Java

Grundlagen

Inhaltsverzeichnis

Änderungen 2.Auflage:
- Fehlerkorrekturen
- Neue Kapitel:
  Collections
  Multimedia

Änderungen 3.Auflage:
- Fehlerkorrekturen
- Anpassungen an Java 2 SDK 1.4.2

1. Einführung

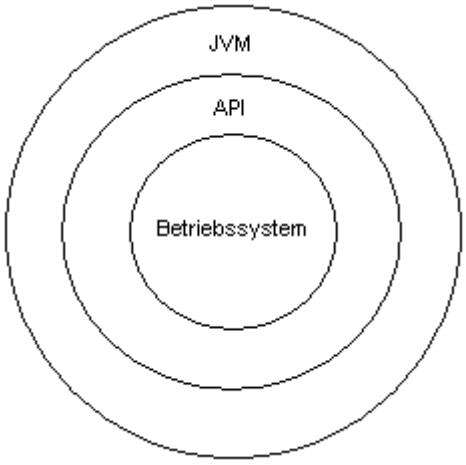
Java ist eine noch sehr junge Programmiersprache des Internets (1995 Vorstellung durch SUN). Hauptanwendungsgebiet sind nach wie vor Applets (kleine Anwendungen, die in Webseiten eingebettet laufen), es werden jedoch auch immer mehr stand-alone Desktop-Applikationen entwickelt. Auf mobilen Geräten sind ebenfalls vermehrt Java-Laufzeitumgebungen verfügbar.

	Internet		
Serverseite:	----------	----------	Clientseite:
Webserver Servlets Serverprogrammierung	Webseiten ⇒	Applets ⇒	Webbrowser JAVA-VM in: Workstations Organizer Handys

Anforderungen:
- Die compilierten Programme sollen möglichst klein sein (rasche Übertraungszeiten im Internet)
- Einfache Programmierung interaktiver Internetanwendungen
- Plattformunabhängigkeit

Realisierung:
Durch Java-Interpreter inklusive Laufzeitsystem (Java Virtual Machine JVM) auf der jeweiligen Zielmaschine (im Webbrowser integriert, als plug-in oder stand-alone). Beim Übersetzen des Java-Quellcodes entsteht plattformunabhängiger *Java-Bytecode* (sehr klein, meist nur einige kB!), dieser wird interpretiert.
Das grafische User-Interface wird von der Wirtsmaschine realisert (java.awt.*, Abstract Windows Toolkit) oder aber von der Java-Maschine selbst (Swing-Klassen, einheitliches Look&Feel, aber langsamer).

Prinzip:



Die Java Virtual Machine stellt eine einheitliche Schnittstelle für alle Plattformen dar!

Vorteile von Java:
- Auf allen bedeutenden Betriebssystemen unterstützt (de facto Plattformunabhängigkeit)
- Einfachere und robustere Sprache als etwa C++
- Hervorragend für Internetanwendungen geeignet (Netzwerkfähigkeit, Sicherheit, Webbrowser,...)
- Mobile Geräte bieten verstärkt Java-Laufzeitumgebungen (Handys, PDAs,...)
- Java verbreitet sich sehr schnell und besitzt zur Zeit ca. den gleichen Marktanteil wie C++ (ca.10-15%)

Nachteile von Java:
- Laufzeitverhalten, Java-Programme sind durchschnittlich ca. 2-3 mal langsamer als C++-Programme
- Microsoft bekämpft das Java-Konzept (.NET Framework, C#)

2. Wesentliche Unterschiede zwischen C/C++ und Java

Datentypen:
Elementare Datentypen haben in Java auf allen Plattformen und unter allen Betriebssystemen fixe Größen. Alle numerischen Datentypen sind vorzeichenbehaftet, es gibt kein *unsigned* wie in C/C++.
Übersicht:

Typ	Bytes	Kleinster Wert	Größter Wert
char	2 (Unicode!)	0	65535
byte	1	-128	127
short	2	-32.768	32767
int	4	-2.147.483.648	2.147.483.647
long	8	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
float	4	1.4e-45	3.4028235e+38
double	8	5e-234	1.7976931348623157e+308

Zusätzlich gibt es noch den Datentyp *boolean* mit den möglichen Werten *true* und *false*.
In Java gibt es weder *struct*, *union*, noch *typedef*, die Zusammenfassung von Daten wird ausschließlich durch Klassendefinitionen erreicht.

In Java gibt es keinen expliziten Zeigertyp!!! Statt Zeiger werden Referenzvariablen verwendet (es sind im Gegensatz zu C++ freie Referenzen möglich!).

C++:

```
int& r;        ⇐ Freie Referenz, nicht möglich, die Referenz muss sofort auf ein Objekt verweisen!
int x = 3;     ⇐ Objekt x
int& r = x;    ⇐ OK, die Referenz r verweist auf das Objekt x!
```

JAVA:

```
public class Auto { ... }
Auto meinAuto;  ⇐ meinAuto ist eine Referenz auf ein Objekt der Klasse Auto
                  Es wurde noch kein Objekt erzeugt, dies muss dynamisch erfolgen!!!
```

Casting bzw. Typkonvertierungen:
Werden in Java wesentlich stärker kontrolliert, eine automatische Konvertierung erfolgt nur, wenn kein Informationsverlust damit verbunden ist.

Konvertieren von elementaren Datentypen
Erweiternde Konvertierung (kein explizites Casting erforderlich!):
byte ⇒ short ⇒ int ⇒ long ⇒ float ⇒ double
char ⇒
Bei Konvertierung von ganzzahligen Datentypen in Fließkommatypen Genauigkeitsverlust möglich!
Einschränkende Konvertierung (in umgekehrter Pfeilrichtung): Explizites Casting notwendig!

Konvertieren von Objekten
Objekte können in andere Objekte konvertiert werden, falls diese durch Vererbung miteinander verbunden sind (kein explizites Casting nötig). Als erweiternde Konvertierung wird vor allem die Umwandlung eines Objektes in ein Objekt einer Basisklasse angesehen (IST EIN – Beziehung!!!)

Konvertieren von Primitivtypen in Objekte und umgekehrt
Geht in Java überhaupt nicht! Dazu gibt es sogenannte Hüllklassen (wrapper classes) für die elementaren Datentypen: Elementare Datentypen werden in Objekte gekapselt, mit entsprechenden Methoden lassen sich daraus die elementaren Datentypen wieder herausholen, zB.:

```
Integer intObjekt = new Integer("36");
int i = intObjekt.intValue();
```

Operatoren:
In Java gibt es kein Operator-Overloading. Für vordefinierte Klassen des SDK existieren sehr wohl zahlreiche überladene Operatoren. Beispiel: + Operator der Klassen *String* bzw. *StringBuffer*.
Im wesentlichen stehen die selben Operatoren wie in C/C++ zur Verfügung. Der Operator >>> erzeugt eine vorzeichenlose (logische) Bitverschiebung nach rechts, es werden Nullen nachgeschoben.
Den Adreßoperator wie in C/C++ gibt es nicht. Bei Bedarf sind statt dessen Hüllklassenobjekte zu verwenden.

Parameterübergabe an Methoden:
In Java sehr einfach geregelt:
Elementare Datentypen: **call by value**, d.h. eine Kopie wird erstellt und der Wert übergeben.
Alle Objekte: **call by reference**, d.h. die Adresse des Objekts wird übergeben.
Im Gegensatz zu C/C++ gibt es keine Defaultparameter, Function Overloading ist sehr wohl möglich!

Kontrollstrukturen:
Entsprechen im wesentlichen den bekannten Kontrollstrukturen aus C/C++:
if( ), while( ) { ... }, do { ... }while( ), for( ) { ... }, switch( ) { case ... }, ...
Bei Vergleichen müssen vollständige boolsche Ausdrücke stehen, zB.: `if(i!= 0) statt if(i) ...`

Speichermanagement:
Dieses erfolgt in Java automatisch: Speicher wird bei Erstellung eines Objekts automatisch zugewiesen, ein im Hintergrund mit geringer Priorität laufender "Garbage Collector" gibt diesen wieder frei, sobald das Objekt nicht mehr benutzt wird (es existiert keine Referenz mehr darauf). Aus diesem Grund benötigt man in Java keine Speicherverwaltungsfunktionen wie in C/C++ (malloc( ), free( ) ...).
Alle Objekte (außer Variablen eines elementaren Datentyps) müssen dynamisch erzeugt werden!

```
public class Auto { ... }
Auto meinAuto;  ⇐ meinAuto ist eine Referenz auf ein Objekt der Klasse Auto
                  Es wurde kein Objekt erzeugt, dies muss dynamisch erfolgen!!!
meinAuto = new Auto();          ⇐ Jetzt wird dynamisch ein Objekt des Typs Auto erstellt
```

Arrays:
Felder sind in Java spezielle Klassenobjekte, bei denen die Arraygrenzen zur Laufzeit kontrolliert werden. Elementare Datentypen werden direkt im Array gespeichert, ein Array von Objekten enthält nur die Referenzen (Verweise) auf die Objekte.
Von den Arrays ist die Klasse ***Vektor*** zu unterscheiden, diese stellt eine Liste von gereihten Objekten (auch verschiedenen Typs) dar.

Strings:
Für das Arbeiten mit Zeichenketten hält Java zwei Klassen bereit.
String für Textkonstanten (die Länge muss nicht konstant sein, Operator + zum Verketten verfügbar)
StringBuffer für veränderliche Zeichenketten
Strings sind eigenständige Objekte, die mittels Stringmethoden als Einheit behandelt werden, sie werden nicht mit einem \0 beendet.

Objektorientierte Programmierung:
Java-Klassen basieren auf der Einfachvererbung, eine gewisse Mehrfachvererbung ist durch sogenannte ***Schnittstellen (Interfaces)*** möglich. Alle Funktionen werden als Methoden (= Funktionen, die fix mit einem Objekt verbunden sind) implementiert, sämtliche Methoden sind automatisch *virtual*!
Eine Sonderstellung besitzen die statischen Methoden (Klassenmethoden), sie entsprechen den normalen globalen Funktionen in C/C++. Sämtliche Methoden werden innerhalb der Klassendefinition implementiert, d.h. es existiert keine Aufteilung in Klassen-definition und Klassenimplementation. Die Zugriffsspezifizierer entsprechen denen in C++ (public, protected,...) und sind explizit vor jedem Klassenmember (Variable oder Methode) zu spezifizieren.

Sonstiges:
Java hat keinen Präprozessor und damit auch kein #define und keine Makros. Auch Konstanten im eigentlichen Sinn gibt es nicht, eine Abhilfe bietet der Modifier ***final***.

WESENTLICH:
In Java muss man sich von der Idee einer einzigen einheitlichen Datei (*.exe) trennen: Eine Anwendung besteht in der Regel aus vielen *.class-Dateien. Wird eine Klasse, die bereits vorhanden ist, benötigt, so kann sie auch schon benutzt werden, ohne dass ein Projekt oder ähnliches dafür anzulegen ist. Wichtig ist nur, dass die erforderlichen Klassen auch gefunden werden.

Die Mechanismen zum Finden von Klassen sind:

classpath: Dieser gibt den Suchpfad zu möglichen *.class-Dateien an. Diese können durchaus auch in einer komprimierten Datei (*.jar...Java Archiv) enthalten sein.

import: Damit ist die Suche in Unterverzeichnissen (auch in Archiv-Dateien) möglich. Das Gruppieren von *.class-Dateien in Unterverzeichnissen erlaubt eine Systematisierung.

Mit plattformspezifischen (native) Compilern ist es möglich, wieder allein lauffähige Programme (*.exe) zu erhalten, diese Vorgangsweise entspricht aber nicht unbedingt dem Wesen von Java.

3. Entwicklungsumgebungen

3.1 Plattformen, Versionen

Seit der Version 1.2 firmieren alle Java-Versionen unter dem offiziellen Oberbegriff *Java 2 Platform*. Seit einiger Zeit werden zudem die Entwicklungssysteme von SUN nicht mehr JDK, sondern *Java 2 SDK* genannt (SDK steht für *Software Development Kit*).
Auf die Unterschiede zwischen den drei unterschiedlichen Java 2-Plattformen, der *J2SE* (*Java 2 Standard Edition*), der *J2ME* (*Java 2 Micro Edition*) und der *J2EE* (*Java 2 Enterprise Edition*), wird hier nicht weiter eingegangen. Wenn nicht anders erwähnt, beziehen sich alle Ausführungen auf die Standard Edition.
Die Micro Edition beschreibt einen eingeschränkten Sprachstandard, mit dem Java auf Geräten wie Mobiltelefonen oder PDAs betrieben werden kann, und die Enterprise Edition enthält eine Reihe zusätzlicher Elemente, die das Entwickeln verteilter, unternehmensweiter Applikationen unterstützen.

3.2 Java 2 SDK (Java 2 Software Development Kit)

Um Java-Programme erstellen zu können, benötigt man im Prinzip nur das Java 2 SDK, welches von Sun für fast alle namhaften Betriebssysteme frei im Internet verfügbar ist (Bsp. j2sdk-1_4_2-windows-i586.exe für Windows). Die aktuelle Version des SDK ist 1.4.2. Für diverse Updates, Dokumentationen, usw. empfiehlt es sich öfter auf die Homepage von Sun zum Thema Java zu schauen (http://java.sun.com).
Die Installation erfolgt vollautomatisch, es empfiehlt sich, das vorgeschlagene Installationsverzeichnis c:\j2sdk1.4.2 zu verwenden. Es sollten ruhig alle Komponenten für die Installation ausgewählt werden, der Festplattenbedarf beträgt ca. 300MB.
Mit dem SDK werden folgende Softwarekomponenten installiert:
- Die JVM für das entsprechende Betriebssystem
- Sämtliche Java-Tools (Compiler, Interpreter, Appletviewer,...)
- Das JRE (Java Runtime Environment), dieses kann auch stand-alone installiert werden
- Zahlreiche Demos zu allen Bereichen

Für das Compilieren der Programme aus der Kommandozeile wie im nächsten Abschnitt erläutert, reicht die folgende Einstellung des Suchpfades auf das *bin*-Unterverzeichnis im SDK-Installationsverzeichnis, dieses enthält neben vielen anderen Tools den Java-Compiler und -Interpreter:
set path = %path%; c:\j2sdk1.4.2\bin
Die Pfadeinstellung sollte in der Systemsteuerung für die Umgebungsvariablen vorgenommen werden.
Die Dokumentation zum Java 2 SDK ist als eigener Download verfügbar, für die Version 1.3 gibt es auch eine Windows-Hilfedatei (SDK131.zip). Die Hilfe ist für eine vernünftige Programmentwicklung absolut unerlässlich!

3.3 IDEs

Für die Erstellung größerer Java-Programme ist eine vernünftige integrierte Entwicklungsumgebung (IDE...Integrated Development Environment) unerlässlich.
Eine einfache, aber vollkommen ausreichende IDE für kleinere Projekte bzw. zum Erlernen von Java stellt *RealJava* dar, diese kann unter www.realj.com für nicht-kommerzielle kostenlos heruntergeladen werden (realjn.exe, ca. 470kB). Eine weitere einfache IDE ist KAWA (www.kawa.com).
Im professionellen Bereich werden sehr umfangreiche IDEs wie zB. der JBuilder von Borland verwendet.

4. Eine erste Java-Applikation

ANMERKUNG
Sehr zu achten ist auf die Groß- und Kleinschreibung, sowohl im Quelltext als auch von Dateinamen, da Java in dieser Beziehung sehr kleinlich ist!

Schritt 1 – Erstellen der Quelltextdatei

In einem eigenen Verzeichnis wird folgende Quelltextdatei als Textfile mit dem Namen *Hello.java* erstellt:

```
/* Ein erstes Java-Programm */
public class Hello
{
  public static void main(String[] args)
  {
    System.out.println("Hello World!");
  }
}
```

Schritt 2 – Compilieren der Quelltextdatei

Dazu öffnet man eine Shell (MS-DOS-Eingabeaufforderung unter Windows) geht in das Verzeichnis mit der Quelltextdatei. Dort wird der Compiler mit folgendem Kommando gestartet:

```
>javac Hello.java
```

Falls die Datei erfolgreich übersetzt werden konnte, erhalten Sie keine Ausgabe auf den Bildschirm, ansonsten werden entsprechende Fehlermeldungen mit den zugehörigen Zeilennummern angezeigt.

Bei erfolgreicher Übersetzung erstellt der Compiler die Datei *Hello.class*, diese enthält den plattformunabhängigen sog. Java-Bytecode, welcher durch das Java-Laufzeitsystem interpretiert werden kann.

Anmerkung: Der Compiler erzeugt den Dateinamen der Bytecodedatei nicht aus dem Dateinamen, sondern aus dem Namen der Klasse + .class. Befinden sich mehrere Klassendefinitionen in einer Quelltextdatei, so wird für jede Klasse eine eigene *.class-Datei erzeugt.
Der Klassenname der Klasse, die als public deklariert ist, muss mit dem Namen der Quelltextdatei übereinstimmen, d.h. der Quelltext jeder als public deklarierten Klasse muss sich in einer eigenen Datei befinden.

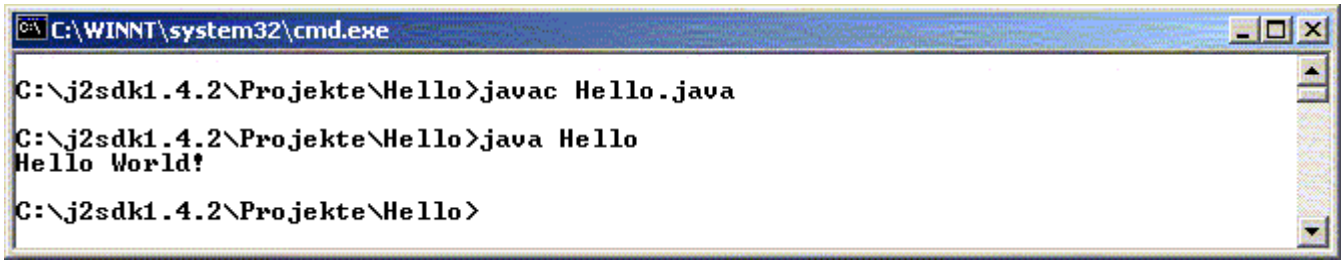
Schritt 3 – Starten der Applikation

Die Applikation wird durch folgendes Kommando gestartet:

```
>java Hello
```

Bei korrekter Ausführung erscheint nun in der nächsten Zeile des Kommandozeileninterpreters die Ausgabe:

```
Hello World!
```



Besprechung der Hello World Applikation:

Die Funktion main ( )

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

Diese stellt wie in C/C++ den Einstiegspunkt der Applikation dar. In Java sieht man sich allerdings mit der Tatsache konfrontiert, dass es ausschließlich Klassen gibt! Das bedeutet unter anderem, dass es keine direkten globalen Funktionen gibt, so wie aus C/C++ bekannt. Die Funktion *main( )* muss daher innerhalb einer Klasse definiert werden. Dies ist mittels des Schlüsselwortes *static* möglich, welches spezifiziert, dass es sich um eine Funktion handelt, zu deren Aufruf ich kein Objekt dieser Klasse benötige, dies entspricht einer globalen Funktion.
Methoden oder Variablen einer Klasse, welche mit dem Schlüsselwort *static* definiert sind, werden als *Klassenmethoden* bzw. *Klassenvariablen* bezeichnet. Solche Methoden bzw. Variablen existieren pro Klasse nur einmal, sind also nicht an die Existenz von Objekten der Klasse gebunden!!!

Kommandozeilenparameter

Wie in C/C++ können dem Programm über einen Aufruf aus der Kommandozeile Parameter übergeben werden.

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

Die Parameterübergabe erfolgt über das Stringarray *args*, die Anzahl der übergebenen Parameter kann über *args.length* ermittelt werden.
arg[0] ist der erste über die Kommandozeile übergebene Parameter als String
arg[1] ist der zweite über die Kommandozeile übergebene Parameter als String

Eine Umwandlung eines Strings in eine Zahl kann zB. durch die Klassenmethode *parseInt( )* der Hüllklasse *Integer* erfolgen.
Beispiel: *int i = Integer.parseInt(args[0]);*

Frage: Welcher wesentliche Unterschied besteht zwischen args[0] in Java und in C/C++ ?

Einfache Ein/Ausgaben

Für die ersten Schritte in einer neuen Sprache benötigt man immer auch I/O-Routinen, um einfache Ein- und Ausgaben vornehmen zu können.
Folgende Ein/Ausgabeströme(Streams) stehen immer zur Verfügung:

System.out	Standard-Output-Stream für die Ausgabe auf der Systemkonsole (vgl. *cout* in C++)
System.in	Standard-Input-Stream für die Eingabe von der Systemkonsole (vgl. *cin* in C++)
System.err	Standard-Error-Stream für die Ausgabe von Fehlermeldungen (vgl. *cerr* in C++)

Mit Hilfe des Kommandos *System.out.println( )* können einfache Ausgaben auf den Bildschirm geschrieben werden. Nach jedem Aufruf wird eine Zeilenschaltung ausgegeben. Mit *System.out.print( )* kann diese auch unterdrückt werden. Beide Methoden erwarten ein einziges Argument, das von beliebigem Typ sein kann, *print( )* und *println( )* sind aber nicht so flexibel wie etwa *printf( )* in C. Mit Hilfe des Plus-Operators können aber immerhin Zeichenketten und numerische Argumente miteinander verknüpft werden, so dass man neben Text auch Zahlen ausgeben kann:

```
System.out.println("1+2=" + (1+2));
```

Leider ist es etwas komplizierter, Daten zeichenweise von der Tastatur zu lesen. Zwar steht der vordefinierte Eingabe-Stream *System.in* zur Verfügung. Er ist aber nicht in der Lage, die eingelesenen Zeichen in primitive Datentypen zu konvertieren. Statt dessen muss zunächst eine Instanz der Klasse *InputStreamReader* und daraus ein *BufferedReader* erzeugt werden. Dieser kann dann dazu verwendet werden, die Eingabe zeilenweise zu lesen und das Ergebnis in einen primitiven Typ umzuwandeln (mehr dazu findet man im Kapitel über Streams).

Beispiel:

```
import java.io.*;

public class Eingabe
{
  public static void main(String[] args) throws IOException
  {
    int a, b, c;
    BufferedReader din = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("Bitte a eingeben: ");
    a = Integer.parseInt(din.readLine());
    System.out.println("Bitte b eingeben: ");
    b = Integer.parseInt(din.readLine());
    c = a + b;
    System.out.println("a+b="+c);
  }
}
```

Namenskonventionen
Wie in allen Programmiersprachen, gibt es auch in Java Konventionen für die Vergabe von Namen. Sie sind zwar nicht zwingend erforderlich, erleichtern aber das Verständnis der Quelltexte ungemein und sollten daher unbedingt eingehalten werden. Die wichtigsten sind:

- Klassennamen beginnen stets mit einem Großbuchstaben. Beispiele sind *String*, *Vector* oder *Hello*. Besteht ein Klassenname aus mehreren Silben, können zur Steigerung der Übersichtlichkeit auch die Folgesilben mit einem Großbuchstaben beginnen. Beispiele dafür wären *HelloWorld*, *KeyAdapter* oder *NoSuchMethodException*. Klassennamen, die nur aus Großbuchstaben bestehen, sind unüblich.
- Methodennamen beginnen mit einem Kleinbuchstaben. Haben sie mehrere Silben, ist die erste oft ein Verb. Weitere beginnen mit einem Großbuchstaben, zB.: *println*, *hasMoreElements* oder *isEnabled*.
- Paketnamen bestehen ausschließlich aus Kleinbuchstaben. Beispiele sind *java.lang*, *javax.swing.event* oder *com.solution42.util* (mehrteilige Paketnamen werden durch Punkte separiert).
- Für Variablennamen gelten dieselben Konventionen wie für Methoden. Es ist unüblich, Membervariablen mit einem Präfix wie z.B. "m_" zu versehen. Auch die Verwendung der ungarischen Notation, bei der Variablen datentypbezogene Namenspräfixe erhalten, ist in Java nicht üblich.

Übungsaufgaben:
1. Erstellen, Übersetzen und Starten Sie die Hello World Applikation.
2. Schreiben Sie ein Programm zur Überprüfung, ob eine Zahl eine Primzahl ist.
3. Schreiben Sie ein Programm, welches alle Primzahlen zwischen 2 bestimmten Werten ausgibt.
4. Schreiben Sie ein Programm zur Ermittlung der Fakultät einer Zahl.
5. Versuchen Sie einige Typkonvertierungen durchzuführen.

Versuchen Sie alle Aufgaben auf 2 Arten zu lösen:
a. Verwenden Sie nur eine Klasse
b. Erstellen Sie für die Berechnungen eine zweite Klasse

Anmerkung: Verwenden Sie intensiv die Hilfe des SDK, sehen Sie sich als Beispiel die Klassen *String*, *StringBuffer* sowie *Integer* an.

5. Klassenbibliotheken des SDK, Packages

Zusammengehörige Java-Klassen können in sogenannte *Packages* zusammengefaßt werden. Das SDK enthält viele Packages mit zahlreichen grundlegenden Klassen.

5.1 Übersicht über die am häufigsten verwendeten Packages:

Package	Beschreibung
java.applet	Enthält die Klasse *Applet* und 3 Schnittstellen. Diese dienen als Schnittstelle zwischen einem Applet und dem Browser oder Appletviewer.
java.awt	Abstract Window Toolkit GUI-Programmierung über die Java-Peer-Klassen. Enthält die GUI-Klassen sowie Klassen zum Laden von Grafiken, Drucken und Ausgeben von grafischen Elementen.
java.awt.datatransfer	Unterstützung des Clipboards mit cut/copy/paste.
java.awt.event	Behandlung von Ereignissen von AWT-Komponenten nach dem neuen Event-Modell.
java.awt.image	Klassen- und Schnittstellensammlung zur Bildbearbeitung.
java.beans	Komponentenmodell von Javasoft, ähnlich wie ActiveX von Microsoft, jedoch plattformneutral.
java.io	Lesen und Schreiben von Daten.
java.lang	Die Klassen dieses Packages werden implizit in jede Java-Applikation und in jedes Applet eingebunden. Enthalten sind die Klasse *Object*, die die Wurzel (Basisklasse) aller Klassen ist, und Klassen zum Erstellen und Verwalten von Threads sowie die erweiterten Datentypen.
java.math	Zwei Klassen mit erweiterten Möglichkeiten zur Berechnung von Zahlen. Normalerweise berechnet man Zahlen mit Methoden aus *java.lang.math*.
java.net	Bietet die Möglichkeit zur Erstellung von Netzwerk-Clients und –Servern mit unterschiedlichen Protokollen.
java.rmi java.rmi.dgc java.rmi.registry java.rmi.server	Die *rmi*-Packages ermöglichen Methodenaufrufe entfernter Objekte und damit die Erstellung verteilter Anwendungen (Remote Method Invocation).
java.security java.security.acl java.security.cert java.security.interfaces java.security.spec	Die *Security*-Packages enthalten Klassen und Schnittstellen, mit deren Hilfe sichere Datenübertragung ermöglicht werden soll.
java.sql	Ermöglicht den Zugriff auf relationale Datenbanken mit SQL-Befehlen.
java.text	Bestimmung und Formatieren von Zahlen, Zahlen-, Währungs- und Datumsformaten sowie Ermittlung von Textbegrenzungen.
java.util	Enthält Tools zum Umwandeln von Zahlen, Währungs- und Datumsformaten, zum Erstellen von Datenstrukturen und die Basisklassen zum Event-Handling.
java.util.zip java.util.jar	Ermöglicht das Zusammenfassen, Komprimieren und Dekomprimieren von Dateien zur beschleunigten Übertragung von Dateien über das WWW.
javax.swing javax.text javax.table javax.tree	All-Java GUI Komponenten. Umfangreiche GUI-Bibliothek, die alles bietet, was von einer modernen Oberfläche erwartet wird.

Neben den Standardklassenbibliotheken findet man auf der bereits erwähnten Homepage von Sun zum Thema Java (http://java.sun.com) noch zusätzliche APIs zum kostenlosen Download.

5.2 Umgebungsvariablen

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 nach Klassen und Packages gesucht werden soll. Im allgemeinen gibt man hier den Punkt (für das jeweils aktuelle Directory) und das Start-Directory der im SDK enthaltenen Packages an. Bei neueren SDK-Versionen ist das jedoch der Standard und der CLASSPATH soll daher *gar nicht* gesetzt werden.
Die Variable JAVA_HOME gibt an, in welchem Directory die Komponenten des SDK zu finden sind. Die explizite Angabe ist ebenfalls meistens *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.
Beispiel für die eventuelle Definition dieser Environment-Variablen (die Details hängen von der jeweiligen Software-Version und deren Konfiguration und Installation ab):
Windows, in der Systemsteuerung:
set path=%PATH%;C:\j2sdk1.4.2\bin (bereits erläutert)
set classpath=.;C:\j2sdk1.4.2\jre\lib\rt.jar;
set java_home=C:\j2sdk1.4.2

5.3 Importieren von Packages

Um Klassen aus den Packages des SDK im Quellcode verwenden zu können, müssen diese importiert werden, dies erfolgt mittels der *import*-Anweisung:

```
import java.applet.*;

public class MyApplet extends Applet
{
   ...
}
```

Der Stern bedeutet, dass alle Klassen und Schnittstellen des betreffenden Packages importiert werden sollen. Bei dem Package *java.applet* macht das durchaus Sinn, da dieses nur aus einer einzigen Klasse und 3 Schnittstellen besteht.

Man kann auch nur einzelne Klassen oder Schnittstellen importieren:

```
import java.applet.Applet;
import java.awt.Button;
```

Es empfiehlt sich jedoch, die Packages des SDK komplett zu importieren. Sinnvoll ist das einzelne Importieren von Klassen eigentlich nur, wenn mehrere Bibliotheken gleichnamige Klassen aufweisen.

5.3 Eigene Packages erstellen

Ein Package, das man selbst anlegt, kann man sich einfach wie ein Unterverzeichnis vorstellen.
Will man zum Beispiel eine Sammlung von Klassen für komplizierte mathematische Berechnungen programmieren, so könnte man diese in einem Unterverzeichnis *Superhirn* erstellen. Um eine Klasse in diesem Unterverzeichnis dem Package *Superhirn* zuzuordnen, fügt man zu Beginn des Quellcodes folgende Zeile ein, vor dieser Zeile dürfen ausschließlich Kommentare stehen:

```
package Superhirn;

public class Berechne {  ... }
```

Wenn man nun aus einer anderen Klasse aus eine Methode der Klasse *Berechne* aufrufen will, müssen folgende Vorrausetzungen erfüllt sein:
- Die Klasse *Berechne* muss mittels
  `import Superhirn.*;` oder `import Superhirn.Berechne;`
  importiert werden.
- Es muss eine Instanz der Klasse *Berechne* erstellt werden.
- Die Methode wird mit Instanzname.Methodenname(Parameter) aufgerufen.

Zugriff auf ein Package

Es gibt 2 Möglichkeiten, um einer Klasse Zugriff auf ein selbst erstelltes Package zu ermöglichen:
- *superhirn* ist ein Unterverzeichnis des Verzeichnisses, in dem die Klasse mit Zugriffswunsch liegt.
- *Superhirn* ist ein Unterverzeichnis eines Verzeichnisses, auf das im CLASSPATH verwiesen wird.

Das public-Attribut

Es gibt noch eine wichtige Besonderheit bei der Deklaration von Klassen, die von anderen Klassen verwendet werden sollen. Damit eine Klasse A eine andere Klasse B einbinden darf, muss eine der beiden folgenden Bedingungen erfüllt sein:
- Entweder gehören A und B zum selben Package oder
- Die Klasse B wurde als *public* deklariert.

Übungsaufgaben:

Erstellen Sie die Klassen für die Übungsaufgaben aus 4. in eigenen Packages.

6. Strings

In Java gibt es 3 Klassen für Strings:
- *String* für Zeichenkettenkonstanten
- *StringBuffer* für veränderliche Zeichenketten
- *StringTokenizer* zum Zerlegen von Zeichenketten in einzelne Teilketten

Details zu den Klassen können in der SDK-Hilfe nachgesehen werden. Um einen Überblick zu bekommen, wie die Hilfe zu einer Klasse aussieht, wurde die Übersicht der Klasse *StringBuffer* ausgedruckt.

Einige einfache Beispiele:

Anhängen an einen String
```
StringBuffer text = new StringBuffer();   // Text ist noch leer
for(int i = 1; i < 10; i++)
{
  text.append(i + ".Zeile\n");            // Die append-Methode sorgt für die
                                          // Bereitstellung von Speicher
}
```

Vergleichen von Strings
```
String text1 = new String("Hallo“);       // Zwei identische Texte
String text2 = new String("Hallo");
String text3 = text1;                     // Referenz text3 verweist auf text1

// ACHTUNG:
if(text1 == text2) ...              // Vergleicht die Referenzen (ungleich!)

if(text1 == text3) ...              // Vergleicht die Referenzen (gleich!)

if(text1.equals(text2)) ...         // Vergleicht die Strings (gleich!)
```

Zugriff auf einzelne Zeichen
```
StringBuffer text = new StringBuffer("Hallo“);

for(int i = 0; i < text.length(); i++)
{
  if(text.charAt(i) == 'A')         // Ist das aktuelle Zeichen ein A ?
  {
     text.setCharAt(i, 'A');        // Wenn ja, dann ersetze es durch ein kleines a
  }
}
```

Übungsaufgaben:
- Testen Sie die angeführten Beispiele durch.
- Schreiben Sie ein Programm, welches eine URL in seine Bestandteile (Protokoll,Host,Pfad,Datei) zerlegt.
  Bsp: http://www.htl.at/lehrer/waser/index.html
  ⇒ Protokoll *http*
  Host *www.htl.at*
  Pfad */lehrer/waser/*
  Datei *index.html*
  Die Eingabe soll sowohl als Kommandozeilenparameter als auch über Tastatur möglich sein.
- Prüfe einen eingegebenen Text auf Palindromeigenschaft (von links und von rechts gleich lesbar, z.B. "Ein Neger mit Gazelle zagt im Regen nie").
- Schreiben Sie ein einfaches Programm für die Caesar-Verschlüsselung:
  Jeder Buchstabe in einem Text wird durch seinen k-ten Nachfolger im Alphabet ersetzt.

7. Arrays

Arrays in Java unterscheiden sich dadurch von Arrays in anderen Programmiersprachen, dass sie Objekte sind. Obwohl dieser Umstand in vielen Fällen vernachlässigt werden kann, bedeutet er dennoch:
- Array-Variablen sind Referenzen.
- Arrays besitzen Methoden und Instanz-Variablen.
- Arrays werden dynamisch zur Laufzeit erstellt.

Dennoch bleibt ein Array immer eine (möglicherweise mehrdimensionale) Reihung von Elementen eines festen Grundtyps. Arrays in Java sind semidynamisch, d.h. ihre Größe kann zur Laufzeit festgelegt, später aber nicht mehr verändert werden.

7.1 Deklaration und Initialisierung

Die Deklaration eines Arrays in Java erfolgt in zwei Schritten:
- Deklaration einer Array-Variablen
- Erzeugen eines Arrays und Zuweisung an die Array-Variable

Die Deklaration eines Arrays entspricht syntaktisch der einer einfachen Variablen, mit dem Unterschied, dass an den Typnamen eckige Klammern angehängt werden:

```
int[] a;
double[] b;
boolean[] c;
```

Wahlweise können die eckigen Klammern auch hinter den Variablennamen geschrieben werden, aber das ist ein Tribut an die Kompatibilität zu C/C++ und sollte in neuen Java-Programmen vermieden werden.

Zum Zeitpunkt der Deklaration wird noch nicht festgelegt, wie viele Elemente das Array haben soll. Dies geschieht erst später bei seiner Initialisierung, die mit Hilfe des new-Operators oder durch Zuweisung eines Array-Literals ausgeführt wird. Sollen also beispielsweise die oben deklarierten Arrays 5, 10 und 15 Elemente haben, würde man das Beispiel wie folgt erweitern:

```
a = new int[5];
b = new double[10];
c = new boolean[15];
```

Ist bereits zum Deklarationszeitpunkt klar, wie viele Elemente das Array haben soll, können Deklaration und Initialisierung zusammen geschrieben werden:

```
int[] a = new int[5];
double[] b = new double[10];
boolean[] c = new boolean[15];
```

Alternativ zur Verwendung des new-Operators kann ein Array auch literal initialisiert werden. Dazu werden die Elemente des Arrays in geschweifte Klammern gesetzt und nach einem Zuweisungsoperator zur Initialisierung verwendet. Die Größe des Arrays ergibt sich aus der Anzahl der zugewiesenen Elemente:

```
int[] x = {1,2,3,4,5};
boolean[] y = {true, true};
```

Das Beispiel generiert ein *int*-Array x mit fünf Elementen und ein *boolean*-Array y mit zwei Elementen. Anders als bei der expliziten Initialisierung mit new muss die Initialisierung in diesem Fall unmittelbar bei der Deklaration erfolgen.

7.2 Zugriff auf Array-Elemente

Dieser erfolgt wie aus C/C++ bekannt über einen Index zwischen 0 und n-1. Der Array-Index muss vom Typ *int* sein. Aufgrund der vom Compiler automatisch vorgenommenen Typkonvertierungen sind auch *short*, *byte* und *char* zulässig. Jedes Array hat eine Instanzvariable *length*, die die Anzahl seiner Elemente angibt. Indexausdrücke werden im Unterschied zu C/C++ vom Laufzeitsystem auf Einhaltung der Array-Grenzen geprüft! Sie müssen größer gleich 0 und kleiner als *length* sein.

```
int[] prim = new int[3];
boolean[] b = {true,false};
prim[0] = 2;
prim[1] = 3;
prim[2] = 5;
System.out.println("prim hat "+prim.length+" Elemente");
System.out.println("b hat "+b.length+" Elemente");
System.out.println(prim[0]);
System.out.println(prim[1]);
System.out.println(prim[2]);
System.out.println(b[0]);
System.out.println(b[1]);
```

Die Ausgabe des Programms ist:

prim hat 3 Elemente
b hat 2 Elemente
2
3
5
true
false

7.3 Mehrdimensionale Arrays

Mehrdimensionale Arrays werden erzeugt, indem zwei oder mehr Paare eckiger Klammern bei der Deklaration angegeben werden. Mehrdimensionale Arrays werden als Arrays von Arrays angelegt. Die Initialisierung erfolgt analog zu eindimensionalen Arrays durch Angabe der Anzahl der Elemente je Dimension.
Der Zugriff auf mehrdimensionale Arrays geschieht durch Angabe aller erforderlichen Indizes, jeweils in eigenen eckigen Klammern. Auch bei mehrdimensionalen Arrays kann eine literale Initialisierung durch Schachtelung der Initialisierungssequenzen erreicht werden. Das folgende Beispiel erzeugt ein Array der Größe 2 * 3 und gibt dessen Elemente aus:

```
int[][] a = new int[2][3];
a[0][0] = 1;
a[0][1] = 2;
a[0][2] = 3;
a[1][0] = 4;
a[1][1] = 5;
a[1][2] = 6;
System.out.println(""+a[0][0]+a[0][1]+a[0][2]);
System.out.println(""+a[1][0]+a[1][1]+a[1][2]);
```

7.4 Vergleichen und Kopieren von Arrays

Beispiele:

```
int feld1[] = new int[2];
int feld2[] = new int[2];
feld1[0] = 1;
feld1[0] = 2;

if(feld1 == feld2) ...   // Vergleicht die Referenzen (ungleich!)
feld2 = feld1;           // Der bei der Initialisierung allokierte Speicher für
                         // feld2 wird freigegeben, feld2 enthält nun die gleiche
                         // Referenz wie feld1 ⇒ Beide Variablen verweisen auf
                         // das gleiche Array! Es wird keine Kopie erzeugt!
                         // Der obige Vergleich der Referenzen liefert nun true

// Das Systemobjekt stellt eine Methode zum Kopieren von Feldern zur Verfügung
// Hierbei wird eine vollständige Kopie des Arrays, auf das feld1 verweist, erzeugt!
System.arraycopy(feld1, 0, feld2, 0, 2); // Siehe SDK-Hilfe
```

Zum echten Vergleichen zweier Arrays kann die Klasse Arrays im Package *java.util* verwendet werden, diese enthält daneben auch noch sehr nützliche Klassenmethoden zum Sortieren, Durchsuchen (Binärsuche) und Auffüllen von Arrays.

import java.util.*;

```
if(Arrays.equals(feld1, feld2))  // Vergleicht Größe und Inhalt der Arrays!
```

Übungsaufgaben:

- Testen Sie die angeführten Beispiele durch.
- Implementieren Sie eine Testnotenstatistik für eine Klasse mit maximal 36 Schülern.
  Der Benutzer soll eine bestimmte Anzahl Testnoten eingeben können, hieraus sollen folgende Daten ermittelt werden:
  - Notendurchschnitt
  - Anzahl von 1er, 2er,...

  Erstellen Sie eine eigene Klasse, welche die entsprechenden Methoden für die Berechnungen enthält.

8. Grafische Benutzeroberflächen

8.1 Grundlegende Eigenschaften

Im Gegensatz zu den meisten anderen Programmiersprachen wurde Java von Anfang an mit dem Anspruch entwickelt, ein vielseitiges, aber einfach zu bedienendes System für die Gestaltung grafischer Oberflächen zur Verfügung zu stellen. Das Resultat dieser Bemühungen steht seit dem SDK 1.0 als Grafikbibliothek unter dem Namen *Abstract Windowing Toolkit* (*AWT*) zur Verfügung.
Die Fähigkeiten des AWT lassen sich grob in vier Gruppen unterteilen:
- Grafische Primitivoperationen zum Zeichnen von Linien, Füllen von Flächen und zur Ausgabe von Text
- Steuerung des Ablaufs auf der Basis von Nachrichten für Tastatur-, Maus- und Fensterereignisse
- Dialogelemente und Funktionen zum portablen Design von Dialogboxen
- Fortgeschrittenere Grafikfunktionen zur Darstellung und Manipulation von Bitmaps,Ausgabe von Sound

Da die grafischen Fähigkeiten Bestandteil der Sprache bzw. ihrer Klassenbibliothek sind, können sie als portabel angesehen werden. Unabhängig von der Zielplattform wird ein GUI-basiertes Programm auf allen verwendeten Systemen gleich oder zumindest ähnlich laufen.

8.2 Von AWT nach Swing

Änderungen mit dem SDK 1.1

Die Entwicklung von grafikorientierten Anwendungen im SDK 1.0 war zwar relativ einfach und erzeugte portable Programme mit grafischer Oberfläche, war aber durch eine Reihe von Restriktionen des AWT eingeschränkt. Vor allem bei der Erstellung großer GUI-Anwendungen wurden die Programme schnell unübersichtlich, und die Performance litt unter dem unzulänglichen Event-Modell der Version 1.0. Mit der Version 1.1 des SDK hat Sun das AWT massiv verändert. Es gab umfangreiche Fehlerbehebungen, und eine Vielzahl von Methodennamen wurden geändert. Vor allem aber wurde das Event-Handling, also der Transfer von GUI-Ereignissen, komplett überarbeitet.
Das neue Schlagwort lautet ***Delegation Based Event Handling***. Es bezeichnet die Fähigkeit des AWT, GUI-Ereignisse an beliebige Objekte weiterzuleiten und dort zu behandeln. Obwohl das Verständnis für diese neuen Techniken etwas schwieriger zu erlangen ist als beim alten Event-Modell, lohnt sich der zusätzliche Aufwand. Die Entwicklung großer GUI-Anwendungen mit einer klaren Trennung zwischen Benutzeroberfläche und Applikationslogik wird so erst möglich gemacht.

Swing

Neben dem AWT gibt es eine zweite Bibliothek für die Gestaltung grafischer Benutzeroberflächen. Sie heißt *Swing*, ist seit dem SDK 1.1 als Add-On verfügbar und seit der Version 1.1 fester Bestandteil des Java Development Kit.
Da Swing-Anwendungen in ihren Möglichkeiten weit über das hinausgehen, was das AWT bietet, werden heute die meisten GUI-Programme mit Swing geschrieben. Dennoch macht es Sinn, sich zunächst mit dem AWT zu beschäftigen. Einerseits ist das AWT einfacher zu erlernen, denn es ist weniger umfangreich als Swing. Gerade als Anfänger muss man nicht so viele neue Konzepte erlernen. Zum anderen werden viele der AWT-Features auch in Swing benötigt, und das mühsam angeeignete Wissen ist nicht völlig verloren. Ob es um die Verwendung von Farben oder unterschiedlichen Schriftarten oder um die Anordnung von Dialogelementen mit Hilfe von Panels und Layout-Managern geht, die zugrunde liegenden Techniken sind in beiden Fällen gleich. Drittens gibt es auch heute noch Anwendungen für das AWT, etwa bei der Applet-Programmierung, oder wenn die Verwendung von plattformspezifischen Dialogelementen zwingend erforderlich ist.

8.3 Anlegen eines einfachen Fensters

Um die Grafikfähigkeiten von Java nutzen zu können, muss das Paket *java.awt* eingebunden werden. Dies geschieht zweckmäßigerweise mit Hilfe folgender Anweisung am Anfang der Klassendefinition:

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

Nun stehen alle Klassen aus dem Package java.awt zur Verfügung.

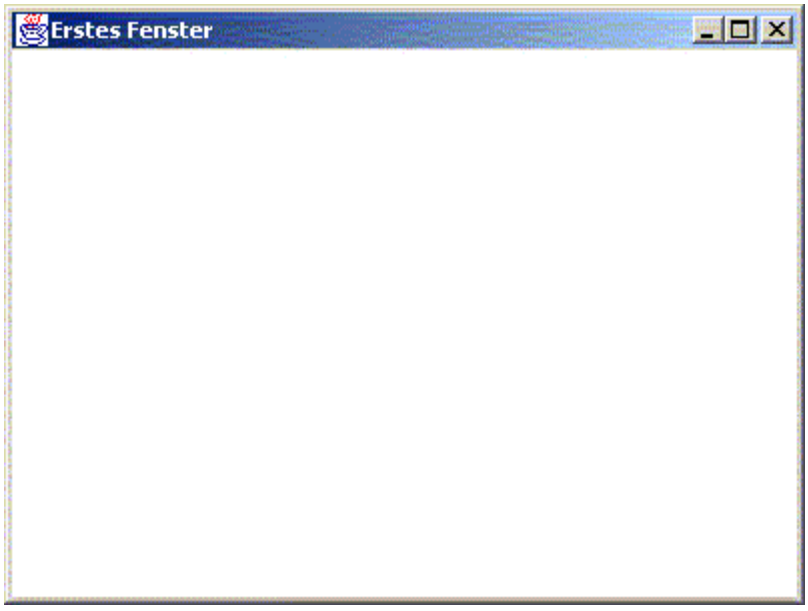
Zur Ausgabe von grafischen Elementen benötigt die Anwendung ein Fenster, auf das die Ausgabeoperationen angewendet werden können. Während bei der Programmierung eines Applets ein Standardfenster automatisch zur Verfügung gestellt wird, muss eine Applikation ihre Fenster selbst erzeugen.
Da die Kommunikation mit einem Fenster über eine Reihe von Callback-Methoden abgewickelt wird, wird eine Fensterklasse in der Regel nicht einfach instanziert. Statt dessen ist es meist erforderlich, eine eigene Klasse aus einer der vorhandenen abzuleiten und die benötigten Interfaces zu implementieren.
Zum Ableiten einer eigenen Fensterklasse wird in der Regel entweder die Klasse *Frame* oder die Klasse *Dialog* verwendet, die beide aus *Window* abgeleitet sind. *Dialog* wird vorwiegend dafür verwendet, Dialogboxen zu erstellen. die über darin enthaltene Komponenten mit dem Anwender kommunizieren. Die wichtigste Klasse zur Ausgabe von Grafiken in Java-Applikationen ist also *Frame*.
Um ein einfaches Fenster zu erzeugen und auf dem Bildschirm anzuzeigen, muss ein neues Element der Klasse *Frame* erzeugt, die gewünschte Größe eingestellt und durch einen Aufruf der Methode *show( )* sichtbar gemacht werden:

```
import java.awt.*;

public class Fenster extends Frame     // Die Klasse Fenster ist von Frame abgeleitet
{
   public Fenster()                    // Im Konstruktor der Klasse Fenster werden der
   {                                   // Fenstertitel und die Grösse(Pixel) gesetzt
      setTitle("Erstes Fenster");
      setSize(400,300);
   }
   public static void main(String args[])
   {
      Fenster fenster = new Fenster(); // Ein Objekt der Klasse Fenster wird erzeugt
      Fenster.show();                  // Das Fenster wird sichtbar gemacht
   }
}
```

Da noch kein Code für die Behandlung von GUI-Events eingebaut ist, bietet das Fenster lediglich das von Windows her bekannte Standardverhalten. Es läßt sich verschieben und in der Größe verändern und besitzt eine Titel- und Menüleiste, die mit einem Systemmenü ausgestattet ist.
Anders als in anderen grafikorientierten Systemen gibt es noch keine Funktionalität zum Beenden des Fensters. Das Beispielprogramm kann daher nur durch einen harten Abbruch seitens des Benutzers (z.B. durch Drücken von [STRG]+[C] in der DOS-Box, aus der das Fenster gestartet wurde) beendet werden.
Um das Schließen des Fenster zu ermöglichen ist eine geeignete Ereignisbehandlung notwendig. Aus diesem Grund sollte man sich zunächst mit dem Event-Handling in Java vertraut machen.

Bildschirmausgabe:



9. Interfaces

Es wurde bereits erwähnt, dass es in Java keine Mehrfachvererbung von Klassen gibt. Der Grund sind die möglichen Schwierigkeiten beim Umgang mit mehrfacher Vererbung und die Einsicht, dass das Erben nichttrivialer Methoden aus mehr als einer Klasse in der Praxis selten zu realisieren ist. Andererseits sah man es als wünschenswert an, dass Klassen eine oder mehrere Schnittstellendefinitionen erben können, und hat mit den Interfaces ein Ersatzkonstrukt geschaffen, das dieses Feature bietet.
Die Ereignisbehandlung in Java macht beispielsweise sehr intensiven Gebrauch von Interfaces. Ereignisempfänger (sog. *EventListeners*) müssen das zum Ereignis passende Empfänger-Interface implementieren

9.1 Definition eines Interfaces

Ein Interface (Schnittstelle) ist eine besondere Form einer Klasse, die ausschließlich abstrakte Methoden und Konstanten enthält. Anstelle des Schlüsselwortes *class* wird ein Interface mit dem Bezeichner *interface* deklariert. Alle Methoden eines Interfaces sind implizit abstrakt und öffentlich. Neben Methoden kann ein Interface auch Konstanten enthalten, die Definition von Konstruktoren ist allerdings nicht erlaubt.

Beispiel: Ein Interface *Groesse*, das die drei Methoden *laenge( )*, *hoehe( )* und *breite( )* enthält:

```
public interface Groesse
{
  public int laenge();
  public int hoehe();
  public int breite();
}
```

Diese Definition ähnelt sehr einer abstrakten Klasse und dient dazu, eine Schnittstelle für den Zugriff auf die räumliche Ausdehnung eines Objekts festzulegen.

9.2 Implementierung eines Interfaces

Durch das bloße Definieren eines Interfaces wird die gewünschte Funktionalität aber noch nicht zur Verfügung gestellt, sondern lediglich beschrieben. Soll diese von einer Klasse tatsächlich realisiert werden, muss sie das Interface implementieren. Dazu erweitert sie die *class*-Anweisung um eine *implements*-Klausel, hinter der der Name des zu implementierenden Interfaces angegeben wird. Der Compiler überprüft, ob alle im Interface geforderten Methoden definitionsgemäß implementiert werden. Zusätzlich "verleiht" er der Klasse einen neuen Datentyp, der ähnliche Eigenschaften wie eine echte Klasse hat.

Eine Implementierung des Interfaces Groesse könnte z. B. von einer Klasse *Auto* vorgenommen werden:

```
public class Auto implements Groesse
{
  public String name;
  public int    leistung;
  public int    laenge;
  public int    hoehe;
  public int    breite;
  public int laenge()
  {
    return this.laenge;
  }
  public int hoehe()
  {
    return this.hoehe;
  }
  public int breite()
  {
    return this.breite;
  }
}
```

Die Klasse wird dazu um drei veränderliche Instanzmerkmale erweitert, die es uns erlauben, die vom Interface geforderten Methoden auf einfache Weise zu implementieren. Ebenso wie die Klasse *Auto* könnte auch jede andere Klasse das Interface implementieren, wie im folgenden Beispiel zu sehen ist:

```
public class FussballPlatz implements Groesse
{
  public int laenge()
  {
    return 105000;
  }
  public int hoehe()
  {
    return 0;
  }
  public int breite()
  {
    return 70000;
  }
}
```

Die Art der Realisierung der vereinbarten Methoden spielt für das Implementieren eines Interfaces keine Rolle. Tatsächlich kommt es häufig vor, dass Interfaces von sehr unterschiedlichen Klassen implementiert und die erforderlichen Methoden auf sehr unterschiedliche Weise realisiert werden.

Anmerkung:
Eine Klasse kann ein Interface auch dann implementieren, wenn sie nicht alle seine Methoden implementiert. In diesem Fall ist die Klasse allerdings als *abstract* zu deklarieren und kann nicht dazu verwendet werden, Objekte zu instanziieren.

9.3 Verwenden eines Interfaces

Nützlich ist ein Interface immer dann, wenn Eigenschaften einer Klasse beschrieben werden sollen, die nicht direkt in seiner normalen Vererbungshierarchie abgebildet werden können. Hätte man beispielsweise *Groesse* als abstrakte Klasse definiert, ergäbe sich eine sehr unnatürliche Ableitungshierarchie, wenn Autos und Fußballplätze daraus abgeleitet wären. Durch Implementieren des *Groesse*-Interfaces können sie die Verfügbarkeit der drei Methoden *laenge( )*, *hoehe( )* und *breite( )* dagegen unabhängig von ihrer eigenen Vererbungslinie garantieren.

Beispiel: Es soll eine Methode *grundflaeche()* entwickelt werden, die zu jedem Objekt, das das Interface Groesse implementiert, dessen Grundfläche (Länge mal Breite) berechnet:

```
public class Test
{
  public static long grundflaeche(Groesse g)    //  Beachte:  Neuer  Datentyp  Groesse!
  {
    return (long)g.laenge() * g.breite();
  }
  public static void main(String[] args)
  {
    Auto2 auto = new Auto();                     // Zuerst erzeugen wir ein Auto...
    auto.laenge = 4235;
    auto.hoehe = 1650;
    auto.breite = 1820;
    FussballPlatz platz = new FussballPlatz();  // Wir erzeugen einen Fußballplatz...

    System.out.println("Auto:  " + grundflaeche(auto));      // Ausgabe
    System.out.println("Platz: " + grundflaeche(platz));
  }
}
```

Das Programm erzeugt zunächst zwei Objekte, die das *Groesse*-Interface implementieren. Anschließend werden sie an die Methode *grundflaeche( )* übergeben, dessen Argument g vom Typ *Groesse* ist. Durch diese Typisierung kann der Compiler sicherstellen, dass nur Objekte "des Typs" *Groesse* an grundflaeche übergeben werden. Das ist genau dann der Fall, wenn das übergebene Objekt dieses Interface implementiert.

Die Ausgabe des Programms ist:
Auto: 7707700
Platz: 7350000000

An diesem Beispiel kann man bereits die wichtigste Gemeinsamkeit zwischen abstrakten Klassen und Interfaces erkennen: Beide können im Programm zur Deklaration von lokalen Variablen, Membervariablen oder Methodenparametern verwendet werden. Eine Interface-Variable ist kompatibel zu allen Objekten, deren Klassen dieses Interface implementieren.

Auch der *instanceof*-Operator kann auf Interfacenamen angewendet werden. Eine alternative Implementierung der Methode *grundflaeche( )*, die mit allen Objekttypen funktioniert, könnte dann etwa so aussehen:

```
public static long grundflaeche(Object o)
{
  long ret = 0;
  if (o instanceof Groesse)
  {
    Groesse g = (Groesse)o;
    ret = (long)g.laenge() * g.breite();
  }

  return ret;
}
```

WICHTIG: Klassen können beliebig viele Interfaces implementieren, diese werden im Sourcecode durch einen Beistrich getrennt.

10. Event-Handling

10.1 Grundlagen

Bei der Programmierung unter einer grafischen Oberfläche erfolgt die Kommunikation zwischen Betriebssystem und Anwendungsprogramm zu einem wesentlichen Teil durch das Versenden von Nachrichten. Die Anwendung wird dabei über alle Arten von Ereignissen und Zustandsänderungen vom Betriebssystem informiert. Dazu zählen beispielsweise Mausklicks, Bewegungen des Mauszeigers, Tastatureingaben oder Veränderungen an der Größe oder Lage eines Fensters.

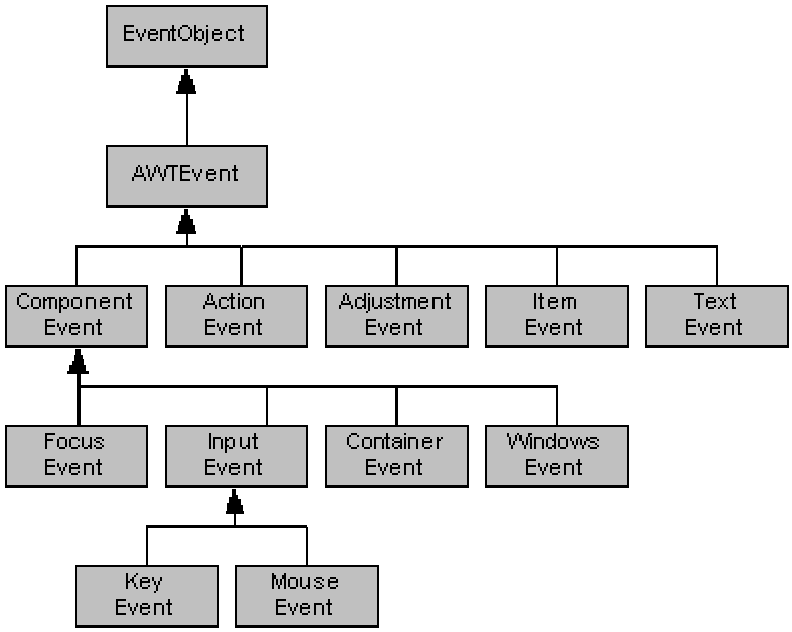
Bei der Verarbeitung des Nachrichtenverkehrs sind zwei verschiedene Arten von Objekten beteiligt. Die *Ereignisquellen (Event Sources)* sind die Auslöser der Nachrichten. Eine Ereignisquelle kann beispielsweise ein Button sein, der auf einen Mausklick reagiert, oder ein Fenster, das mitteilt, dass es über das Systemmenü geschlossen werden soll. Die Reaktion auf diese Nachrichten erfolgt in den speziellen *Ereignisempfängern (den EventListeners)*; das sind Objekte, die das zum Ereignis passende Empfänger-Interface implementieren. Damit ein Ereignisempfänger die Nachrichten einer bestimmten Ereignisquelle erhält, muss er sich bei dieser registrieren.
Dieses Kommunikationsmodell nennt sich *Delegation Event Model* oder *Delegation Based Event Handling* und wurde mit der Version 1.1 des SDK eingeführt. Im Gegensatz zum alten Modell, bei dem jedes Ereignis die Verteilermethode *handleEvent( )* der Klasse *Component* durchlaufen musste und von ihr an die verschiedenen Empfänger verteilt wurde, hat dieses neue Modell zwei wesentliche Vorteile:

- Es verringert den Nachrichtenverkehr, da nur noch die Ereignisse transportiert werden, für die es Empfänger gibt. Dadurch erhöht sich potentiell die Performance des Nachrichtentransports im AWT.
- Es erlaubt eine klare Trennung zwischen Programmcode zur Oberflächengestaltung und solchem zur Implementierung der Anwendungslogik. Es erleichtert dadurch die Erzeugung von robustem Code, der auch in großen Programmen den Entwurf sauber strukturierter Ereignishandler ermöglicht.

10.2 Ereignistypen

Im Gegensatz zur Version 1.0 werden ab dem SDK 1.1 die Ereignistypen nicht mehr durch eine einzige Klasse *Event* repräsentiert, sondern durch eine Hierarchie von Ereignisklassen, die aus der Klasse *java.util.EventObject* abgeleitet sind. Die Klasse *java.util.EventObject* fungiert damit als allgemeine Basisklasse aller Arten von Ereignissen, die zwischen verschiedenen Programmteilen ausgetauscht werden können. Ihre einzige nennenswerte Fähigkeit besteht darin, das Objekt zu speichern, das die Nachricht ausgelöst hat, und durch Aufruf der Methode *getSource( )* eines Ereignisobjekts anzugeben:

Hierarchie der Ereignisklassen:



Die Dokumentation zum SDK unterteilt diese Klassen in zwei große Hierarchien. Unterhalb der Klasse *ComponentEvent* befinden sich alle Low-Level-Ereignisse. Sie sind für den Transfer von elementaren Nachrichten zuständig, die von Fenstern oder Dialogelementen stammen. Die übrigen Klassen *ActionEvent*, *AdjustmentEvent*, *ItemEvent* und *TextEvent* werden als semantische Ereignisse bezeichnet. Sie sind nicht an ein bestimmtes GUI-Element gebunden, sondern übermitteln höherwertige Ereignisse wie das Ausführen eines Kommandos oder die Änderung eines Zustands.

Ein Problem der AWT-Designer war es, einen guten Kompromiß zwischen der Größe und der Anzahl der zu implementierenden Ereignisklassen zu finden. Da der Ableitungsbaum sehr unübersichtlich geworden wäre, wenn für jedes Elementarereignis eine eigene Klasse implementiert worden wäre, hat man sich teilweise dazu entschlossen, mehrere unterschiedliche Ereignisse durch eine einzige Klasse zu realisieren. So ist beispielsweise die Klasse *MouseEvent* sowohl für Mausbewegungen als auch für alle Arten von Klick- oder Drag-Ereignissen zuständig. Mit Hilfe der Methode *getID( )* und der in den jeweiligen Eventklassen definierten symbolischen Konstanten kann das Programm herausfinden, um welche Art von Ereignis es sich handelt.

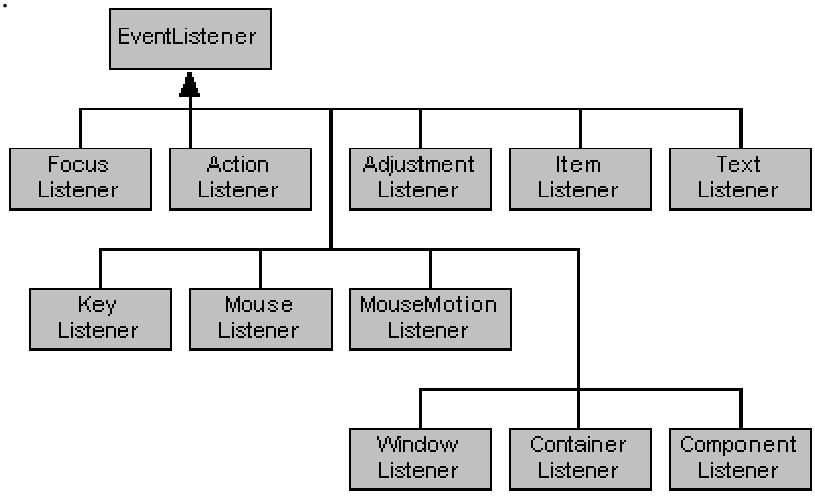
Die Eventklassen enthalten keine frei zugänglichen Felder. Statt dessen sind die Eigenschaften durch *set( )-/get( )*-Methoden gekapselt, die je nach Ereignisklasse unterschiedlich sind. So gibt es beispielsweise in *MouseEvent* die Methode *getPoint( )*, mit der die Mauszeigerposition abgefragt werden kann, in *KeyEvent* die Methode *getKeyChar( )* zur Bestimmung der gedrückten Taste und in der übergeordneten Klasse *InputEvent* die Methode *getModifiers( )*, mit der sowohl für Maus- als auch für Tastaturevents der Zustand der Sondertasten bestimmt werden kann.

10.3 Ereignisempfänger

Damit ein Objekt Nachrichten empfangen kann, muss es eine Reihe von Methoden implementieren, die von der Nachrichtenquelle, bei der es sich registriert hat, aufgerufen werden können. Um sicherzustellen, dass diese Methoden vorhanden sind, müssen die Ereignisempfänger bestimmte *Interfaces* implementieren, die aus der Klasse *EventListener* des Pakets *java.util* abgeleitet sind. Diese *EventListener-Interfaces* befinden sich im Paket *java.awt.events*.

Je Ereignisklasse gibt es ein *EventListener-Interface*. Es definiert eine separate Methode für jede Ereignisart dieser Ereignisklasse. So besitzt beispielsweise das Interface *MouseListener* die Methoden *mouseClicked( )*, *mouseEntered( )*, *mouseExited( )*, *mousePressed( )* und *mouseReleased( )*, die bei Auftreten des jeweiligen Ereignisses aufgerufen werden.

Hierarchie der EventListener-Interfaces:



Jede der Methoden eines Listener-Interfaces enthält als einziges Argument ein Objekt des zugehörigen Ereignistyps.

10.4 Ereignisquellen

Die Ereignisse stammen von den Ereignisquellen, also von Fenstern, Dialogelementen oder höheren Programmobjekten. Eine Ereignisquelle sendet aber nur dann Ereignisse an einen Ereignisempfänger, wenn dieser sich bei der Ereignisquelle registriert hat. Fehlt eine solche Registrierung, wird die Ereignisquelle keine Ereignisse senden und der Empfänger folglich auch keine erhalten.

Die Registrierung erfolgt mit speziellen Methoden, an die ein Objekt übergeben wird, das das jeweilige EventListener-Interface implementiert. So gibt es beispielsweise eine Methode *addMouseListener( )* in der Klasse *Component*, mit der ein Objekt, das das Interface *MouseListener* implementiert, sich für den Empfang von Mausereignissen bei der Komponente registrieren lassen kann. Nach erfolgter Registrierung wird bei jedem Mausereignis die jweilige Methode *mouseClicked( )*, *mouseEntered( )*, *mouseExited( )*, *mousePressed( )* oder *mouseReleased( )* aufgerufen.

Die Ereignisquellen unterstützen das Multicasting von Ereignissen. Dabei kann eine Ereignisquelle nicht nur einen einzelnen *EventListener* mit Nachrichten versorgen, sondern eine beliebige Anzahl von ihnen. Jeder Aufruf von *addXYZListener* registriert dabei einen weiteren Listener in der Ereignisquelle.

10.5 Adapterklassen

Eine Adapterklasse in Java ist eine Klasse, die ein vorgegebenes Interface mit leeren Methodenrümpfen implementiert. Adapterklassen können verwendet werden, wenn aus einem Interface lediglich ein Teil der Methoden benötigt wird, der Rest aber uninteressant ist. Dazu leitet man einfach eine neue Klasse aus der Adapterklasse ab, anstatt das zugehörige Interface zu implementieren, und überlagert die benötigten Methoden. Alle übrigen Methoden des Interfaces werden von der Basisklasse zur Verfügung gestellt.
Zu jedem der Low-Level-Ereignisempfänger steht eine passende Adapterklasse zur Verfügung. So gibt es die Adapterklassen *FocusAdapter*, *KeyAdapter*, *MouseAdapter*, *MouseMotionAdapter*, *ComponentAdapter*, *ContainerAdapter* und *WindowAdapter*.

10.6 Beispiele für Ereignisse und Ereignisempfänger

Key-Ereignisse

Eigenschaft	**Klasse, Interface oder Methode**
Ereignisklasse	*KeyEvent*
Listener-Interface	*KeyListener*
Registrierungsmethode	*addKeyListener*
Mögliche Ereignisquellen	*Component*
Ereignismethode	**Bedeutung**
KeyPressed	Eine Taste wurde gedrückt.
KeyReleased	Eine Taste wurde losgelassen.
KeyTyped	Eine Taste wurde gedrückt und wieder losgelassen.

Mouse-Ereignisse

Eigenschaft	**Klasse, Interface oder Methode**
Ereignisklasse	*MouseEvent*
Listener-Interface	*MouseListener*
Registrierungsmethode	*addMouseListener*
Mögliche Ereignisquellen	*Component*
Ereignismethode	**Bedeutung**
MouseClicked	Eine Maustaste wurde gedrückt und wieder losgelassen.
MouseEntered	Der Mauszeiger betritt die Komponente.
MouseExited	Der Mauszeiger verläßt die Komponente.
MousePressed	Eine Maustaste wurde gedrückt.
MouseReleased	Eine Maustaste wurde losgelassen.

10.7 Entwurfsmuster für den Nachrichtenverkehr

Als Basis soll ein einfaches Programm dienen, das folgende Anforderungen erfüllt:

Nach dem Start soll das Programm ein Fenster mit dem Titel »Nachrichtentransfer« auf dem Bildschirm anzeigen. Das Fenster soll einen grauen Hintergrund haben und in der Client-Area die Meldung »Zum Beenden bitte ESC drücken ...« anzeigen. Nach Drücken der Taste [ESC] soll das Fenster geschlossen und das Programm beendet werden. Andere Tastendrücke, Mausklicks oder ähnliche Ereignisse werden ignoriert.

Basis der Programme ist der folgende Quellcode:

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

public class Fenster extends Frame
{
  public static void main(String[] args)
  {
    Fenster wnd = new Fenster();
  }

  public Fenster()
  {
    super("Nachrichtentransfer");  // super bezeichnet die Basisklasse!!!
    setBackground(Color.lightGray);
    setSize(300,200);
    setLocation(200,100);
    setVisible(true);
  }

  public void paint(Graphics g)     // Diese Methode wird ausgeführt, wenn das Fenster
                                    // neu gezeichnet werden muss
  {
    g.setFont(new Font("Serif",Font.PLAIN,18));              // Fonts siehe SDK-Hilfe
    g.drawString("Zum Beenden bitte ESC drücken...",10,50); // Gibt einen Text aus
  }
}
```

Bildschirmausgabe:

Das Programm erfüllt die ersten der obengenannten Anforderungen, ist aber mangels Event-Handler noch nicht in der Lage, per [ESC] beendet zu werden. Um die Nachfolgeversionen vorzubereiten, ist bereits die Anweisung import java.awt.event.* hinzugefügt.

Variante 1: Implementierung eines EventListener-Interfaces

Bei der ersten Variante gibt es nur eine einzige Klasse, *Fenster*. Sie ist einerseits eine Ableitung der Klasse *Frame*, um ein Fenster auf dem Bildschirm darzustellen und zu beschriften. Andererseits implementiert sie das Interface *KeyListener*, das die Methoden *keyPressed( )*, *keyReleased( )* und *keyTyped( )* definiert. Der eigentliche Code zur Reaktion auf die Taste [ESC] steckt in der Methode *keyPressed( )*, die immer dann aufgerufen wird, wenn eine Taste gedrückt wurde. Mit der Methode *getKeyCode( )* der Klasse *KeyEvent* wird auf den Code der gedrückten Taste zugegriffen und dieser mit der symbolischen Konstante VK_ESCAPE verglichen. Stimmen diese überein, wurde [ESC] gedrückt, und das Programm wird beendet.

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

public class Fenster extends Frame implements KeyListener
{
   public static void main(String[] args)
   {
      Fenster wnd = new Fenster();
   }
   public Fenster()
   {
      super("Nachrichtentransfer");
      setBackground(Color.lightGray);
      setSize(300,200);
      setLocation(200,100);
      setVisible(true);
      addKeyListener(this);    // Hier wird der Ereignisempfänger registriert!!!
   }
   public void paint(Graphics g)
   {
      g.setFont(new Font("Serif",Font.PLAIN,18));
      g.drawString("Zum Beenden bitte ESC drücken...",10,50);
   }
   public void keyPressed(KeyEvent event)
   {
      if (event.getKeyCode() == KeyEvent.VK_ESCAPE)
      {
         setVisible(false);
         dispose();             // Gibt alle Ressourcen des Fensters frei
         System.exit(0);        // Beendet das Programm
      }
   }
   public void keyReleased(KeyEvent event)
   {
   }
   public void keyTyped(KeyEvent event)
   {
   }
}
```

Die Verbindung zwischen der Ereignisquelle (in diesem Fall der Fensterklasse *Fenster*) und dem Ereignisempfänger (ebenfalls die Klasse *Fenster*) erfolgt über den Aufruf der Methode *addKeyListener( )* der Klasse *Frame*. Diese Implementierung ist sehr naheliegend, denn sie ist einfach zu implementieren und erfordert keine weiteren Klassen.

Nachteile:

- Es besteht keine Trennung zwischen GUI-Code und Applikationslogik (unübersichtlich, schwer wartbar).
- Für jeden Ereignistyp muss eine passende Listener-Klasse registriert werden. Da viele der *EventListener*-Interfaces mehr als eine Methode definieren, werden dadurch schnell viele leere Methodenrümpfe in der Fensterklasse zu finden sein. In diesem Beispiel sind es schon *keyReleased( )* und *keyTyped( )*, bei zusätzlichen Interfaces würden schnell weitere hinzukommen.

Variante 2: Lokale und anonyme Klassen
Die zweite Alternative bietet eine bessere Lösung. Sie basiert auf der Verwendung lokaler bzw. anonymer Klassen und kommt ohne die Nachteile der vorigen Version aus. Sie ist das in der Dokumentation des SDK empfohlene Entwurfsmuster für das Event-Handling in kleinen Programmen oder bei Komponenten mit einfacher Nachrichtenstruktur.

Lokale Klassen
Dabei wird innerhalb einer bestehenden Klasse X eine neue Klasse Y definiert, die nur innerhalb von X sichtbar ist. Objektinstanzen von Y können damit auch nur innerhalb von X erzeugt werden. Anders herum kann Y auf die Membervariablen von X zugreifen.
Die Anwendung lokaler Klassen für die Ereignisbehandlung besteht darin, mit ihrer Hilfe die benötigten EventListener zu implementieren. Dazu wird in dem GUI-Objekt, das einen Event-Handler benötigt, eine lokale Klasse definiert und aus einer passenden Adapterklasse abgeleitet. Nun braucht nicht mehr das gesamte Interface implementiert zu werden (denn die Methodenrümpfe werden ja aus der Adapterklasse geerbt), sondern lediglich die tatsächlich benötigten Methoden. Da die lokale Klasse zudem auf die Membervariablen und Methoden der Klasse zugreifen kann, in der sie definiert wurde, lassen sich auf diese Weise sehr schnell die benötigten Ereignisempfänger zusammenbauen.

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

public class Fenster extends Frame
{
  public static void main(String[] args)
  {
    Fenster wnd = new Fenster();
  }
  public Fenster()
  {
    super("Nachrichtentransfer");
    setBackground(Color.lightGray);
    setSize(300,200);
    setLocation(200,100);
    setVisible(true);
    addKeyListener(new MyKeyListener());
  }
  public void paint(Graphics g)
  {
    g.setFont(new Font("Serif",Font.PLAIN,18));
    g.drawString("Zum Beenden bitte ESC drücken...",10,50);
  }
  class MyKeyListener extends KeyAdapter //Lokale Klasse,abgeleitet von Adapterklasse
  {
    public void keyPressed(KeyEvent event)
    {
      if (event.getKeyCode() == KeyEvent.VK_ESCAPE)
      {
        setVisible(false);
        dispose();
        System.exit(0);
      }
    }
  }
}
```

Vorteil: Es werden keine unnützen Methodenrümpfe erzeugt, aber trotzdem verbleibt der Ereignisempfängercode wie im vorigen Beispiel innerhalb der Ereignisquelle. Dieses Verfahren ist also immer dann gut geeignet, wenn es von der Architektur oder der Komplexität der Ereignisbehandlung her sinnvoll ist, Quelle und Empfänger zusammenzufassen.

Anonyme Klassen

Die häufigste Anwendung lokaler Klassen innerhalb von Methoden besteht darin, diese *anonym* zu definieren. Dabei erhält die Klasse keinen eigenen Namen, sondern Definition und Instanzierung erfolgen in einer kombinierten Anweisung. Eine anonyme Klasse ist also eine Einwegklasse, die nur einmal instanziiert werden kann. Anonyme Klassen werden normalerweise aus anderen Klassen abgeleitet oder erweitern existierende Interfaces.

Das folgende Beispiel ist eine leichte Variation des vorigen. Es zeigt die Verwendung einer anonymen Klasse, die aus KeyAdapter abgeleitet wurde, als Ereignisempfänger. Zum Instanziierungszeitpunkt erfolgt die Definition der überlagernden Methode *keyPressed( )*, in der der Code zur Reaktion auf das Drücken der Taste [ESC] untergebracht wird.

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

public class Fenster extends Frame
{
  public static void main(String[] args)
  {
    Fenster wnd = new Fenster();
  }

  public Fenster()
  {
    super("Nachrichtentransfer");
    setBackground(Color.lightGray);
    setSize(300,200);
    setLocation(200,100);
    setVisible(true);
    addKeyListener(
      new KeyAdapter() {
        public void keyPressed(KeyEvent event)
        {
          if (event.getKeyCode() == KeyEvent.VK_ESCAPE)
          {
            setVisible(false);
            dispose();
            System.exit(0);
          }
        }
      }
    );
  }

  public void paint(Graphics g)
  {
    g.setFont(new Font("Serif",Font.PLAIN,18));
    g.drawString("Zum Beenden bitte ESC drücken...",10,50);
  }
}
```

Vorteilhaft bei dieser Vorgehensweise ist der verminderte Aufwand, denn es muss keine separate Klassendefinition angelegt werden. Statt dessen werden die wenigen Codezeilen, die zur Anpassung der Adapterklasse erforderlich sind, dort eingefügt, wo die Klasse instanziert wird, nämlich beim Registrieren des Nachrichtenempfängers. Anonyme Klassen haben einen ähnlichen Einsatzbereich wie lokale, empfehlen sich aber vor allem, wenn sehr wenig Code für den Ereignisempfänger benötigt wird. Bei aufwendigeren Ereignisempfängern ist die explizite Definition einer benannten Klasse dagegen vorzuziehen.

Variante 3: Trennung von GUI- und Anwendungscode

Eine Trennung zwischen dem Programmcode, der für die Oberfläche zuständig ist, und solchem, der für die Anwendungslogik zuständig ist, ist absolut wünschenswert. Es wird eine bessere Modularisierung des Programms erreicht, und der Austausch oder die Erweiterung von Teilen des Programms wird erleichtert.

Das Delegation Event Model wurde auch mit dem Designziel entworfen, eine solche Trennung zu ermöglichen bzw. zu erleichtern. Der Grundgedanke dabei ist es, auch Nicht-Komponenten die Reaktion auf GUI-Events zu ermöglichen. Dies wurde dadurch erreicht, dass jede Art von Objekt als Ereignisempfänger registriert werden kann, solange es die erforderlichen *Listener*-Interfaces implementiert. Damit ist es möglich, die Anwendungslogik vollkommen von der grafischen Oberfläche abzulösen und in Klassen zu verlagern, die eigens für diesen Zweck entworfen wurden.

Das nachfolgende Beispiel zeigt diese Vorgehensweise, indem es das Beispielprogramm in die drei Klassen *MyApp*, *MainFrameCommand* und *MainFrameGUI* aufteilt. *MyApp* enthält nur noch die *main( )*-Methode und dient lediglich dazu, die anderen beiden Klassen zu instanzieren. *MainFrameGUI* realisiert die GUI-Funktionalität und stellt das Fenster auf dem Bildschirm dar. *MainFrameCommand* spielt die Rolle des Kommandointerpreters, der immer dann aufgerufen wird, wenn im Fenster ein Tastaturereignis aufgetreten ist.

Die Verbindung zwischen beiden Klassen erfolgt durch Aufruf der Methode *addKeyListener* in *MainFrameGUI*, an die das an den Konstruktor übergebene *MainFrameCommand*-Objekt weitergereicht wird. Dazu ist es erforderlich, dass das Hauptprogramm den Ereignisempfänger *cmd* zuerst instanziert, um ihn bei der Instanzierung des GUI-Objekts *gui* übergeben zu können.

Umgekehrt benötigt natürlich auch das Kommando-Objekt Kenntnis über das GUI-Objekt, denn es soll ja das zugeordnete Fenster schließen und das Programm beenden. Der scheinbare Instanzierungskonflikt durch diese zirkuläre Beziehung ist aber in Wirklichkeit gar keiner, denn bei jedem Aufruf einer der Methoden von *MainFrameCommand* wird an das *KeyEvent*-Objekt der Auslöser der Nachricht übergeben, und das ist in diesem Fall stets das *MainFrameGUI*-Objekt *gui*. So kann innerhalb des Kommando-Objekts auf alle öffentlichen Methoden des GUI-Objekts zugegriffen werden.

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

public class Fenster
{
  public static void main(String[] args)
  {
    MainFrameCommand cmd = new MainFrameCommand();
    MainFrameGUI     gui = new MainFrameGUI(cmd);
  }
}

class MainFrameGUI extends Frame
{
  public MainFrameGUI(KeyListener cmd)
  {
    super("Nachrichtentransfer");
    setBackground(Color.lightGray);
    setSize(300,200);
    setLocation(200,100);
    setVisible(true);
    addKeyListener(cmd);     // Der Ereignisempfänger wird registriert
  }
```

```
   public void paint(Graphics g)
   {
      g.setFont(new Font("Serif",Font.PLAIN,18));
      g.drawString("Zum Beenden bitte ESC drücken...",10,50);
   }
}

class MainFrameCommand implements KeyListener
{
   public void keyPressed(KeyEvent event)
   {
      Frame source = (Frame)event.getSource(); // Ermitteln der Quelle des Ereignisses!
      if (event.getKeyCode() == KeyEvent.VK_ESCAPE)
      {
         source.setVisible(false);
         source.dispose();
         System.exit(0);
      }
   }

   public void keyReleased(KeyEvent event)
   {
   }

   public void keyTyped(KeyEvent event)
   {
   }
}
```

Diese Designvariante ist vorwiegend für größere Programme geeignet, bei denen eine Trennung von Programmlogik und Oberfläche sinnvoll ist. Für sehr kleine Programme oder solche, die wenig Ereigniscode haben, sollte eher eine der vorherigen Varianten angewendet werden, wenn diese zu aufwendig ist.

Variante 4: Überlagern der Event-Handler in den Komponenten

Als letzte Möglichkeit, auf Nachrichten zu reagieren, soll das Überlagern der Event-Handler in den Ereignisquellen selbst aufgezeigt werden.

Jede Ereignisquelle besitzt eine Reihe von Methoden, die für das Aufbereiten und Verteilen der Nachrichten zuständig sind. Soll eine Nachricht weitergereicht werden, so wird dazu zunächst innerhalb der Nachrichtenquelle die Methode *processEvent( )* aufgerufen. Diese verteilt die Nachricht anhand ihres Typs an spezialisierte Methoden, deren Name sich nach dem Typ der zugehörigen Ereignisklasse richtet. So ist beispielsweise die Methode *processActionEvent( )* für das Handling von Action-Events und *processMouseEvent( )* für das Handling von Mouse-Events zuständig:

Beide Methodenarten können in einer abgeleiteten Klasse überlagert werden, um die zugehörigen Ereignisempfänger zu implementieren. Wichtig ist dabei, dass in der abgeleiteten Klasse die gleichnamige Methode der Basisklasse aufgerufen wird, um das Standardverhalten sicherzustellen. Wichtig ist weiterhin, dass sowohl *processEvent( )* als auch *processActionEvent( )* usw. nur aufgerufen werden, wenn der entsprechende Ereignistyp für diese Ereignisquelle aktiviert wurde. Dies passiert in folgenden Fällen:

- Wenn ein Ereignisempfänger über die zugehörige *addEventListener( )*-Methode registriert wurde.
- Wenn der Ereignistyp explizit durch Aufruf der Methode *enableEvents( )* aktiviert wurde.

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

public class Fenster extends Frame
{
  public static void main(String[] args)
  {
    Fenster wnd = new Fenster();
  }
  public Fenster()
  {
    super("Nachrichtentransfer");
    setBackground(Color.lightGray);
    setSize(300,200);
    setLocation(200,100);
    setVisible(true);
    enableEvents(AWTEvent.KEY_EVENT_MASK);
  }
  public void paint(Graphics g)
  {
    g.setFont(new Font("Serif",Font.PLAIN,18));
    g.drawString("Zum Beenden bitte ESC drücken...",10,50);
  }
  public void processKeyEvent(KeyEvent event)
  {
    if (event.getID() == KeyEvent.KEY_PRESSED)
    {
      if (event.getKeyCode() == KeyEvent.VK_ESCAPE)
      {
        setVisible(false);
        dispose();
        System.exit(0);
      }
    }
    super.processKeyEvent(event);  // Aufruf des Basisklassenhandlers
  }
}
```

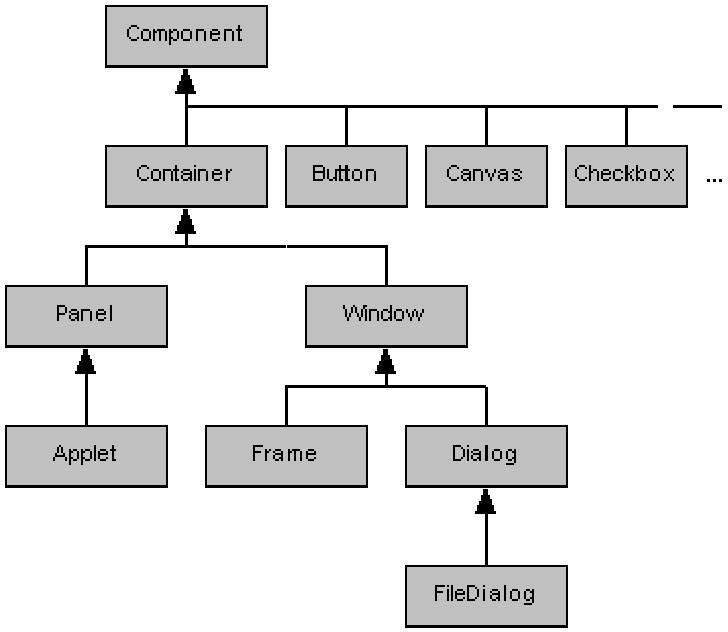
Diese Art der Ereignisbehandlung ist nur sinnvoll, wenn Fensterklassen oder Dialogelemente überlagert werden und ihr Aussehen oder Verhalten signifikant verändert wird. Das hier vorgestellte Verfahren umgeht das Delegation Event Model vollständig und hat damit die gleichen inhärenten Nachteile wie das Event-Handling des alten SDK. Es sollte daher bis auf Ausnahmefälle nicht verwendet werden.

11. Fensterklassen und wichtige Steuerelemente

11.1 Fensterklassen

Das Abstract Windowing Toolkit von Java enthält verschiedene Fensterklassen, die über eine gemeinsame Vererbungshierarchie miteinander in Verbindung stehen.

Hierarchie der Fensterklassen:



Component ist eine abstrakte Klasse, deren Aufgabe darin besteht, ein Programmelement zu repräsentieren, das eine Größe und Position hat und das in der Lage ist, eine Vielzahl von Ereignissen zu senden und auf diese zu reagieren.

Container ist ebenfalls eine abstrakte Klasse. Sie ist dafür zuständig, innerhalb einer Komponente andere Komponenten aufzunehmen. *Container* stellt Methoden zur Verfügung, um Komponenten hinzuzufügen oder sie zu entfernen und realisiert in Zusammenarbeit mit den *LayoutManager*-Klassen die Positionierung und Anordnung der Komponenten. *Container* und *Component* bilden zusammen ein *Composite-Pattern*.

Panel ist die einfachste konkrete Klasse mit den Eigenschaften von *Component* und *Container*. Sie wird verwendet, um eine Sammlung von Dialogelementen mit einem vorgegebenen Layout zu definieren und wiederverwendbar zu machen. Nützlich ist die Klasse auch beim Layouten von Dialogen, weil sie es ermöglicht, Teildialoge zu schachteln und mit eigenen Layoutmanagern zu versehen.

Die für die Entwicklung von *Applets* wichtige Klasse *Applet* ist eine direkte Unterklasse von *Panel*. Sie erweitert zwar die Funktionalität der Klasse *Panel* um Methoden, die für das Ausführen von Applets von Bedeutung sind, bleibt aber letzlich ein Programmelement, das eine Größe und Position hat, auf Ereignisse reagieren kann und in der Lage ist, weitere Komponenten aufzunehmen.

Die Klasse *Window* abstrahiert ein Top-Level-Window ohne Rahmen, Titelleiste und Menü. Sie ist für Anwendungen geeignet, die ihre Rahmenelemente selbst zeichnen oder die volle Kontrolle über das gesamte Fenster benötigen.

Die Klasse *Frame* repräsentiert ein Top-Level-Window mit Rahmen, Titelleiste und optionalem Menü. Einem Frame kann ein Icon zugeordnet werden, das angezeigt wird, wenn das Fenster minimiert wird. Es kann eingestellt werden, ob das Fenster vom Anwender in der Größe verändert werden kann. Zusätzlich besteht die Möglichkeit, das Aussehen des Mauszeigers zu verändern.

Die zweite aus Window abgeleitete Klasse ist *Dialog*. Sie ist dafür vorgesehen, modale oder nicht-modale Dialoge zu realisieren. Ein modaler Dialog ist ein Fenster, das immer im Vordergrund des Fensters bleibt, von dem es erzeugt wurde, und das alle übrigen Fensteraktivitäten und Ereignisse so lange blockiert, bis es geschlossen wird. Ein nicht-modaler Dialog kann mit anderen Fenstern koexistieren und erlaubt es, im aufrufenden Fenster weiterzuarbeiten.
Die unterste Klasse in der Hierarchie der Fenster ist *FileDialog*. Sie stellt den Standard-Dateidialog des jeweiligen Betriebssystems zur Verfügung. Dieser kann beim Laden oder Speichern einer Datei zur Eingabe oder Auswahl eines Dateinamens verwendet werden. Gegenüber den Möglichkeiten des Standard-Dateidialogs, wie er vom Windows-API zur Verfügung gestellt wird, sind die Fähigkeiten der Klasse *FileDialog* allerdings etwas eingeschränkt.

11.2 Wichtige Steuerelemente: Menüs, Textfelder und Buttons

Jedes Programm mit grafischer Benutzeroberfläche bedient sich allseits bekannter Steuerelemente wie Menüs, Buttons, Eingabefelder, Listfeldern, ...
Im folgenden soll ein kleines Programm mit einem einfachen Menü, einem Textfeld und einem Button als Ausgangspunkt für weitere Experimente dienen:

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

class MainMenu extends MenuBar   // Menüklasse
{
   public MainMenu(ActionListener actionListener)
   {
      Menu m;

      // Menü 1
      m = new Menu("Menü 1");
      m.add(new MenuItem("Eins"));
      m.add(new MenuItem("Zwei"));
      add(m);
      m.addActionListener(actionListener);
      //Menü 2
      m = new Menu("Menü 2");
      m.add(new MenuItem("Drei"));
      m.addSeparator();
      m.add(new MenuItem("Vier"));
      add(m);
      m.addActionListener(actionListener);
  }
}

public class MyApp extends Frame implements ActionListener  // Fensterklasse
{
   Button button1 = new Button("Button 1");
   TextField textField1 = new TextField(10);

   public static void main(String[] args)
   {
      MyApp wnd = new MyApp();
   }

   public MyApp()
   {
      super("Menüs");                  // Aufruf des Basisklassenkonstruktors
      setLocation(100,100);            // Fensterposition festlegen
      setSize(400,300);                // Fenstergröße festlegen
      setMenuBar(new MainMenu(this));  // Menü erzeugen und festlegen
      addWindowListener(new MyWindowListener());   // Ereignis-Empfänger registrieren
```

```
      // Neues Panel erzeugen und hinzufügen
      Panel myPanel = new Panel();
      add(myPanel);

      // Button hinzufügen
      button1.addActionListener(this);
      myPanel.add(button1);

      // Textfeld hinzufügen
      myPanel.add(textField1);

      // Layout Manager einrichten
      setLayout(new FlowLayout());

      // Fenster anzeigen
      setVisible(true);
   }

   // Methode des Action Listener-Interface
   public void actionPerformed(ActionEvent event)
   {
      Object quelle = event.getSource();  // Ermitteln der Quelle des Ereignisses

      String cmd = event.getActionCommand();
      if(cmd.equals("Eins"))
      {
         System.out.println("Menüpunkt 1 ausgewählt");
      }
      if(cmd.equals("Drei"))
      {
         System.out.println("Menüpunkt 3 ausgewählt");
      }

      if(quelle == button1)
      {
         System.out.println("Button 1 gedrückt");
         System.out.println("Text in Textfeld: " + textField1.getText());
      }
   }

   // Window-Listener Interface als lokale Klasse implementieren
   class MyWindowListener extends WindowAdapter
   {
      public void windowClosing(WindowEvent event)
      {
         event.getWindow().dispose();
         System.exit(0);
      }
   }
}
```

11.3 Die Layout-Manager

In vielen grafischen Oberflächen wird die Anordnung der Elemente eines Dialoges durch Angabe *absoluter Koordinaten* vorgenommen. Dabei wird für jede Komponente manuell oder mit Hilfe eines Ressourcen-Editors pixelgenau festgelegt, an welcher Stelle im Dialog sie zu erscheinen hat.
Da Java-Programme auf vielen unterschiedlichen Plattformen mit unterschiedlichen Ausgabegeräten laufen sollen, war eine solche Vorgehensweise für die Designer des AWT nicht akzeptabel. Sie wählten statt dessen den Umweg über einen Layoutmanager, der für die Anordnung der Dialogelemente verantwortlich ist. Um einen Layoutmanager verwenden zu können, wird dieser dem Fenster mit der Methode *setLayout( )* zugeordnet. Er ordnet dann die per *add( )* übergebenen Elemente auf dem Fenster an.

Jeder Layoutmanager implementiert seine eigene Logik bezüglich der optimalen Anordnung der Komponenten:

- Das *FlowLayout* ordnet Dialogelemente nebeneinander in einer Zeile an. Wenn keine weiteren Elemente in die Zeile passen, wird mit der nächsten Zeile fortgefahren.
- Das *GridLayout* ordnet die Dialogelemente in einem rechteckigen Gitter an, dessen Zeilen- und Spaltenzahl beim Erstellen des Layoutmanagers angegeben wird.
- Das *BorderLayout* verteilt die Dialogelemente nach Vorgabe des Programms auf die vier Randbereiche und den Mittelbereich des Fensters.
- Das *CardLayout* ist in der Lage, mehrere Unterdialoge in einem Fenster unterzubringen und jeweils einen davon auf Anforderung des Programms anzuzeigen.
- Das *GridBagLayout* ist ein komplexer Layoutmanager, der die Fähigkeiten von GridLayout erweitert und es ermöglicht, mit Hilfe von Bedingungsobjekten sehr komplexe Layouts zu erzeugen.

Neben den gestalterischen Fähigkeiten eines Layoutmanagers bestimmt in der Regel die Reihenfolge der Aufrufe der *add( )*-Methode des Fensters die tatsächliche Anordnung der Komponenten auf dem Bildschirm. Wenn nicht - wie es z.B. beim *BorderLayout* möglich ist - zusätzliche Positionierungsinformationen an das Fenster übergeben werden, ordnet der jeweilige Layoutmanager die Komponenten in der Reihenfolge ihres Eintreffens an.

Um komplexere Layouts realisieren zu können, als die Layoutmanager sie in ihren jeweiligen Grundausprägungen bieten, gibt es die Möglichkeit, Layoutmanager zu schachteln. Auf diese Weise kann auch ohne Vorgabe fester Koordinaten fast jede gewünschte Komponentenanordnung realisiert werden. Sollte auch diese Variante nicht genau genug sein, so bietet sich schließlich durch Verwendung eines *Null-Layouts* die Möglichkeit an, Komponenten durch Vorgabe fester Koordinaten zu plazieren.

Übungsaufgaben:

- Testen Sie die angeführten Beispiele durch.
- Schreiben Sie ein vollständiges Programm für die Caesar-Chiffrierung und Dechiffrierung mit einer ansprechenden grafischen Benutzeroberfläche (Menüs, Eingabefelder, Buttons).

12. Applets

12.1 Grundlagen

Oft steht die Entwicklung von Applets im Vordergrund und ist der Hauptgrund für die Beschäftigung mit der Sprache. In letzter Zeit ist allerdings ein Trend zu beobachten, bei dem Java zunehmend auch als Sprache für die Anwendungsentwicklung an Bedeutung gewinnt.

Tatsächlich unterscheiden sich Applets und Applikationen gar nicht so stark voneinander, wie man vermuten könnte. Bis auf wenige Ausnahmen werden sie mit denselben Werkzeugen und Techniken konstruiert. Vereinfacht kann man sagen, dass Java-Applikationen eigenständige Programme sind, die zur Ausführung den Stand-Alone-Java-Interpreter benötigen, während Java-Applets aus HTML-Seiten heraus aufgerufen werden und zur Ausführung einen Web-Browser benötigen.

Die wichtigsten Unterschiede kann man in einer kurzen Liste zusammenfassen:

- Das Hauptprogramm eines Applets wird immer aus der Klasse *Applet* abgeleitet. Bei einer Applikation ist es prinzipiell gleichgültig, woraus die Hauptklasse abgeleitet wird.
- Eine Applikation wird gestartet, indem vom Java-Interpreter die Klassenmethode *main( )* aufgerufen wird. Das Starten eines Applets wird dadurch erreicht, dass der Web-Browser die Applet-Klasse instanziiert und die Methoden *init( )* und *start( )* aufruft.
- Aus Sicherheitsgründen darf ein Applet in der Regel weder auf Dateien des lokalen Rechners zugreifen noch externe Programme auf diesem starten. Eine Ausnahme bilden signierte Applets. Für eine Applikation gelten diese Beschränkungen nicht.
- Ein Applet arbeitet immer grafik- und ereignisorientiert. Bei einer Applikation dagegen ist es möglich, auf die Verwendung des AWT zu verzichten und alle Ein-/Ausgaben textorientiert zu erledigen.
- Applets bieten einige zusätzliche Möglichkeiten im Bereich des Benutzerschnittstellen-Designs, die so bei Applikationen nicht ohne weiteres zu finden sind. Die Ausgabe von Sound beispielsweise ist standardmäßig auf Applets beschränkt.

12.2 Die Klasse java.awt.Applet

Wie schon erwähnt, wird das Hauptprogramm eines Applets aus der Klasse *Applet* des Pakets *java.applet* abgeleitet. *Applet* ist nun aber keine der abstrakten Basisklassen der Java-Klassenbibliothek, sondern eine konkrete Grafikklasse am Ende der Klassenhierarchie. *Applet* ist aus der Klasse *Panel* abgeleitet.

Ein Applet ist also ein rechteckiges Bildschirmelement, das eine Größe und Position hat, Ereignisse empfangen kann und in der Lage ist, grafische Ausgaben vorzunehmen. Tatsächlich kommt dies dem Anspruch eines Applets, ein eigenständiges Programm zu sein, das innerhalb eines fensterartigen Ausschnitts in einem grafikfähigen Web-Browser läuft, sehr entgegen. Durch die Vererbungshierarchie erbt ein Applet bereits von seinen Basisklassen einen großen Teil der Fähigkeiten, die es zur Ausführung benötigt. Die über die Fähigkeiten von Container, Component und Panel hinaus erforderliche Funktionalität, die benötigt wird, um ein Objekt der Klasse *Applet* als Hauptmodul eines eigenständigen Programms laufen zu lassen, wird von der Klasse *Applet* selbst zur Verfügung gestellt.

12.3 Initialisierung und Endebehandlung

Die Kommunikation zwischen einem Applet und seinem Browser läuft auf verschiedenen Ebenen ab. Nach dem Laden wird das Applet zunächst instanziert und dann initialisiert. Anschließend wird es gestartet, erhält GUI-Events und wird irgendwann wieder gestoppt. Schließlich wird das Applet vom Browser nicht mehr benötigt und zerstört.

Zu jedem dieser Schritte gibt es eine korrespondierende Methode, die vor dem entsprechenden Statuswechsel aufgerufen wird. Die aus *Applet* abgeleitete Klasse ist dafür verantwortlich, diese Methoden zu überschreiben und mit der erforderlichen Funktionalität auszustatten.

Instanziierung des Applets
Bevor ein Applet aktiv werden kann, muss der Browser zunächst ein Objekt der abgeleiteten Applet-Klasse instanzieren. Hierzu ruft er den parameterlosen Default-Konstruktor der Klasse auf:

```
public Applet()
```

Üblicherweise wird dieser in der abgeleiteten Klasse nicht überlagert, sondern von Applet geerbt. Notwendige Initialisierungen von Membervariablen werden später erledigt.

Initialisierung des Applets
Nach der Instanziierung ruft der Browser die Methode *init( )* auf, um dem Applet die Möglichkeit zu geben, Initialisierungen vorzunehmen:

```
public void init()
```

init( ) wird während der Lebensdauer eines Applets genau einmal aufgerufen, nachdem die Klassendatei geladen und das Applet instanziert wurde. Innerhalb von *init( )* können Membervariablen initialisiert, Images oder Fonts geladen oder Parameter ausgewertet werden.

Starten des Applets
Nachdem die Initialisierung abgeschlossen ist, wird die Methode *start( )* aufgerufen, um die Ausführung des Applets zu starten:

```
public void start()
```

Im Gegensatz zur Initialisierung kann das Starten eines Applets mehrfach erfolgen. Wenn der Browser eine andere Web-Seite lädt, wird das Applet nicht komplett zerstört, sondern lediglich gestoppt. Bei erneutem Aufruf der Seite wird es dann wieder gestartet und die Methode *start( )* erneut aufgerufen.

Stoppen des Applets
Durch Aufrufen der Methode *stop( )* zeigt der Browser dem Applet an, dass es gestoppt werden soll:

```
public void stop()
```

Wie erwähnt, geschieht dies immer dann, wenn eine andere Seite geladen wird. Während der Lebensdauer eines Applets können die Methoden *start( )* und *stop( )* also mehrfach aufgerufen werden. Keinesfalls darf ein Applet also innerhalb von *stop( )* irgendwelche endgültigen Aufräumarbeiten durchführen und Ressourcen entfernen, die es bei einem nachträglichen Neustart wieder benötigen würde.

Zerstören des Applets
Wenn ein Applet ganz bestimmt nicht mehr gebraucht wird (z.B. weil der Browser beendet wird), ruft der Browser die Methode *destroy( )* auf:

```
public void destroy()
```

Diese kann überschrieben werden, um Aufräumarbeiten zu erledigen, die erforderlich sind, wenn das Applet nicht mehr verwendet wird. Eine typische Anwendung von *destroy( )* besteht beispielsweise darin, einen Thread zu zerstören, der bei der Initialisierung eines Applets erzeugt wurde.

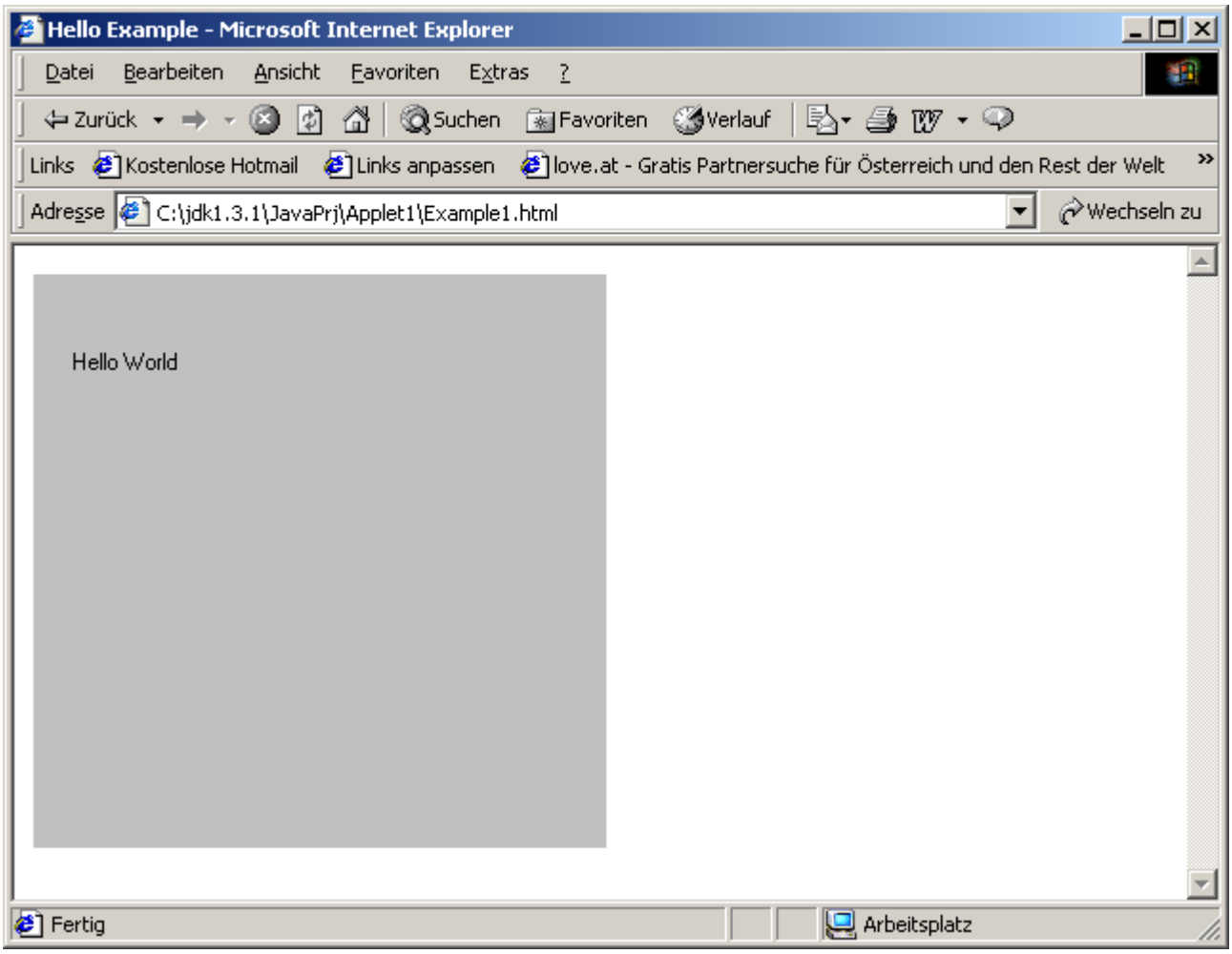
12.4 Ein erstes Beispiel

Das folgende Beispiel zeigt ein sehr einfaches Applet, das den Text »Hello, world« ausgibt:

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

public class Hello extends Applet
{
   public void paint(Graphics g)     // Diese Methode wird ausgeführt, wenn das Fenster
   {                                 // neu gezeichnet werden muss
      g.drawString("Hello, world",20,50);  // Textausgabe Position x=20,y=50
   }
}
```

Bildschirmausgabe:



12.5 Weitere Methoden der Klasse *Applet*

Methoden zum Nachrichtentransfer
Neben den speziellen Methoden kann ein Applet alle Nachrichten erhalten, die an ein Component-Objekt versendet werden. Hierzu zählen Mouse-, MouseMotion-, Key-, Focus- und Component-Events ebenso wie die Aufforderung an das Applet, seinen Clientbereich neu zu zeichnen. Bezüglich Reaktion auf die Events und die Registrierung und Programmierung geeigneter Listener-Klassen verhält sich ein Applet genauso wie jedes andere Fenster.

s**howstatus( )**
Mit der Methode *showStatus( )* kann das Applet einen Text in die Statuszeile des HTML-Browsers schreiben, der das Applet ausführt:

```
public void showStatus(String msg)
```

getParameterInfo( )
Die Klasse *Applet* besitzt eine Methode *getParameterInfo( )*, die in abgeleiteten Klassen überschrieben werden sollte:

```
public String[][] getParameterInfo()
```

getParameterInfo() kann vom Browser aufgerufen werden, um Informationen über die vom Applet akzeptierten Parameter zu erhalten. Der Rückgabewert von *getParameterInfo( )* ist ein zweidimensionales Array von Strings. Jedes Element des Arrays beschreibt einen Parameter des Applets durch ein Subarray mit drei Elementen. Das erste Element gibt den Namen des Parameters an, das zweite seinen Typ und das dritte eine textuelle Beschreibung des Parameters. Die Informationen sollten so gestaltet sein, dass sie für *menschliche* Benutzer verständlich sind; eine programmgesteuerte Verwendung ist eigentlich nicht vorgesehen.

Beispiel:
```
public String[][] getParameterInfo()
{
   String ret =
   {
      {"redwidth","int","Breite eines roten Segments"},
      {"whitewidth","int","Breite eines weissen Segments"}
   };

   return ret;
}
```

getAppletInfo( )
Ähnlich der Methode *getParameterInfo( )* gibt es eine Methode *getAppletInfo( )*, mit der die Anwendung Informationen über das Applet zur Verfügung stellen sollte:

```
public String getAppletInfo()
```

Die Sprachspezifikation gibt an, dass hier ein String zurückgegeben werden sollte, der Angaben zum Applet selbst, zur aktuellen Version und zum Autor des Applets macht.

Beispiel:
```
public String getAppletInfo()
{
   return "AppletBeispiel Ver. 1.0 (C) 2001 Manfred Muster";
}
```

12.6 Einbinden des Applets in ein HTML-Dokument

Das Einbinden eines Applets in ein HTML-Dokument erfolgt unter Verwendung des APPLET-Tags, es wird also durch <APPLET> eingeleitet und durch </APPLET> beendet. Zwischen den beiden Marken kann ein Text stehen, der angezeigt wird, wenn das Applet nicht aufgerufen werden kann. Ein applet-fähiger Browser ignoriert den Text.

Beispiel:
```
<APPLET CODE="Hello.class" WIDTH=300 HEIGHT=200>
Hier steht das Applet Hello
</APPLET>
```

Ein Applet-Tag wird wie normaler Text in die Browser-Ausgabe eingebunden. Das Applet belegt soviel Platz auf dem Bildschirm, wie durch die Größenangaben WIDTH und HEIGHT reserviert wurde. Soll das Applet in einer eigenen Zeile stehen, müssen separate Zeilenschaltungen in den HTML-Code eingebaut werden (beispielsweise <p> oder
), oder es muss ein Tag verwendet werden, dessen Ausgabe in einer eigenen Zeile steht (z.B. <h1> bis <h6>).

Neben dem Ersatztext besitzt ein Applet-Tag weitere Parameter:
- Eine Reihe von Parametern, die innerhalb von <APPLET> untergebracht werden und die Größe und Anordnung des Applets bestimmen. CODE, WIDTH und HEIGHT müssen dabei in jedem Fall angegeben werden, außerdem gibt es noch einige optionale Parameter.
- Eine Liste von PARAM-Tags, die zwischen <APPLET> und </APPLET> stehen und zur Parameterübergabe an das Applet verwendet werden.

Zwischen beiden Parameterarten besteht ein grundsätzlicher Unterschied. Während die Parameter der ersten Gruppe (also CODE, WIDTH und HEIGHT) vom Browser interpretiert werden, um die visuelle Darstellung des Applets zu steuern, werden die Parameter der zweiten Gruppe an das Applet weitergegeben. Der Browser übernimmt bei ihnen nur die Aufbereitung und die Übergabe an das Applet, führt aber selbst keine Interpretation der Parameter aus.

Die Parameter des Applet-Tags
Der wichtigste Parameter des Applet-Tags heißt CODE und gibt den Namen der Applet-Klasse an. Bei der Angabe des CODE-Parameters sind einige Dinge zu beachten:

- Der Klassenname sollte inklusive der Extension .class angegeben werden.
- Groß- und Kleinschreibung müssen eingehalten werden.
- Die Klassendatei muss im aktuellen bzw. angegebenen Verzeichnis zu finden sein.

Alternativ kann die Klassendatei auch in einem der Verzeichnisse liegen, die in der Umgebungsvariablen CLASSPATH angegeben wurden. CLASSPATH enthält eine Liste von durch Kommata getrennten Verzeichnissen, die in der Reihenfolge ihres Auftretens durchsucht werden. CLASSPATH spielt außerdem beim Aufruf des Compilers eine Rolle, die Umgebungsvariable dient dazu, die importierten Pakete zu suchen.
Das Applet-Tag hat zwei weitere nichtoptionale Parameter WIDTH und HEIGHT, die die Höhe und Breite des für das Applet reservierten Bildschirmausschnitts angeben. Innerhalb des Applets steht ein Rechteck dieser Größe als Ausgabefläche zur Verfügung.

Das Applet-Tag besitzt weitere optionale Parameter. Diese dienen zur Konfiguration des Applets und zur Beeinflussung der Darstellung des Applets und des umgebenden Textes.

Überblick über die verfügbaren Parameter:

Parameter	Bedeutung
CODEBASE	Hier kann ein alternatives Verzeichnis für das Laden der Klassendateien angegeben werden. Fehlt diese Angabe, wird das Dokumentenverzeichnis verwendet.
ARCHIVE	Angabe eines JAR-Archivs, aus dem die Klassendateien und sonstigen Ressourcen des Applets geladen werden sollen.
OBJECT	Name einer Datei, die den serialisierten Inhalt des Applets enthält
ALT	Alternativer Text für solche Browser, die zwar das Applet-Tag verstehen, aber Java nicht unterstützen
NAME	Eindeutiger Name für das Applet. Er kann zur Unterscheidung mehrerer, miteinander kommunizierender Applets auf einer Web-Seite verwendet werden.
ALIGN	Vertikale Anordnung des Applets in einer Textzeile. Hier kann einer der Werte left, right, top, texttop, middle, absmiddle, baseline, bottom oder absbottom angegeben werden.
VSPACE	Rand über und unter dem Applet
HSPACE	Rand links und rechts vom Applet

Parameterübergabe an Applets
Neben den Parametern des Applet-Tags gibt es die Möglichkeit, Parameter an das Applet selbst zu übergeben. Dazu kann innerhalb von <APPLET> und </APPLET> das optionale Tag <PARAM> verwendet werden. Jedes PARAM-Tag besitzt die beiden Parameter *name* und *value*, welche den Namen und den Wert des zu übergebenden Parameters angeben.
Innerhalb des Applets können die Parameter mit der Methode *getParameter( )* der Klasse *Applet* abgefragt werden:

```
public String getParameter(String name)
```

Für jeden angegebenen Parameter liefert *getParameter( )* den zugehörigen Wert als String. Numerische Parameter müssen vor der weiteren Verwendung also erst konvertiert werden. Wird der angegebene Parameter nicht gefunden, gibt die Methode null zurück.

Beispiel: Die Hintergrundfarbe des Applets wird durch einen Parameter übergeben
parameter.html:

```
<HTML>
   <HEAD>
      <TITLE>Applet mit Parameter</TITLE>
   </HEAD>
   <BODY>
      <H1>Mein Applet</H1>
      <HR>
      <P>
         <APPLET CODE="Parameter.class" WIDTH="300" HEIGHT="300">
         <param name=background value="grau">
         </APPLET>
      </P>
      <HR>
   </BODY>
</HTML>
```

Parameter.java:

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

public class Parameter extends Applet
{
   String background;

   public void init()
   {
      background = getParameter("Background");
      if(background.equals("grau"))
      {
         setBackground(Color.gray);
      }
      if(background.equals("gelb"))
      {
         setBackground(Color.yellow);
      }
   }

   public void paint(Graphics g)
   {
      g.drawString("Der Hintergrund ist " + background, 20, 20);
   }
}
```

Übungsaufgaben:

- Testen Sie die angeführten Beispiele durch.
- Schreiben Sie ein Applet zur Durchführung der Grundrechenarten mit Textfeldern und Buttons.
- Implementieren Sie ein einfaches Zeichenprogramm als Applet.

13. Streams

13.1 Konzept

Java stellt für den sequentiellen Datentransport eine Reihe von Klassen zur Verfügung, deren Instanzen sogenannte Streams repräsentieren. Einen Stream (Datenstrom) kann man sich wie einen Kanal von einer Datenquelle zur Software bzw. von der Software zu einem Datenziel vorstellen.
Vergleiche dazu die Standard I/O Streams in C++: cin, cout, cerr
Alle Streamklassen des Pakets *java.io* sind direkte oder indirekte Subklassen einer der vier Stream-Basisklassen: *InputStream*, *OutputStream*, *Reader* und *Writer*.

- InputStream Uninterpretierter Byte-Eingabe-Stream
- OutputStream Uninterpretierter Byte-Ausgabe-Stream
- Reader Uninterpretierter Zeichen-Eingabe-Stream
- Writer Uninterpretierter Zeichen-Ausgabe-Stream

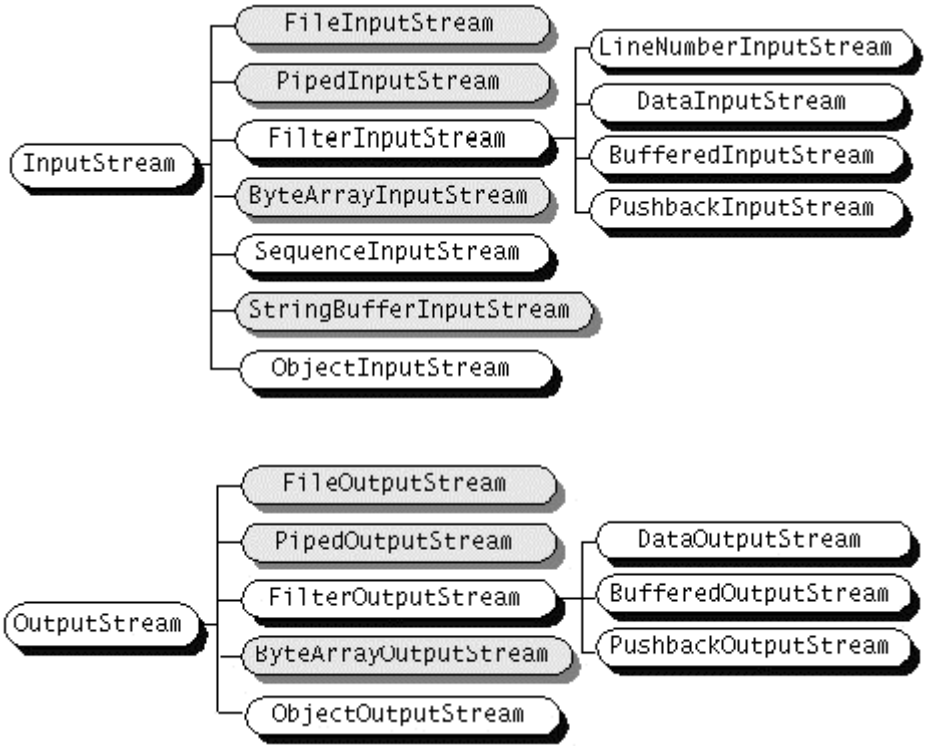
InputStream und OutputStream ermöglichen die Übertragung einzelner Bytes und kompletter Blöcke von Bytes, wogegen Reader und Writer den Transfer von einzelnen Zeichen und Blöcken von Zeichen erlauben. Die generelle Unterscheidung zwischen Bytes und Zeichen ist notwendig, da Java den Unicode-Zeichensatz verwendet, bei dem Zeichencodes 16 bit breit sind.

Datenquellen und -ziele
Für den Datentransport von bestimmten Quellen und zu bestimmten Zielen stehen in *java.io* einige Klassen zur Verfügung, welche von den Stream-Basisklassen abgeleitet sind. Als mögliche Datenquellen bzw. -ziele, sogenannte data sinks, bietet java.io Byte- und Character-Arrays, Strings bzw. StringBuffers, sowie Dateien an. Verschiedene Klassen aus *java.net* ermöglichen eine Datenübertragung in Netzwerken mittels Streams. Natürlich kann man weitere Quellen und Ziele für die streamorientierte Datenübertragung nutzbar machen, indem man eigene Subklassen der Stream-Basisklassen erzeugt.

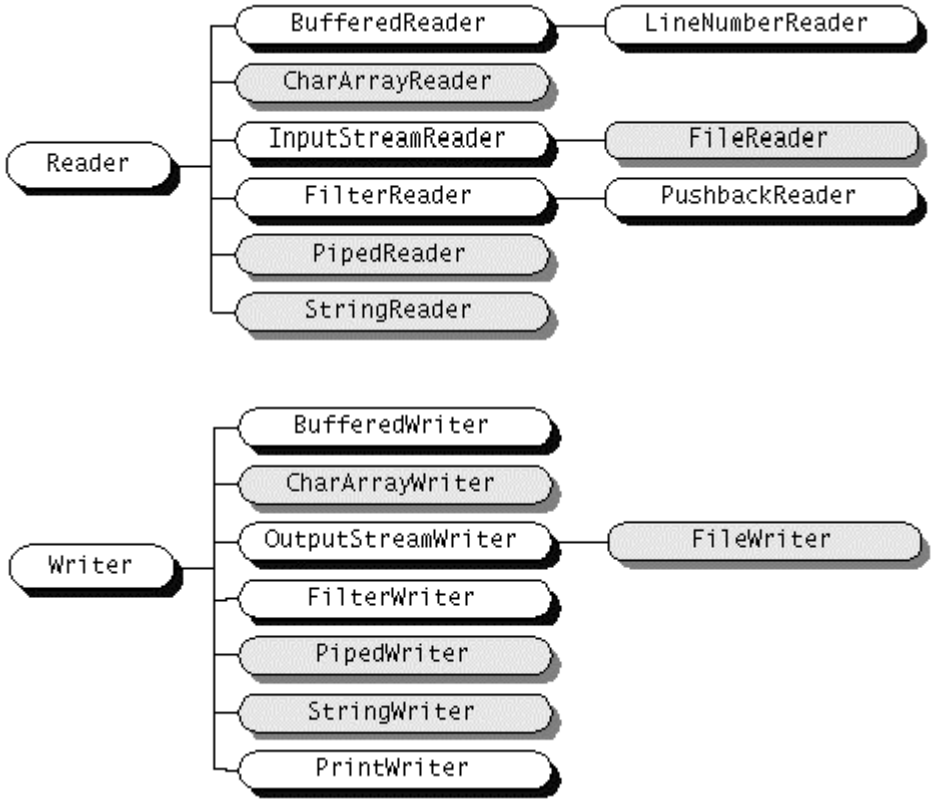
13.2 Grobübersicht der verfügbaren Stream-Klassen

Klassenhierachien:

Byte-Streams:



Character Streams:



Kurzbeschreibungen der wichtigsten Klassen:

Puffern von Daten

Die Streamklassen *BufferedInputStream*, *BufferedOutputStream*, *BufferedReader* und *BufferedWriter* sorgen für eine Zwischenspeicherung von Daten mit dem Ziel, die Anzahl der Zugriffe auf eine Datenquelle bzw. ein Datenziel herabzusetzen.

Das Filterkonzept

Ein Filterstream ist ein Stream, welcher einen anderen Stream referenziert, dessen Lese- bzw. Schreibmethoden er benutzt, um seine eigenen Lese- bzw. Schreibmethoden auszuführen. Der Filterstream kann dabei Operationen auf den gelesenen bzw. zu schreibenden Daten ausführen, er kann die Daten puffern oder auch zusätzliche Funktionalität, wie z.B. wortweises Lesen, bereitstellen.
Java stellt zur Unterstützung der Entwicklung von Filterstreams die Basisklassen *FilterInputStream*, *FilterOutputStream*, *FilterReader*, sowie *FilterWriter* zur Verfügung.

Vermitteln zwischen byte- und zeichenorientierten Streams

Die von *Reader* bzw. *Writer* abgeleiteten Klassen *InputStreamReader* und *OutputStreamWriter* stellen eine Brücke zwischen zeichenorientierten und byteorientierten Streams dar. Jede Instanz von *InputStreamReader* referenziert eine Instanz von *InputStream*. Fordert man durch Aufruf einer Lesemethode von einem *InputStreamReader*-Objekt Zeichen an, werden aus dem referenzierten InputStream Bytes gelesen und in Zeichen umgewandelt, welche dann vom InputStreamReader an den Aufrufer der Lesemethode geliefert werden. Der umgekehrte Fall besteht bei der Klasse *OutputStreamWriter*. Jedes *OutputStreamWriter*-Objekt referenziert eine Instanz von *OutputStream*. Beim Schreiben von Zeichen in einen OutputStreamWriter werden diese in Bytes konvertiert, welche in den referenzierten OutputStream ausgegeben werden.

Pipes

Pipes - wörtlich: Röhren - dienen zum Datentransport zwischen Threads. Eine Pipe besteht aus zwei Streams, einem *PipedInputStream* und einem *PipedOutputStream* bzw. einem *PipedReader* und einem *PipedWriter*, die miteinander verknüpft sind.

Textuelle Ausgabe von Daten
Mit den Klassen *PrintStream* und *PrintWriter* ist es möglich, eine Stringrepräsentation von Daten elementarer Datentypen, sowie von Objekten auszugeben. Ein PrintStream konvertiert dabei auszugebende Zeichen in Bytes bzw. Bytefolgen unter Verwendung der Standard-Zeichencodierung des jeweiligen Systems und schreibt diese in einen anderen OutputStream. Ein PrintWriter kann seine Daten sowohl in einen PrintWriter als auch in einen PrintStream schreiben.
Die Objekte *System.out* und *System.err* des Pakets java.lang sind Instanzen der Klasse PrintOutputStream.

Lesen und Schreiben von Zahlen, Strings und Wahrheitswerten
Mit den Klassen *DataInputStream* und *DataOutputStream* ist es möglich, Werte der elementaren Datentypen von Java, sowie Strings aus einem byteorientierten Stream zu lesen bzw. in einen solchen zu schreiben.

Lesen und Schreiben von Files
Die Klassen *FileReader* und *FileWriter* dienen zum Lesen und Schreiben von Textfiles, analog dazu existieren die Klassen *FileInputStream* sowie *FileOutputstream* zur Bearbeitung von Binärfiles. Zur Angabe von Dateinamen und Verzeichnissen wird die Klasse *File* verwendet.

Lesen und Schreiben beliebiger Objekte
Die Klassen *ObjectInputStream* und *ObjectOutputStream* besitzen die gleiche Funktionalität wie *DataInputStream* und *DataOutputStream*, ermöglichen aber auch noch die Ein- bzw. Ausgabe von Objekten anderer Klassen als String.

ANMERKUNG:
Das Stream-Klassenkonzept von Java bietet die Möglichkeit, Streams zu verketten oder zu schachteln. Die Verkettung von Streams ermöglicht es, mehrere Dateien zusammenzufassen und für den Aufrufer als einen einzigen Stream darzustellen. Das Schachteln von Streams erlaubt die Konstruktion von Filtern, die bei der Ein- oder Ausgabe bestimmte Zusatzfunktionen übernehmen, beispielsweise das Puffern von Zeichen, das Zählen von Zeilen oder die Interpretation binärer Daten. Beide Konzepte sind mit normalen Sprachmitteln realisiert und können selbst erweitert werden. Es ist ohne weiteres möglich, eigene Filter zu schreiben, die den Ein- oder Ausgabestrom analysieren und anwendungsbezogene Funktionalitäten realisieren.

<u>Einige wichtige Methoden für Streams:</u>

read( )	Lesen aus einem Stream. Die Methode existiert in vielen Varianten. Alle *read( )*-Methoden sind so aufgebaut, dass sie warten müssen, bis *alle* geforderten Eingaben verfügbar sind.Wird zB. beim Einlesen in einen Puffer das Streamende erreicht, so wird die Anzahl der tatsächlich gelesenen Bytes geliefert. Ein nochmaliger Aufruf liefert –1 (Streamende erreicht).
write( )	Schreiben in einen Stream. Alle Varianten müssen warten, bis die *gesamte* angeforderte Ausgabe geschrieben ist. Analog zu *read( )* kann auch ein Versatz in einem Puffer angegeben werden (Ausgabe eine Pufferteils).
available( ) **ready( )**	Liefern die Anzahl der Bytes bzw. Zeichen, die ohne Blockierung (d.h. ohne Warten gelesen werden können. *Beachte*: Manche Streams liefern immer 0 zurück (liegt in der Natur dieser Streams).
flush( )	Leerung der Ausgabe durch gepufferten Cache (hat nicht jeder Ausgabestream!)
close( )	Schließen des Streams sowie Freigabe der damit verbundenen Ressourcen.

13.3 Beispiele

13.3.1 Eingabe von der Systemkonsole

Der vordefinierte Eingabe-Stream *System.in* ist nicht in der Lage, die eingelesenen Zeichen in primitive Datentypen zu konvertieren. Statt dessen muss zunächst eine Instanz der Klasse *InputStreamReader* und daraus ein *BufferedReader* erzeugt werden. Dieser kann dann dazu verwendet werden, die Eingabe zeilenweise zu lesen und das Ergebnis in einen primitiven Typ umzuwandeln.

Beispiel:

```
import java.io.*;

public class Eingabe
{
   public static void main(String[] args)
   {
      int a, b, c;

      try
      {
         BufferedReader din = new BufferedReader(new InputStreamReader(System.in));
         System.out.println("Bitte a eingeben: ");
         a = Integer.parseInt(din.readLine());
         System.out.println("Bitte b eingeben: ");
         b = Integer.parseInt(din.readLine());
         c = a + b;
         System.out.println("a+b="+c);
      }
      catch (Exception e)
      {
         System.out.println(e.toString());   // Automatische Fehlermeldung
      }
   }
}
```

Exception Handling:

Java erweitert das bereits in C++ eingeführte Konzept des *Structured Exception Handling*. Statt für jede Programmzeile Fehlerabfragen durchzuführen, was den Quellcode äußerst unleserlich macht, können komplette Programmblöcke zu einer Einheit zusammengefaßt werden, und alle möglichen Maßnahmen bei Auftreten von Ausnahmesituationen, welche beim Ablauf dieses Programmblocks behandelt werden sollen, an einer zentralen Stelle auscodiert werden.

Syntax:

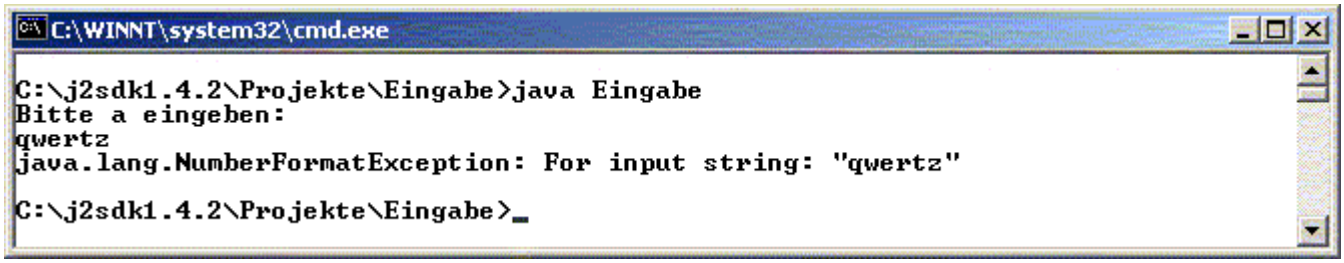
```
try
{
  Anweisung; ...
}
catch(Ausnahmetyp x)
{
  Anweisung; ...
}
```

Der try-Block enthält eine oder mehrere Anweisungen, bei deren Ausführung ein Fehler des Typs *Ausnahmetyp* auftreten kann. In diesem Fall wird die normale Programmausführung unterbrochen, und der Programmablauf fährt mit der ersten Anweisung nach der catch-Klausel fort, die den passenden Ausnahmetyp deklariert hat. Hier kann nun Code untergebracht werden, der eine angemessene Reaktion auf den Fehler realisiert.
In C++ ist dies optional möglich, Java geht so weit, dass es für viele Operationen eine Ausnahmebehandlung zwingend vorschreibt! Beispiel: Methode *readLine( )* der Klasse *BufferedReader( ).*

Falls man obigen Quellcode ohne *try-catch* implementiert, so erzeugt der Java-Compiler folgende Fehlermeldung für Zeile 13:

```
C:\j2sdk1.4.2\JavaPrj\Eingabe\Eingabe.java:13: unreported exception java.io.IOExceptio
n; must be caught or declared to be thrown
                  a = Integer.parseInt(din.readLine());
```

Beispiel für das Autreten einer Exception (statt einer Zahl wird ein Text eingegeben):



Anstatt die Ausnahme mit *try-catch* abzufangen, besteht für die Methode alternativ die Möglichkeit, die Ausnahme weiterzureichen (werfen ⇒ *throw*). In diesem Fall muss sich der Aufrufer des Programmblocks um die Ausnahme kümmern.

Beispiel: `public static void main(String[] args) throws IOException ...`

Die Anzahl der möglichen Exceptions ist enorm, die dazugehörigen Klassen sind wie erwartet in einer Klassenhierachie zu finden (siehe SDK-Hilfe).

13.3.2 Erzeugen eines Textfiles

```
import java.io.*;
public class TextFile
{
  public static void main(String[] args)
  {
    BufferedWriter f;
    String s;
    try
    {
      f = new BufferedWriter(new FileWriter("buffer.txt"));
      for (int i = 1; i <= 10; ++i)
      {
        s = "Dies ist die " + i + ". Zeile";
        f.write(s);
        f.newLine();
      }
      f.close();
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Erstellen der Datei");
    }
  }
}
```

Übungsaufgaben:

- Schreiben Sie ein Programm, das Zeichen für Zeichen den Text einer Datei einliest und die Häufigkeit der darin vorkommenden Vokale a,e,i,o,u bestimmt (oder allgemein für ein bestimmtes Textzeichen in %).
- Schreiben Sie ein Programm, das eine Datei kopiert.
- Schreiben Sie ein Programm, das Textdateien mit Zeilennummern versieht.

14. Threads

14.1 Grundlagen

Kaum eine wichtige Programmiersprache der letzten Jahre hat das Konzept der *Nebenläufigkeit* innerhalb der Sprache implementiert. Mit Nebenläufigkeit bezeichnet man die Fähigkeit eines Systems, zwei oder mehr Vorgänge gleichzeitig oder quasi-gleichzeitig ausführen zu können. Lediglich ADA stellt sowohl parallele Prozesse als auch Mechanismen zur Kommunikation und Synchronisation zur Verfügung, die direkt in die Sprache eingebettet sind. Durch Weiterentwicklungen im Bereich der Betriebssystemtechnologie wurde allerdings das Konzept der *Threads* immer populärer und auf der Basis von Library-Routinen auch konventionellen Programmiersprachen zur Verfügung gestellt.

Java hat Threads direkt in die Sprache integriert und mit den erforderlichen Hilfsmitteln als Konstrukt zur Realisierung der Nebenläufigkeit implementiert. Ein Thread ist ein eigenständiges Programmfragment, das parallel zu anderen Threads laufen kann. Ein Thread ähnelt damit einem *Prozeß*, arbeitet aber auf einer feineren Ebene. Während ein Prozeß das Instrument zur Ausführung eines kompletten Programms ist, können innerhalb dieses Prozesses mehrere Threads parallel laufen. Der Laufzeit-Overhead zur Erzeugung und Verwaltung eines Threads ist relativ gering und kann in den meisten Programmen vernachlässigt werden. Ein wichtiger Unterschied zwischen Threads und Prozessen ist der, dass alle Threads eines Programmes sich einen gemeinsamen Adreßraum teilen, also auf die selben Variablen zugreifen, während die Adreßräume unterschiedlicher Prozesse streng voneinander getrennt sind.

Die Implementierung von Threads war eine explizite Anforderung an das Design der Sprache. Threads sollen unter anderem die Implementierung grafischer Anwendungen erleichtern, die durch Simulationen komplexer Abläufe oft inhärent nebenläufig sind. Threads dienen auch dazu, die Bedienbarkeit von Dialoganwendungen zu verbessern, indem rechenintensive Anwendungen im Hintergrund ablaufen.

Threads werden in Java durch die Klasse *Thread* und das Interface *Runnable* implementiert. In beiden Fällen wird der Thread-Body, also der parallel auszuführende Code, in Form der überschriebenen Methode *run( )* zur Verfügung gestellt. Die Kommunikation kann dann durch Zugriff auf die Instanz- oder Klassenvariablen oder durch Aufruf beliebiger Methoden, die innerhalb von *run( )* sichtbar sind, erfolgen. Zur Synchronisation stellt Java das aus der Betriebssystemtheorie bekannte Konzept des *Monitors* zur Verfügung, das es erlaubt, kritische Regionen innerhalb korrekt geklammerter Programmfragmente und Methoden zu kapseln und so den Zugriff auf gemeinsam benutzte Datenstrukturen zu koordinieren.

Darüber hinaus bietet Java Funktionen zur Verwaltung von Threads. Diese erlauben es, Threads in Gruppen zusammenzufassen, zu priorisieren und Informationen über Eigenschaften von Threads zu gewinnen. Das Scheduling kann dabei wahlweise *preemptiv* oder *nichtpreemptiv* implementiert sein. Die Sprachspezifikation legt dies nicht endgültig fest, aber in den meisten Java-Implementierungen wird dies von den Möglichkeiten des darunter liegenden Betriebssystems abhängen.

14.2 Erstellen von Threads

1. Variante

- Ableitung einer Klasse von der Klasse Thread

```
class MyThread extends Thread
{
  public void run()
  {
    ...  // Thread-Tätigkeit
  }
}
```

- Erstellung und Ausführung

```
MyThread m1 = new MyThread();
M1.start(); // Aufruf der Methode run()
```

2. Variante
Aufgrund der Einfachvererbung in Java ist Variante 1 nicht immer möglich. So müssen zB. Applets von der Klasse *Applet* abgeleitet sein, eine weitere Ableitung von der Klasse *Thread* ist nicht mehr möglich, da Mehrfachvererbung nicht unterstützt wird. In diesem Fall hat man das Interfacee *runnable* zu implementieren:

- Implementierung des Interfaces *Runnable*

```
class MyApplet extends Applet implements Runnable
{
  public void run()
  {
    ...  // Thread-Tätigkeit
  }
}
```

- Eine Instanz von Thread muss erstellt werden und die *run( )*-Methode muss aufgerufen werden, zB. im Konstruktor (hier der Klasse MyApplet)

```
Thread myThread = new Thread(this); // Anderer Konstruktor!
myThread.start();
```

oder in einer anderen Klasse

```
class MyClass extends ImportKlasse implements Runnable
{
  ...
}

class AnotherClass
{
  MyClass myClass = new MyClass(); // Neue Instanz der Klasse MyClass
  Thread t1 = new Thread(myClass); // Diese an Thread übergeben
  t1.start();                      // Thread starten
  ...
}
```

14.3 Abbrechen eines Threads

Normalerweise wird ein Thread dadurch beendet, dass das Ende seiner *run( )*-Methode erreicht ist. In manchen Fällen ist es jedoch erforderlich, den Thread von außen abzubrechen. Die bis zum SDK 1.1 übliche Vorgehensweise bestand darin, die Methode *stop( )* der Klasse Thread aufzurufen. Dadurch wurde der Thread abgebrochen und aus der Liste der aktiven Threads entfernt.
Mit dem SDK 1.2 wurde die Methode *stop( )* als deprecated markiert, d.h. sie sollte nicht mehr verwendet werden. Der Grund dafür liegt in der potentiellen Unsicherheit des Aufrufs, denn es ist nicht voraussagbar und auch nicht definiert, an welcher Stelle ein Thread unterbrochen wird, wenn ein Aufruf von *stop( )* erfolgt. Es kann nämlich insbesondere vorkommen, dass der Abbruch innerhalb einer kritischen Region erfolgt (die mit dem *synchronized*-Schlüsselwort geschützt wurde) oder in einer anwendungsspezifischen Transaktion auftritt, die aus Konsistenzgründen nicht unterbrochen werden darf.

Eine alternative Methode, einen Thread abzubrechen, besteht darin, im Thread selbst auf Unterbrechungsanforderungen zu reagieren. So könnte beispielsweise eine Membervariable *cancelled* eingeführt und beim Initialisieren des Threads auf *false* gesetzt werden. Mit Hilfe einer Methode *cancel( )* kann der Wert der Variable zu einem beliebigen Zeitpunkt auf *true* gesetzt werden. Aufgabe der Bearbeitungsroutine in *run( )* ist es nun, an geeigneten Stellen diese Variable abzufragen und für den Fall, dass sie *true* ist, die Methode *run( )* konsistent zu beenden.

In der Klasse Thread gibt es Methoden, die einen solchen Mechanismus standardmäßig unterstützen:

```
public void interrupt()
public boolean isInterrupted()
public static boolean interrupted()
```

Durch Aufruf von *interrupt( )* wird ein Flag gesetzt, das eine Unterbrechungsanforderung signalisiert. Durch Aufruf von *isInterrupted( )* kann der Thread festellen, ob das Abbruchflag gesetzt wurde und der Thread beendet werden soll. Die statische Methode *interrupted( )* stellt den Status des Abbruchsflags beim aktuellen Thread fest. Ihr Aufruf entspricht dem Aufruf von *currentThread( ).isInterrupted( )*, setzt aber zusätzlich das Abbruchflag auf seinen ursprünglichen Wert *false* zurück.

Beispiel: In einem separaten Thread werden ununterbrochen Textzeilen auf dem Bildschirm ausgegeben. Das Hauptprogramm soll den Thread erzeugen und nach 2 Sekunden durch einen Aufruf von *interrupt( )* eine Unterbrechungsanforderung erzeugen. Der Thread gibt die aktuelle Zeile fertig aus und terminiert.

```
public class Text extends Thread
{
   int count = 0;
   public void run()
   {
      while (true)
      {
         if (isInterrupted())
         {
            break;
         }
         printLine(++count);
      }
   }
   private void printLine(int count)
   {
      System.out.print(count + ": ");      //Zeile ausgeben
      for (int i = 0; i < 30; ++i)
      {
         System.out.print(i == count % 30 ? "* " : ". ");
      }
      System.out.println();
      try
      {
         Thread.sleep(100);     //100 ms warten
      }
      catch (InterruptedException e)
      {
         interrupt();
      }
   }
   public static void main(String[] args)
   {
      Text t = new Text();
      {
         t.start(); //Thread starten
         try
         {
            Thread.sleep(2000); // 2 Sekunden warten
         }
         catch (InterruptedException e) {}
         t.interrupt();   //Thread unterbrechen
      }
   }
}
```

Man erkennt, dass der Thread tatsächlich parallel zum Hauptprogramm ausgeführt wird. Nach dem Aufruf von *start()* beginnt einerseits die Zählschleife mit der Bildschirmausgabe, aber gleichzeitig fährt das Hauptprogramm mit dem Aufruf der *sleep( )*-Methode fort. Beide Programmteile laufen also parallel ab.
Würde die *stop( )*-Methode zum Abbruch des Treads verwendet werden, so wäre die Programmausgabe bei Threadabbruch völlig undefiniert, d.h. es könnte die letzte Zeile nur teilweise ausgegeben werden!
ANMERKUNG: Falls der Aufruf von *interrupt( )* während des Aufrufs von *sleep( )* geschieht, so wird die *sleep( )*-Methode mit einer *InterruptedException* beendet und das Abbruchflag wird dabei gelöscht, daher wird innerhalb der *catch*-Klausel die Methode *interrupt( )* erneut aufgerufen!

14.4 Anhalten eines Threads

Die Klasse *Thread* besitzt zwei Methoden *suspend( )* und *resume( )*, mit deren Hilfe es möglich ist, einen Thread vorübergehend anzuhalten und anschließend an der Unterbrechungsstelle fortzusetzen. Beide Methoden sind nicht ganz ungefährlich und können unbemerkt Deadlocks verursachen. Sie wurden daher mit dem SDK 1.2 als deprecated markiert und sollten nicht mehr verwendet werden. Ihre Funktionalität sollte - wenn erforderlich - manuell nachgebildet werden.

14.5 Synchronisationsprobleme

Wenn man sich mit Nebenläufigkeit beschäftigt, muss man sich in aller Regel auch mit Fragen der *Synchronisation* nebenläufiger Prozesse beschäftigen. In Java erfolgt die Kommunikation zweier Threads auf der Basis gemeinsamer Variablen, die von beiden Threads erreicht werden können. Führen beide Prozesse Änderungen auf den gemeinsamen Daten durch, so müssen sie synchronisiert werden, denn andernfalls können undefinierte Ergebnisse entstehen.

Beispiel: Zwei Threads zählen einen gemeinsamen Zähler hoch:

```
public class Test extends Thread
{
   static int count = 0;

   public static void main(String[] args)
   {
      Thread t1 = new Test();
      Thread t2 = new Test();
      t1.start();
      t2.start();
   }

   public void run()
   {
      while (true)
      {
         count++;
         for(int i = 0; i < 10000000; i++);
         System.out.println(count);
      }
   }
}
```

Die Programmausgabe ergibt nun nicht wie erwartet, eine Folge von aufsteigenden Zahlen, mitunter kann es passieren, dass eine Zahl übersprungen wird, oder doppelt ausgegeben wird.
Beide Prozesse greifen unsynchronisiert auf die gemeinsame Klassenvariable *count* zu. Da die Zählschleife für i nicht *atomar* ist, kommt es zu dem Fall, dass diese mitten in der Ausführung unterbrochen wird und der Scheduler mit dem anderen Thread fortfährt. Um den Effekt zu verstärken, wurde mit Absicht eine Schleife mit sehr vielen Durchläufen eingebaut.
Um diese Art von Inkonsistenzen zu beseitigen, bedarf es der *Synchronisation* der beteiligten Prozesse.

Monitore
Zur Synchronisation nebenläufiger Prozesse hat Java das Konzept des *Monitors* implementiert. Ein Monitor ist die Kapselung einer *kritischen Region* (also eines Programmteils, der nur von jeweils einem Thread zur Zeit durchlaufen werden darf) mit Hilfe einer automatisch verwalteten Sperre. Diese Sperre wird beim Betreten des Monitors gesetzt und beim Verlassen wieder zurückgenommen. Ist sie beim Eintritt in den Monitor bereits von einem anderen Thread gesetzt, so muss der aktuelle Thread warten, bis der Konkurrent die Sperre freigegeben und den Monitor verlassen hat.

Das Monitor-Konzept wird mit Hilfe des in die Sprache integrierten Schlüsselworts *synchronized* realisiert. Durch *synchronized* kann entweder eine komplette Methode oder ein Block innerhalb einer Methode geschützt werden. Der Eintritt in den so deklarierten Monitor wird durch das Setzen einer Sperre auf einer Objektvariablen erreicht. Bezieht sich *synchronized* auf eine komplette Methode, wird als Sperre die *this*-Referenz verwendet, andernfalls ist eine Objektvariable explizit anzugeben.

Anwendung von *synchronized* auf einen Block von Anweisungen
Die naheliegende Lösung, die Anweisungen durch einen synchronized-Block auf der Variablen this zu synchronisieren, funktioniert leider nicht. Da die Referenz this für jeden der beiden Threads, die ja unterschiedliche Instanzen repräsentieren, neu vergeben wird, wäre für jeden Thread der Eintritt in den Monitor grundsätzlich erlaubt.
Statt dessen verwendet man zB. die Methode *getClass( )*, die uns ein Klassenobjekt beschafft (ein und dasselbe für alle Instanzen), mit dem die Klassenvariable *count* geschützt werden kann.

```
public class Test extends Thread
{
   static int count = 0;

   public static void main(String[] args)
   {
      Thread t1 = new Test();
      Thread t2 = new Test();
      t1.start();
      t2.start();
   }
   public void run()
   {
      while (true)
      {
         synchronized(getClass())
         {
            count++;
            for(int i = 0; i < 10000000; i++);
            System.out.println(count);
         }
      }
   }
}
```

Anwendung von *synchronized* auf eine Methode
Alternativ könnte eine Methode *inkrement( )* atomar definiert werden, welche innerhalb der *run( )*-Methode zyklisch aufgerufen wird. Diese muss allerdings static deklariert werden, da die Methode ja für beide Thread-Instanzen sonst extra existieren würde und keine Synchronisation erfolgen würde!

```
static public synchronized void inkrement()
{
   count++;
   for(int i = 0; i < 10000000; i++);
   System.out.println(count);
}
```

wait( )* und *notify( )

Neben dem Monitorkonzept stehen mit den Methoden *wait( )* und *notify( )* der Klasse *Object* noch weitere Synchronisationsprimitive zur Verfügung. Zusätzlich zu der bereits erwähnten Sperre, die einem Objekt zugeordnet ist, besitzt ein Objekt nämlich auch noch eine *Warteliste*. Dabei handelt es sich um eine (möglicherweise leere) Menge von Threads, die vom Scheduler unterbrochen wurden und auf ein Ereignis warten, um fortgesetzt werden zu können.

Sowohl *wait( )* als auch *notify( )* dürfen nur aufgerufen werden, wenn das Objekt bereits gesperrt ist, also nur innerhalb eines *synchronized*-Blocks für dieses Objekt. Ein Aufruf von *wait( )* nimmt die bereits gewährten Sperren (temporär) zurück und stellt den Thread, der den Aufruf von *wait( )* verursachte, in die Warteliste des Objekts. Dadurch wird er unterbrochen und im Scheduler als *wartend* markiert. Ein Aufruf von *notify( )* entfernt einen (beliebigen) Thread aus der Warteliste des Objekts, stellt die (temporär) aufgehobenen Sperren wieder her und führt ihn dem normalen Scheduling zu. *wait( )* und *notify( )* sind damit für elementare Synchronisationsaufgaben geeignet, bei denen es weniger auf die Kommunikation als auf die Steuerung der zeitlichen Abläufe ankommt.

Beispiel: *Producer/Consumer-Problem*. Ein Prozeß arbeitet dabei als Produzent, der Fließkommazahlen »herstellt«, und ein anderer als Konsument, der die produzierten Daten verbraucht. Die Kommunikation zwischen beiden erfolgt über ein gemeinsam verwendetes *Vector*-Objekt, das die produzierten Elemente zwischenspeichert und als Medium für die *wait( )*-/*notify( )*-Aufrufe dient:

```
import java.util.*;

class Producer extends Thread
{
  private Vector v;
  public Producer(Vector v)
  {
    this.v = v;
  }
  public void run()
  {
    String s;
    while (true)
    {
      synchronized(v)
      {
        s = "Wert " + Math.random();
        v.addElement(s);
        System.out.println("Produzent erzeugte " + s);
        v.notify();
      }
      try
      {
        Thread.sleep((int)(100*Math.random()));
      }
      catch (InterruptedException e)
      {
      }
    }
  }
}

class Consumer extends Thread
{
  private Vector v;
  public Consumer(Vector v)
  {
    this.v = v;
  }
```

```
  public void run()
  {
    while(true)
    {
      synchronized(v)
      {
        if(v.size() < 1)
        {
          try
          {
            v.wait();
          }
          catch (InterruptedException e)
          {
          }
        }
        System.out.print(" Konsument fand " + (String)v.elementAt(0));
        v.removeElementAt(0);
        System.out.println(" (verbleiben: " + v.size() + ")");
      }
      try
      {
        Thread.sleep((int)(100*Math.random()));
      }
      catch (InterruptedException e)
      {
      }
    }
  }
}
public class Test
{
  public static void main(String[] args)
  {
    Vector v = new Vector();
    Producer p = new Producer(v);
    Consumer c = new Consumer(v);
    p.start();
    c.start();
  }
}
```

Wirkungsweise: Mit *synchronized(v)* erhält ein Thread den Monitor, d.h. ein Zugriff auf v ist für andere Threads einstweilen nicht möglich (Sperre). Die Sperre wird erst bei Verlassen des kritischen Bereichs aufgehoben. Betritt nun der Consumer-Thread den kritischen Bereich, so ist für den Producer der Zugriff auf v gesperrt, er muss warten. Falls nun der Consumer kein Erzeugnis antrifft und darauf wartet, würde eine typische Deadlock-Situation auftreten. Durch *wait( )* wird die Sperre kurzfristig aufgehoben und zwar solange, bis eine Benachrichtigung mittels *notify( )* (Producer hat ein neues Produkt erzeugt) erfolgt.
Um die Arbeitsverteilung zwischen den Prozessen etwas interessanter zu gestalten, werden beide gezwungen, nach jedem Schritt eine kleine Pause einzulegen. Da die Wartezeit zufällig ausgewählt wird, kann es durchaus dazu kommen, dass der Produzent eine größere Anzahl an Elementen anhäuft, die der Konsument noch nicht abgeholt hat. Der umgekehrte Fall ist natürlich nicht möglich, da der Konsument warten muss, wenn keine Elemente verfügbar sind.
wait( ) und *notify( )* sind für elementare Synchronisationsaufgaben geeignet, bei denen es weniger auf die Kommunikation als auf die Steuerung der zeitlichen Abläufe ankommt.

Übungsaufgaben:
- Testen Sie die angeführten Beispiele durch.
- Schreiben Sie ein Applet, welches laufend Datum und Uhrzeit ausgibt.
- Schreiben Sie ein Applet für eine einfache Animation (zB. springender Punkt).

15. Netzwerkprogrammierung

15.1 Protokolle

Um überhaupt Daten zwischen zwei oder mehr Partnern austauschen zu können, müssen sich die Teilnehmer auf ein gemeinsames *Protokoll* geeinigt haben. Als *Protokoll* bezeichnet man die Menge aller Regeln, die notwendig sind, um einen kontrollierten und eindeutigen Verbindungsaufbau, Datenaustausch und Verbindungsabbau gewährleisten zu können. Es gibt sehr viele Protokolle mit sehr unterschiedlichen Ansprüchen. Ein weitverbreitetes Architekturmodell, das *ISO/OSI-7-Schichten-Modell*, unterteilt sie in Abhängigkeit von ihrem Abstraktionsgrad in sieben Schichten:

Die derzeit in Java verfügbaren Netzwerkfähigkeiten basieren alle auf den Internet-Protokollen TCP/IP (Transport Control Protocol/Internet Protocol) (bzw. TCP/UDP User Datagram Protocol). Dieses Kapitel beschäftigt sich ausschließlich mit der TCP/IP-Familie von Protokollen. Hierfür wird häufig eine etwas vereinfachte Unterteilung in vier Ebenen vorgenommen:

- Die unterste Ebene repräsentiert die physikalischen Geräte. Sie wird durch die Netzwerkhardware und -leitungen und die unmittelbar darauf laufenden Protokolle wie beispielsweise *Ethernet*, *FDDI* oder *ATM* repräsentiert.
- Die zweite Ebene repräsentiert die *Netzwerkschicht*. Sie wird in TCP/IP-Netzen durch das IP-Protokoll und dessen Kontrollprotokolle (z.B. *ICMP*) implementiert.
- Die dritte Ebene stellt die *Transportschicht* dar. Sie wird durch die Protokolle TCP bzw. UDP repräsentiert.
- Die oberste Ebene steht für die große Klasse der *Anwendungsprotokolle*. Dazu zählen beispielsweise FTP zum Filetransfer, SMTP zumMail-Versand und HTTP zur Übertragung von Web-Seiten.

Das ISO/OSI-7-Schichten-Modell

Layer 7: Application Layer (Applikationsschicht
Layer 6: Presentation Layer (Darstellungsschicht)
Layer 5: Session Layer (Sitzungsschicht)
Layer 4: Transport Layer (Transportschicht)
Layer 3: Network Layer (Netzwerkschicht)
Layer 2: Data Link Layer (Datensicherungsschicht)
Layer 1: Physical Layer (Physikalische Schicht)

Das vereinfachte 4-Ebenen-Modell

Layer 4: Applikationschicht
Layer 3: Transportschicht (TCP)
Layer 2: Netzwerkschicht (IP)
Layer 1: Verbindungsschicht

Die TCP/IP-Protokollfamilie wird sowohl in lokalen Netzen (Intranet) als auch im Internet verwendet. Alle Eigenarten des Transportweges werden auf der zweiten bzw. dritten Ebene ausgeglichen, und die Anwendungsprotokolle merken *prinzipiell* keinen Unterschied zwischen lokalen und globalen Verbindungen. Wurde beispielsweise ein SMTP-Mailer für den Versand von elektronischer Post in einem auf TCP/IP basierenden Unternehmensnetz entwickelt, so kann dieser im Prinzip auch dazu verwendet werden, eine Mail ins Internet zu versenden.

- Das IP-Protokoll stellt eine ungesicherte Punkt-zu-Punkt-Verbindung zwischen 2 Rechnern in unterschiedlichen Netzwerken (Internet!) zur Verfügung.
- TCP ist ein *verbindungsorientiertes* Protokoll, das auf der Basis von IP eine sichere und fehlerfreie Punkt-zu-Punkt-Verbindung realisiert.

Daneben gibt es ein weiteres Protokoll oberhalb von IP, das als *UDP* (*User Datagram Protocol*) bezeichnet wird. Im Gegensatz zu TCP ist UDP ein *verbindungsloses* Protokoll, bei dem die Anwendung selbst dafür sorgen muss, dass Pakete fehlerfrei und in der richtigen Reihenfolge beim Empfänger ankommen. Sein Vorteil ist die größere Geschwindigkeit gegenüber TCP. In Java wird UDP durch die Klassen *DatagramPacket* und *DatagramSocket* abgebildet. Im folgenden werden allerdings ausschließlich die populäreren Protokolle auf der Basis von TCP/IP behandelt.

15.2 Adressierung von Daten

IP-Adressen

Der Absender von Daten hat anzugeben, wohin die Daten zu transportieren sind, der Empfänger sollte erkennen, von wem die Daten stammen. In TCP/IP-Netzwerken (Internet, Intranet) erfolgt die Adressierung auf IP-Ebene (Schicht 3) mit einer 32 bit langen IP-Adresse (vgl. IPv6 128 bit).
Eine IP-Adresse besteht aus Netzwerk-ID und Host-ID, je nach Größe der Host-ID unterscheidet man:

Klasse A: Netzwerk-ID 8 bit (beginnt 0...) Host-ID 24 bit (extrem große Netze)
Klasse B: Netzwerk-ID 16 bit (beginnt 10...) Host-ID 16 bit (mittlere Netze bis ca. 65000 Rechner)
Klasse C: Netzwerk-ID 24 bit (beginnt 110...) Host-ID 8 bit (kleine Netze bis ca. 250 Rechner)
Beispiele: 192.168.1.7, 127.0.0.1 (local host)...Beide werden nicht ins Internet geroutet

Domain-Namen

Während IP-Adressen für Computer sehr leicht zu verarbeiten sind, gilt das nicht unbedingt für die Menschen, die damit arbeiten müssen. Wer kennt schon die Telefonnummern und Ortsnetzkennzahlen von allen Leuten, mit denen er Telefonate zu führen pflegt? Um die Handhabung der IP-Adressen zu vereinfachen, wurde daher das *Domain Name System* eingeführt (kurz *DNS*), das numerischen IP-Adressen sprechende Namen wie *www.htl.at* oder *java.sun.com* zuordnet.
Anstelle der IP-Adresse können bei den Anwendungsprotokollen nun wahlweise die symbolischen Namen verwendet werden. Sie werden mit Hilfe von *Name-Servern* in die zugehörige IP-Adresse übersetzt, bevor die Verbindung aufgebaut wird. Zwar kann sich hinter ein und derselben IP-Adresse mehr als ein Name-Server-Eintrag befinden. In umgekehrter Richtung ist die Zuordnung aber eindeutig, d.h. zu jedem symbolischen Namen kann eindeutig die zugehörige IP-Adresse (und damit das Netz und der Host) bestimmt werden, zu der der Name gehört (Beispiel www.htl.at ⇒ 193.154.0.16).

15.3 Ports und Applikationen

Bei den üblichen Internet-Diensten läuft die Kommunikation zwischen zwei Rechnern auf der Basis einer Client-Server-Beziehung ab:

- Der Server stellt einen Dienst zur Verfügung, der von anderen Rechnern genutzt werden kann. Dabei läuft er im Hintergrund und wartet, dass eine Verbindung von einem anderen Rechner zu ihm aufgebaut wird. Das Protokoll wird vom Server festgelegt. Meistens laufen mehrere Server-Dienste auf einem Rechner.
- Der Client baut bei Bedarf nach einem Dienst (zB. Abruf einer Web-Seite) eine Verbindung zu einem geeigneten Server auf. Die Adresse dieses Servers muss bekannt sein, der Client muss sich an das Protokoll halten, damit der Datenaustausch funktioniert.

Um die unterschiedlichen Dienste auf einem Server unterscheiden zu können, gibt es ein weiteres Adressierungsmerkmal: die Portnummer. Diese ist 16 bit breit und erlaubt somit Werte von 0...65535, wobei die Nummern 0...1023 auf UNIX-Maschinen für Anwendungen mit Supervisor(root)-Rechten reserviert sind. Viele Dienste haben eine reservierte Portnummer (sogenannte *well known ports*).

Einige *well known ports*:

Name	Port	Transport	Beschreibung
Echo	7	tcp/udp	Gibt jede Zeile zurück, die der Client sendet
Daytime	13	tcp/udp	Liefert ASCII-String mit Datum und Uhrzeit
Chargen	19	tcp/udp	Generiert ununterbrochen Zeichen
Ftp	21	tcp	Versenden und Empfangen von Dateien
Telnet	23	tcp	Interaktive Session mit entferntem Host
Smtp	25	tcp	Versenden von E-Mails
Time	37	tcp/udp	Liefert die aktuelle Uhrzeit als Anzahl der Sekunden seit 1.1.1900
Whois	43	tcp	Einfacher Namensservice
Gopher	70	tcp/udp	Quasi-Vorgänger von WWW
Finger	79	tcp	Liefert Benutzerinformationen
Www	80	tcp/udp	Der Web-Server
Pop3	110	tcp/udp	Übertragen von Mails
Nntp	119	tcp	Übertragen von Usenet-News

15.4 Request for Comments

Die meisten der allgemein zugänglichen Protokolle sind in sogenannten *Request For Comments* (kurz *RFCs*) beschrieben. RFCs sind Dokumente des *Internet Activity Board* (*IAB*), in denen Entwürfe, Empfehlungen und Standards zum Internet beschrieben sind. Auch Anmerkungen, Kommentare oder andere informelle Ergänzungen sind darin zu finden.
Alle bekannten Protokolle, wie beispielsweise FTP, SMTP, NNTP, MIME, DNS, HMTL oder HTTP, sind in einschlägigen RFCs beschrieben. Sie sind nicht immer einfach zu lesen, aber oftmals die einzige verläßliche Quelle für die Implementierung eines bestimmten Protokolls. Es gibt viele Server im Internet, die RFCs zur Verfügung stellen. Bekannte Beispiele sind http://rfc.fh-koeln.de/, http://www.cis.ohio-state.edu/hypertext/information/rfc.html.

Beispiele für RFCs:

Protokoll	**Zuständige RFCs**
IP	RFC791, RFC1060
ICMP	RFC792
TCP	RFC793
UDP	RFC768
DNS	RFC1034, RFC1035, RFC2136, RFC974, RFC1101, RFC1812
ARP / RARP	RFC826, RFC903
SMTP	RFC821, RFC822
POP3	RFC1939
HTTP 1.0 / 1.1	RFC1945, RFC2068

15.5 Client-Sockets

Adressierung

Zur Adressierung von Rechnern im Netz wird die Klasse *InetAddress* des Pakets *java.net* verwendet. Ein *InetAddress*-Objekt enthält sowohl eine IP-Adresse als auch den symbolischen Namen des jeweiligen Rechners. Die beiden Bestandteile können mit den Methoden *getHostName( )* und *getHostAddress( )* abgefragt werden. Mit Hilfe von *getAddress( )* kann die IP-Adresse auch direkt als byte-Array mit vier Elementen beschafft werden:

```
String getHostName()
String getHostAddress()
byte[] getAddress()
```

Um ein *InetAddress*-Objekt zu generieren, stehen die beiden statischen Methoden *getByName( )* und *getLocalHost( )* zur Verfügung:

```
public static InetAddress getByName(String host)
  throws UnknownHostException
public static InetAddress getLocalHost()
  throws UnknownHostException
```

getByName( ) erwartet einen String mit der IP-Adresse oder dem Namen des Hosts als Argument, *getLocalHost( )* liefert ein *InetAddress*-Objekt für den eigenen Rechner. Beide Methoden lösen eine Ausnahme des Typs *UnknownHostException* aus, wenn die Adresse nicht ermittelt werden kann. Das ist insbesondere dann der Fall, wenn kein DNS-Server zur Verfügung steht, der die gewünschte Namensauflösung erledigen könnte (beispielsweise weil die Dial-In-Verbindung zum Provider gerade nicht besteht).
Eine Besonderheit ist der Name *localhost*. Dies ist eine Pseudo-Adresse für den eigenen Host. Sie ermöglicht das Testen von Netzwerkanwendungen, auch wenn keine wirkliche Netzwerkverbindung besteht (TCP/IP muss allerdings korrekt installiert sein). Sollen wirkliche Adressen verarbeitet werden, muss natürlich eine Verbindung zum Netz (insbesondere zum DNS-Server) aufgebaut werden können.

Aufbau einer einfachen Socket-Verbindung

Als *Socket* bezeichnet man eine streambasierte Programmierschnittstelle zur Kommunikation zweier Rechner in einem TCP/IP-Netz. Sockets wurden Anfang der achtziger Jahre für die Programmiersprache C entwickelt und mit Berkeley UNIX 4.1/4.2 allgemein eingeführt. Das Übertragen von Daten über eine Socket-Verbindung ähnelt dem Zugriff auf eine Datei:

- Zunächst wird eine Verbindung aufgebaut.
- Dann werden Daten gelesen und/oder geschrieben.
- Schließlich wird die Verbindung wieder abgebaut.

Während die Socket-Programmierung in C eine etwas mühsame Angelegenheit darstellt, ist es in Java recht einfach geworden. Im wesentlichen sind dazu die beiden Klassen *Socket* und *ServerSocket* erforderlich. Sie repräsentieren Sockets aus der Sicht einer Client- bzw. Server-Anwendung.

Im folgenden wird die Klasse *Socket* näher behandelt. Diese besitzt verschiedene Konstruktoren, mit denen ein neuer Socket erzeugt werden kann. Die wichtigsten von ihnen sind:

```
public Socket(String host, int port)
  throws UnknownHostException, IOException
public Socket(InetAddress address, int port)
  throws IOException
```

Beide Konstruktoren erwarten als erstes Argument die Übergabe des Hostnamens, zu dem eine Verbindung aufgebaut werden soll. Dieser kann entweder als Domainname in Form eines Strings oder als Objekt des Typs *InetAddress* übergeben werden. In diesem Fall kann das übergebene *InetAddress*-Objekt wiederverwendet werden, und die Adressauflösung muss nur einmal erfolgen. Wenn der Socket nicht geöffnet werden konnte, gibt es eine Ausnahme des Typs *IOException* bzw. *UnknownHostException* (wenn das angegebene Zielsystem nicht angesprochen werden konnte).

Der zweite Parameter des Konstruktors ist die Portnummer. Nachdem die Socket-Verbindung erfolgreich aufgebaut wurde, kann mit den beiden Methoden *getInputStream( )* und *getOutputStream( )* je ein Stream zum Empfangen und Versenden von Daten beschafft werden:

```
public InputStream getInputStream()
  throws IOException

public OutputStream getOutputStream()
  throws IOException
```

Diese Streams können entweder direkt verwendet oder mit Hilfe der Filterstreams in einen bequemer zu verwendenden Streamtyp geschachtelt werden. Nach Ende der Kommunikation sollten sowohl die Eingabe- und Ausgabestreams als auch der Socket selbst mit *close( )* geschlossen werden.

Beispiel 1 Abruf des DayTime-Services

Eine Verbindung zum *DayTime*-Service auf Port 13 wird hergestellt. Dieser Service läuft auf fast allen UNIX-Maschinen und kann gut zu Testzwecken verwendet werden. Nachdem der Client die Verbindung aufgebaut hat, sendet der DayTime-Server einen String mit dem aktuellen Datum und der aktuellen Uhrzeit und beendet dann die Verbindung.

```
import java.net.*;
import java.io.*;
public class DayTimeClient
{
   public static void main(String[] args)
   {
      if (args.length!= 1)
      {
         System.err.println("Usage: java DayTimeClient <host>");
         System.exit(1);
      }
```

```
      try
      {
         Socket sock = new Socket(args[0], 13);  // Verbinden auf Port 13
         InputStream in = sock.getInputStream();
         int len;
         byte[] b = new byte[100];
         while ((len = in.read(b))!= -1)
         {
            System.out.write(b, 0, len);
         }
         in.close();
         sock.close();
      }
      catch (IOException e)
      {
         System.err.println(e.toString());
         System.exit(1);
      }
   }
}
```

Die Ausgabe des Programms ist beispielsweise:
Sat Aug 25 16:58:37 2001

Beispiel 2 Zugriff auf einen Webserver
Die Kommunikation mit einem Web-Server erfolgt über das HTTP-Protokoll, wie es in den RFCs 1945 und 2068 beschrieben wurde. Ein Web-Server läuft normalerweise auf TCP-Port 80 (manchmal läuft er zusätzlich auch auf dem UDP-Port 80) und kann wie jeder andere Server über einen Client-Socket angesprochen werden. An diesere Stelle wird nicht auf Details eingegangen, sondern nur die einfachste und wichtigste Anwendung eines Web-Servers gezeigt, nämlich das Übertragen einer Seite. Ein Web-Server ist in seinen Grundfunktionen ein recht einfaches Programm, dessen Hauptaufgabe darin besteht, angeforderte Seiten an seine Clients zu versenden. Kompliziert wird er vor allem durch die Vielzahl der mittlerweile eingebauten Zusatzfunktionen, wie beispielsweise Logging, Server-Scripting, Server-Side-Includes, Security- und Tuning-Features usw.
Fordert ein Anwender in seinem Web-Browser eine Seite an, so wird diese Anfrage vom Browser als *GET*-Transaktion an den Server geschickt. Um beispielsweise die Seite *http://www.htl.at/index.htm* zu laden, wird folgendes Kommando an den Server www.htl.at gesendet: `GET /index.html`
Der erste Teil gibt den Kommandonamen an, dann folgt die gewünschte Datei. Die Zeile muss mit einer \r\n-Sequenz abgeschlossen werden, ein einfaches \n reicht nicht aus. Der Server versucht nun die angegebene Datei zu laden und überträgt sie an den Client. Ist der Client ein Web-Browser, wird er den darin befindlichen HTML-Code interpretieren und auf dem Bildschirm anzeigen. Befinden sich in der Seite Verweise auf Images, Applets oder Frames, so fordert der Browser die fehlenden Seiten in weiteren GET-Transaktionen von deren Servern ab.
Die Struktur des GET-Kommandos wurde mit der Einführung von HTTP 1.0 etwas erweitert. Zusätzlich werden nun am Ende der Zeile eine Versionskennung und wahlweise in den darauffolgenden Zeilen weitere Headerzeilen mit Zusatzinformationen mitgeschickt. Nachdem die letzte Headerzeile gesendet wurde, folgt eine leere Zeile (also ein alleinstehendes \r\n), um das Kommandoende anzuzeigen. HTTP 1.0 ist weit verbreitet, und das obige Kommando würde von den meisten Browsern in folgender Form gesendet werden: `GET /index.html HTTP/1.0`
Wird HTTP/1.0 verwendet, ist auch die Antwort des Servers etwas komplexer. Anstatt lediglich den Inhalt der Datei zu senden, liefert der Server seinerseits einige Headerzeilen mit Zusatzinformationen, wie beispielsweise den Server-Typ, das Datum der letzten Änderung oder den MIME-Typ der Datei. Auch hier ist jede Headerzeile mit einem \r\n abgeschlossen, und nach der letzten Headerzeile folgt eine Leerzeile. Erst dann beginnt der eigentliche Dateiinhalt.

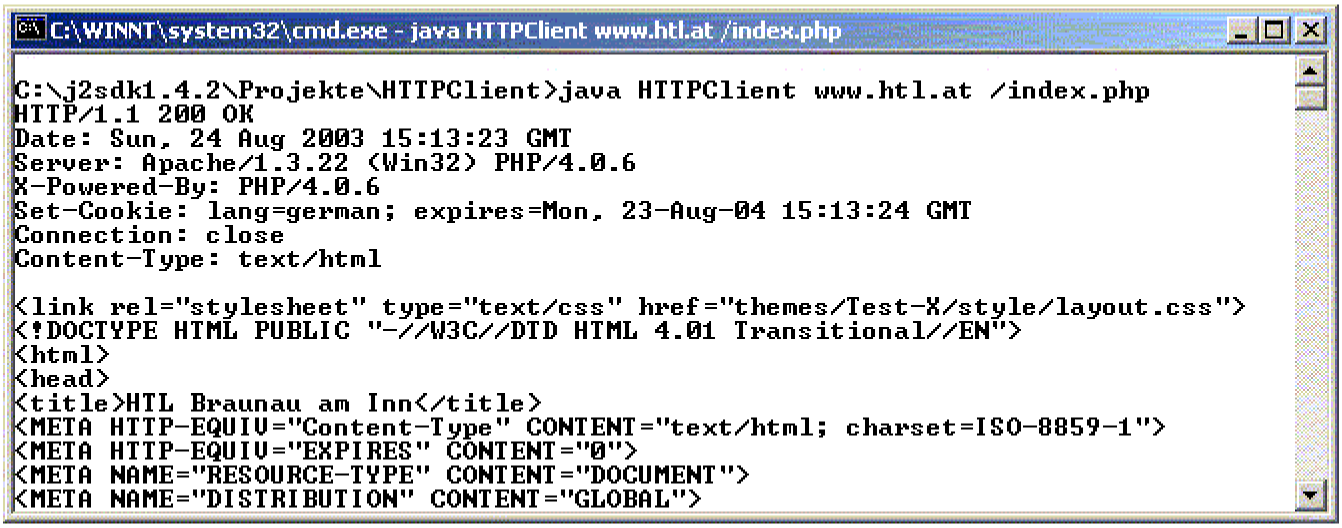
Das folgende Programm kann dazu verwendet werden, eine Datei von einem Web-Server zu laden. Es wird mit einem Host- und einem Dateinamen als Argument aufgerufen und lädt die Seite vom angegebenen Server. Das Ergebnis wird (mit allen Headerzeilen) auf dem Bildschirm angezeigt.

```
import java.net.*;
import java.io.*;
public class HTTPClient
{
  public static void main(String[] args)
  {
    if (args.length!= 2)
    {
      System.err.println("Usage: java HTTPClient <host> <file>");
      System.exit(1);
    }
    try
    {
      Socket sock = new Socket(args[0], 80);
      OutputStream out = sock.getOutputStream();
      InputStream in = sock.getInputStream();

      String s = "GET " + args[1] + " HTTP/1.0" + "\r\n\r\n";  // GET-Kommando senden
      out.write(s.getBytes());

      int len;                          // Ausgabe lesen und anzeigen
      byte[] b = new byte[100];
      while ((len = in.read(b))!= -1)
      {
        System.out.write(b, 0, len);
      }

      in.close();                       // Programm beenden
      out.close();
      sock.close();
    }
    catch(IOException e)
    {
      System.err.println(e.toString());
      System.exit(1);
    }
  }
}
```



15.5 Server Sockets

Der wesentliche Unterschied zu einem Client-Socket liegt in der Art des Verbindungsaufbaus, für den es eine spezielle Klasse *ServerSocket* gibt. Diese Klasse stellt Methoden zur Verfügung, um auf einen eingehenden Verbindungswunsch zu warten und nach erfolgtem Verbindungsaufbau einen Socket zur Kommunikation mit dem Client zurückzugeben. Bei der Klasse *ServerSocket* sind im wesentlichen der Konstruktor und die Methode *accept( )* von Interesse:

```
public ServerSocket(int port)
  throws IOException

public Socket accept()
  throws IOException
```

Der Konstruktor erzeugt einen ServerSocket für einen bestimmten Port, also einen bestimmten Typ von Serveranwendung. Anschließend wird die Methode *accept( )* aufgerufen, um auf einen eingehenden Verbindungswunsch zu warten. *accept( )* blockiert so lange, bis sich ein Client bei der Serveranwendung anmeldet (also eine Verbindung zu unserem Host unter der Portnummer, die im Konstruktor angegeben wurde, aufbaut). Ist der Verbindungsaufbau erfolgreich, liefert *accept( )* ein *Socket*-Objekt, das wie bei einer Client-Anwendung zur Kommunikation mit der Gegenseite verwendet werden kann. Anschließend steht der ServerSocket für einen weiteren Verbindungsaufbau zur Verfügung oder kann mit *close( )* geschlossen werden.

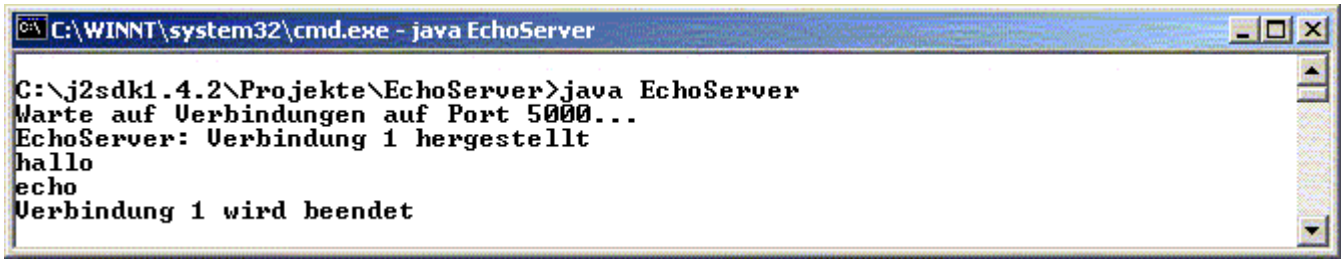
Beispiel: Implementierung eines einfachen ECHO-Servers, der auf Port 7 auf Verbindungswünsche wartet. Alle eingehenden Daten sollen unverändert an den Client zurückgeschickt werden. Zur Kontrolle werden sie ebenfalls auf die Konsole ausgegeben.
Das Programm soll folgende Funktionalitäten aufweisen:
- Der Server soll mehr als einen Client gleichzeitig bedienen, wobei die Clients durchnumeriert werden.
- Beim Verbindungsaufbau soll der Client eine Begrüßungsmeldung erhalten.
- Für jeden Client soll ein eigener Thread angelegt werden.

```
import java.net.*;
import java.io.*;

public class EchoServer
{
  public static void main(String[] args)
  {
    int cnt = 0;
    try
    {
      System.out.println("Warte auf Verbindungen auf Port 7...");
      ServerSocket echod = new ServerSocket(7);
      while(true)
      {
        Socket socket = echod.accept();
        (new EchoClientThread(++cnt, socket)).start();
      }
    }
    catch(IOException e)
    {
      System.err.println(e.toString());
      System.exit(1);
    }
  }
}
```

```
class EchoClientThread extends Thread
{
   private int name;
   private Socket socket;
   public EchoClientThread(int name, Socket socket)
   {
      this.name = name;
      this.socket = socket;
   }
   public void run()
   {
      String msg = "EchoServer: Verbindung " + name;
      System.out.println(msg + " hergestellt");
      try
      {
         InputStream in = socket.getInputStream();
         OutputStream out = socket.getOutputStream();
         out.write((msg + "\r\n").getBytes());
         int c;
         while ((c = in.read())!= -1)
         {
            out.write((char)c);
            System.out.print((char)c);
         }
         System.out.println("Verbindung " + name + " wird beendet");
         socket.close();
      }
      catch (IOException e)
      {
         System.err.println(e.toString());
      }
   }
}
```

Beispiel für eine Programmausgabe:



Im Hauptprogramm wird der ServerSocket erzeugt und in einer Schleife jeweils mit *accept( )* auf einen Verbindungswunsch gewartet. Nach dem Verbindungsaufbau erfolgt die weitere Bearbeitung nicht mehr im Hauptprogramm, sondern es wird ein neuer Thread mit dem Verbindungs-Socket als Argument erzeugt. Dann wird der Thread gestartet und erledigt die gesamte Kommunikation mit dem Client. Beendet der Client die Verbindung, wird auch der zugehörige Thread beendet. Das Hauptprogramm braucht sich nur um den Verbindungsaufbau zu kümmern und ist von der eigentlichen Client-Kommunikation vollständig befreit. Das Programm ist ein geeigneter Ausgangspunkt für eigene Experimente.

Übungsaufgaben:
- Testen Sie die angeführten Beispiele durch.
- Schreiben Sie einen Client für den Echo-Server.
- Schreiben Sie ein Applet, das den Austausch von Textnachrichten zwischen Rechnern ermöglicht.
- Implementieren Sie einen einfachen Webserver.

16. Collections

16.1 Grundlagen und Konzepte

Als *Collections* bezeichnet man Datenstrukturen, die dazu dienen, *Mengen* von Daten aufzunehmen und zu verarbeiten. Da es sich um sehr wichtige und häufig benötigte Datenstrukturen handelt, werden die vom SDK-Framework zur Verfügung gestellten Collections in diesem Kapitel relativ ausführlich erläutert.
Die Daten werden gekapselt abgelegt, und der Zugriff ist nur mit Hilfe vorgegebener Methoden möglich. Typische Collections sind *Listen (Lists)*, *Mengen (Sets), Stapel (Stacks)*, *Warteschlangen (Queues)*, *Bäume (Trees)* und *Hashtabellen (Maps)*.
Einfache Programmiersprachen bieten Collections meist nur in Form von Arrays. In Java und anderen objektorientierten Sprachen gibt es dagegen eine ganze Reihe unterschiedlicher Collections.
Java kennt zwei unterschiedliche *Collection-APIs*. Während beider Funktionalität sich beträchtlich überschneidet, weisen ihre Schnittstellen teilweise recht große Unterschiede auf. Einerseits gibt es seit dem SDK 1.0 die "traditionellen" Collections mit den Klassen *Vektor*, *Stack*, *Dictionary*, *Hashtable*, *BitSet*. Andererseits wurde mit dem SDK 1.2 parallel zu den vorhandenen Klassen ein neues Collection-API eingeführt.

16.2 Die Collection-Klassen des SDK 1.0

Die Klasse Vektor

Die Klasse *Vector* aus dem Package *java.util* ist die Java-Repräsentation einer linearen Liste. Die Liste kann Elemente beliebigen Typs enthalten, und ihre Länge ist zur Laufzeit veränderbar. *Vector* erlaubt das Einfügen von Elementen an beliebiger Stelle und bietet sowohl sequentiellen als auch wahlfreien Zugriff auf die Elemente. Das SDK realisiert einen *Vector* als Array von Elementen des Typs *Object*. Daher sind Zugriffe auf vorhandene Elemente und das Durchlaufen der Liste schnelle Operationen. Löschungen und Einfügungen, die die interne Kapazität des Arrays überschreiten, sind dagegen relativ langsam, weil Teile des Arrays umkopiert werden müssen.

Erzeugen eines Vektors
Das Erzeugen eines neuen Vektors kann mit Hilfe des parameterlosen Konstruktors erfolgen:

```
public Vector();
```

Nach dem Anlegen ist ein *Vector* vorerst leer, d.h.er enthält keine Elemente. Durch Aufruf von *isEmpty( )* kann geprüft werden, ob ein *Vector* leer ist; *size( )* liefert die Anzahl der Elemente:

```
public final boolean isEmpty();
public final int     size();
```

Elemente können an beliebiger Stelle in die Liste eingefügt werden. Ein *Vector* erlaubt die Speicherung beliebiger Objekttypen, denn die Einfüge- und Zugriffsmethoden arbeiten mit Instanzen der Klasse *Object*. Da jede Klasse letztlich aus *Object* abgeleitet ist, können auf diese Weise beliebige Objekte in die Liste eingefügt werden.

ANMERKUNG
Leider ist der Zugriff auf die gespeicherten Elemente damit natürlich nicht *typsicher*. Der Compiler kann nicht wissen, welche Objekte an welcher Stelle im *Vector* gespeichert wurden, und geht daher davon aus, dass beim Zugriff auf Elemente eine Instanz der Klasse *Object* geliefert wird. Mit Hilfe des Typkonvertierungsoperators muss diese dann in das ursprüngliche Objekt zurückverwandelt werden. Die Verantwortung für korrekte Typisierung liegt damit beim Entwickler. Mit Hilfe des Operators *instanceof( )* kann bei Bedarf zumindest eine Laufzeit-Typüberprüfung vorgeschaltet werden.

Einfügen von Elementen
Neue Elemente können wahlweise an das Ende des Vektors oder an einer beliebigen anderen Stelle eingefügt werden. Das Einfügen am Ende erfolgt mit der Methode *addElement( )*:

```
public void addElement(Object obj);
```

In diesem Fall wird das Objekt *obj* an das Ende der bisherigen Liste von Elementen angehängt. Soll ein Element dagegen an einer beliebigen Stelle innerhalb der Liste eingefügt werden, ist die Methode *insertElementAt( )* zu verwenden:

```
public void insertElementAt(Obj obj, int index) throws ArrayIndexOutOfBoundsException
```

Diese Methode fügt das Objekt *obj* an der Position *index* in den Vektor ein, wobei Indizes mit 0 beginnen. Alle bisher an dieser oder einer dahinterliegenden Position befindlichen Elemente werden um eine Position weitergeschoben.

Zugriff auf Elemente
Ein Vektor bietet sowohl *sequentiellen* als auch *wahlfreien* Zugriff auf seine Elemente. Für den sequentiellen Zugriff bietet es sich an, den im nachfolgenden Abschnitt beschriebenen *Iterator* zu verwenden. Der wahlfreie Zugriff erfolgt mit einer der Methoden *firstElement( )*, *lastElement( )* oder *elementAt( )*:

```
public Object firstElement() throws ArrayIndexOutOfBoundsException;
public Object lastElement() throws ArrayIndexOutOfBoundsException;
public Object elementAt(int index) throws ArrayIndexOutOfBoundsException;
```

firstElement( ) liefert das erste Element, *lastElement( )* das letzte. Mit Hilfe von *elementAt( )* wird auf das Element an Position *index* zugegriffen. Alle drei Methoden verursachen eine Ausnahme, wenn das gesuchte Element nicht vorhanden ist.

Der Vektor als Iterator
Für den sequentiellen Zugriff auf die Elemente des Vektors steht ein *Iterator* zur Verfügung. Ein Iterator ist eine Abstraktion für den aufeinanderfolgenden Zugriff auf alle Elemente einer komplexen Datenstruktur. Ein Iterator für die traditionellen Collection-Klassen wird in Java durch das Interface *Enumeration* zur Verfügung gestellt und deshalb in der Java-Welt oft auch als *Enumerator* bezeichnet.
Das Interface *Enumeration* besitzt die Methoden *hasMoreElements( )* und *nextElement( )*. Nach der Initialisierung zeigt ein *Enumeration*-Objekt auf das erste Element der Aufzählung. Durch Aufruf von *hasMoreElements( )* kann geprüft werden, ob weitere Elemente in der Aufzählung enthalten sind, und *nextElement( )* setzt den internen Zeiger auf das nächste Element:

```
public boolean hasMoreElements()
public Object  nextElement() throws NoSuchElementException;
```

In der Klasse *Vector* liefert die Methode *elements( )* einen Enumerator für alle Elemente, die sich im Vektor befinden:

```
public Enumeration elements()
```

Das folgende Beispiel verdeutlicht die Anwendung der Klasse *Vector* :

```
import java.util.*;
public class VektorTest
{
  public static void main(String[] args)
  {
    Vector v = new Vector();
     v.addElement("eins");
     v.addElement("drei");
     v.insertElementAt("zwei",1);
     for (Enumeration el=v.elements(); el.hasMoreElements(); )
     {
       System.out.println((String)el.nextElement());
     }
  }
}
```

Das Programm fügt die Werte "eins", "zwei" und "drei" in einen Vektor ein und gibt sie anschließend aus.

ANMERKUNG
Ein Enumerator ist immer dann nützlich, wenn die Elemente eines zusammengesetzten Datentyps nacheinander aufgezählt werden sollen. Enumeratoren werden in Java noch an verschiedenen anderen Stellen zur Verfügung gestellt, beispielsweise in den Klassen *Hashtable* oder *StringTokenizer*.

Die Klasse Stack

Ein Stack (Stapel) ist eine Datenstruktur, die nach dem *LIFO-Prinzip* (last-in-first-out) arbeitet. Die Elemente werden am vorderen Ende einer Liste eingefügt und von dort auch wieder entnommen. Das heißt, die zuletzt eingefügten Elemente werden zuerst entnommen und die zuerst eingefügten zuletzt.
In Java ist ein *Stack* eine Ableitung eines Vektors, der um neue Zugriffsfunktionen erweitert wurde, um das Verhalten eines Stacks zu implementieren. Obwohl dies eine ökonomische Vorgehensweise ist, bedeutet es, dass ein *Stack* alle Methoden eines Vektors erbt und damit auch wie ein Vektor verwendet werden kann.

Erzeugen eines Stacks

Der Konstruktor der Klasse *Stack* ist parameterlos:

```
public Stack();
```

Das Anfügen neuer Elemente wird durch einen Aufruf der Methode *push( )* erledigt und erfolgt *wie* üblich am oberen Ende des Stacks. Die Methode liefert als Rückgabewert das eingefügte Objekt:

```
public Object push(Object item);
```

Zugriff auf das oberste Element

Der Zugriff auf das oberste Element kann mit einer der Methoden *pop( )* oder *peek( )* erfolgen. Beide liefern das oberste Element des Stacks, *pop( )* entfernt es anschließend vom Stack:

```
public Object pop();
public Object peek();
```

Des weiteren gibt es eine Methode *empty( )*, um festzustellen, ob der Stack leer ist, und eine Methode *search( )*, die nach einem beliebigen Element sucht und als Rückgabewert die Distanz zwischen gefundenem Element und oberstem Stack-Element angibt:

```
public boolean empty();
public int     search(Object o);
```

Das folgende Beispiel verdeutlicht die Anwendung eines Stacks:

```
import java.util.*;
public class StackTest
{
  public static void main(String[] args)
  {
    Stack s = new Stack();

    s.push("Erstes Element");
    s.push("Zweites Element");
    s.push("Drittes Element");
    while (true)
    {
      try
      {
        System.out.println((String)s.pop());
      }
      catch (EmptyStackException e)
      {
        break;
      }
    }
  }
}
```

Das Programm erzeugt einen Stack und fügt die Werte "Erstes Element", "Zweites Element" und "Drittes Element" ein. Anschließend entfernt es so lange Elemente aus dem Stack, bis die Ausgabeschleife durch eine Ausnahme des Typs *EmptyStackException* beendet wird. Durch die Arbeitsweise des Stacks werden die Elemente in der umgekehrten Eingabereihenfolge ausgegeben.

Die Klasse Hashtable

Die Klasse *Hashtable* ist eine Implementierung der abstrakten Klasse *Dictionary*. Diese stellt einen *assoziativen* Speicher dar, der *Schlüssel* auf *Werte* abbildet und über den Schlüsselbegriff einen effizienten Zugriff auf den Wert ermöglicht. Ein Dictionary speichert also immer zusammengehörige Paare von Daten, bei denen der Schlüssel als Name des zugehörigen Wertes angesehen werden kann. Über den Schlüssel kann später der Wert sehr schnell wiedergefunden werden.
Da ein Dictionary auf unterschiedliche Weise implementiert werden kann, haben die Java-Designer entschieden, dessen abstrakte Eigenschaften in einer Basisklasse zusammenzufassen. Die Implementierung *Hashtable* benutzt das Verfahren der *Schlüsseltransformation*, also die Verwendung einer Transformationsfunktion (auch *Hash-Funktion* genannt), zur Abbildung von Schlüsseln auf Indexpositionen eines Arrays. Weitere Implementierungen der Klasse *Dictionary*, etwa auf der Basis binärer Bäume, gibt es in Java derzeit nicht.
Neben den erwähnten abstrakten Eigenschaften besitzt *Hashtable* noch die konkreten Merkmale *Kapazität* und *Ladefaktor*. Die Kapazität gibt die Anzahl der Elemente an, die insgesamt untergebracht werden können. Der Ladefaktor zeigt dagegen an, bei welchem Füllungsgrad die Hash-Tabelle vergrößert werden muss. Das Vergrößern erfolgt automatisch, wenn die Anzahl der Elemente innerhalb der Tabelle größer ist als das Produkt aus Kapazität und Ladefaktor. Seit dem SDK 1.2 darf der Ladefaktor auch größer als 1 sein. In diesem Fall wird die *Hashtable* also erst dann vergrößert, wenn der Füllungsgrad größer als 100 % ist und bereits ein Teil der Elemente in den Überlaufbereichen untergebracht wurde.

ANMERKUNG
Wichtig bei der Verwendung der *Dictionary*-Klassen ist, dass das Einfügen und der Zugriff auf Schlüssel nicht auf der Basis des Operators = =, sondern mit Hilfe der Methode *equals( )* erfolgt. Schlüssel müssen daher lediglich *inhaltlich* gleich sein, um als identisch angesehen zu werden. Eine Referenzgleichheit ist dagegen nicht erforderlich.

Erzeugen einer Hashtable
Eine Instanz der Klasse *Hashtable* kann mit Hilfe eines parameterlosen Konstruktors angelegt werden:

```
public Hashtable();
```

Das Einfügen von Elementen erfolgt durch Aufruf der Methode *put( )*:

```
public Object put(Object key, Object value);
```

Dieser Aufruf fügt das Schlüssel-Werte-Paar *(key, value)* in die Hashtable ein. Weder *key* noch *value* dürfen dabei *null* sein. Falls bereits ein Wertepaar mit dem Schlüssel *key* enthalten ist, wird der bisherige Wert gegen den neuen ausgetauscht, und *put( )* liefert in diesem Fall den Wert zurück, der bisher dem Schlüssel zugeordnet war. Falls der Schlüssel bisher noch nicht vorhanden ist, ist der Rückgabewert *null*.

Einfügen von Elementen
Der Zugriff auf ein Element erfolgt mit Hilfe der Methode *get( )* über den ihm zugeordneten Schlüssel. Die Methode *get( )* erwartet ein Schlüsselobjekt und liefert den dazu passenden Wert. Falls der angegebene Schlüssel nicht enthalten war, ist der Rückgabewert *null*:

```
public Object get(Object key);
```

Zusätzlich zu den bisher erwähnten Methoden gibt es noch zwei weitere mit den Namen *contains( )* und *containsKey( )*. Sie überprüfen, ob ein bestimmter Wert bzw. ein bestimmter Schlüssel in der Hashtable enthalten ist:

```
public boolean contains(Object value);
public boolean containsKey(Objec key);
```

Der Rückgabewert ist *true*, falls das gesuchte Element enthalten ist, andernfalls ist er *false*.

ANMERKUNG
Bei der Verwendung dieser Funktionen ist zu beachten, dass die Suche nach einem Wert viel ineffizienter ist als die Suche nach einem Schlüssel. Während der Schlüssel über die Transformationsfunktion sehr schnell gefunden wird, erfordert die Suche nach einem Wert einen sequentiellen Durchlauf durch die Tabelle.

Hashtable als Iterator
In der Klasse *Hashtable* gibt es zwei Iteratoren, die zur Auflistung von Schlüsseln und Werten verwendet werden können:

```
public Enumeration elements();
public Enumeration keys();
```

Die Methode *elements( )* liefert einen Iterator für die Auflistung aller Werte in der Hashtable. In welcher Reihenfolge die Elemente dabei durchlaufen werden, ist nicht definiert. Da eine Hash-Funktion die Eigenschaft hat, Schlüssel gleichmäßig über den verfügbaren Speicher zu verteilen, ist davon auszugehen, dass die Iteratoren ihre Rückgabewerte in einer zufälligen Reihenfolge liefern.
Analog zu *elements( )* liefert *keys( )* eine Auflistung aller Schlüssel, die sich in der Hash-Tabelle befinden. Wie üblich liefern beide Methoden ein Objekt, welches das Interface *Enumeration* implementiert. Wie bereits erklärt, erfolgt der Zugriff daher mit Hilfe der Methoden *hasMoreElements( )* und *nextElement( )*.

Das folgende Beispiel verdeutlicht die Anwendung einer Hashtable:

```
import java.util.*;
public class EmailAliases
{
  public static void main(String[] args)
  {
    Hashtable h = new Hashtable();
    // Pflege der Aliase
    h.put("Fritz","f.mueller@test.de");
    h.put("Franz","fk@b-blabla.com");
    h.put("Paula","user0125@mail.uofm.edu");
    h.put("Lissa","lb3@gateway.fhdto.northsurf.dk");

    //Ausgabe
    Enumeration e = h.keys();
    while (e.hasMoreElements())
    {
      String alias = (String)e.nextElement();
      System.out.println(alias + " --> " + (String)h.get(alias));
    }
  }
}
```

Das Programm legt eine leere Hashtable an, die zur Aufnahme von Mail-Aliasen verwendet werden soll. Dazu soll zu jeder E-Mail-Adresse ein kurzer Aliasname gepflegt werden, unter dem die lange Adresse später angesprochen werden kann. Das Programm legt zunächst die Aliase »Fritz«, »Franz«, »Paula« und »Lissa« an und assoziiert jeden mit der zugehörigen E-Mail-Adresse. Anschließend durchläuft es alle Schlüssel und gibt zu jedem den dazu passenden Wert aus.

Die Klasse Properties

Die Klasse *Properties* ist von *Hashtable* abgeleitet und repräsentiert ein auf String-Paare spezialisiertes Dictionary, das es erlaubt, seinen Inhalt auf einen externen Datenträger zu speichern oder von dort zu laden. Ein solches Objekt wird auch als *Property-Liste* (oder *Eigenschaften-Liste*) bezeichnet. Zur Instanzierung stehen zwei Konstruktoren zur Verfügung:

```
public Properties();
public Properties(Properties defaults);
```

Der erste legt eine leere Property-Liste an, der zweite füllt sie mit den übergebenen Default-Werten. Der Zugriff auf die einzelnen Elemente erfolgt mit den Methoden *getProperty( )* und *propertyNames( )*:

```
public String      getProperty(String key);
public String      getProperty(String key, String defaultValue);
public Enumeration propertyNames()
```

Die erste Variante von *getProperty( )* liefert die Eigenschaft mit der Bezeichnung *key*. Ist sie nicht vorhanden, wird *null* zurückgegeben. Die zweite Variante gibt aber den Standardwert *defaultValue* zurück, wenn die gesuchte Eigenschaft nicht gefunden wurde. Mit *propertyNames( )* kann ein *Enumeration*-Objekt beschafft werden, mit dem alle Eigenschaften der Property-Liste aufgezählt werden können.

Zum Speichern und Lasen einer Property-Liste stehen die Methoden *store( )* und *load( )* zur Verfügung:

```
public void store(OutputStream out, String header) throws IOException;
public void load(InputStream in) throws IOException;
```

Property-Dateien sind Textdateien mit einem recht einfachen Aufbau (siehe SDK-Dokumentation).

Die Klasse BitSet

Die Klasse *BitSet( )* dient dazu, *Mengen ganzer Zahlen* zu repräsentieren. Sie erlaubt es, Zahlen einzufügen oder zu löschen und zu testen, ob bestimmte Werte in der Menge enthalten sind. Die Klasse bietet zudem die Möglichkeit, Teil- und Vereinigungsmengen zu bilden.
Ein BitSet erlaubt aber auch noch eine andere Interpretation. Sie kann nämlich auch als eine Menge von Bits, die entweder gesetzt oder nicht gesetzt sein können, angesehen werden. In diesem Fall entspricht das Einfügen eines Elements dem Setzen eines Bits, das Entfernen dem Löschen und die Vereinigungs- und Schnittmengenoperationen den logischen ODER- bzw. UND-Operationen.
Diese Analogie wird insbesondere dann deutlich, wenn man eine Menge von ganzen Zahlen in der Form repräsentiert, dass in einem booleschen Array das Element an Position *i* genau dann auf *true* gesetzt wird, wenn die Zahl *i* Element der repräsentierten Menge ist. Mit *BitSet* bietet Java nun eine Klasse, die sowohl als Liste von Bits als auch als Menge von Ganzzahlen angesehen werden kann.

Elementweise Operationen

Ein neues Objekt der Klasse *BitSet* kann mit dem parameterlosen Konstruktor angelegt werden. Die dadurch repräsentierte Menge ist zunächst leer.

```
public BitSet();
```

Das Einfügen einer Zahl (bzw. das Setzen eines Bits) erfolgt mit Hilfe der Methode *set( )*. Das Entfernen einer Zahl (bzw. das Löschen eines Bits) erfolgt mit der Methode *clear( )*. Die Abfrage, ob eine Zahl in der Menge enthalten ist (bzw. die Abfrage des Zustands eines Bits), erfolgt mit Hilfe der Methode *get( )*:

```
public void    set(int bit);
public void    clear(int bit);
public boolean get(int bit);
```

Mengenorientierte Operationen

Die mengenorientierten Operationen benötigen zwei Mengen als Argumente, nämlich das aktuelle Objekt und eine weitere Menge, die als Parameter übergeben wird. Das Ergebnis der Operation wird dem aktuellen Objekt zugewiesen. Das Bilden der Vereinigungsmenge (bzw. die bitweise ODER-Operation) erfolgt durch Aufruf der Methode *or( )*, das Bilden der Durchschnittsmenge (bzw. die bitweise UND-Operation) mit Hilfe von *and( )*. Zusätzlich gibt es die Methode *xor( )*, die ein bitweises Exklusiv-ODER durchführt.
Deren mengentheoretisches Äquivalent ist die Vereinigung von Schnittmenge und Schnittmenge der Umkehrmengen.

Seit dem SDK 1.2 gibt es zusätzlich die Methode *andNot( )*, mit der die Bits der Ursprungsmenge gelöscht werden, deren korrespondierendes Bit in der Argumentmenge gesetzt ist.

```
public void or(BitSet set);
public void and(BitSet set);
public void xor(BitSet set);
public void andNot(BitSet set)
```

16.3 Die Collection-Klassen des SDK 1.2

Wie in 16.1 erläutert, gibt es seit dem SDK 1.0 die "traditionellen" Collections mit den Klassen *Vector*, *Stack*, *Dictionary*, *Hashtable* und *BitSet*. Obwohl sie ihren Zweck durchaus erfüllen, gab es einige Kritik am Collections-Konzept des SDK 1.0. Zunächst wurde die geringe Vielseitigkeit kritisiert, etwa im Vergleich zum mächtigen Collection-Konzept der Sprache SmallTalk. Zudem galten die SDK-1.0-Collections als nicht sehr performant, fast alle wichtigen Methoden sind *synchronized* (siehe Threads). Zudem gab es Detailschwächen, die immer wieder kritisiert wurden. Als Beispiel kann die Wahl der Methodennamen des *Enumeration*-Interfaces genannt werden. Oder die Implementierung der Klasse *Stack*, die als Ableitung von *Vector* weit über ihr eigentliches Ziel hinausschießt.
Diese Kritik wurde von den Java-Designern zum Anlaß genommen, das Collection-Konzept im Rahmen der Version 1.2 neu zu überdenken. Herausgekommen ist dabei eine Sammlung von gut 20 Klassen und Interfaces im Package *java.util*, die das Collections-Framework des SDK 1.2 bilden. Wer bei dieser Zahl erschreckt, sei getröstet. Letztlich werden im wesentlichen die drei Grundformen *Set*, *List* und *Map* realisiert. Die große Anzahl ergibt sich aus der Bereitstellung verschiedender Implementierungsvarianten, Interfaces und einiger abstrakter Basisklassen, mit denen die Implementierung eigener Collections vereinfacht werden kann.

Zunächst wird die grundlegenden Arbeitsweise der Collection-Typen erläutert:

- **Eine List ist eine beliebig große Liste von Elementen beliebigen Typs, auf die sowohl wahlfrei als auch sequentiell zugegriffen werden kann.**
- **Ein Set ist eine (doublettenlose) Menge von Elementen, auf die mit typischen Mengenoperationen zugegriffen werden kann.**
- **Eine Map ist eine Abbildung von Elementen eines Typs auf Elemente eines anderen Typs, also eine Menge zusammengehöriger Paare von Objekten.**

Jede dieser Grundformen existiert als Interface unter dem oben angegebenen Namen. Zudem gibt es jeweils eine oder mehrere konkrete Implementierungen. Sie unterscheiden sich in den verwendeten Datenstrukturen und Algorithmen und damit in ihrer Eignung für verschiedene Anwendungsfälle. Weiterhin gibt es eine abstrakte Implementierung des Interfaces, mit dessen Hilfe das Erstellen eigener Collections erleichtert wird.

ANMERKUNG
Im Gegensatz zu den SDK 1.1-Klassen sind die Collections des SDK 1.2 aus Performancegründen durchgängig unsynchronisiert. Soll also von mehr als einem Thread gleichzeitig auf eine Collection zugegriffen werden (Collections sind häufig Kommunikationsmittel zwischen gekoppelten Threads), so ist unbedingt darauf zu achten, die Zugriffe selbst zu synchronisieren. Andernfalls können leicht Programmfehler und Dateninkonsistenzen entstehen.

Namensgebung
Die Namensgebung der Interfaces und Klassen folgt einem einfachen Schema. Das Interface hat immer den allgemein verwendeten Namen der zu implementierenden Collection, also beispielsweise *List*, *Set* oder *Map*. Jede Implementierungsvariante stellt vor den Namen dann eine spezifische Bezeichnung, die einen Hinweis auf die Art der verwendeten Datenstrukturen und Algorithmen geben soll. So gibt es für das Interface *List* beispielsweise die Implementierungsvarianten *LinkedList* und *ArrayList* sowie die abstrakte Implementierung *AbstractList*. Wenn man dieses Schema einmal begriffen hat, verliert der »Collection-Zoo« viel von seinem Schrecken.

Das Basisinterface Collection

Die Interfaces spielen eine wichtige Rolle, denn sie beschreiben bereits recht detailliert die Eigenschaften der folgenden Implementierungen. Dabei wurde im SDK 1.2 eine weitreichende Design-Entscheidung getroffen. Um nicht für jede denkbare Collection-Klasse ein eigenes Interface definieren zu müssen, wurde ein Basisinterface *Collection* geschaffen, aus dem die meisten Interfaces abgeleitet wurden. Es faßt die wesentlichen Eigenschaften einer großen Menge unterschiedlicher Collections zusammen:

Methoden des *Collection*-Interfaces:

```
int      size();
boolean  isEmpty();
boolean  contains(Object o);
Iterator iterator();
Object[] toArray();
Object[] toArray(Object[] a);
boolean  add(Object o);
boolean  remove(Object o);
boolean  containsAll(Collection c);
boolean  addAll(Collection c);
boolean  removeAll(Collection c);
boolean  retainAll(Collection c);
void     clear();
boolean  equals(Object o);
int      hashCode();
```

Zusätzlich fordert die SDK-1.2-Spezifikation für jede Collection-Klasse zwei Konstruktoren (was ja leider nicht im Rahmen der Interface-Definition sichergestellt werden kann). Ein parameterloser Konstruktor wird verwendet, um eine leere Collection anzulegen. Ein weiterer, der ein einzelnes *Collection*-Argument besitzt, dient dazu, eine neue Collection anzulegen und mit allen Elementen aus der als Argument übergebenen Collection zu füllen. Die Interfaces *List* und *Set* sind direkt von *Collection* abgeleitet, *SortedSet* ist ein unmittelbarer Nachfolger von *Set*. Lediglich die Interfaces *Map* und das davon abgeleitete Interface *SortedMap* wurden nicht aus *Collection* abgeleitet.
Der Vorteil dieses Designs ist natürlich, dass eine flexible Schnittstelle für den Umgang mit *Mengen von Objekten* zur Verfügung steht. Wenn eine Methode einen Rückgabewert vom Typ *Collection* besitzt, können die Aufrufer dieser Methode auf die zurückgegebenen Elemente - unabhängig vom Typ der tatsächlich zurückgegebenen Collection-Klasse - einheitlich zugreifen. Selbst wenn eine andere *Implementierung* gewählt wird, ändert sich für den Aufrufer nichts, solange das zurückgegebene Objekt das *Collection*-Interface implementiert.

Soweit zur Theorie. Der Nachteil dieses Designs besteht darin, dass längst nicht alle tatsächlichen Collections eine Schnittstelle besitzen, wie sie in *Collection* definiert wurde. Während diese für eine Liste noch passen mag, besitzt eine Queue oder ein Stack gänzlich anders arbeitende Zugriffsroutinen. Dieser Konflikt wurde dadurch zu lösen versucht, dass die Methoden *zum Ändern* der Collection *optional* sind. Während also *contains( )*, *containsAll( )*, *equals( )*, *hashCode( )*, *isEmtpy( )*, *size( )* und *toArray( )* obligatorisch sind, müssen die übrigen Methoden in einer konkreten Implementierung nicht unbedingt zur Verfügung gestellt werden, sondern können weggelassen, ersetzt oder ergänzt werden.
Aus Kapitel 9 ist allerdings bekannt, dass alle definierten Methoden eines Interfaces in einer konkreten Implementierung zur Verfügung gestellt werden müssen. Davon sind natürlich auch die Collection-Klassen nicht ausgenommen. Wenn sich diese entschließen, eine optionale Methode nicht zu realisieren, so muss diese zwar implementiert werden, löst aber beim Aufruf eine Exception des Typs *UnsupportedOperation-Exception* aus.

Collections des Typs List

Abstrakte Eigenschaften
Eine Collection vom Typ *List* ist eine geordnete Menge von Objekten, auf die entweder sequentiell oder über ihren Index (ihre Position in der Liste) zugegriffen werden kann. Wie bei Arrays hat das erste Element den Index 0 und das letzte den Index *size( ) - 1*. Es ist möglich, an einer beliebigen Stelle der Liste ein Element einzufügen oder zu löschen. Die weiter hinten stehenden Elemente werden dann entsprechend nach rechts bzw. links verschoben. Des weiteren gibt es Methoden, um Elemente in der Liste zu suchen.
Das Interface *List* ist direkt aus *Collection* abgeleitet und erbt somit dessen Methoden. Zusätzlich gibt es einige neue Methoden, die zum wahlfreien Zugriff auf die Elemente benötigt werden. Um Elemente in die Liste einzufügen, können die Methoden *add( )* und *addAll( )* verwendet werden:

```
void    add(int index, Object element);
boolean add(Object o);
boolean addAll(Collection c);
boolean addAll(int index, Collection c);
```

Mit *add( )* wird ein einfaches Element in die Liste eingefügt. Wenn die Methode mit einem einzelnen Object als Parameter aufgerufen wird, hängt sie das Element an das Ende der Liste an. Wird zusätzlich der Index angegeben, so wird das Element an der spezifizierten Position eingefügt und alle übrigen Elemente um eine Position nach rechts geschoben. Mit *addAll( )* kann eine komplette Collection in die Liste eingefügt werden. Auch hier können die Elemente wahlweise an das Ende angehängt oder an einer beliebigen Stelle in der Liste eingefügt werden.
Der Rückgabewert von *add( )* ist *true*, wenn die Liste durch den Aufruf von *add( )* verändert, also das Element hinzugefügt wurde. Er ist *false*, wenn die Liste nicht verändert wurde. Das kann beispielsweise dann der Fall sein, wenn die Liste keine Doubletten (doppelte Elemente) erlaubt und ein bereits vorhandenes Element noch einmal eingefügt werden soll. Konnte das Element dagegen aus einem anderen Grund nicht eingefügt werden, wird eine Ausnahme des Typs *UnsupportedOperationException*, *ClassCastException* oder *IllegalArgumentException* ausgelöst.

Das Löschen von Elementen kann mit den Methoden *remove( )*, *removeAll( )* und *retainAll( )* erfolgen:

```
Object  remove(int index);
Boolean remove(Object o);
boolean removeAll(Collection c);
boolean retainAll(Collection c);
```

An *remove( )* kann dabei wahlweise der Index des zu löschenden Objekts oder das Objekt selbst übergeben werden. Mit *removeAll( )* werden alle Elemente gelöscht, die auch in der als Argument übergebenen Collection enthalten sind, und *retainAll( )* löscht alle Elemente außer den in der Argument-Collection enthaltenen.

Implementierungen
Das Interface *List* wird seit dem SDK 1.2 von verschiedenen Klassen implementiert:

- **Die Klasse AbstractList ist eine abstrakte Basisklasse, bei der alle optionalen Methoden die Ausnahme UnsupportedOperationException auslösen und diverse obligatorische Methoden als *abstract* deklariert wurden. Sie dient als Basisklasse für eigene List-Implementierungen.**
- **Die Klasse LinkedList realisiert eine Liste, deren Elemente als doppelt verkettete lineare Liste gehalten werden. Ihre Einfüge- und Löschoperationen sind im Prinzip (viele Elemente vorausgesetzt) performanter als die der ArrayList. Der wahlfreie Zugriff ist dagegen normalerweise langsamer.**

- **Die Klasse ArrayList implementiert die Liste als Array von Elementen, das bei Bedarf vergrößert wird. Hier ist der wahlfreie Zugriff schneller, aber bei großen Elementzahlen kann das Einfügen und Löschen länger dauern als bei einer LinkedList.**
- **Aus Gründen der Vereinheitlichung implementiert seit dem SDK 1.2 auch die Klasse Vector das List-Interface. Neben den bereits in 16.2 erwähnten Methoden besitzt ein SDK 1.2-Vector also auch die entsprechenden Methoden des List-Interfaces.**

Soll im eigenen Programm eine Liste verwendet werden, stellt sich die Frage, welche der genannten Implementierungen dafür am besten geeignet ist. Während die Klasse *AbstractList* nur als Basisklasse eigener Listenklassen sinnvoll verwendet werden kann, ist die Entscheidung für eine der drei übrigen Klassen von den Spezifika der jeweiligen Anwendung abhängig. Bleibt die Liste klein, wird hauptsächlich wahlfrei darauf zugegriffen; überwiegen die lesenden gegenüber den schreibenden Zugriffen deutlich, so liefert die *ArrayList* die besten Ergebnisse. Ist die Liste dagegen sehr groß und werden häufig Einfügungen und Löschungen vorgenommen, ist wahrscheinlich die *LinkedList* die bessere Wahl. Wird von mehreren Threads gleichzeitig auf die Liste zugegriffen, kann die Klasse *Vector* verwendet werden, denn ihre Methoden sind bereits weitgehend als *synchronized* deklariert.

Das folgende Beispiel zeigt das Anlegen und Bearbeiten zweier unterschiedlicher Listen:

```
import java.util.*;
public class ListTest
{
   static void fillList(List list)
   {
      for(int i = 0; i < 10; ++i)
      {
         list.add("" + i);
      }
      list.remove("3");
      list.remove("5");
   }

   static void printList(List list)
   {
      for(int i = 0; i < list.size(); ++i)
      {
         System.out.println((String)list.get(i));
      }
      System.out.println("---");
   }

   public static void main(String[] args)
   {
      LinkedList list1 = new LinkedList();     //Erzeugen der LinkedList
      fillList(list1);
      printList(list1);
      ArrayList list2 = new ArrayList();       //Erzeugen der ArrayList
      fillList(list2);
      printList(list2);
      list2.remove("0");
      list1.removeAll(list2);                  //Test von removeAll
      printList(list1);
   }
}
```

Die Funktion des Beispielprogramms ist als Übung wiederum selbständig herauszufinden.

Iteratoren

Das Interface Iterator

Auf die Collections der Prä-1.2-SDKs konnte mit Hilfe des *Enumeration*-Interfaces und seiner beiden Methoden *hasMoreElements( )* und *nextElement( )* zugegriffen werden. Da Objekte, die dem Durchlaufen von Collections dienen, überall in der Informatik als *Iteratoren* bezeichnet werden, wurde die Namensgebung des Interfaces und seiner Methoden vielfach kritisiert. Die Designer der Collections der Version 1.2 haben sich nun dem allgemeinen Sprachgebrauch angepaßt und das Interface zum Durchlaufen der Elemente einer Collection als *Iterator* bezeichnet und mit folgenden Methoden ausgestattet:

```
Boolean hasNext();
Object  next();
void    remove();
```

hasNext( ) gibt genau dann *true* zurück, wenn der Iterator mindestens ein weiteres Element enthält. *next( )* liefert das nächste Element bzw. löst eine Ausnahme des Typs *NoSuchElementException* aus, wenn es keine weiteren Elemente gibt. Wie beim *Enumeration*-Interface ist der Rückgabewert als *Object* deklariert und muss daher auf den passenden Typ gecastet werden. Als neues Feature (gegenüber einer Enumeration) bietet ein Iterator die Möglichkeit, die Collection während der Abfrage zu ändern, indem das zuletzt geholte Element mit der Methode *remove( )* gelöscht wird. Bei allen Collections, die das Interface Collection implementieren, kann ein Iterator zum Durchlaufen aller Elemente mit der Methode *iterator( )* beschafft werden.

ANMERKUNG
Dies ist auch gleichzeitig die einzige erlaubte Möglichkeit, die Collection während der Verwendung eines Iterators zu ändern. Alle direkt ändernden Zugriffe auf die Collection machen das weitere Verhalten des Iterators undefiniert.

Das folgende Beispiel zeigt die Benutzung eines Iterators:

```
import java.util.*;
public class IteratorTest
{
  public static void main(String[] args)
  {
    ArrayList list = new ArrayList();        //Füllen der Liste
    for (int i = 1; i <= 20; ++i)
    {
      list.add("" + i);
    }
    Iterator it = list.iterator();           //Löschen von Elementen über Iterator
    while (it.hasNext())
    {
      String s = (String) it.next();
      if (s.startsWith("1"))
      {
        it.remove();
      }
    }
    it = list.iterator();                    //Ausgeben der verbleibenden Elemente
    while (it.hasNext())
    {
      System.out.println((String) it.next());
    }
  }
}
```

Die Funktion des Beispielprogramms ist als Übung wiederum selbständig herauszufinden.

Das Interface ListIterator
Neben dem *Iterator*-Interface gibt es das davon abgeleitete Interface *ListIterator*. Es steht nur bei Collections des Typs *List* (und davon abgeleiteten Klassen) zur Verfügung und bietet zusätzlich die Möglichkeit, die Liste in beiden Richtungen zu durchlaufen, auf den Index des nächsten oder vorigen Elements zuzugreifen, das aktuelle Element zu verändern und ein neues Element hinzuzufügen:

```
boolean hasPrevious();
Object  previous();
int     nextIndex();
int     previousIndex();
void    add(Object o);
void    set(Object o);
```

Mit *hasPrevious( )* kann bestimmt werden, ob es *vor* der aktuellen Position ein weiteres Element gibt; der Zugriff darauf würde mit *previous( )* erfolgen. Die Methoden *nextIndex( )* und *previousIndex* liefern den Index des nächsten bzw. vorigen Elements des Iterators. Wird *previousIndex( )* am Anfang des Iterators aufgerufen, ist sein Rückgabewert -1. Wird *nextIndex( )* am Ende aufgerufen, liefert es *size( )* als Rückgabewert. Mit *add( )* kann ein neues Element an der Stelle in die Liste eingefügt werden, die unmittelbar vor dem nächsten Element des Iterators liegt. *set( )* erlaubt es, das durch den letzten Aufruf von *next( )* bzw. *previous( )* beschaffte Element zu ersetzen.

ANMERKUNG
Ebenso wie beim Interface *Collection* sind die *ändernden* Methoden der Iteratoren optional. Falls ein Iterator eine dieser Methoden nicht zur Verfügung stellen will, löst er bei ihrem Aufruf eine Ausnahme des Typs *UnsupportedOperationException* aus. Das gilt für die Methoden *add( )*, *set( )* und *remove( )*.

Collections des Typs Set

Abstrakte Eigenschaften
Ein Set ähnelt einer List, erlaubt aber im Gegensatz zu dieser keine doppelten Elemente. Genauer gesagt, in einem Set darf es zu keinem Zeitpunkt zwei Elemente x und y geben, für die x.equals(y) wahr ist. Ein Set entspricht damit im mathematischen Sinn einer *Menge*, in der ja ebenfalls jedes Element nur einmal vorkommen kann. Ein Set hat allerdings keine spezielle Schnittstelle, sondern erbt seine Methoden von der Basisklasse *Collection*.
Die Unterschiede zu *List* treten zutage, wenn man versucht, mit *add( )* ein Element einzufügen, das bereits vorhanden ist (bei dem also die *equals*-Methode wie zuvor beschrieben *true* ergibt). In diesem Fall wird es nicht erneut eingefügt, sondern *add( )* gibt *false* zurück. Auch für die Methoden *addAll( )* und den Konstruktor *Set(Collection c)* gilt, dass sie kein Element doppelt einfügen und damit insgesamt die Integritätsbedingung einer Menge erhalten.
Ein weiterer Unterschied zu *List* ist, dass ein Set keinen ListIterator, sondern lediglich einen einfachen Iterator erzeugen kann. Dessen Elemente haben keine definierte Reihenfolge. Sie kann sich durch wiederholtes Einfügen von Elementen im Laufe der Zeit sogar ändern.

ANMERKUNG
Besondere Vorsicht ist geboten, wenn ein Set dazu benutzt wird, *veränderliche Objekte* zu speichern (sie werden auch als *mutable* bezeichnet). Wird ein Objekt, das in einem Set gespeichert ist, so verändert, dass der equals-Vergleich mit einem anderen Element danach einen anderen Wert ergeben könnte, so gilt der komplette Set als undefiniert (er ist in seiner Integrität verletzt) und darf nicht mehr benutzt werden!
Zentrale Ursache für dieses Problem ist die Tatsache, dass Objektvariablen intern als *Zeiger* auf die zugehörigen Objekte dargestellt werden. Auf ein Objekt, das in einem Set enthalten ist, kann somit auch von außen zugegriffen werden, wenn nach dem Einfügen des Objekts noch ein Verweis darauf zur Verfügung steht.

Implementierungen
Das *Set*-Interface wird im SDK nur von den Klassen *HashSet* und *AbstractSet* implementiert. Die abstrakte Implementierung dient lediglich als Basisklasse für eigene Ableitungen. In der voll funktionsfähigen *HashSet*-Implementierung werden die Elemente intern in einer HashMap gespeichert. Vor jedem Einfügen wird geprüft, ob das einzufügende Element bereits in der HashMap enthalten ist. Die Performance der Einfüge-, Lösch- und Zugriffsmethoden ist im Mittel *konstant*. Neben den oben erwähnten Standardkonstruktoren besitzt die Klasse *HashSet* zwei weitere Konstruktoren, deren Argumente direkt an den Konstruktor der HashMap weitergereicht werden:

```
HashSet(int initialCapacity);
HashSet(int initialCapacity, float loadFactor);
```

Das folgende Beispiel zeigt die Anwendung der Klasse *HashSet* am Beispiel der Generierung eines Lottotips:

```
import java.util.*;
public class LottoTips // 6 aus 49
{
  public static void main(String[] args)
  {
    HashSet set = new HashSet(10);
    int doubletten = 0;
    //Lottozahlen erzeugen
    while (set.size() < 6)
    {
      int num = (int)(Math.random() * 49) + 1; //Math.random liefert x,mit 0 <= x < 1
      if (!set.add(new Integer(num)))
      {
        doubletten++;
      }
    }
    //Lottozahlen ausgeben
    Iterator it = set.iterator();
    while (it.hasNext())
    {
      System.out.println(((Integer)it.next()).toString());
    }
    System.out.println("Ignorierte Doubletten: " + doubletten);
  }
}
```

Ein HashSet wird so lange mit Zufallszahlen zwischen 1 und 49 gefüllt, bis die Anzahl der Elemente 6 ist. Da in einen Set keine Elemente eingefügt werden können, die bereits darin enthalten sind, ist dafür gesorgt, dass jede Zahl nur einmal im Tip auftaucht. Der Zähler *doubletten* wird durch den Rückgabewert *false* von *add( )* getriggert. Er zählt, wie oft eine Zahl eingefügt werden sollte, die bereits enthalten war.

Collections des Typs Map

Abstrakte Eigenschaften
Eine Collection des Typs *Map* realisiert einen assoziativen Speicher, der Schlüssel auf Werte abbildet. Sowohl Schlüssel als auch Werte sind Objekte eines beliebigen Typs. Je Schlüssel gibt es entweder keinen oder genau einen Eintrag in der Collection. Soll ein Schlüssel-Wert-Paar eingefügt werden, dessen Schlüssel bereits existiert, wird dieses nicht neu eingefügt. Es wird lediglich dem vorhandenen Schlüssel der neue Wert zugeordnet. Der Wert wird also praktisch *ausgetauscht*.

Das Interface *Map* ist nicht von *Collection* abgeleitet. Es definiert folgende Methoden:

```
int               size();
boolean           isEmpty();
boolean           containsKey(Object key);
boolean           containsValue(Object value);
Object            get(Object key);
Object            put(Object key, Object value);
Object            remove(Object key);
void              putAll(Map t);
void              clear();
boolean           equals(Object o);
int               hashCode() ;

public Set        keySet();
public Collection values();
public Set        entrySet();
```

Die Methoden *size( )*, *isEmpty( )*, *remove( )*, *clear( )*, *equals( )* und *hashCode( )* sind mit den gleichnamigen Methoden des *Collection*-Interfaces identisch und brauchen daher nicht noch einmal erklärt zu werden.
Mit Hilfe von *put( )* wird ein neues Schlüssel-Wert-Paar eingefügt bzw. dem bereits vorhandenen Schlüssel ein neuer Wert zugeordnet. Die Methode *putAll( )* macht das für alle Paare der als Argument übergebenen Map. Auch hier gilt die Regel, dass ein Schlüssel, der bereits vorhanden ist, nicht neu eingefügt wird. Lediglich sein zugeordneter Wert wird ausgetauscht. Mit der Methode *get( )* kann der Wert zu einem Schlüssel beschafft werden. Da der Rückgabewert vom Typ *Object* ist, muss er auf den erwarteten Wert gecastet werden.

Falls der angegebene Schlüssel nicht in der Map enthalten ist, wird *null* zurückgegeben. Hier ist Vorsicht geboten, denn ein *null*-Wert wird auch dann zurückgegeben, wenn die Map das Einfügen von *null*-Werten erlaubt und ein solcher diesem Schlüssel explizit zugeordnet wurde. Um diese beiden Fälle zu unterscheiden, kann die Methode *containsKey( )* verwendet werden. Sie gibt genau dann *true* zurück, wenn der gewünschte Schlüssel in der Map enthalten ist. Andernfalls liefert sie *false*. Analog dazu kann mit *containsValue( )* festgestellt werden, ob ein bestimmter Wert mindestens einmal in der Map enthalten ist.

Im Vergleich zum *Collection*-Interface fällt auf, dass eine Map keine Methode *iterator( )* besitzt. Statt dessen kann sie drei unterschiedliche Collections erzeugen, die dann natürlich ihrerseits dazu verwendet werden können, einen Iterator zu liefern. Diese Collections werden als *Views*, also als *Sichten* auf die Collection bezeichnet:

- **Die Methode keySet( ) liefert die Menge der Schlüssel. Da per Definition keine doppelten Schlüssel in einer Map auftauchen, ist diese Collection vom Typ Set.**
- **Die Methode values( ) liefert die Menge der Werte der Map. Da Werte sehr wohl doppelt enthalten sein können, ist der Rückgabewert lediglich vom Typ Collection, wird also typischerweise eine Liste oder eine anonyme Klasse mit entsprechenden Eigenschaften sein.**
- **Die Methode entrySet( ) liefert eine Menge von Schlüssel-Wert-Paaren. Jedes Element dieser Menge ist vom Typ MapEntry, d.h. es implementiert das lokale Interface Entry des Interfaces Map. Dieses stellt u.a. die Methoden getKey( ) und getValue( ) zur Verfügung, um auf die beiden Komponenten des Paares zuzugreifen.**

Neben den im Interface definierten Methoden sollte eine konkrete Map-Implementierung zwei Konstruktoren zur Verfügung stellen. Ein leerer Konstruktor dient dazu, eine leere Map zu erzeugen. Ein zweiter Konstruktor, der ein einzelnes Map-Argument erwartet, erzeugt eine neue Map mit denselben Schlüssel-Wert-Paaren wie die als Argument übergebene Map.

Implementierungen
Das SDK stellt mehrere Implementierungen des *Map*-Interfaces zur Verfügung:

- **Mit AbstractMap steht eine abstrakte Basisklasse für eigene Ableitungen zur Verfügung.**
- **Die Klasse HashMap implementiert das Interface auf der Basis einer *Hashtabelle*. Dabei wird ein Speicher fester Größe angelegt, und die Schlüssel werden mit Hilfe der *Hash-Funktion*, die den Speicherort direkt aus dem Schlüssel berechnet, möglichst gleichmäßig auf die verfügbaren Speicherplätze abgebildet. Da die Anzahl der potentiellen Schlüssel meist wesentlich größer als die Anzahl der verfügbaren Speicherplätze ist, können beim Einfügen Kollisionen auftreten, die mit geeigneten Mitteln behandelt werden müssen (bei der HashMap werden alle kollidierenden Elemente in einer verketteten Liste gehalten).**
- **Die altbekannte Klasse Hashtable implementiert seit dem SDK 1.2 ebenfalls das Map-Interface. Ihre Arbeitsweise entspricht im Prinzip der von HashMap, allerdings mit dem Unterschied, dass ihre Methoden synchronisiert sind und dass es nicht erlaubt ist, null-Werte einzufügen (die HashMap läßt dies zu).**

Das folgende Beispiel ist sehr ähnlich dem in 16.2 gezeigten Beispiel zur Verwaltung von Mail-Aliasen:

```
import java.util.*;
public class EmailAliases2
{
  public static void main(String[] args)
  {
    HashMap h = new HashMap();
    //Pflege der Aliase
    h.put("Fritz","f.mueller@test.de");
    h.put("Franz","fk@b-blabla.com");
    h.put("Paula","user0125@mail.uofm.edu");
    h.put("Lissa","lb3@gateway.fhdto.northsurf.dk");
    //Ausgabe
    Iterator it = h.entrySet().iterator();
    while(it.hasNext())
    {
      Map.Entry entry = (Map.Entry)it.next();
      System.out.println(
        (String)entry.getKey() + " --> " + (String)entry.getValue()
      );
    }
  }
}
```

Der wichtigste Unterschied liegt in der Ausgabe der Ergebnisse. Während bei der Hashtable eine Enumeration verwendet wurde, muss man sich hier durch Aufruf von *entrySet( )* zunächst eine Menge der Schlüssel-Wert-Paare beschaffen. Diese liefert dann einen Iterator, der für jedes Element ein Objekt des Typs *Map.Entry( )* zurückgibt. Dessen Methoden *getKey( )* und *getValue( )* liefern den Schlüssel bzw. Wert des jeweiligen Eintrags. Alternativ hätte man auch mit *keySet( )* die Menge der Schlüssel durchlaufen und mit *get( )* auf den jeweils aktuellen Wert zugreifen können:

```
Iterator it = h.keySet().iterator();
while(it.hasNext())
{
  String key = (String)it.next();
  System.out.println(
    key + " --> " + (String)h.get(key)
  );
}
```

Sortierte Collections

Comparable und Comparator

Die bisher vorgestellten Set- und Map-Collections waren unsortiert, d.h. ihre Iteratoren haben die Elemente in keiner bestimmten Reihenfolge zurückgegeben. Im Gegensatz dazu gibt ein List-Iterator die Elemente in der Reihenfolge ihrer Indexnummern zurück. Im SDK gibt es nun auch die Möglichkeit, die Elemente eines Set oder einer Map zu sortieren. Dabei kann entweder die *natürliche Ordnung* der Elemente verwendet werden, oder die Elemente können mit Hilfe eines expliziten Vergleichsobjekts sortiert werden.

Sortieren mittels der natürlichen Ordnung der Elemente

Bei der natürlichen Ordnung muss sichergestellt sein, dass alle Elemente der Collection eine *compareTo( )*-Methode besitzen und je zwei beliebige Elemente miteinander verglichen werden können, ohne dass es zu einem Typfehler kommt. Dazu müssen die Elemente das Interface *Comparable* aus dem Package *java.lang* implementieren:

```
public int compareTo(Object o);
```

Comparable besitzt lediglich eine einzige Methode *compareTo( )*, die aufgerufen wird, um das aktuelle Element mit einem anderen zu vergleichen.

- **compareTo( ) muss einen Wert kleiner 0 zurückgeben, wenn das aktuelle Element *vor* dem zu vergleichenden liegt.**
- **compareTo( ) muss einen Wert größer 0 zurückgeben, wenn das aktuelle Element *hinter* dem zu vergleichenden liegt.**
- **compareTo( ) muss 0 zurückgeben, wenn das akt. Element und das zu vergleichende *gleich* sind.**

ANMERKUNG

Während in älteren SDKs bereits einige Klassen eine *compareTo( )*-Methode besaßen, wird seit dem SDK 1.2 das *Comparable*-Interface bereits von vielen der eingebauten Klassen implementiert, etwa von *String*, *Character*, *Double* usw.

Sortieren mittels eines expliziten Vergleichsobjekts

Die zweite Möglichkeit, eine Menge von Elementen zu sortieren, besteht darin, an den Konstruktor der Collection-Klasse ein Objekt zu übergeben, welches die Schnittstelle *Comparator* implementiert. *Comparator* ist ein Interface, das lediglich eine einzige Methode *compare( )* definiert:

```
public int compare(Object o1, Object o2);
```

Das übergebene *Comparator*-Objekt übernimmt die Aufgabe einer »Vergleichsfunktion«, deren Methode *compare( )* immer dann aufgerufen wird, wenn bestimmt werden muss, in welcher Reihenfolge zwei beliebige Elemente zueinander stehen.

SortedSet und TreeSet

Zur Realisierung von sortierten Mengen wurde aus *Set* das Interface *SortedSet* abgeleitet. Es erweitert das Basisinterface um einige interessante Methoden:

```
Object    first();
Object    last();
SortedSet headSet(Object toElement);
SortedSet subSet(Object fromElement, Object toElement);
SortedSet tailSet(Object fromElement);
```

Mit *first( )* und *last( )* kann das (gemäß der Sortierordnung) erste bzw. letzte Element der Menge beschafft werden. Die übrigen Methoden dienen dazu, aus der Originalmenge Teilmengen auf der Basis der Sortierung der Elemente zu konstruieren:

- **headSet( ) liefert alle Elemente, die echt kleiner als das als Argument übergebene Element sind.**
- **TailSet( ) liefert alle Elemente, die größer oder gleich dem als Argument übergebenen Element sind.**
- **SubSet( ) liefert alle Elemente, die größer oder gleich fromElement und kleiner als toElement sind.**

Ein parameterloser Konstruktor erzeugt eine leere Menge, deren (zukünftige) Elemente bezüglich ihrer natürlichen Ordnung sortiert werden.

- **Ein Konstruktor mit einem Argument des Typs Comparator erzeugt eine leere Menge, deren Elemente bezüglich der durch den Comparator vorgegebenen Ordnung sortiert werden.**
- **Ein Konstruktor mit einem Argument vom Typ Collection erzeugt eine Menge, die alle eindeutigen Elemente der als Argument übergebenen Collection in ihrer natürlichen Ordnung enthält.**
- **Ein Konstruktor mit einem Argument des Typs SortedSet erzeugt eine Menge mit denselben Elementen und derselben Sortierung wie die als Argument übergebene Menge.**

Die einzige Klasse im SDK 1.2 und 1.3, die das Interface *SortedSet* implementiert, ist *TreeSet*. Sie implementiert die sortierte Menge mit Hilfe der Klasse *TreeMap*. Diese verwendet einen sog. *Red-Black-Tree* als Datenstruktur, dabei handelt es sich um einen Baum, der durch spezielle Operationen davor geschützt wird, im Extremfall zu einer linearen Liste zu entarten. Alle Basisoperationen (Einfügen, Suchen, Löschen) können in logarithmischer Zeit bezüglich der Anzahl der Elemente des Baums ausgeführt werden und sind damit auch bei großen Elementzahlen recht schnell.

Interessant ist die Fähigkeit, einen sortierten Iterator zu erzeugen. Das folgende Programmbeispiel erzeugt eine sortierte Menge und fügt einige Strings unsortiert ein:

```
import java.util.*;
public class SortTest
{
  public static void main(String[] args)
  {
    //Konstruieren des Sets
    TreeSet s = new TreeSet();
    s.add("Kiwi");
    s.add("Kirsche");
    s.add("Ananas");
    s.add("Zitrone");
    s.add("Grapefruit");
    s.add("Banane");
    //Sortierte Ausgabe
    Iterator it = s.iterator();
    while (it.hasNext())
    {
      System.out.println((String)it.next());
    }
  }
}
```

Der Iterator gibt die Elemente in alphabetischer Reihenfolge aus. Der Grund dafür ist, dass die Klasse *String* seit dem SDK 1.2 das *Comparable*-Interface implementiert und eine Methode *compareTo( )* zur Verfügung stellt, mit der die Zeichenketten in lexikographischer Ordnung angeordnet werden.

Sollen die Elemente unserer Menge dagegen rückwärts sortiert werden, ist die vorhandene *compareTo( )*-Methode dazu nicht geeignet. Stattdessen kann ein *Comparator*-Objekt an den Konstruktor übergeben werden, dessen *compare( )*-Methode so implementiert wurde, dass zwei zu vergleichende Strings genau dann als aufsteigend beurteilt werden, wenn sie gemäß ihrer lexikographischen Ordnung absteigend sind.

Das folgende Listing zeigt dies am Beispiel der Klasse *ReverseStringComparator*:

```
import java.util.*;
public class ReverseSortTest
{
  public static void main(String[] args)
  {
    //Konstruieren des Sets
    TreeSet s = new TreeSet(new ReverseStringComparator());
    s.add("Kiwi");
    s.add("Kirsche");
    s.add("Ananas");
    s.add("Zitrone");
    s.add("Grapefruit");
    s.add("Banane");
    //Rückwärts sortierte Ausgabe
    Iterator it = s.iterator();
    while(it.hasNext())
    {
      System.out.println((String)it.next());
    }
  }
}

class ReverseStringComparator implements Comparator
{
  public int compare(Object o1, Object o2)
  {
    return ((String)o2).compareTo((String)o1);
  }
}
```

Mit Hilfe eines Comparators kann eine beliebige Sortierung der Elemente eines SortedSet erreicht werden. Wird ein Comparator an den Konstruktor übergeben, so wird die *compareTo( )* -Methode überhaupt nicht mehr verwendet, sondern die Sortierung erfolgt ausschließlich mit Hilfe der Methode *compare( )* des *Comparator*-Objekts. So können beispielsweise auch Elemente in einem SortedSet gespeichert werden, die das *Comparable*-Interface nicht implementieren.

SortedMap und TreeMap

Neben einem sortierten Set gibt es auch eine sortierte Map. Das Interface *SortedMap* ist analog zu *SortedSet* aufgebaut und enthält folgende Methoden:

```
Object     first();
Object     last();
SortedMap  headMap(Object toElement);
SortedMap  subMap(Object fromElement, Object toElement);
SortedMap  tailMap(Object fromElement);
```

Als konkrete Implementierung von *SortedMap* gibt es die Klasse *TreeMap*, die analog zu *TreeSet* arbeitet. Die Methoden *keySet( )* und *entrySet( )* liefern Collections, deren Iteratoren ihre Elemente in aufsteigender Reihenfolge abliefern. Auch bei einer SortedMap kann wahlweise mit der natürlichen Ordnung der Schlüssel gearbeitet werden oder durch Übergabe eines *Comparator*-Objekts an den Konstruktor eine andere Sortierfolge erzwungen werden.

Die Klasse Collections

Im Package *java.util* gibt es eine Klasse *Collections* (man achte auf das »s« am Ende), die eine große Anzahl statischer Methoden zur Manipulation und Verarbeitung von Collections enthält. Darunter finden sich Methoden zum Durchsuchen, Sortieren, Kopieren und Synchronisieren von Collections sowie solche zur Extraktion von Elementen mit bestimmten Eigenschaften. Im Folgenden werden nur einige der interessanten Methoden dieser Klasse betrachtet (für weitere Informationen siehe SDK-Dokumentation).

Sortieren und Suchen
Die Klasse *Collections* enthält zwei Methoden *sort( )*:

```
static void sort(List list);
static void sort(List list, Comparator c);
```

Mit Hilfe von *sort( )* können beliebige Listen sortiert werden. Als Argument werden die Liste und wahlweise ein Comparator übergeben. Fehlt der Comparator, wird die Liste in ihrer natürlichen Ordnung sortiert. Dazu müssen alle Elemente das *Comparable*-Interface implementieren und ohne Typfehler paarweise miteinander vergleichbar sein. Gemäß SDK-Dokumentation verwendet diese Methode ein modifiziertes Mergesort, das auch im Worst-Case eine Laufzeit von *n*log(n)* hat (auch bei der Klasse *LinkedList*) und damit auch für große Listen geeignet sein sollte.

Das folgenden Beispiel gezeigt, wie man Elemente einer Liste mit Hilfe der Methode *sort( )* sortieren kann:

```
import java.util.*;
public class SortTest2
{
  public static void main(String[] args)
  {
    //Konstruieren des Sets
    List l = new ArrayList();
    l.add("Kiwi");
    l.add("Kirsche");
    l.add("Ananas");
    l.add("Zitrone");
    l.add("Grapefruit");
    l.add("Banane");
    //Unsortierte Ausgabe
    Iterator it = l.iterator();
    while(it.hasNext())
    {
      System.out.println((String)it.next());
    }
    System.out.println("---");
    //Sortierte Ausgabe
    Collections.sort(l);
    it = l.iterator();
    while(it.hasNext())
    {
      System.out.println((String)it.next());
    }
  }
}
```

Muss in einer großen Liste wiederholt gesucht werden, macht es Sinn, diese einmal zu sortieren und anschließend eine *binäre Suche* zu verwenden. Dabei wird das gewünschte Element durch eine Intervallschachtelung mit fortgesetzter Halbierung der Intervallgröße immer weiter eingegrenzt, und das gesuchte Element ist nach spätestens log(n) Schritten gefunden.

Die binäre Suche wird mit Hilfe der Methoden *binarySearch( )* realisiert:

```
static int binarySearch(List list, Object key);
static int binarySearch(List list, Object key, Comparator c);
```

Auch hier gibt es wieder eine Variante, die gemäß der natürlichen Ordnung vorgeht, und eine zweite, die einen expliziten Comparator erfordert.

Synchronisieren von Collections
Es wurde bereits mehrfach erwähnt, dass die neuen Collections des SDK 1.2 nicht threadsicher sind und deshalb aus Performancegründen auf den Gebrauch des Schlüsselworts *synchronized* weitgehend verzichtet wurde. Damit in einer Umgebung, bei der von mehr als einem Thread auf eine Collection zugegriffen werden kann, nicht alle Manipulationen vom Aufrufer synchronisiert werden müssen, gibt es einige Methoden, die eine unsynchronisierte Collection in eine synchronisierte verwandeln:

```
static Collection synchronizedCollection(Collection c);
static List       synchronizedList(List list);
static Map        synchronizedMap(Map m);
static Set        synchronizedSet(Set s);
static SortedMap  synchronizedSortedMap(SortedMap m);
static SortedSet  synchronizedSortedSet(SortedSet s);
```

Die Methoden erzeugen jeweils aus der als Argument übergebenen Collection eine synchronisierte Variante und geben diese an den Aufrufer zurück. Erreicht wird dies, indem eine neue Collection desselben Typs erzeugt wird, deren sämtliche Methoden synchronisiert sind. Wird eine ihrer Methoden aufgerufen, leitet sie den Aufruf innerhalb eines *synchronized*-Blocks einfach an die als Membervariable gehaltene Original-Collection weiter.

Erzeugen unveränderlicher Collections
Analog zum Erzeugen von synchronisierten Collections gibt es einige Methoden, mit denen aus gewöhnlichen Collections *unveränderliche* Collections erzeugt werden können:

```
static Collection unmodifiableCollection(Collection c)
static List       unmodifiableList(List list)
static Map        unmodifiableMap(Map m)
static Set        unmodifiableSet(Set s)
static SortedMap  unmodifiableSortedMap(SortedMap m)
static SortedSet  unmodifiableSortedSet(SortedSet s)
```

Auch hier wird jeweils eine Wrapper-Klasse erzeugt, deren Methodenaufrufe an die Original-Collection delegiert werden. Alle Methoden, mit denen die Collection verändert werden könnte, werden so implementiert, dass sie beim Aufruf eine Ausnahme des Typs *UnsupportedOperationException* auslösen.

Übungsaufgaben:

- Testen Sie die angeführten Beispiele durch.

- Implementieren Sie eine Klasse *Telefonbuch* zur Verwaltung von Personen mit Telefonnummern, gemäß folgender Schnittstelle:

```
class Person extends Object  // Implementiert nicht Comparable!
{
     ...
     Person(String forename, String surname, String city, String phonenumber)
     //public String getForename()
     //public XXX getXXX()
     ...
}

class PhoneBook extends Object
{
   public static final int FORENAME = 0;
   public static final int SURNAME = 1;
   public static final int CITY = 2;
   public void insert(Person p)  // Ein Einfügen macht die Sortierung zunichte
   public void delete(int n)
   public int size()
   public boolean isSorted()     // true zwischen dem letzten sort() und insert(p)
   public int getLastSortMode()
   public List searchAll(String searchStr, int mode) // mode = FORENAME,SURNAME,CITY
   public Person searchFirst(String searchStr, int mode)
   public void sortAll(int mode)                     // Sortiert intern
   public ListIterator getIterator()
   public void printAll(OutputStream o) // Gibt alle Personen in einen Stream aus
                                        // z.B. System.out
   public Person searchByNumber(String number)
}
```

Hinweise:

Verwenden Sie zur internen Verwaltung der Personen ein Objekt der Klasse *ArrayList*. Zum Sortieren benötigen Sie drei Comparator-Objekte (für jeden Modus eines). Schreiben Sie dazu drei *Comparator*-Klassen. Ebenso benötigen Sie die Klasse *Collections*.
Insert( ) fügt nicht sortiert ein; eine eventuelle Sortierung wird nach einem Einfügevorgang ungültig. Durch den Aufruf von *isSorted( )* kann erfragt werden, ob das Telefonbuch gerade sortiert ist. Sortiert wird immer aufsteigend. *printAll( )* und *searchAll( )* sind mit Iteratoren zu implementieren.
Bei *searchAll( )* wird eine neue Liste erzeugt, die genau diejenigen Objekte enthält, die den searchStr beinhalten (je nach Modus).
Die Methode *getIterator( )* gibt einen ListIterator der internen ArrayList zurück.
Die Methode *getLastSortMode( )* gibt den zuletzt verwendeten Modus zurück.
Implementieren Sie keine static-Methoden.

Die Klasse Telefonbuch soll auch eine effiziente Suche bei gegebener Telefonnummer ermöglichen! Beachten Sie, dass jede Person genau eine Telefonnummer hat und jede Telefonnummer eindeutig ist. (Fehlerbehandlung bei *insert( )*!). Verwenden Sie dazu eine zusätzliche Collection-Klasse zur ArrayList, die in der Lage ist Schlüssel (Telefonnummern) auf Werte (Personen) abzubilden (nicht Hashtable!). Achten Sie darauf, dass für jede Person genau eine Referenz in beiden Collection-Objekten enthalten ist!

Schreiben Sie eine Klasse *TelefonbuchTester* zum Testen der Klasse *Telefonbuch*.

- Implementieren Sie eine eigene Variante einer Queue (Warteschlange), eventuell mit Prioritäten

17. Multimedia

17.1 Bilder

Laden und Anzeigen einer Bitmap
Das Anzeigen einer Bitmap kann in zwei Schritte unterteilt werden:

- **das Laden der Bitmap von einem externen Speichermedium oder aus dem Netzwerk**
- **die eigentliche Ausgabe auf den Bildschirm**

Das Laden erfolgt mit der Methode *getImage( )*, die eine Instanz der Klasse *Image* zurückgibt. Das *Image*-Objekt kann dann mit der Methode *drawImage( )* der Klasse *Graphics* angezeigt werden.
getImage( ) gibt es in verschiedenen Varianten, die sich dadurch unterscheiden, aus welcher Quelle sie die Bitmap-Daten laden. In einer Java-*Applikation* wird in der Regel die Methode *getImage( )* aus der Klasse *ToolKit* verwendet. Sie erwartet den Namen einer lokalen Datei als Parameter:

```
public Image getImage(String filename);
```

ANMERKUNG
getImage( ) versteht in der aktuellen Version des AWT die beiden Bitmap-Typen *gif* und *jpeg*. Andere Grafikformate, wie etwa das unter Windows gebräuchliche *bmp*-Format, werden nicht unterstützt, sondern müssen bei Bedarf konvertiert werden.

Neben den *getImage( )*-Methoden gibt es seit dem SDK 1.2 auch zwei Methoden mit dem Namen *createImage( )* in der Klasse *ToolKit*:

```
public abstract Image createImage(String filename);
public abstract Image createImage(URL url);
```

Sie laden ein Image bei jeder Verwendung neu und führen (im Gegensatz zu *getImage( )*) kein Caching des Bildes durch. Die SDK-Dokumentation empfiehlt sie gegenüber *getImage( )*, weil bei deren Verwendung Speicherlecks durch das unbegrenzte Zwischenspeichern der Bilddaten entstehen können.

Das ToolKit für die aktuelle Umgebung kann mit der Methode *getToolKit( )* der Klasse *Component* beschafft werden:

```
public Toolkit getToolkit();
```

Der gesamte Code zum Laden einer Bitmap *java.gif* sieht daher so aus:

```
Image img;
img = getToolkit().getImage("java.gif");
```

Um das Image anzuzeigen, kann die Methode *drawImage( )* der Klasse *Graphics* aufgerufen werden:

```
public boolean drawImage(Image img, int x, int y, ImageObserver observer) ;
```

drawImage( ) gibt es in unterschiedlichen Ausprägungen. Die hier vorgestellte Variante erwartet das anzuzeigende Image-Objekt und die Position (x,y), an der die linke obere Ecke der Bitmap plaziert werden soll. Das zusätzlich angegebene Objekt *observer* dient zur Übergabe eines *ImageObserver*-Objektes, mit dem der Ladezustand der Bitmaps überwacht werden kann. Hier kann die *this*-Referenz, also eine Referenz auf das Fensterobjekt, übergeben werden. Weitere Varianten von *drawImage( )*, die ein Bild sogar skalieren und spiegeln können, werden in der SDK-Hilfe beschrieben.

Das folgende Listing ist ein einfaches Beispiel für das Laden und Anzeigen der Bitmap *duke.gif*. Alle erforderlichen Aktionen erfolgen innerhalb der *paint( )*-Methode:

```
public void paint(Graphics g)
{
   Image img;
   img = getToolkit().getImage("java.gif");
   g.drawImage(img,40,40,this);
}
```

Bildschirmausgabe:

ANMERKUNG
Die gewählte Vorgehensweise ist nicht besonders effizient, denn die Bitmap wird bei jedem Aufruf von *paint( )* neu geladen. Besser ist es, die benötigten Bitmaps einmal zu laden und dann im Speicher zu halten. Obwohl man vermuten könnte, dass dies die Ladezeit des Fensters unannehmbar verlängern würde, ist der Konstruktor der Klasse eine gute Stelle dafür. Der Aufruf von *getImage( )* lädt die Bitmap nämlich noch nicht, sondern bereitet das Laden nur vor. Der eigentliche Ladevorgang erfolgt erst, wenn die Bitmap beim Aufruf von *drawImage( )* tatsächlich benötigt wird.

17.2 Die Klasse MediaTracker

Manchmal kann es sinnvoll sein, den tatsächlichen Ladevorgang des Bildes abzuwarten, bevor im Programm fortgefahren wird. Wird zum Beispiel die Größe der Bitmap benötigt, um sie korrekt auf dem Bildschirm anordnen oder skalieren zu können, muss das Programm warten, bis das Image vollständig erzeugt ist.
Für diese Zwecke steht die Klasse *MediaTracker* zur Verfügung, die das Laden eines oder mehrerer Bilder überwacht. Dazu wird zunächst eine Instanz der Klasse angelegt:

```
public MediaTracker(Component comp);
```

Als Komponente wird die *this*-Referenz des aktuellen Fensters übergeben. Anschließend werden durch Aufruf von *addImage( )* alle Bilder, deren Ladevorgang überwacht werden soll, an den MediaTracker übergeben:

```
public void addImage(Image img, int id);
```

Der zusätzlich übergebene Parameter *id* kann dazu verwendet werden, einen Namen zu vergeben, unter dem auf das Iamge zugegriffen werden kann. Zusätzlich bestimmt er die Reihenfolge, in der die Images geladen werden. Bitmaps mit kleineren Werten werden zuerst geladen.

Der MediaTracker bietet eine Reihe von Methoden, um den Ladezustand der Bilder zu überwachen. Hier wird nur nur die Methode *waitForAll( )* betrachtet. Sie wartet, bis alle Images vollständig geladen sind:

```
public void waitForAll() throws InterruptedException;
```

Nach Abschluß des Ladevorgangs sendet *waitForAll( )* eine Ausnahme des Typs *InterruptedException*. Das vollständige Beispielprogramm zur Anzeige von *java.gif* sieht nun so aus:

```
import java.awt.*;
import java.awt.event.*;
public class Image2 extends Frame
{
  private Image img;
  public static void main(String[] args)
  {
    Image2 wnd = new Image2();
  }

  public Image2()
  {
    super("Image2");
    setBackground(Color.lightGray);
    setSize(250,150);
    setVisible(true);
    //WindowListener
    addWindowListener(new WindowClosingAdapter(true));
    //Bild laden
    img = getToolkit().getImage("java.gif");
    MediaTracker mt = new MediaTracker(this);
    mt.addImage(img, 0);
    try
    {
      //Warten, bis das Image vollständig geladen ist,
      mt.waitForAll();
    }
    catch (InterruptedException e)
    {
      //nothing
    }
    repaint();
  }

  public void paint(Graphics g)
  {
    if (img!= null)
    {
      g.drawImage(img,40,40,this);
    }
  }
}
```

17.3 Sound

Soundausgabe in Applets

Das SDK bietet auch einige Möglichkeiten, Sound auszugeben. Hierbei muss klar zwischen dem SDK 1.2 und seinen Vorgängern unterschieden werden. Während das SDK 1.2 die Soundausgabe sowohl Applikationen als auch Applets ermöglicht, war diese vorher nur für Applets möglich. Dabei war die Ausgabe auf gesampelte Sounds beschränkt, die im AU-Format vorliegen mussten. Das AU-Format stammt aus der Sun-Welt und legt ein Sample im Format 8 Bit Mono, Sampling-Rate 8 kHz, µ-law-Kompression ab. Seit dem SDK 1.2 werden dagegen auch die Sample-Formate WAV und AIFF sowie die MIDI-Formate

Typ 0 und *Typ 1* und RMF unterstützt. Zudem gibt es einige Shareware- oder Freeware-Tools, die zwischen verschiedenen Formaten konvertieren können (z.B. CoolEdit oder GoldWave). Mit dem SDK 1.3 wurden die Fähigkeiten erneut erweitert. Mit der nun im SDK enthaltenen Sound-Engine kann Musik nicht nur wiedergegeben, sondern auch aufgenommen und bearbeitet werden, und es ist möglich, Zusatzgeräte wie Mixer, Synthesizer oder andere Audiogeräte anzusteuern.

Die Ausgabe von Sound ist denkbar einfach und kann auf zwei unterschiedliche Arten erfolgen. Zum einen stellt die Klasse *Applet* die Methode *play( )* zur Verfügung, mit der eine Sound-Datei geladen und abgespielt werden kann:

```
public void play(URL url);
public void play(URL url, String name);
```

Hierbei kann entweder die *URL* einer Sound-Datei oder die Kombination von Verzeichnis-URL und Dateinamen angegeben werden. Üblicherweise wird zur Übergabe der Verzeichnis-URLs eine der Applet-Methoden *getCodeBase( )* oder *getDocumentBase( )* verwendet. Diese liefern eine URL des Verzeichnisses, aus dem das Applet gestartet wurde bzw. in dem die aktuelle HTML-Seite liegt:

```
public URL getCodeBase();
public URL getDocumentBase() ;
```

Der Nachteil dieser Vorgehensweise ist, dass die Sound-Datei bei jedem Aufruf neu geladen werden muss. In der zweiten Variante wird zunächst durch einen Aufruf von *getAudioClip( )* ein Objekt der Klasse *AudioClip* beschafft, das dann beliebig oft abgespielt werden kann:

```
public getAudioClip(URL url, String name);
```

AudioClip stellt die drei Methoden *play( )*, *loop( )* und *stop( )* zur Verfügung:

```
public void play();
public void loop();
public void stop();
```

play( ) startet die zuvor geladene Sound-Datei und spielt sie genau einmal ab. *loop( )* startet sie ebenfalls, spielt den Sound in einer Endlosschleife aber immer wieder ab. Durch Aufruf von *stop( )* kann diese Schleife beendet werden. Es ist auch möglich, mehr als einen Sound gleichzeitig abzuspielen. So kann beispielsweise eine Hintergrundmelodie in einer Schleife immer wieder abgespielt werden, ohne dass die Ausgabe von zusätzlichen Vordergrund-Sounds beeinträchtigt würde.
Das folgende Beispiel ist eine neue Variante des »Hello, World«-Programms. Anstatt der textuellen Ausgabe stellt das Applet zwei Buttons zur Verfügung, mit denen die Worte »Hello« und »World« abgespielt werden können:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class HWApplet extends Applet implements ActionListener
{
  Button    hello;
  Button    world;
  AudioClip helloClip;
  AudioClip worldClip;

  public void init()
  {
    super.init();
    setLayout(new FlowLayout());
```

```
    hello = new Button("Hello");
    hello.addActionListener(this);
    add(hello);
    world = new Button("World");
    world.addActionListener(this);
    add(world);
    helloClip = getAudioClip(getCodeBase(),"hello.wav");
    worldClip = getAudioClip(getCodeBase(),"world.wav");
  }

  public void actionPerformed(ActionEvent event)
  {
    String cmd = event.getActionCommand();
    if(cmd.equals("Hello"))
    {
      helloClip.play();
    }
    else
    {
      if(cmd.equals("World"))
      {
        worldClip.play();
      }
    }
  }
}
```

Soundausgabe in Applikationen

Seit dem SDK 1.2 kann nicht nur in Applets, sondern auch in Applikationen Sound ausgegeben werden. Dazu bietet die Klasse *Applet* eine statische Methode *newAudioClip( )*:

```
public static AudioClip newAudioClip(URL url);
```

Da es sich um eine Klassenmethode handelt, kann sie auch außerhalb eines Applets aufgerufen werden. Das folgende Beispiel spielt eine Sounddatei maximal 10 Sekunden lang ab:

```
import java.net.*;
import java.applet.*;
public class PlaySound
{
  public static void main(String[] args)
  {
    if(args.length >= 1)
    {
      try
      {
        URL url = new URL(args[0]);
        AudioClip clip = Applet.newAudioClip(url);
        clip.play();
        try
        {
          Thread.sleep(10000);       // 10 Sekunden warten
        }
        catch (InterruptedException e) {}
      }
      catch (MalformedURLException e) { System.out.println(e.toString()); }
    }
  }
}
```

Übungsaufgaben:

- Testen Sie die angeführten Beispiele durch.

18. Quellenverzeichnis

Nur wenige der angeführten Beispiele stammen aus meiner Hand, die meisten stammen aus dem Buch *Handbuch der Java-Programmierung* von Guido Krüger, das wirklich empfehlenswert ist. Im folgenden sind meine Quellen aufgelistet, sowie einige interessante Links:

Bücher, Unterlagen:
- *Handbuch der Java-Programmierung*, Guido Krüger 3. Auflage © 2002
  ca. 1200 Seiten, Addison Wesley ISBN 3-8273-1949-8
  Die HTML-Version steht zum freien Download zur Verfügung (siehe Link)
- *Das Java-Grundlagenbuch* © 1999
  ca. 930 Seiten, DATA BECKER ISBN 3-8158-1384-0
- *Internetprogrammierung mit Java*, Seminarunterlagen von F.Brandl (HTBLA Linz) © 2000

Links:

- http://java.sun.com	Webseite von Sun zum Thema Java (die Java-Seite!)
- http://www.javabuch.de	Homepage des Buches *Handbuch der Java-Programmierung*
- http://www.c-lab.de/java/21Tage	Homepage des Buches *Java in 21 Tagen*
- http://www.selfjava.de	Ein guter Online-Kurs für Java
- http://www.jcreator.com	Homepage von *JCreator*

Folgende Dokumente bzw. Software-Pakete werden von mir zur Verfügung gestellt:

- Skriptum	*Java – 3.Auflage.pdf*
- Java 2 SDK 1.4.2_01 für Windows 98/NT4/ME/2000/XP	*j2sdk-1_4_2_01-windows-i586.zip*
- Java 2 SDK 1.4.2 Dokumentation	*j2sdk-1_4_2-doc.zip*
- *JCreator*, eine einfache Entwicklungsumgebung für Java (Freeware)	*jcrea250.zip*
- *Handbuch der Java-Programmierung*, das Buch im HTML-Format	*hjp3html.zip*, *hjp3exam.zip*
- *Java in 21 Tagen*, das Buch im HTML-Format	*java21.zip*
- *SelfJava*, der Java-Kurs im HTML-Format	*selfjava.zip*