilenweise, sondern in graphischer Form in Bildschirmfenstern darzustellen.

Graphics
: eine Klasse für graphische Darstellungen.

Hostname
: ein Name oder eine Nummer, die einen Rechner innerhalb des Internet oder Intranet eindeutig bezeichnet. Der Name kommt vom englischen Wort *host* für Gastgeber.

Hypertext Markup Language (HTML)
: das Format, in dem Text-Informationen mit Verknüpfungen in Form von sogenannten Hypertext-Links über das WWW übertragen werden (siehe auch WWW Was ist das).

Hypertext Transfer Protocol (HTTP)
: das Protokoll, nach dem die Übertragung zwischen Web-Servern und Web-Browsern erfolgt.

int
: Datentyp für ganze Zahlen im normalen Wertebereich, vom englischen Wort *integer* für ganz.

InputStream
: die Oberklasse für alle Klassen für die Byte-orientierte Eingabe.

Interface (Schnittstelle)
: etwas Ähnliches wie eine Oberklasse, in der jedoch nur die Deklarationen der Methoden festgelegt sind, die von der Unterklasse implementiert werden müssen; erlaubt im Gegensatz zu Oberklassen auch eine mehrfache Vererbung.

Internet
: ein weltweites Netz von miteinander verbundenen Computernetzen. Die Übertragung erfolgt mit dem Protokoll TCP/IP. Über das Internet laufen viele verschiedene Dienste: *Telnet* für den Zugriff auf andere Rechner, *File Transfer Protocol* (FTP) für die Übertragung von Dateien, *Electronic Mail* (E-Mail) für den Austausch von elektronischen Briefen, das *World Wide Web* (WWW) für den Zugriff auf Informationssysteme, die *Usenet* Newsgruppen für Diskussionen, und viele andere. Der Name Internet kommt von der englischen Bezeichnung *interconnected networks* und bedeutet miteinander verbundene Netzstrukturen.

Intranet
: ein internes lokales Computernetz, das die gleiche Technologie wie das Internet verwendet. Der Name kommt vom lateinischen Wort *intra* für innerhalb und vom englischen Wort *network* für Netz.

Iso-Latin-1
: ein Zeichensatz, der nur die westeuropäischen Buchstaben, Ziffern und Sonderzeichen umfasst.

Java
: eine Insel in Indonesien, eine in Amerika übliche Bezeichnung für Kaffee und eine moderne, objektorientierte Programmiersprache

Java Bean
: ein Java-Programm (Klasse) mit genau festgelegten Konventionen für die Schnittstellen, die eine Wiederverwendung dieser Klasse in anderen Programmen ermöglichen. Der Name ist ein Wortspiel, das Kaffeebohne bedeutet.

Java Database Connectivity (JDBC)
: eine Schnittstelle für den Zugriff auf Datenbanken von Java-Programmen aus.

Java Development Kit JDK
: die Software, die für die Erstellung, Übersetzung und Ausführung von Java-Programmen notwendig ist; enthält unter anderem den Java-Compiler, das Java Runtime Environment JRE und diverse Hilfsprogramme. Der Name bedeutet übersetzt Java-Entwicklungs-Werkzeug.

Java Runtime Environment JRE
: die Software, die für die Ausführung von Java-Programmen notwendig ist; enthält unter anderem die Java Virtual Machine JVM und die Klassenbibliothek. Der Name bedeutet übersetzt Java-Laufzeit-Umgebung.

Java Virtual Machine JVM
: die Software, die notwendig ist, um ein Java-Binärprogramm (Bytecode) auf einem Computer auszuführen. Der Name bedeutet virtuelle Java-Maschine und kommt daher, dass der Computer, der direkt nur Windows- oder Macintosh- oder Unix-Binärprogramme ausführen kann, mit Hilfe der JVM so wirkt, als ob er Java-Bytecode ausführen könnte, also als ob er eine Java-Maschine wäre.

Kapselung
: der Schutz von Datenfeldern gegen falsche Werte und von programminternen Methoden gegen falsche Aufrufe durch andere Programme. Dies wird auch als *Data-Hiding* (Daten verstecken) bezeichnet.

Klasse
: eine Menge von gleichartigen Objekten und ein Programm, das die Eigenschaften von solchen Objekten beschreibt.

Klassenbibliothek
: eine Sammlung von Programmen (Klassen), die für verschiedene Anwendungen eingesetzt werden können.

Konstruktor
: ein Programmteil, der ein Objekt erzeugt, also einen Speicherbereich für ein Objekt bereitstellt und die Datenfelder des Objekts auf ihre Anfangswerte setzt. Der Name kommt vom englischen Wort *constructor* für Baumeister.

Layout-Manager
: eine Klasse, die für die Anordnung von GUI-Komponenten in einem Container sorgt.

Listener
: eine Klasse, die auf Aktionen des Benutzers (*Events*) in einem Graphischen User-Interface wartet.

long
: Datentyp für besonders große ganze Zahlen, vom englischen Wort für lang.

Math
: eine Klasse für mathematische Funktionen.

Methode
: ein Programmteil, der eine Aktion beschreibt, die von einem Objekt oder mit einem Objekt ausgeführt werden kann; in manchen Programmiersprachen auch als Unterprogramm, Subroutine, Prozedur oder Funktion bezeichnet.

Oberklasse (Superklasse)
: eine Klasse, die den allgemeinen Fall beschreibt, von dem dann mittels Vererbung Unterklassen gebildet werden können.

Object
: eine Klasse, die allgemeine Objekteigenschaften enthält und als Oberklasse für alle Klassen dient.

Objekt
: ein Exemplar, das Eigenschaften hat und Aktionen durchführen kann. In der objektorientierten Programmierung werden Objekte mit Programmen beschrieben, die Klassen genannt werden.

Objektorientierte Analyse (OOA)
: die Überlegung, aus welchen Objekten und Klassen eine Aufgabenstellung besteht, und welche Eigenschaften und Aktionen diese Objekte haben, vorerst noch unabhängig von der verwendeten Programmiersprache.

Objektorientiertes Design (OOD)
: das Konzept, wie das Ergebnis der objektorientieretn Analyse am besten in einer bestimmten Programmiersprache realisiert werden kann.

Objektorientierte Programmierung (OOP)
: das Schreiben von Programmen, in denen die Eigenschaften und Aktionen von Objekten bzw. von Klassen von Objekten festgelegt werden, in einer Programmiersprache.

Open Database Connectivity (ODBC)
: eine Schnittstelle für den Zugriff auf Datenbanken von Programmen aus.

OutputStream
: die Oberklasse für alle Klassen für die Byte-orientierte Ausgabe.

Paket (package)
: eine Menge von zusammengehörenden Klassen.

Polymorphismus
: siehe Überschreiben.

Portnummer
: eine Nummer, die eindeutig angibt, an welches Programm innerhalb eines Rechners (siehe Hostname) eine Information übertragen werden soll.

println
: eine Methode für die Ausgabe einer Textzeile, von den englischen Wörtern für drucken und Zeile.

private
: kann nur innerhalb derselben Klasse angesprochen werden, vom englischen Wort für privat.

Programm
: eine Anweisung an einen Computer, welche Aktionen er wann ausführen soll. In der objektorientierten Programmierung werden Programme meist Klassen genannt.

Programmiersprache
: eine für Menschen lesbare und für Computer mit Hilfe eines Compilers verständliche Sprache, mit der ein Mensch in Form eines Programms festlegen kann, welche Aktionen ein Computer ausführen soll.

protected
: kann von allen Klassen aus angesprochen werden, die im selben Paket liegen oder eine Unterklasse dieser Klasse sind, vom englischen Wort für geschützt.

Protokoll
: eine Festlegung, wie die Übertragung von Informationen zwischen zwei Programmen oder Systemen erfolgen soll.

public
: kann von allen Klassen aus angesprochen werden, vom englischen Wort für öffentlich.

Reader
: die Oberklasse für alle Klassen für die textorientierte Eingabe.

Referenz
: eine Angabe, wo ein bestimmtes Datenfeld zu finden ist; in manchen Programmiersprachen auch als *Pointer* (Zeiger) bezeichnet. In Java werden Objekte und Felder (*arrays*) immer über Referenzen angesprochen.

Relationale Datenbank
: eine Datenbank, bei der die Informationen in Form von Relationen gespeichert sind, d.h. in Form von Tabellen und von Beziehungen zwischen Tabellen.

Runnable
: ein Interface für Programme, die als selbständige Threads ablaufen können.

Serializable
: ein Interface für Objekte, die als Folge von Bytes gespeichert oder übertragen werden können.

Servlet
: ein Java-Programm, das innerhalb eines Web-Servers abläuft und die Ausgabe an einen Web-Browser sendet.

Server
: ein Computer oder Programm, das die Informationen oder Dienste anbietet, die von einem Client nachgefragt werden. Der Name kommt vom englischen Wort für einen Kunden bedienen.

sleep
: eine Methode für eine Zeitdauer, in der ein Programm keine Aktivität ausführt, vom englischen Wort für schlafen.

Signatur (Signature)
: Kennzeichen, Unterschrift; bei Methoden: das, was die Methode eindeutig kennzeichnet, nämlich Name und Parameterliste.

Socket
: eine Verbindung zwischen einem Programm auf einem Computer über das Internet oder Intranet zu einem anderen Programm auf einem anderen Computer. Der Name kommt vom englischen Wort für Steckdose.

Source-Programm (Quellprogramm)
: das von Menschen geschriebene Programm, das dann vom Compiler in das am Computer ausführbare Binärprogramm übersetzt wird.

static
: eine Eigenschaft oder Methode, die nicht je einmal pro Objekt, sondern von Anfang an nur einmal für die Klasse existiert.

Statement (Anweisung)
: ein einzelner Schritt, der innerhalb eines Programms ausgeführt wird.

String
: eine Klasse für Zeichenketten, also für Texte oder Teile von Texten.

Structured Query Language (SQL)
: eine Sprache für die Datenabfrage und Datenänderung in Datenbanken.

Syntax (Grammatik)
: die Regeln, wie die Wörter, Zahlen und Sonderzeichen in einem Source-Programm geschrieben werden müssen, damit es vom Compiler richtig verstanden und richtig übersetzt werden kann.

System
: eine Klasse für Systemkomponenten und Systemfunktionen.

Sun Microsystems
: die EDV-Firma, in der die Programmiersprache Java entwickelt wurde.

Swing
: ein umfangreiches Paket von Java-Klassen für Graphische User-Interfaces, bei denen das Aussehen komplett innerhalb des Java-Programms festgelegt wird. Dies wird auch als leichtgewichtige Klassen bezeichnet (englisch *light-weight*). Der Produktname *Swing* kommt vom englischen Wort für Schaukel und für schwungvolle Musik.

TCP/IP
: das Protokoll, nach dem die Übertragung im Internet und in Intranets erfolgt, eine Abkürzung für Transaction Control Protocol / Internet Protocol.

Thread
: ein Programmteil, der zeitlich unabhängig von anderen Programmteilen abläuft, und eine Klasse, mit der solche Abläufe gesteuert werden können; vom englischen Wort für Faden.

Timeout
: eine maximale Wartezeit, vom englischen Wort für Zeitüberschreitung.

toString
: die Methode, die Objekte durch einen für Menschen lesbaren Text darstellt.

Überladen
: die Definition von mehreren Methoden oder Konstruktoren, die den gleichen Namen, aber verschiedene Parameterlisten haben und deshalb als verschiedene Methoden gelten.

Überschreiben
: die Definition einer Methode in einer Unterklasse, die für Objekte dieses Typs die von der Oberklasse geerbte Methode ersetzt. Dies wird auch als Polymorphismus bezeichnet, vom griechischen Wort für viele Gestalten.

Unicode
: ein Zeichensatz, der alle Schriftzeichen der Menschheit umfasst (nicht nur die westeuropäischen).

Uniform Resource Locator (URL)
: eine Adresse, unter der eine Information oder Datei über das Internet erreichbar ist; außerdem eine Klasse für den Zugriff auf solche Informationen. Der Name bedeutet eine einheitliche Ortsangabe für Ressourcen.

Unterklasse (Subklasse)
: eine Klasse, die einen Spezialfall einer Oberklasse beschreibt und alle Eigenschaften und Methoden von der Oberklasse erbt (Vererbung).

Utility
ein Hilfsprogramm, vom englischen Wort für nützlich.

Vererbung
eine Beziehung der Art "Unterklasse ist ein Spezialfall von Oberklasse".

Web-Browser
ein Computer bzw. das dort laufende Client-Programm, mit dem der Benutzer auf Web-Pages und eventuell auch auf andere Internet-Dienste zugreifen kann.

Web-Page
eine über das WWW auf einem Web-Server angebotene Information, meist in Form eines HTML-Files.

Web-Server
ein Computer bzw. das dort laufende Server-Programm, das Web-Pages über das WWW anbietet.

World-Wide Web (WWW)
ein Informationssystem, das einen weltweiten Zugriff auf Informationen bietet, die meist in der Form von HTML-Files (Web-Pages) angeboten werden. Der Zugriff erfolgt nach dem Prinzip von Server und Client über das Internet mit dem Protokoll HTTP. Der Name bedeutet so viel wie weltweites Spinnennetz oder weltweites Gewebe. (siehe auch WWW Was ist das).

Writer
die Oberklasse für alle Klassen für die textorientierte Ausgabe.

14 Referenzen

:

- Online-Informationen über Java:
  - Online-Dokumentation (API) des JDK [Seite 8] - auch on-line auf http://java.sun.com/docs/
  - http://java.sun.com/ - auch unter http://www.javasoft.com/
  - http://www.gamelan.com/ - auch unter http://www.developer.com/java/
  - http://www.sourceforge.net/
  - Java Tutorial auf http://java.sun.com/docs/books/tutorial/index.html
  - Java FAQ auf http://sunsite.unc.edu/javafaq/javafaq.html
  - Java Programmers FAQ auf http://www.afu.com/javafaq.html
  - Java Glossary auf http://mindprod.com/
  - deutsche Java-FAQ auf http://www.dclj.de/
  - Java-Einführung von Hubert Partl auf http://www.boku.ac.at/javaeinf/
  - Handbuch der Java-Programmierung von Guido Krüger auf http://www.javabuch.de/ - auch unter http://www.gkrueger.com/
  - Java ist auch eine Insel von Christian Ullenboom auf http://java-tutor.com/
  - Java Dokumentation von Brit Schröter und Johann Plank auf http://www.selfjava.de/
  - interaktives Java Tutorial von Bradley Kjells auf http://www.gailer-net.de/tutorials/java/java-toc.html
  - Thinking in Java von Bruce Eckel auf http://www.BruceEckel.com/
  - Java Tutorial von Prof.Baldwin auf http://www.dickbaldwin.com/
  - Java Developers Almanac von Patrick Chan auf http://javaalmanac.com/
  - Liste weiterer Links beim dmoz Open Directory Project auf http://dmoz.org/Computers/Programming/Languages/Java
  - siehe auch Google-Suche über Java
- Online-Informationen über Internet, WWW und HTML:
  - HTML-Einführung von H.Partl auf http://www.boku.ac.at/htmleinf/
  - W3-Consortium auf http://www.w3.org/
  - siehe auch Google-Suche über HTML
- Newsgruppen über Java und Web-Pages:
  - comp.lang.java.help
  - comp.lang.java.programmer
  - comp.lang.java.gui
  - und andere Spezialgruppen in der Hierarchie comp.lang.java.*
  - comp.infosystems.www.authoring.html.misc
  - comp.infosystems.www.authoring.site-design
  - und andere Gruppen in der Hierarchie comp.infosystems.*
  - de.comp.lang.java
  - de.comm.infosystems.www.authoring.misc
  - und andere Gruppen in den Hierarchien de.comm.* und de.comp.*
  - siehe auch Google Groups Suche (ehemals AltaVista, DejaNews)

Beilage:

- Musterlösungen der Übungsaufgaben

JAVA - EINFÜHRUNG

Musterlösungen

Hubert Partl

Inhaltsverzeichnis

Java Einführung

von Hubert Partl, ZID BOKU Wien

Musterlösungen zu den Übungsaufgaben

Wenn Sie Java erfolgreich lernen wollen, empfehle ich Ihnen dringend, die Musterlösungen erst dann auszudrucken und anzusehen, wenn Sie bereits *alle* Übungsbeispiele selbständig fertig programmiert haben, also erst am Ende des Kurses.

Es gibt immer mehrere Möglichkeiten, eine Aufgabe zu programmieren. Die hier gezeigten Musterlösungen sind also nicht notwendiger Weise "besser" als das, was Sie selbst geschrieben haben. Außerdem hat sich gezeigt, dass manche von diesen Beispielen in bestimmten Java-Implementierungen *nicht* fehlerfrei funktionieren.

1 HelloWorld-Applikation

1.1 HelloWorld.java

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

    System.out.println("Hello World!");
  }
}
```

1.2 HelloText.java

```
public class HelloText {

  public String messageText = "Hello World!";
  // or: private ...

  public void printText() {
    System.out.println (messageText);
  }

  public static void main (String[] args) {
    HelloText h = new HelloText();
    h.printText();
  }
}
```

1.3 MultiText.java

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

    HelloText engl = new HelloText();
    HelloText germ = new HelloText();
    germ.messageText = "Hallo, liebe Leute!";
    HelloText cat = new HelloText();
    cat.messageText = "Miau!";

    engl.printText();
    germ.printText();
    engl.printText();
    cat.printText();
    engl.printText();
  }
}
```

2 Programmdokumentation (javadoc)

2.1 HelloDoc.java

```
/**
* ein einfaches Hello-World-Programm.
* <p>
* Im Gegensatz zum kurzen, rein statischen "HelloWorld"
* ist dieses Programm ein Musterbeispiel
* für eine <b>objekt-orientierte</b> Java-Applikation.
*
* @author Hubert Partl
* @version 99.9
* @since JDK 1.0
* @see HelloWorld
*/
public class HelloDoc {

  /** der Text, der gedruckt werden soll.
  */
  public String messageText = "Hello World!";

  /** druckt den Text messageText auf System.out aus.
  *   @see #messageText
  */
  public void printText() {
    System.out.println (messageText);
  }

  /** Test des HelloDoc-Objekts.
  */
  public static void main (String[] args) {
    HelloDoc h = new HelloDoc();
    h.printText();
  }
}
```

3 Quadratzahlen

3.1 Quadrat.java

```
public class Quadrat {
  public static void main (String[] args) {
    int zahl, quad;
    System.out.println(" Quadratzahlen:");
    for ( zahl=1; zahl<=20; zahl++ ) {
      quad = zahl * zahl;
      System.out.println (zahl + " * " + zahl +
        " = " + quad );
    }
  }
}
```

4 Steuer

4.1 Steuer.java

```
public class Steuer {
  public static void main (String[] args) {
    double brutto = 100; /* inkl. USt. */
    double prozentsatz = 20;
    System.out.println (brutto + " brutto");
    double steuer = brutto * prozentsatz / (100.0 + prozentsatz);
    System.out.println (steuer + " USt.");
    double netto = brutto - steuer;
    System.out.println (netto + " netto");
  }
}
```

5 Sparbuch

5.1 Bank.java

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

    double invest = 10000.0;
    double rate = 4.0;
    double factor = (100.0 + rate) / 100.0;
    double amount;
    int numYears = 10;

    System.out.println("Investment =      " + invest);
    System.out.println("Interest Rate =   " + rate);

    amount=invest;
    for (int year=1; year<=numYears; year++) {
      amount = amount * factor;
      System.out.println ("Year " + year +
        "   Amount = " + amount );
    }
  }
}
```

6 erweitertes Sparbuch

6.1 BankEx.java

```
import java.text.*;

public class BankEx {
  public static void main (String[] args) {
    double invest, rate, factor, amount;
    int numYears;

    DecimalFormat df = new DecimalFormat("#,###,##0.00");
    DecimalFormat intf = new DecimalFormat("00");

    if (args.length != 2) {
      System.out.println ("Usage: java BankEx amount rate");
      System.exit(1);
    }

    try {
      invest = Double.valueOf(args[0]).doubleValue();
      rate = Double.valueOf(args[1]).doubleValue();
      factor = (100.0 + rate) / 100.0;
      numYears = 10;

      System.out.println("Investment =      " +
        df.format(invest) );
      System.out.println("Interest Rate =   " +
        df.format(rate) );

      amount=invest;
      for (int year=1; year<=numYears; year++) {
        amount = amount * factor;
      System.out.println ("Year " + intf.format(year) +
          "   Amount = " + df.format(amount) );
      }
    } catch (Exception e) {
        System.out.println("*** error: " + e);
    }
  }
}
```

6.2 BankEx2.java

```
import java.text.*;

public class BankEx2 {
  public static void main (String[] args) {
    double invest, rate, factor, amount;
    int numYears;
    DecimalFormat df = new DecimalFormat("#,###,##0.00");
    DecimalFormat intf = new DecimalFormat("00");

    try {
      invest = Double.valueOf(args[0]).doubleValue();
    } catch (Exception e) {
      // ArrayIndexOutOfBoundsException or NumberFormatException
      System.out.println("Investment not specified * " + e);
      invest = 1000;
    }
```

```
    try {
      rate = Double.valueOf(args[1]).doubleValue();
    } catch (Exception e) {
      // ArrayIndexOutOfBoundsException or NumberFormatException
      System.out.println("Rate not specified * " + e);
      rate = 3.5;
    }

    factor = (100.0 + rate) / 100.0;
    numYears = 10;
    System.out.println("Investment =      " +
      df.format(invest) );
    System.out.println("Interest Rate =   " +
      df.format(rate) );

    amount=invest;
    for (int year=1; year<=numYears; year++) {
      amount = amount * factor;
      System.out.println ("Year " + intf.format(year) +
        "   Amount = " + df.format(amount) );
    }
  }
}
```

7 Kurs

7.1 Kurs.java

```
public class Kurs {
  private String kursTitel = null;
  private boolean kostenlos = false;
  private int anzahl = 0;
  private String[] teilnehmer;

  public Kurs (String kursTitel, int maxAnzahl) {
    setKursTitel(kursTitel);
    teilnehmer = new String [maxAnzahl];
  }

  private void setKursTitel (String kursTitel) {
    this.kursTitel = kursTitel;
  }
  public String getKursTitel() {
    return kursTitel;
  }
  public void setKostenlos (boolean kostenlos) {
    this.kostenlos = kostenlos;
  }
  public boolean isKostenlos() {
    return kostenlos;
  }
  // no setTeilnehmer(String[])
  public String[] getTeilnehmer() {
    return teilnehmer;
  }
  private void setTeilnehmer (int i, String name)
      throws ArrayIndexOutOfBoundsException {
    this.teilnehmer[i] = name;
  }
  public String getTeilnehmer (int i)
      throws ArrayIndexOutOfBoundsException {
    return teilnehmer[i];
  }
  public void addTeilnehmer (String name)
      throws ArrayIndexOutOfBoundsException {
    if (anzahl >= teilnehmer.length )
      throw new ArrayIndexOutOfBoundsException();
    else {
      anzahl++;
      setTeilnehmer (anzahl-1, name);
    }
  }
  // no setAnzahl(int)
  public int getAnzahl() {
    return anzahl;
  }
  public int getMaxAnzahl() {
    return teilnehmer.length;
  }
  public boolean equals (Object other) {
   return ( other instanceof Kurs &&
     ((Kurs)other).getKursTitel().equals (this.kursTitel) );
  }
  public String toString() {
```

```
    String s = getKursTitel();
    if ( isKostenlos() )
      s = s + " (kostenlos)";
    s = s + ": " + getAnzahl() + " Teilnehmer, "
      + (getMaxAnzahl()-getAnzahl()) + " Plätze frei";
    return s;
  }

  public static void main (String[] args) {
    Kurs java1 = new Kurs ("Java Einfuehrung", 15);
    java1.setKostenlos(true);
    Kurs java2 = new Kurs ("Java fuer Fortgeschrittene", 8);
    java1.addTeilnehmer("Hubert");
    java2.addTeilnehmer("Hubert");
    java1.addTeilnehmer("Clemens");
    java1.addTeilnehmer("Ernst");
    java2.addTeilnehmer("Markus");
    System.out.println (java1);
    for (int i=0; i < java1.getAnzahl(); i++)
      System.out.println ("  " + java1.getTeilnehmer(i));
    System.out.println (java2);
    for (int i=0; i < java2.getAnzahl(); i++)
      System.out.println ("  " + java2.getTeilnehmer(i));
  }
}
```

8 Person und Student

8.1 Person.java

```
public class Person {

   private String vorname;
   private String zuname;

   public Person (String zuname, String vorname) {
      this.setZuname(zuname);
      this.setVorname(vorname);
   }

   protected void setZuname (String s) {
      this.zuname = s;
   }
   public String getZuname() {
      return this.zuname;
   }

   protected void setVorname (String s) {
      this.vorname = s;
   }
   public String getVorname() {
      return this.vorname;
   }

   public String toString() {
      return this.vorname + " " + this.zuname;
   }

}
```

8.2 Student.java

```
public class Student extends Person {

   private String uni;

   public Student (String zuname, String vorname,
         String uni) {
      super (zuname, vorname);
      this.uni = uni;
    }

   protected void setUni (String uni) {
      this.uni = uni;
   }
   public String getUni() {
      return this.uni;
   }

  public String toString() {
      String s;
      s = super.toString() + ", studiert an " + uni;
      return s;
   }
}
```

8.3 PersStud.java

```
public class PersStud {
  public static void main (String[] args) {
    Person georg, willi, anna, barbara;

    anna = new Person ("Schmidt", "Anna");
    barbara = new Student ("Paulus", "Barbara", "TU Wien");
    georg = new Student ("Fischer", "Georg", "BOKU");
    willi = new Person ("Schlosser", "Wilhelm");

    System.out.println();
    System.out.println("Vornamen:");
    System.out.println( anna.getVorname() );
    System.out.println( barbara.getVorname() );
    System.out.println( georg.getVorname() );
    System.out.println( willi.getVorname() );
    System.out.println();
    System.out.println("Komplette Informationen:");
    System.out.println( anna );
    System.out.println( barbara );
    System.out.println( georg );
    System.out.println( willi );
    System.out.println();
    }
}
```

9 Konto

9.1 KontoNichtGedecktException.java

```
public class KontoNichtGedecktException
   extends IllegalArgumentException {}
```

9.2 Konto.java

```
public class Konto {

   private Person inhaber;
   private double guthaben = 0.0;

   public Konto (Person inhaber) {
      this.setInhaber(inhaber);
   }

   private void setInhaber (Person inhaber) {
      this.inhaber = inhaber;
   }
   public Person getInhaber() {
      return this.inhaber;
   }

   private void setGuthaben (double neuerBetrag) {
        this.guthaben = neuerBetrag;
   }
   public double getGuthaben() {
      return this.guthaben;
   }

   public void einzahlen (double betrag) {
      this.setGuthaben( this.guthaben + betrag );
   }

   public void abheben (double betrag) {
      double neuerBetrag = this.guthaben - betrag;
      if (neuerBetrag >= 0.0) {
        this.setGuthaben( neuerBetrag );
      }
   }

   public String toString() {
      return this.inhaber + ": " + this.guthaben + " Euro";
   }

   public static void main (String[] args) {

     Person hubert = new Person ("Partl", "Hubert");
     Konto kontoH = new Konto (hubert);
     System.out.println(kontoH);
     kontoH.einzahlen(1000.0);
     System.out.println(kontoH);
     kontoH.abheben(500.0);
     System.out.println(kontoH);
     kontoH.abheben(9999.0);
     System.out.println(kontoH);

     Student anna = new Student ("Fleißig", "Anna", "Uni Wien");
```

```
      Konto kontoA = new Konto (anna);
      System.out.println(kontoA);
      kontoA.einzahlen(1000.0);
      System.out.println(kontoA);
      kontoA.abheben(500.0);
      System.out.println(kontoA);

   }
}
```

10 Katzenmusik

10.1 Katzenmusik.java

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

    LautesTier tier1 = new Katze();
    LautesTier tier2 = new Hund();
    for (int i=1; i<=10; i++) {
      tier1.gibtLaut();
      tier2.gibtLaut();
    }
  }
}
```

10.2 LautesTier.java

```
public interface LautesTier {
  public void gibtLaut() ;
}
```

10.3 Katze.java

```
public class Katze implements LautesTier {
  public void gibtLaut() {
    System.out.println("Miau!");
  }
}
```

10.4 Hund.java

```
public class Hund implements LautesTier {
  public void gibtLaut() {
    System.out.println("Wau wau!");
  }
}
```

11 Layout-Manager

11.1 LayoutTests.java

```
import java.awt.* ;

public class LayoutTests extends Frame {

  private Button b1, b2;
  private Label lab;
  private TextField tf;

  // activate (un-comment) one of the following lines:
  private FlowLayout myLayout = new FlowLayout();
  // private FlowLayout myLayout = new FlowLayout( FlowLayout.LEFT );
  // private GridLayout myLayout = new GridLayout(2,2);
  // private GridLayout myLayout = new GridLayout(4,1);
  // private BorderLayout myLayout = new BorderLayout();

  public void init() {
    setLayout ( myLayout );
    b1 = new Button("ok");
    add(b1,"West");
    b2 = new Button("cancel");
    add(b2,"South");
    lab = new Label("Hallo!");
    add(lab,"Center");
    tf = new TextField("",20);
    add(tf,"East");
    setSize(300,200); // or: pack();
    setVisible(true);
  }

  public static void main (String[] args) {
    LayoutTests f = new LayoutTests();
    f.init();
  }

}
```

12 Canvas Verkehrsampel

12.1 TrafficCanvas.java

```
import java.awt.*;

public class TrafficCanvas extends Canvas {

  public Dimension getMinimumSize() {
    return new Dimension(100,260);
  }
  public Dimension getPreferredSize() {
    return getMinimumSize();
  }

  public void paint (Graphics g) {
    g.setColor (Color.black);
    g.fillRect (10, 10, 80, 240);
    g.setColor (Color.red);
    g.fillOval (20, 20, 60, 60);
    g.setColor (Color.yellow);
    g.fillOval (20, 100, 60, 60);
    g.setColor (Color.green);
    g.fillOval (20, 180, 60, 60);
  }

  public static void main (String[] args) {
    Frame f = new Frame("Traffic Light Test");
    TrafficCanvas theLight = new TrafficCanvas();
    f.setLayout ( new FlowLayout() );
    f.add(theLight);
    f.setSize(200,300); // or: pack();
    f.setVisible(true);
    // no event handling, use Ctrl-C to stop this program.
  }
}
```

13 einfache GUI-Applikation

13.1 Gui.java

```
import java.awt.*;
import java.awt.event.*;

public class Gui extends Frame
  implements ActionListener, WindowListener {

  Label titleLabel;
  Button openButton, closeButton;
  Panel northPanel, southPanel;

  public Gui (String s) {
    super(s);
  }

  public void init () {

    setLayout (new BorderLayout() );

    // north panel = title label

    northPanel = new Panel();
    northPanel.setLayout (new FlowLayout() );
    titleLabel = new Label ("GUI Exercise ");
    northPanel.add(titleLabel);
    add (northPanel, "North");

    // south panel = two buttons

    openButton = new Button(" Open ");

    closeButton = new Button(" Close ");
    closeButton.addActionListener (this);
    closeButton.setActionCommand ("close");

    southPanel = new Panel();
    southPanel.setLayout (new FlowLayout() );
    southPanel.add(openButton);
    southPanel.add(closeButton);
    add (southPanel, "South");

    addWindowListener(this);
    pack(); // setSize(500,500);
    setVisible(true);
  }

  public static void main (String[] args) {
    Gui f = new Gui("Test");
    f.init();
  }

  public void actionPerformed (ActionEvent e) {
    String which = e.getActionCommand();
    if ( which.equals("close") ) {
      dispose();
      System.exit(0);
    }
```

```
  }
  public void windowClosing (WindowEvent e) {
    dispose();
    System.exit(0);
  }
  public void windowClosed (WindowEvent e) { }
  public void windowOpened (WindowEvent e) { }
  public void windowIconified (WindowEvent e) { }
  public void windowDeiconified (WindowEvent e) { }
  public void windowActivated (WindowEvent e) { }
  public void windowDeactivated (WindowEvent e) { }
}
```

14 HelloWorld Applet

14.1 HelloApp.java

```
import java.awt.*;
import java.applet.*;

public class HelloApp extends Applet {

  private String s = "Hello World!";

  public void paint (Graphics g) {
    g.drawString (s, 25, 25);
  }
}
```

14.2 hello.html

```
<html>
<head>
<title>Example</title>
</head>
<body>

<h1>Example</h1>

<p align=center>
<applet code="HelloApp.class" width=200 height=100
        alt="Hello World!">
  Hello World!
</applet>
</p>

</body>
</html>
```

15 Applet Thermostat (in einer Klasse)

15.1 Thermo.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Thermo extends Applet
  implements ActionListener {

  private int temp = 22;
  private Label tempDisplay;
  private Button plusButton, minusButton;
  private Panel centerPanel, southPanel;

  public void init () {

    setLayout (new BorderLayout() );

    // center panel = temperature display

    tempDisplay = new Label("00", Label.CENTER);
    Font bigFont = new Font ("Helvetica", Font.BOLD, 20);
    tempDisplay.setFont(bigFont);
    showTemp();

    centerPanel = new Panel();
    centerPanel.setLayout (new FlowLayout() );
    centerPanel.add(tempDisplay);
    add (centerPanel, "Center");

     // south panel = two buttons

    minusButton = new Button(" -1");
    minusButton.addActionListener (this);
    minusButton.setActionCommand ("minus");

    plusButton = new Button("+1");
    plusButton.addActionListener (this);
    plusButton.setActionCommand ("plus");

    southPanel = new Panel();
    southPanel.setLayout (new FlowLayout() );
    southPanel.add(minusButton);
    southPanel.add(plusButton);
    add (southPanel, "South");

    setVisible(true);
  }

  private void showTemp() {
    tempDisplay.setBackground(Color.white);
    if (temp<20) {
      tempDisplay.setForeground(Color.blue);
    }
    else if (temp>25) {
      tempDisplay.setForeground(Color.red);
    }
    else {
```

```
      tempDisplay.setForeground(Color.magenta);
    }
    tempDisplay.setText("" + temp);
  }

  public void actionPerformed (ActionEvent e) {
    String which = e.getActionCommand();
    if ( which.equals("minus") ) {
      temp--;
    }
    else if ( which.equals("plus") ) {
      temp++;
    }
    showTemp();
  }

}
```

15.2 Thermo.html

```
<html>
<applet code="Thermo.class" width=200 height=100>
  <p>Without Java, you cannot play with the temperature, sorry.
</applet>
</html>
```

16 Applet Thermostat (in drei Klassen)

16.1 ThermoModel.java

```
public class ThermoModel {

  private int temp = 22;

  public void setTemp (int temp) {
    this.temp = temp;
  }

  public void addTemp (int diff) {
    temp = temp + diff;
  }

  public int getTemp() {
    return temp;
  }
}
```

16.2 ThermoView.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ThermoView extends Applet {

  private ThermoModel model;
  private ThermoController controller;
  private Label tempDisplay;
  private Button plusButton, minusButton;
  private Panel centerPanel, southPanel;

  public void init () {

    model = new ThermoModel();
    model.setTemp(22);
    controller = new ThermoController (this, model);

    setLayout (new BorderLayout() );

    // center panel = temperature display

    tempDisplay = new Label("00", Label.CENTER);
    Font bigFont = new Font ("Helvetica", Font.BOLD, 20);
    tempDisplay.setFont(bigFont);
    showTemp();

    centerPanel = new Panel();
    centerPanel.setLayout (new FlowLayout() );
    centerPanel.add(tempDisplay);
    add (centerPanel, "Center");

     // south panel = two buttons

    minusButton = new Button(" -1");
    minusButton.addActionListener ( controller );
    minusButton.setActionCommand ("minus");
```

```
    plusButton = new Button("+1");
    plusButton.addActionListener ( controller );
    plusButton.setActionCommand ("plus");

    southPanel = new Panel();
    southPanel.setLayout (new FlowLayout() );
    southPanel.add(minusButton);
    southPanel.add(plusButton);
    add (southPanel, "South");

    setVisible(true);
  }

  protected void showTemp() {
    int temp = model.getTemp();
    tempDisplay.setBackground(Color.white);
    if (temp<20) {
      tempDisplay.setForeground(Color.blue);
    }
    else if (temp>25) {
      tempDisplay.setForeground(Color.red);
    }
    else {
      tempDisplay.setForeground(Color.magenta);
    }
    tempDisplay.setText("" + temp);
  }
}
```

16.3 ThermoController.java

```
import java.awt.*;
import java.awt.event.*;

public class ThermoController
  implements ActionListener {

  private ThermoView view;
  private ThermoModel model;

  public ThermoController (ThermoView v, ThermoModel m) {
    this.view = v;
    this.model = m;
  }

  public void actionPerformed (ActionEvent e) {
    String which = e.getActionCommand();
    if ( which.equals("minus") ) {
      model.addTemp(-1);
    }
    else if ( which.equals("plus") ) {
      model.addTemp(+1);
    }
    view.showTemp();
  }
}
```

16.4 ThermoView.html

```
<html>
<applet code="ThermoView.class" width=200 height=100>
  <p>Without Java, you cannot play with the temperature, sorry.
</applet>
</html>
```

17 Applet Verkehrsampel

17.1 TrafficLight.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class TrafficLight extends Applet
  implements ActionListener {

  private Label titleLabel;
  private TheLight theLight; // defined below
  private Button nextButton, stopButton;
  private Panel northPanel, centerPanel, southPanel;

  public void init () {

    theLight = new TheLight();

    setLayout (new BorderLayout() );

    // north panel = title label

    northPanel = new Panel();
    northPanel.setLayout (new FlowLayout() );
    titleLabel = new Label ("Traffic Light Exercise ");
    northPanel.add(titleLabel);
    add (northPanel, "North");

    // center panel = traffic light drawing canvas

    centerPanel = new Panel();
    centerPanel.setLayout (new FlowLayout() );
    centerPanel.add(theLight);
    add (centerPanel, "Center");

    // south panel = two buttons

    nextButton = new Button("Next");
    nextButton.addActionListener (this);
    nextButton.setActionCommand ("next");

    stopButton = new Button("Stop");
    stopButton.addActionListener (this);
    stopButton.setActionCommand ("stop");

    southPanel = new Panel();
    southPanel.setLayout (new FlowLayout() );
    southPanel.add(nextButton);
    southPanel.add(stopButton);
    add (southPanel, "South");

    setVisible(true);
  }

  public void actionPerformed (ActionEvent e) {
    String which = (String) e.getActionCommand();
    if ( which.equals("next") ) {
      theLight.nextState();
```

```
      theLight.repaint();
    }
    if ( which.equals("stop") ) {
      theLight.setState(TheLight.RED);
      theLight.repaint();
    }
  }
}
```

17.2 TheLight.java

```
import java.awt.*;

public class TheLight extends Canvas {

  public static final int RED = 1;
  public static final int REDYELLOW = 2;
  public static final int GREEN = 3;
  public static final int YELLOW = 4;
  private static final int MAXSTATE = 4;

  private int lightState = RED;

  public int getState() {
     return lightState;
  }
  public void setState (int state) {
    lightState=state;
  }
  public void nextState() {
    lightState++;
    if (lightState>MAXSTATE) {
      lightState=1;
    }
  }

  public Dimension getMinimumSize() {
    return new Dimension(100,260);
  }
  public Dimension getPreferredSize() {
    return getMinimumSize();
  }

  public void paint (Graphics g) {
    switch (lightState) {
      case RED:
        paintLights (g, true, false, false);
        break;
      case REDYELLOW:
        paintLights (g, true, true, false);
        break;
      case GREEN:
        paintLights (g, false, false, true);
        break;
      case YELLOW:
        paintLights (g, false, true, false);
        break;
      }
    }

  private void paintLights (Graphics g,
```

```
      boolean red, boolean yellow, boolean green) {
    g.setColor (Color.black);
    g.fillRect (10, 10, 80, 240);
    if (red) {
      g.setColor (Color.red);
      g.fillOval (20, 20, 60, 60);
    }
    if (yellow) {
      g.setColor (Color.yellow);
      g.fillOval (20, 100, 60, 60);
    }
    if (green) {
      g.setColor (Color.green);
      g.fillOval (20, 180, 60, 60);
    }
  }
}
```

17.3 TrafficLight.html

```
<html>
<applet code="TrafficLight.class" width=300 height=400>
  <p>Without Java, you cannot play with the traffic light, sorry.
</applet>
</html>
```

18 Blinklicht

18.1 BlinkLight.java

```
import java.awt.* ;
import java.awt.event.*;
import java.applet.* ;

public class BlinkLight extends Applet
  implements Runnable {

  private boolean runFlag;
  private Image img;
  private Graphics g;
  private Thread t = null;
  private int width, height, diameter, x1, y1;

  public void init() {

    Dimension d = getSize();
    img = createImage(d.width, d.height);
    g=img.getGraphics();
    width = d.width;
    height = d.height;
    if (width < height) {
      diameter = 2 * width / 3;
    }
    else {
      diameter = 2 * height / 3;
    }
    x1 = (width - diameter) / 2;
    y1 = (height - diameter) / 2;
  }

  public void start() {
    if (t == null) {
      t = new Thread (this);
      t.start();
    }
  }
  public void stop() {
    if (t != null) {
      runFlag = false; // t.stop();
      t=null;
    }
  }

  public void run () {
    boolean onOff = false;
    runFlag = true;
    // costant background of image
    g.setColor (Color.black);
    g.fillRect (0, 0, width, height);

    while (runFlag) {

      // prepare image
      if (onOff) {
        g.setColor (Color.yellow);
      }
```

```
      else { // also needed to bypass an error of some 1.1.x compiler optimizations
        g.setColor (Color.black);
      }
      g.fillOval (x1, y1, diameter, diameter);
      // paint the new image (without flicker)
      repaint();

      onOff = ! onOff;
      // wait 1 second
      try { Thread.sleep(1000); }
      catch (InterruptedException e) {}
    }
  }

  public void update (Graphics g) {
    paint(g); // no clear necessary
  }

  public void paint (Graphics g) {
    if (img != null)
      g.drawImage(img,0,0,null);
  }

}
```

18.2 BlinkLight.html

```
<html>
<applet code="BlinkLight.class" width=100 height=300>
  <p>Without Java, you cannot play with the blinking light, sorry.
</applet>
</html>
```

19 zeilenweises Ausdrucken eines Files

19.1 PrintFile.java

```
import java.io.*;

public class PrintFile {
  public static void main (String[] args) {
    String thisLine;
    try {
      BufferedReader in = new BufferedReader (
        new FileReader ("PrintFile.java") );
      while( (thisLine = in.readLine()) != null ) {
        System.out.println(thisLine);
      }
      in.close();
    } catch (Exception e) {
        System.out.println("error " + e);
    }
  }
}
```

20 zeichenweises Kopieren eines Files

20.1 CopyFile.java

```
import java.io.*;

public class CopyFile {
  public static void main (String[] args) {
    int ch;
    try {
      BufferedInputStream in = new BufferedInputStream (
        new FileInputStream ("CopyFile.java") );
      BufferedOutputStream out = new BufferedOutputStream (
        new FileOutputStream ("test.out") );
      while( (ch = in.read()) != -1 ) {
        out.write(ch);
      }
      in.close();
      out.flush();
      out.close();
    } catch (Exception e) {
      System.out.println("error " + e);
    }
  }
}
```

20.2 CopyFast.java

```
import java.io.*;

public class CopyFast {
  public static void main (String[] args) {
    byte buff[] = null;
    int length = 0;
    try {
      File f = new File("CopyFast.java");
      length = (int) f.length();
      buff = new byte[length];
      FileInputStream in = new FileInputStream (f);
      FileOutputStream out = new FileOutputStream ("test.out");
      int copied = 0;
      while (copied < length) {
        copied = copied +
         in.read ( buff, copied, length-copied );
      }
      out.write (buff, 0, length);
      in.close();
      out.flush();
      out.close();
    } catch (Exception e) {
      System.out.println("error " + e);
    }
  }
}
```

20.3 CopyText.java

```
import java.io.*;

public class CopyText {
  public static void main (String[] args) {
    int ch;
    try {
      BufferedReader in = new BufferedReader (
        new FileReader ("CopyText.java") );
      BufferedWriter out = new BufferedWriter (
        new FileWriter ("test.out") );
      while( (ch = in.read()) != -1 ) {
        out.write(ch);
      }
      in.close();
      out.flush();
      out.close();
    } catch (Exception e) {
        System.out.println("error " + e);
    }
  }
}
```

21 Lesen eines Files über das Internet

21.1 PrintURL.java

```
import java.io.*;
import java.net.*;

public class PrintURL {
  public static void main (String[] args) {
    String thisLine;
    String thisURL="http://java.sun.com/products/jdk/1.1/README";

    System.out.println("*** Connecting to " + thisURL);
    System.out.println();
    try {
      BufferedReader in = new BufferedReader (
        new InputStreamReader
          ( (new URL(thisURL)).openStream(), "8859_1" ) );
      while( (thisLine = in.readLine()) != null ) {
        System.out.println(thisLine);
      }
      in.close();
      System.out.println();
      System.out.println("*** Connection finished.");
    } catch (Exception e) {
        System.out.println("error " + e);
    }
  }
}
```

22 Datenbank-Abfragen

22.1 GebJahr.java

```
import java.sql.*;

public class GebJahr {
 public static void main (String[] args) {
  try {
    System.out.println("* Treiber laden");
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    System.out.println("* Datenbank-Verbindung beginnen");
    Connection con = DriverManager.getConnection
        ("jdbc:odbc:TestDB1", "", "");
    con.setReadOnly(true);
    System.out.println("* Statement beginnen");
    Statement stmt = con.createStatement();

    System.out.println("* Abfrage beginnen");
    ResultSet rs = stmt.executeQuery
      ("SELECT Vorname, Geburtsjahr FROM Mitarbeiter ORDER BY Vorname");
    System.out.println("* Ergebnisse anzeigen");
    while (rs.next()) {
      System.out.println( rs.getString(1) + " " + rs.getInt(2) );
    }

    System.out.println("* Statement beenden");
    stmt.close();
    System.out.println("* Datenbank-Verbindung beenden");
    con.close();
  } catch (Exception e) {
      System.out.println("*** Fehler: " + e);
  }
 }
}
```

22.2 Gehalt.java

```
import java.sql.*;

public class Gehalt {
 public static void main (String[] args) {
  double gehalt, summe, durchschnitt;
  int anzahl;
  try {
    System.out.println("* Treiber laden");
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    System.out.println("* Datenbank-Verbindung beginnen");
    Connection con = DriverManager.getConnection
        ("jdbc:odbc:TestDB1", "", "");
    con.setReadOnly(true);
    System.out.println("* Statement beginnen");
    Statement stmt = con.createStatement();

    System.out.println("* Abfrage beginnen");
    ResultSet rs = stmt.executeQuery
      ("SELECT Gehalt FROM Mitarbeiter");
    System.out.println("* Ergebnis berechnen");
    summe = 0.0;
    anzahl = 0;
```

```
    while (rs.next()) {
      gehalt=rs.getDouble(1);
      summe = summe + gehalt;
      anzahl = anzahl + 1;
    }
    durchschnitt = summe / (double) anzahl ;
    System.out.println("Durchscnittsgehalt = " + durchschnitt);

    System.out.println("* Statement beenden");
    stmt.close();
    System.out.println("* Datenbank-Verbindung beenden");
    con.close();
  } catch (Exception e) {
      System.out.println("*** Fehler: " + e);
  }
 }
}
```