Intensivkurs Java

Prof. Dr. Karl Friedrich Gebhardt

Auflage vom 10. Oktober 2019

Prof. Dr. K. F. Gebhardt

Tel: 0711-667345-11(16)(15)(12)
Fax: 0711-667345-10
email: kfg@lehre.dhbw-stuttgart.de

Vorwort

Das vorliegende Skriptum ist die Arbeitsunterlage für einen drei- bis viertägigen Java Intensivkurs. Es eignet sich daher nur bedingt zum Selbstudium.

Sprachkenntnisse einer höheren Programmiersprache (z.B. C, Pascal, FORTRAN, BASIC) sind erfahrungsgemäß eine Voraussetzung, um den Stoff in drei bis vier Tagen zu erarbeiten.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel wird an Hand von drei Beispielen gezeigt, wie ein Java-Programm aussieht, wie es übersetzt und zum Laufen gebracht wird. Bei der Gelegenheit wird ein Überblick über die wichtigsten Aspekte von Java gegeben. Im zweiten Abschnitt geben wir einige Informationen über die Geschichte und den Entwicklungsstand von Java. Schließlich werden die wesentlichen Züge objekt-orientierter Programmierung zusammengefasst.

Java ist als Sprache nicht allzu schwierig, wesentlich leichter jedenfalls als C oder C++. Aber mit der Sprache wird eine **Anwendungs-Programmier-Schnittstelle (Kernklassen-Bibliothek, *application programming interface, API, Java platform)*** mitgeliefert, die sehr mächtige Programmier-Werkzeuge zur Verfügung stellt, aber die eben auch gelernt werden müssen. Das API besteht inzwischen (Java SE 5) aus 166 Paketen mit insgesamt 3562 Klassen. Das API ist auch unter der Bezeichnung **JFC *(Java Foundation Classes)*** bekannt.

Damit immer klar ist, was ein Schlüsselwort der Sprache ist oder was aus dem API kommt, verwenden wir in unseren Beispielen deutsche Worte oder Begriffe für die eigenen Namen von Klassen, Objekten und Methoden.

1.1 Erste Java-Programme

1.1.1 "Hello-World"-Programm

Pflichtübung für jede Sprache ist ein "Hello-World"-Programm:

```
public class   GutenTag
   {
   public static void   main (String[] argument)
      {
      System.out.println ("Guten Tag!");
      }
   }
```

Wie jedes Java-Programm besteht das Programm aus einer **öffentlichen Klassendefinition** `public class`, wobei die Implementation der Klasse zwischen geschweiften Klammern steht.

Die Datei, in der diese Klassendefinition steht, muss denselben Namen haben wie die Klasse mit der Erweiterung `.java`:

```
GutenTag.java
```

Diese Datei wird mit

```
$ javac GutenTag.java
```

übersetzt, wobei eine Datei `GutenTag.class` generiert wird. Diese Datei enthält den berühmten **Bytecode**, der aus Maschinen-Instruktionen für die ***Java Virtual Machine*** **(JVM)** besteht. Um das Programm laufen zu lassen, wird der Java-Interpreter (bzw die JVM) mit dem Namen der Klasse aufgerufen (`$` sei der System-Prompt.):

```
$ java GutenTag
```

Das ergibt:

```
Guten Tag!
```

Die Klasse `GutenTag` enthält eine Methode, Funktion oder Prozedur `main`, die angibt, wo der Java-Interpreter mit der Ausführung des Programms zu beginnen hat.

Die Methode `main` ist `public` deklariert, damit sie außerhalb der Übersetzungseinheit bekannt ist und gestartet werden kann. Ferner ist sie `static` deklariert, womit sie als eine **Klassenmethode** ***(class method)*** definiert wird, die ohne Referenz auf ein Objekt der Klasse aufgerufen werden kann. Jede Methode in Java muss einen Rückgabewert haben. Da `main` nichts zurückgibt, muss als Rückgabewert `void` spezifiziert werden. Übergabeparameter von `main` ist ein Feld `argument` von Zeichenketten `String`, das eventuell beim Programmaufruf mitgegebene Argumente enthält.

Die Implementation von `main` steht zwischen geschweiften Klammern und besteht aus einer Zeile, in der aus der API-Klasse `System` das Standard-Ausgabe Streamobjekt `out` und dessen Methode `println (String s)` angesprochen werden.

1.1.2 Produkt von zwei Zahlen

Mit folgendem Programm können wir interaktiv das Produkt von zwei Zahlen berechnen. Dieses Programm gestaltet sich in Java wesentlich komplizierter als in anderen Sprachen:

```
// Produkt von zwei Zahlen

import java.io.*;

public class Produkt
   {
```

```
public static void   main (String[] argument)
   {
   double[] zahl = new double[2];

   for (int i = 0; i < 2; i++)
      {
      System.out.print ("Bitte Zahl ");
      System.out.print (i + 1);
      System.out.print (" eingeben: ");
      System.out.flush ();
      try
         {
         zahl[i] = new Double (
            new BufferedReader (
            new InputStreamReader (System.in))
            .readLine ()).doubleValue ();
         // oder ausführlich:
         //InputStreamReader  s = new InputStreamReader (System.in);
         //BufferedReader  t = new BufferedReader (s);
         //Double d = new Double (t.readLine ());
         //zahl[i] = d.doubleValue ();
         }
      catch (IOException e)
         {
         System.err.println ("IOException wurde geworfen!");
         }
      catch (NumberFormatException e)
         {
         System.err.println ("NumberFormatException wurde geworfen!");
         }
      }

   System.out.print ("Das Produkt von ");
   System.out.print (zahl[0]);
   System.out.print (" mit ");
   System.out.print (zahl[1]);
   System.out.print (" beträgt ");
   System.out.println (zahl[0] * zahl[1]);
   }
}
```

Die erste Zeile ist ein Kommentar. Die Zeichen `//` lassen einen Kommentar beginnen, der am Ende der Zeile aufhört. Eine zweite Möglichkeit zu kommentieren besteht in der Verwendung von C-Kommentaren zwischen den Zeichenfolgen `/*` und `*/`. Sogenannte `javadoc`-Kommentare `/**  */` werden von dem Programm `javadoc` verarbeitet. Damit kann eine Dokumentation des Source-Codes als HTML-Datei erstellt werden. `javadoc`-Kommentare können nicht geschachtelt werden.

Die Zeile `import java.io.*` erlaubt dem Programmierer, abgekürzte Namen für die Klassen der Ein- und -Ausgabe-Ströme zu verwenden.

Wir müssen wieder eine Klasse – hier `Produkt` – mit einer Methode `main` definieren.

In der Zeile `double ...` wird ein Feld `zahl` von Gleitkommazahlen definiert. Ferner wird mit `new` Speicher für zwei Gleitkommazahlen angelegt.

Mit der iterativen Kontrollstruktur `for` werden Gleitkommazahlen von der Standard-Eingabe eingelesen und im Feld `zahl` gespeichert. Die entscheidenden Zeilen sind

```
zahl[i] = new Double (
   new BufferedReader (
   new InputStreamReader (System.in))
   .readLine ()).doubleValue ();
```

Da die Standard-Eingabe `System.in` keine geeigneten Methoden zur Verfügung stellt, um Zeichenketten oder Zahlen zu lesen, legen wir mit

```
new InputStreamReader (System.in)
```

ein anonymes Objekt der Klasse `InputStreamReader` an, mit dem wir wiederum ein anonymes Objekt der Klasse `BufferedReader` anlegen, die wenigstens eine Methode `readLine ()` zum Einlesen von Zeichenketten zur Verfügung stellt. Anwendung dieser Methode gibt uns ein anonymes Objekt der Klasse `String` zurück, mit dem wir ein anonymes Objekt der Klasse `Double` anlegen. Die Methode `doubleValue ()` der Klasse `Double` gibt uns dann eine Gleitkommazahl vom Typ `double` zurück.

Die Methode `readLine ()` kann eine ***Exception*** vom Typ `IOException` werfen. Der Konstruktor `Double (String s)` der Klasse `Double` kann eine Exception vom Typ `NumberFormatException` werfen. Die `IOException` muss entweder in einem `try-catch`-Block abgefangen oder deklariert werden (Genaueres siehe Kapitel "Ausnahmebehandlung").

Nachdem beide Zahlen eingelesen sind, wird schließlich das Ergebnis ausgegeben.

```
$ javac Produkt.java
$ java Produkt
Bitte Zahl 1 eingeben: 3.2
Bitte Zahl 2 eingeben: 6
Das Produkt von 3.2 mit 6 beträgt 19.200000000000003
$
```

Das nächste Beispiel wäre in einer anderen Sprache sehr aufwendig zu programmieren. In Java wird es Dank des API relativ einfach.

1.1.3 Schmier-Programm

Bisher haben wir die Java-Programme immer als **Anwendung** ***(application)*** gestartet. Das folgende Programm kann auch innerhalb eines **Applet-Viewers** ***(applet viewer)*** oder Web-Browsers gestartet werden, da wir eine Klasse definiert haben, die von der Klasse `JApplet` erbt. Diese und alle anderen Klassen die mit "`J`" beginnen, befinden sich in dem Paket `javax.swing`, das alle Komponenten der graphischen Benutzeroberfläche enthält.

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

public class   Schmier extends JApplet
   implements ActionListener, ItemListener, MouseMotionListener
   {
   private Container behälter;
   private int altesX = 0;
   private int altesY = 0;
   private Color  aktuelleFarbe = Color.black;
   private JButton   loeschKnopf;
   private JComboBox farbWahl;
   private JButton   endeKnopf;

   private static final String   schwarz = "Schwarz";
   private static final String   rot = "Rot";
   private static final String   gelb = "Gelb";
   private static final String   grün = "Grün";
   private static final String   blau = "Blau";

   public void init ()
      {
      behälter = this.getContentPane ();
      behälter.setLayout (new FlowLayout ());
      behälter.setBackground (Color.gray);

      loeschKnopf = new JButton ("Löschen");
      loeschKnopf.addActionListener (this);
      loeschKnopf.setForeground (Color.black);
      loeschKnopf.setBackground (Color.lightGray);
      behälter.add (loeschKnopf);

      farbWahl = new JComboBox ();
      farbWahl.addItemListener (this);
      farbWahl.addItem (schwarz);
      farbWahl.addItem (rot);
      farbWahl.addItem (gelb);
      farbWahl.addItem (grün);
      farbWahl.addItem (blau);
```

```
   farbWahl.setForeground (Color.black);
   farbWahl.setBackground (Color.lightGray);
   behälter.add (new JLabel ("Farbe: "));
   behälter.add (farbWahl);

   behälter.addMouseMotionListener (this);
   }

public void actionPerformed (ActionEvent ereignis)
   {
   Object   ereignisQuelle = ereignis.getSource ();
   if (ereignisQuelle == loeschKnopf)
      {
      repaint ();
      }
   else if (ereignisQuelle == endeKnopf)
      {
      System.exit (0);  // Nur bei Applikationen erlaubt,
                        // nicht erlaubt bei Applets
      }
   }

public void itemStateChanged (ItemEvent e)
   {
   if (e.getItem () == schwarz) aktuelleFarbe = Color.black;
   else if (e.getItem () == rot) aktuelleFarbe = Color.red;
   else if (e.getItem () == gelb) aktuelleFarbe = Color.yellow;
   else if (e.getItem () == grün) aktuelleFarbe = Color.green;
   else if (e.getItem () == blau) aktuelleFarbe = Color.blue;
   }

public void mouseDragged (MouseEvent e)
   {
   Graphics g = behälter.getGraphics ();
   g.setColor (aktuelleFarbe);
   g.drawLine (altesX, altesY, e.getX (), e.getY ());
   altesX = e.getX ();
   altesY = e.getY ();
   }

public void mouseMoved (MouseEvent e)
   {
   altesX = e.getX ();
   altesY = e.getY ();
   }

public void addiere (JComponent komponente)
   {
   behälter.add (komponente);
```

```
    }

  public static void   main (String[] argument)
    {
    Schmier  s = new Schmier ();
    s.init ();
    s.endeKnopf = new JButton ("Ende");
    s.endeKnopf.addActionListener (s);
    s.endeKnopf.setForeground (Color.black);
    s.endeKnopf.setBackground (Color.lightGray);
    s.addiere (s.endeKnopf);

    JFrame   f = new JFrame ("Schmier");
    f.pack ();  // Trick: Erzeugt Peer-Frame

    f.getContentPane ().add (s, BorderLayout.CENTER);
    f.setSize (600, 400);
    s.start ();
    f.setVisible (true);
    }

  }
```

Um auf Ereignisse wie das Drücken der Maus oder das Anklicken eines Knopfs reagieren zu können, werden von der Klasse `Schmier` verschiedene *Ereignis-Empfänger* implementiert: `ActionListener`, `ItemListener` und `MouseMotionListener`. D.h. diese Schnittstellen geben an, welche Methoden implementiert werden müssen, damit das empfangene Ereignis behandelt werden kann. Das sind beim `ActionListener` die Methode `actionPerformed`, bei `ItemListener` die Methode `itemStateChanged` und bei `MouseMotionListener` die Methoden `mouseDragged` und `mouseMoved`.

Die Ereignisse werden von den verschiedenen GUI-Komponenten `JButtons`, `JComboBox` und dem Behälter `JApplet` bei entsprechenden Aktionen des Benutzers erzeugt.

Die Methoden

```
XXXEreignisErzeuger.addXXXListener (XXXEreignisEmpfänger)
```

sagen dem Ereignis-Erzeuger, wer das Ereignis empfängt und behandelt.

Damit die Ereignis-Klassen bekannt sind, muss das Statement

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

gegeben werden.

Damit das Programm als Applet laufen kann, muss die Klasse `Schmier` von der Klasse `JApplet` erben:

```
public class   Schmier extends JApplet
```

Der Name der Klasse `JApplet` wird durch das Statement

```
import javax.swing.*;
```

bekannt gemacht.

Um `Schmier` auch als Anwendung laufen lassen zu können, haben wir die Methode `main` definiert, die die graphischen Voraussetzungen schafft und die Methoden `init` und `start` aufruft. (Bem.: Nicht mit jedem Applet – insbesondere wenn Parameter übergeben werden – geht das so einfach.)

Um in einem `JApplet` oder einem `JFrame` GUI-Komponenten unterbringen zu können, muss man sich von den Objekten dieser Klassen mit der Methode `getContentPane ()` einen GUI-Komponenten-Behälter (Klasse `Container`) holen.

Übersetzt wird dieses Programm wieder mit:

```
$ javac Schmier.java
```

Als Anwendung kann es laufen mit:

```
$ java Schmier
```

Um es aber als Applet laufen zu lassen, müssen wir es in einer HTML-Datei referenzieren. Ein Beispiel dafür ist die Datei `Schmier.html`:

```
<HTML>
<HEAD>
<TITLE>Das Schmier-Applet</TITLE>
</HEAD>
<BODY>
Wenn Sie einen Java-fähigen Browser benutzen,
bitte schmieren Sie im unten angegebenen Applet rum.
Ansonsten haben Sie Pech gehabt.
<P>
<APPLET code="Schmier.class" width=600 height=400>
</APPLET>
</BODY>
</HTML>
```

Mit dem Applet-Viewer `appletviewer` können wir nun das Applet anschauen:

```
$ appletviewer Schmier.html
```

Alternativ kann man die Datei `Schmier.html` auch mit einem Web-Browser anschauen.

```
$ netscape file:‘pwd‘/Schmier.html
```

1.2 Geschichte und Entwicklungsstand von Java

Prinzip: *It is more important that Java programs are easy to* ***read*** *than to* ***write.***

- 1990: James Gosling entwickelt bei Sun eine Architektur-unabhängige, C++ ähnliche, aber einfachere Sprache "Oak" für Unterhaltungs-Elektronik-Software.
- 1993: Das Java-Team (Gosling, Joy, Steele, Tuck, Yellin, van Hoff) entwickelt Oak unter dem Namen "Java" weiter für das Internet.
- 1995: Erstes Java Developer Kit (JDK 1.0)
- 1996: JDK 1.0.2 erste brauchbare Version
- 1997: Version JDK 1.1 (innere Klassen, ziemliche Änderungen im API)
- 1998: Version JDK 1.2 heißt jetzt Java 2 (Compiler wurde strenger, `swing`)
- 1999: Java 2, J2SE, J2ME, J2EE
  Versionsbezeichnung:
  - Java 2 SDK
    SDK: Software Development Kit, SE: Standard Edition, ME: Micro Edition, EE: Enterprise Edition
- Die aktuelle Version ist `Java SE 6`

Wir werden in diesem Skriptum nicht diskutieren, welche Syntax- oder Bibliotheks-Varianten in welcher Version möglich waren oder sind, sondern werden versuchen, die gerade aktuellste Version darzustellen.

Merkmale von Java:

- Plattformunabhängigkeit (Einsatz in allen möglichen Systemen: Mobiltelefon, Modem, Drucker, Fernsehen, Waschmaschine, Prozessautomatisierung, Bankensoftware)
- JVM (Java Virtual Machine) kann Betriebssystem für alle Computer sein. Ist etwa 200 KByte groß.
- Funktionalität und Interaktivität für WWW
- Software muss nicht mehr gespeichert werden, sondern kann übers Netz auf aktuellem Stand geholt werden.
- Java:
  - einfach
  - objekt-orientiert
  - verteilt
  - interpretiert
  - robust
  - sicher

- architekturunabhängig
- portabel
- hochleistungsfähig
- Multithread-fähig
- dynamisch

1.3 Objektorientierte Programmierung

Das Schlagwort "objektorientiert..." beinhaltet üblicherweise vier Aspekte:

- Identität von Objekten
- Klassifizierung
- Polymorphismus
- Vererbung

Unter **Identität** versteht man, dass **Daten** als **Attribute** zu diskreten, unterscheidbaren Einheiten, sogenannten **Objekten** gehören. Ein Objekt ist z.B. ein spezieller Abschnitt in einem Text, eine spezielle Wiedergabe eines Musikstücks, das Sparkonto Nr.2451, ein Kreis in einer Zeichnung, ein Eingabekanal, ein Sensor, ein Aktor, eine Person. Objekte können konkreter oder konzeptioneller Natur sein (z.B. Herstellungsanleitung, Rezeptur, Verfahrensweise). Jedes Objekt hat seine Identität. Selbst wenn die Werte aller Attribute von zwei Objekten gleich sind, können sie doch verschiedene Objekte sein. In der Datenverarbeitung werden solche Objekte häufig dadurch unterschieden, dass sie verschiedene Speicherplätze belegen.

Ein besonderes Problem für die Datenverarbeitung ist die Tatsache, dass reale Objekte i.a. eine Lebensdauer haben, die über die Programmlaufzeit hinausgeht (**Persistenz** von Objekten).

Der Zugang zur realen Welt geschieht beim objektorientierten Ansatz über Objekte, nicht über Funktionen. Für Objekte wird ein **Verhalten nach außen (Interface, Methoden)** und eine **interne Repräsentation (Datenstruktur)** durch Datenelemente definiert. Das Verhalten von Objekten ist im Laufe der Zeit sehr konstant oder kann sehr konstant gehalten werden. Wenn sich das Verhalten ändert, dann äußert sich das meistens durch eine schlichte Erweiterung des Interfaces mit weiteren Methoden. Alte Methoden können oft erhalten bleiben.

Eine Verhaltensänderung hat manchmal zur Folge, dass die interne Repräsentation (Datenstruktur) von Objekten geändert werden muss. Da aber das bisherige Verhalten fast immer auch mit der neuen Datenstruktur emuliert werden kann, ist die Datenstruktur von Objekten weniger konstant als ihr Verhalten. Das Verhalten von Objekten hat einen allgemeinen Charakter unabhängig von speziellen Datenverarbeitungsaufgaben.

Funktionen sind Lösungen konkreter Automatisierungsaufgaben. Diese verändern sich laufend, oder es kommen neue Aufgaben dazu. Beruhen Funktionen direkt auf der Datenstruktur, dann müssen alle Funktionen geändert werden, wenn es notwendig wird, die Datenstruktur zu ändern. Beruhen Funktionen aber auf dem Verhalten, muss an den Funktionen nichts geändert werden.

Klassifizierung bedeutet, dass Objekte mit denselben Attributen (Datenstruktur) und demselben Verhalten (Operationen, Methoden) als zu einer Klasse gehörig betrachtet werden. Die

Klasse ist eine Abstraktion des Objekts, die die für die gerade vorliegende Anwendung wichtigen Eigenschaften des Objekts beschreibt und den Rest ignoriert. Die Wahl von Klassen ist letztlich willkürlich und hängt von der Anwendung ab. Jede Klasse beschreibt eine möglicherweise unendliche Menge von Objekten, wobei jedes Objekt eine **Instanz** seiner Klasse ist. Attribute und Verhalten werden in *einer* Klasse zusammen verwaltet, was die Wartung von Software wesentlich erleichtert.

Jede Methode, die für eine Klasse geschrieben wird, steht überall dort zur Verfügung, wo ein Objekt der Klasse verwendet wird. Hierin liegt die Ursache für den Gewinn bei der Software-Entwicklung. Denn der Entwickler wird dadurch gezwungen, die Methoden allgemein anwendbar und "wasserdicht" zu schreiben, da er damit rechnen muss, dass die Methode auch an ganz anderen Stellen angewendet wird, als wofür sie zunächst entworfen wurde. Der Schwerpunkt liegt auf dem Problem**bereich** (***problem domain***) und nicht auf dem gerade zu lösenden (Einzel-)Problem.

Abstrakte Datentypen (data abstraction), Datenkapselung (information hiding) bedeutet, dass kein direkter Zugriff auf die Daten möglich ist. Die Daten sind nur über Zugriffsfunktionen zugänglich. Damit ist eine nachträgliche Änderung der Datenstruktur relativ leicht möglich durch Änderung der Software nur in einer lokalen Umgebung, nämlich der Klasse. Die Zugriffsfunktionen sollten so sorgfältig definiert werden, dass sie ein wohldefiniertes und beständiges Interface für den Anwender einer Klasse bilden.

Polymorphismus bedeutet, dass dieselbe Operation unterschiedliche Auswirkung bei verschiedenen Klassen hat. Die Operation "lese" könnte bei einer Klasse *Textabschnitt* bedeuten, dass ein Textabschnitt aus einer Datenbank geholt wird. Bei einer Klasse *Sparkonto* wird durch "lese" der aktuelle Kontostand ausgegeben. Bei einer Klasse *Sensor* bedeutet "lese", dass der Wert des Sensors angezeigt wird, bei einer Klasse *Aktor*, dass ein Stellwert eingegeben werden soll.

Eine spezifische Implementation einer Operation heißt **Methode**. Eine Operation ist eine Abstraktion von analogem Verhalten verschiedener Arten von Objekten. Jedes Objekt "weiß", wie es seine Operation auszuführen hat. Der Anwender von solchen Objekten muss sich nicht darum kümmern.

Vererbung ist die gemeinsame Nutzung von Attributen und Operationen innerhalb einer Klassenhierarchie. Girokonto und Sparkonto haben die Verwaltung eines Kontostands gemeinsam. Um die Wiederholung von Code zu vermeiden, können beide Arten von Konten von einer Klasse `Konto` *erben*. Oder: Alle Sensoren haben gewisse Gemeinsamkeiten, die an spezielle Sensoren wie Temperatursensoren, Drucksensoren usw weitervererbt werden können. Durch solch ein Design werden Code-Wiederholungen vermieden. Wartung und Veränderung von Software wird erheblich sicherer, da im Idealfall nur an *einer* Stelle verändert oder hinzugefügt werden muss. Ohne Vererbung ist man gezwungen, Code zu kopieren. Nachträgliche Änderungen sind dann an vielen Stellen durchzuführen. Mit Vererbung wird von einer oder mehreren Basisklassen geerbt. Das andere Verhalten der erbenden Klasse wird implementiert, indem Daten ergänzt werden, zusätzliche Methoden geschrieben werden oder geerbte Methoden überschrieben ("überladen" ist hier falsch) werden. **Polymorphismus von Objekten** bedeutet, dass ein Objekt sowohl als Instanz seiner Klasse alsauch als Instanz von Basisklassen seiner Klasse betrachtet werden kann. Es muss möglich sein, Methoden dynamisch zu binden. Hierin liegt die eigentliche Mächtigkeit objekt-orientierter Programmierung.

Beispiel Messtechnik:

Bei der Programmierung einer Prozessdatenverarbeitung müssen Messdaten, z.B. Temperaturen

erfasst werden.

Bei einem **funktionalen** Angang des Problems wird man versuchen, eine Funktion zur Erfassung einer Temperatur zu entwickeln, die eine Analog/Digital-Wandler-Karte anzusprechen hat, die Rohdaten eventuell linearisiert und skaliert.

Beim **objekt-basierten** Ansatz muss man zunächst geeignete **Objekte** bezüglich der Temperaturerfassung suchen. Hier bietet sich der **Temperatursensor** als Objekt an, für den die Klasse Temperatursensor definiert werden kann. Der nächste Schritt ist, die **Repräsentation** eines Temperatursensors zu definieren, d.h. festzulegen, durch welche Daten oder Attribute ein Temperatursensor repräsentiert wird. Als Datenelemente wird man wohl den aktuellen Temperaturwert, verschiedene Skalierungsfaktoren, Werte für den erlaubten Messbereich, Eichkonstanten, Kenngrößen für die Linearisierung, Sensortyp, I/O-Kanal-Adressen oder -Nummern definieren.

Erst im zweiten Schritt wird man die Funktionen (jetzt **Methoden** genannt) zur Skalierung, Linearisierung und Bestimmung des Temperaturwerts und eventuell Initialisierung des Sensors definieren und schließlich programmieren. Ferner wird man dem Konzept der **Datenkapselung** folgend Zugriffsfunktionen schreiben, mit denen die Daten des Sensors eingestellt und abgefragt werden können.

Beim *objekt-orientierten* Denken wird man versuchen, zunächst einen allgemeinen Sensor zu definieren, der Datenelemente hat, die für jeden Sensor (Temperatur, Druck usw) relevant sind. Ein Temperatursensor wird dann alles *erben*, was ein allgemeiner Sensor hat und nur die Temperaturspezifika ergänzen. Weiter könnte der Tempertursensor relativ allgemein gestaltet werden, so dass bei Anwendung eines NiCrNi-Thermoelements das Thermoelement vom allgemeinen Temperatursensor erbt.

Dieses Beispiel soll zeigen, dass der objekt-basierte oder -orientierte Ansatz den Systementwickler veranlasst (möglicherweise gar zwingt), eine Aufgabe allgemeiner, tiefer, umfassender zu durchdringen. Ferner ist beim funktionalen Ansatz nicht unmittelbar klar, was z.B. beim Übergang von einem Sensor zum anderen zu ändern ist. Im allgemeinen muss man die Messfunktion neu schreiben. Bei einer gut entworfenen Objektklasse sind nur Werte von Datenelementen zu ändern. Ferner wird es bei einer gut angelegten Klasse offensichtlich sein, welche Daten zu ändern sind.

1.4 Übungen

Übung Guten Tag: Schreiben Sie die Klasse `GutenTag` ab. Übersetzen Sie das Programm und lassen Sie es laufen.

Übung Produkt: Schreiben Sie die Klasse `Produkt` ab. Übersetzen Sie das Programm und lassen Sie es laufen. Geben Sie auch mal nicht-numerische Werte ein. Ändern Sie das Programm so ab, daß dividiert wird. Teilen Sie eine positive und eine negative Zahl durch Null.

Übung Schmier: Schreiben Sie die Klasse `Schmier` ab. Übersetzen Sie das Programm und lassen Sie es als Anwendung und als Applet laufen.

1. Erweitern Sie das Programm um die Farben Zyan (`Color.cyan`) und Magenta (`Color.magenta`).
2. Beim Farbwahlmenü soll die Hintergrund-Farbe die aktuelle Schmier-Farbe sein. Die Vordergrund-Farbe soll so sein, dass man sie sehen kann, d.h. Weiß oder Schwarz oder vielleicht etwas noch Intelligenteres.

3. Wenn das Fenster überdeckt, vergrößert oder verkleinert wird, dann bleibt die Zeichnung i.A. nicht erhalten, weil wir nicht dafür gesorgt haben, dass in diesen Fällen die Zeichnung wiederhergestellt wird. In dieser Übung wollen wir diesen Mangel beheben.

   Unsere Zeichnung besteht aus einer großen Anzahl kurzer Striche, die wir uns in einem Objekt der Klasse `java.util.ArrayList` merken. Die Methoden dieser Klasse finden sich am Ende von Kapitel "Datentypen" und im Anhang. Dazu ist es aber notwendig für die Striche eine eigene Klasse zu definieren, z.B. `Strich`.

   Ferner müssen wir unsere graphische Benutzeroberfläche etwas erweitern. Insbesondere benötigen wir ein `JPanel`, auf das wir unsere Zeichnung malen.

   Folgende Schritte sind nacheinander durchzuführen:

   (a) In der Klasse `Schmier` bekommt `behälter` ein `BorderLayout`.

   (b) Definiere ein Datenelement vom Typ `JPanel` z.B mit Namen `menue`. Dieses wird instanziiert und mit

   ```
   behälter.add (menue, BorderLayout.NORTH)
   ```

   oben in `behälter` eingefügt.

   (c) Nun werden alle GUI-Komponenten anstatt in `behälter` in `menue` eingefügt.

   (d) Schreiben Sie eine Klasse `Zeichnung`, die von `JPanel` erbt. Sie hat ein Datenelement vom Typ `ArrayList`, das dann alle Strichobjekte aufnehmen soll. Dies muss instanziiert werden.

   (e) Schreiben Sie in der Klasse `Zeichnung` eine Methode `loesche`, die die `ArrayList` neu instanziiert. Und dann die Methode `repaint ()` aufruft.

   (f) Definieren Sie in der Klasse `Schmier` ein Datenelement vom Typ `Zeichnung`, instanziieren Sie es und fügen Sie es in `behälter` an der Stelle `BorderLayout.CENTER` ein.

   (g) Rufen Sie in der Methode `init` (letzte Zeile) die Methode `addMouseMotionListener` anstatt für `behälter` für das `Zeichnung`-Objekt auf.

   (h) In der Methode `actionPerformed` wird, wenn der Knopf "Löschen" gedrückt wurde, für das `Zeichnung`-Objekt nur die Methode `loesche ()` aufgerufen.

   (i) In der Methode `mouseDragged` wird das `Graphics`-Objekt anstatt vom `behälter` von dem `Zeichnung`-Objekt geholt.

   (j) Das Programm sollte jetzt wieder getestet werden.

   (k) Schreiben Sie die Klasse `Strich`.

   (l) Schreiben Sie in der Klasse `Zeichnung` eine Methode `addiere`, um Objekte vom Typ `Strich` an die `ArrayList` zu addieren.

   (m) In der Methode `mouseDragged` wird ein `Strich`-Objekt angelegt und mit `addiere` der Zeichnung hinzugefügt.

   (n) Für eine `JComponente` – und das ist unsere Klasse `Zeichnung` – wird die Methode `paintComponent (Graphics g)` aufgerufen, wenn die Komponente neu dargestellt werden soll. Also müssen wir in `Zeichnung` die Methode

   ```
   public void paintComponent (Graphics g)
      {
      super.paintComponent (g);
      ...
      ...
      }
   ```

definieren und so überschreiben, dass der Inhalt des `ArrayList`-Objekts neu gezeichnet wird.

4. Wahrscheinlich kommt bei Ihnen die Codesequenz

```
g.setColor (...);
g.drawLine (...);
```

mindestens zweimal vor. Solche Codewiederholungen sind schwer wartbar und fehleranfällig. Der Code gehört eigentlich in die Klasse `Strich`. Definieren Sie also in dieser Klasse eine Methode

`public void zeichne (Graphics g) ...`,

die einen Strich zeichnet. Verwenden Sie diese Methode anstatt der oben erwähnten Codesequenz.

5. Schreiben Sie eine Klasse `DoppelStrich`, die von der Klasse `Strich` erbt und die jeden Strich doppelt zeichnet (z.B. um drei Pixel versetzt). Wenden Sie diese Klasse an, wobei nur in der Methode `mouseDragged` an einer einzigen Stelle der Klassenname `Strich` durch `DoppelStrich` ersetzt werden muss.

6. Bieten Sie für die beiden Stricharten ein Auswahlmenü an.

7. Bieten Sie weitere "Strich"-arten an, d.h. irgendwelche interessante Verbindungen zwischen zwei Punkten (Rechtecke, Kreise usw.). Die Methoden der Klasse `Graphics` können Sie da auf Ideen bringen.

8. Und nun wird es richtig schön objektorientiert gemacht: Wahrscheinlich haben Sie zur Unterscheidung der Stricharten in der Methode `mouseDragged` einen `if-else`-Konstrukt oder ein `switch` verwendet. Solche Konstrukte sind schwer zu warten. Ferner ist Ihre interessante Verbindung vielleicht so, dass es nicht vernünftig ist, sie von `Strich` erben zu lassen. Wir wollen diese Probleme unter Ausnutzung des Polymorphismus folgendermaßen lösen:

   (a) Definieren Sie in einer eigenen Datei eine Schnittstelle, die alle möglichen Verbindungen repräsentiert und folgende Methoden deklariert:

```
public interface  Verbindung
   void  zeichne (Graphics g)
   Verbindung  newVerbindung (...)
   String   toString ()
```

   (b) Lassen Sie `Strich` und alle anderen Verbindungstypen diese Schnittstelle implementieren.

   Dabei gibt `newVerbindung` ein neues `Strich`-Objekt bzw. entsprechend andere Objekte zurück.

   `toString` gibt einfach den Klassennamen zurück, also etwa `Strich`.

   (c) Ersetzen Sie in `paintComponent ()` die Klasse `Strich` durch die Schnittstelle `Verbindung`.

(d) Definieren Sie für jeden Verbindungstyp ein konstantes Objekt. Diese Objekte addieren Sie anstatt der Strings an die Auswahl-Combobox für die Verbindungstypen. (`JComboBox` ruft für jedes Item `toString` auf, um es darzustellen.)

(e) Falls nicht schon geschehen, definieren Sie ein Klassenelement vom Typ `Verbindung` etwa mit Namen `aktuelleVerbindung`. In `itemStateChanged` belegen Sie dieses Klassenelement einfach mit `e.getItem ()`, wenn `itemStateChanged` von der entsprechenden Combobox aufgerufen wird.

(f) Ersetzen Sie in `mouseDragged` das `if-else`- oder `switch`-Konstrukt durch einen `newVerbindung`-Aufruf für die aktuelle Verbindung.

9. Verwenden Sie für die Ereignisbehandlung dedizierte innere Klassen.

10. Erstellen Sie eine Verbindung, die blinkt (etwa `BlinkStrich`).

11. Erstellen Sie einen Thread, der jede Sekunde die Anzahl der Verbindungen ausgibt.

12. Erstellen Sie weitere Verbindungen, die aktiv sind. Versuchen Sie dabei das Threading durch Komposition zu realisieren, um Code-Wiederholungen zu vermeiden.

Kapitel 2

Datentypen

In diesem Kapitel behandeln wir zunächst die Programmstruktur, den Namensraum und dann die **primitven Datentypen** oder **Standardtypen** ***(primitive type)***, **Felder** ***(array)*** und den Unterschied zu **Klassen** ***(class)*** und **Schnittstellen** ***(interface)***.

2.1 Programmstruktur

2.1.1 `main`-Methode

Ein Java-Programm besteht aus einer oder mehreren **Klassen**- und **Schnittstellen**-Definitionen. Damit das Programm als eigenständige Anwendung laufen kann muss mindestens eine Klasse die Methode `main` definieren. Sie muss folgenden Prototyp haben:

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

Der Java-Interpreter startet das `main` der Klasse, für die er aufgerufen wird. `main` läuft bis zu seinem natürlichen Ende oder, bis ein `return` erreicht wird. Wenn Threads gestartet wurden, dann läuft der Interpreter, bis alle Threads zum Ende gekommen sind.

Das einzige Argument von `main` ist ein Feld von Zeichenketten. Die Elemente des Feldes sind die Argumente, die beim Aufruf des Java-Interpreters nach dem Klassennamen eventuell mitgegeben werden. Das erste Argument ist *nicht* der Klassenname, sondern wirklich das erste Argument. Die Anzahl der übergebenen Argumente ist über `argument.length` zugänglich. Das folgende erweiterte Hello-World-Programm demonstriert das:

```
public class   GutenTagArg
   {
   public static void   main (String[] argument)
      {
      System.out.println ("Guten Tag!");
      for (int i = 0; i < argument.length; i++)
         {
```

```
        System.out.print ("Argument ");
        System.out.print (i);
        System.out.print (" war: ");
        System.out.println (argument[i]);
        }
     System.out.println ();
     }
  }
```

Der Rückgabewert von `main` ist `void`, d.h. `main` kann nichts zurückgeben. Wenn trotzdem ein Wert zurückgegeben werden soll, ist das über den Systemaufruf

```
System.exit (intWert)
```

möglich, der allerdings das Verlassen des Interpreters auf jeden Fall zur Folge hat, gleichgültig, ob noch irgendwelche Threads laufen. (`exit` ist in Applets nicht erlaubt.)

2.1.2 Umgebungsvariable

Umgebungsvariable sind nur über die Methode `getProperty` der Klasse `System` zugänglich, z.B.

```
String heimVerzeichnis = System.getProperty ("user.home");
```

Weitere Umgebungsvariable sind:

```
user.name
user.dir
java.version
java.home
java.class.version
java.class.path
os.name
```

Nicht alle Umgebungsvariable sind von Applets lesbar.

Bemerkung: Weitere, allerdings stark systemabhängige Programmierung ist möglich unter Verwendung der Klasse `Runtime`. Mit der statischen Methode `getRuntime ()` bekommt man ein Objekt für die aktuelle Plattform:

```
Runtime  r = Runtime.getRuntime ();
```

Dieses Objekt kann verwendet werden, um z.B. Systemaufrufe abzusetzen:

```
r.exec ("Systemkommando");
```

Das funktioniert nur mit Kommandos, die nicht in die jeweilige Shell eingebaut sind. Z.B. muss man unter Windows bei solchen Kommados (wie z.B. `dir`) extra den Kommando-Prozessor aufrufen:

```
r.exec ("cmd /c dir");
```

Bemerkung: Dieser Aufruf gibt ein Objekt vom Typ `Process` zurück, über das man Zugriff auf die Standard-Ein- und Ausgabe-Ströme des Systemkommandos hat:

```
Process  p = r.exec ("Systemkommando");
InputStream stdout = p.getInputStream ();
InputStream stderr = p.getErrorStream ();
OutputStream   stdin = p.getOutputStream ();
```

2.1.3 Namensraum

In Java ist jeder Name weltweit eindeutig. Damit kann es globale Variable und Methoden schlicht nicht geben. Jede Variable und jede Methode kann nur innerhalb einer Klasse definiert werden. Variable und Methoden einer Klasse werden mit dem Sammelbegriff **Klassenelement** ***(class element, field)*** bezeichnet.

Ferner gehört jede Klasse zu einem **Paket** ***(package)***. Jedes Klassenelement muss mit seinem vollen Namen angesprochen werden, der sich aus dem Namen des Pakets, der Klasse und des Klassenelements selbst zusammensetzt:

```
PaketName.KlassenName.variablenName
PaketName.KlassenName.methodenName ()
```

Es ist Konvention, den ersten Buchstaben

von Klassen groß,
von Klassenelementen klein

zu schreiben.

Jede kompilierte Klasse wird in einer eigenen Datei mit der Erweiterung `.class` gespeichert. Der Paketname leitet sich ab aus dem vollen qualifizierten Namen des Verzeichnisses, wo die übersetzte Klasse gespeichert ist.

Also wenn die übersetzte Klasse `GutenTagArg` in der Datei

```
kfg/lang/java/Einleitung/Beispiele/GutenTagArg.class
```

auf dem Rechner `amadeus` mit dem Internet-Domain-Namen

```
ba-stuttgart.de
```

gespeichert ist, dann lautet der Paketname:

```
de.ba-stuttgart.kfg.lang.java.Einleitung.Beispiele
```

Da der **Name** mit dem **Ort** einer Übersetztungseinheit gekoppelt ist, wird automatisch die Eindeutigkeit des Namens garantiert, da zwei verschiedene Dinge schlicht nicht am selben Ort sein können.

Eine Quellcode-Datei mit der Erweiterung `.java` kann mehrere Klassen enthalten, wovon nur eine `public` deklariert sein darf. Der Name dieser Klasse muss der Name der Datei sein (um `.java` erweitert).

Klassenpfad `CLASSPATH`

In der Betriebssystemvariablen `CLASSPATH` werden alle Verzeichnisse aufgeführt, in denen und darunter der Compiler und Interpreter nach Klassendateien suchen soll. Jede Klasse steht unter ihrem relativ zu den in `CLASSPATH` angegebenen Verzeichnissen voll qualifizierten Namen zur Verfügung.

Unix:
```
setenv CLASSPATH .:/home/kfg/jvKlassen:/usr/local/jvKlassen
```

Windows:
```
SET CLASSPATH=.;a:\kfg\jvKlassen;c:\lokal\jvKlassen
```

Die Pfade der System-Klassen werden automatisch an den `CLASSPATH` angehängt. Mit der Systemeigenschaft `java.class.path` erhält man die geordnete Liste der schließlich verwendeten Pfade.

Der `CLASSPATH` wird vom Interpreter `java` und vom Compiler `javac` verwendet. Dort gibt es allerdings auch die Option `-classpath` oder `-cp`, mit der Suchverzeichnisse angegeben werden können. Diese Angabe hat zur Folge, dass der `CLASSPATH` keine Rolle spielt. Daher emphiehlt es sich oft, den `CLASSPATH` mit anzugeben:

Unix:
```
java -classpath bahn/apl:kj.jar:$CLASSPATH Anwendung
```

Windows:
```
java -classpath bahn\apl;kj.jar;"%CLASSPATH%" Anwendung
```

Statement `package`

Mit dem `package`-Statement kann man angeben, zu welchem Paket eine Übersetzungseinheit gehört. Das Statement muss nach Kommentaren als erstes Statement in einer Datei erscheinen:

```
package  lang.java.konten;
```

Der Name kann relativ zu einem `CLASSPATH`-Verzeichnis sein. Die übersetzten Dateien müssen dann in dem Verzeichnis liegen, das dem Paket entspricht. Mit dem `package`-Statement legt man die Lage der Bytecode Datei (`.class`) im Verzeichnisbaum fest.

Wenn das Statement fehlt, wird ein Default-Paket angenommen.

Ein Paket ist eine Entwurfs-Einheit, die mehrere Klassen als zusammengehörig betrachtet. Insbesondere können Namenskonflikte vermieden werden.

Statement `import`

Mit dem Import-Statement können Klassen so zur Verfügung gestellt werden, dass sie unter einem abgekürzten Namen verwendbar sind. Das Statement spart Schreibarbeit. Auf keinen Fall bedeutet es, dass irgendwelche Klassen "eingelesen" werden.

Beliebig viele Import-Statements sind erlaubt. Sie müssen nach dem optionalen `package`-Statement, aber vor allen Klassen- oder Schnittstellen-Definitionen erscheinen.

Es gibt drei Formen:

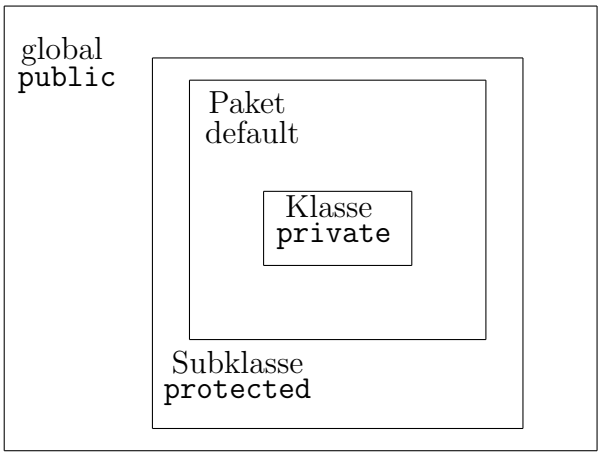
1. `import` *Pa.cka.ge.name*`;`
   z.B.:
   `import kfg.lang.java.Einleitung.Beispiele;`
   Damit kann man das Paket unter dem Namen seiner letzten Komponente verwenden, hier z.B. `Beispiele`, also etwa `Beispiele.GutenTagArg` .
2. `import` *Pa.cka.ge.name.Klasse*`;`
   z.B.:
   `import kfg.lang.java.Einleitung.Beispiele.GutenTagArg;`
   Damit kann man eine Klasse direkt mit ihrem Namen ansprechen und verwenden, hier z.B. die Klasse `GutenTagArg` .
3. `import` *Pa.cka.ge.name*`.*;`
   z.B.:
   `import kfg.lang.java.Einleitung.Beispiele.*;`
   Diese Form wird üblicherweise verwendet. Damit kann man *alle* Klassen des Pakets direkt unter ihrem Namen ansprechen. Wenn es zu Namenskonflikten kommt, müssen diese durch qualifizierte Namen aufgelöst werden.

Zugriffsregeln und Sichtbarkeit

- Ein Paket und seine Klassen können verwendet werden, wenn die entsprechenden Verzeichnisse und Dateien die richtigen Leseberechtigungen des Betriebssystems haben, oder wenn sie über das Netz geladen werden können.
- Alle Klassen und Schnittstellen stehen allen anderen Klassen und Schnittstellen des gleichen Pakets zur Verfügung.
- Klassen und Schnittstellen können `public` oder ***default*** (d.h. ohne **Modifikator *(modifier)***) deklariert werden. In einer Datei kann höchstens *eine* Klasse `public` deklariert werden. Eine `public` Klasse ist von anderen Paketen aus verwendbar, sofern ihr Paket zugänglich ist. Eine ***default***-Klasse kann nur im gleichen Quellfile verwendet werden.
- `private` deklarierte Klassenelemente sind nur in der eigenen Klasse bekannt.
- ***default*** (d.h. ohne Modifikator) deklarierte Klassenelemente sind in anderen Klassen desselben Pakets bekannt. Dieses Sichtbarkeitsniveau heißt auch emdeu Paket-Zugriff ***(package access)***.
- Ein Klassenelement ist in anderen Paketen bekannt, wenn seine Klasse dort bekannt ist und wenn es `public` deklariert ist.

- Ein Klassenelement ist innerhalb einer abgeleiteten Klasse, die zu einem *anderen* Paket gehört, nur bekannt, wenn es mindestens `protected` deklariert ist.
- Innerhalb einer Klasse kann auf alle Klassenelemente der Klasse zugegriffen werden.

Die **Sichtbarkeit** ***(scope)*** von Klassenelementen kann in folgender Graphik zusammengefasst werden.



Lokale Variable

Innerhalb von Blöcken (geschweiften Klammern {}) können auch Variable definiert werden. Diese sind nur in dem betreffenden Block bekannt. Sie können in keiner Weise veröffentlicht werden. Argumente von Methoden sind auch lokale Variable.

2.2 Konstanten

Jede Variable, die als `final` deklariert wird, ist eine Konstante und muss bei der Definition initialisiert werden. Die C-Konvention der Großbuchstaben bei Konstanten wird beibehalten.

```
public class   Mathematik
   {
   public static final double EULER = 2.7182;
   public final double FAKTOR;
   }
```

`static final` Variable sind Klassenvariable und können direkt über die Klasse angesprochen werden. `final` Variable (nicht `static`) können nur über ein Objekt der Klasse angesprochen werden; sie können nur in einem Konstruktor oder Instanz-Initialisierer einmal mit einem Wert belegt werden und können damit von Objekt zu Objekt unterschiedlich sein.

Auf `static` Variable und Methoden wird im Kapitel "Die Java-Klasse" näher eingegangen.

Bemerkung: Konstanten sollten wirklich Konstanten sein, d.h. niemals von der ProgrammierIn geändert werden. Eine nachträgliche Änderung kann Probleme machen, da manche Compiler die Konstanten hart in den Bytecode schreiben, so dass die Übersetzung der Klasse, die die Konstante definiert, nicht genügt.

2.3 Unicode

Zeichen, Zeichenketten und Identifikatoren (z.B. Variable, Methoden, Klassennamen) werden mit 16-Bit Unicode-Zeichen dargestellt.

Die ersten 256 Zeichen (0x0000 bis 0x00FF) sind identisch mit den Zeichen von ISO8859-1 (Latin-1).

Da die meisten Systeme nicht alle 34000 definierten Unicode-Zeichen darstellen können, können Java-Programme mit speziellen Unicode-Escape-Sequenzen geschrieben werden. Jedes Unicode-Zeichen kann dargestellt werden als `\uxxxx`, wobei `xxxx` eine Folge von ein bis vier hexadezimalen Zeichen ist.

Ferner werden die C-Escape-Sequenzen unterstützt wie `\n`, `\t` und `\xxx`, wobei `xxx` drei Octalzeichen sind.

Unicode-Escape-Sequenzen können überall im Programm erscheienen. Andere Escape-Sequenzen können nur in Zeichen und Zeichenketten erscheinen. Ferner werden Unicode-Escape-Sequenzen *vor* anderen Escape-Sequenzen bearbeitet. D.h. die Unicode-Sequenz `\u0022` würde erst durch `"` ersetzt. Wenn `"` innerhalb eines Strings auftaucht (`"\""`), muss daher auch die Unicode-Sequenz mit einem Backslash eingeleitet werden (`"\\u0022"`).

2.4 Standarddatentypen

Folgende **Standardtypen** ***(primitive data type)*** sind in Java definiert:

Typ	enthält	Default	Bit	Bereich
`boolean`	Bool'scher Wert	`false`	1	`true` oder `false`
`char`	Unicode-Zeichen	`\u0`	16	`\u0000` bis `\uFFFF`
`byte`	ganze Zahl	0	8	`-128` bis `127`
`short`	ganze Zahl	0	16	`-32768` bis `32767`
`int`	ganze Zahl	0	32	`-2147483648` bis `2147483647`
`long`	ganze Zahl	0	64	`-9223372036854775808` bis `9223372036854775807`
`float`	IEEE-754 Zahl	0.0	32	±`3.40282347E+38` bis ±`1.40239846E-45`
`double`	IEEE-754 Zahl	0.0	64	±`1.79769313486223157OE+308` ±`4.94065645841246544E-324`

Bemerkungen:

1. `boolean`-Werte können nicht von einem oder in einen anderen Typ gewandelt werden.

2. Werte vom Typ `char` haben kein Vorzeichen. Alle sonstigen ganzzahligen Typen haben ein Vorzeichen.

3. Eine Konstante vom Typ `long` kann mit einem `L` oder `l` unterschieden werden (`387L`).

4. `float` und `double` Gleitkommazahlen können unterschieden werden durch anhängen von `F`, `f` bzw `D`, `d` (`1.435F` oder `3.957D`).

5. In den Klassen `java.lang.Float` und `java.lang.Double` sind die Konstanten `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, `NaN` ***(not a number)***, `MAX_VALUE` (größter Absolutwert), `MIN_VALUE` (kleinster Absolutwert) definiert.

6. Ganzzahlige Division durch Null wirft eine `ArithmeticException`. Gleitkommadivision wirft niemals eine Exception, sondern die Resultate sind eventuell `POSITIVE_INFINITY` oder `NEGATIV_INFINITY`.

7. Einerseits gibt es einen besonderen Aufzählungstyp, den wir am Ende des Kapitels behandeln. Andererseits gibt es im API eine Schnittstelle `java.util.Enumeration`, die es erlaubt, Collections zu druchlaufen.

2.5 Referenzdatentypen

Referenztypen ***(reference data type)*** sind **Objekte** ***(object)*** (d.h. vom Typ einer Klasse) und **Felder** ***(array)***. Diese Typen werden nur als Referenz ***(by reference)*** behandelt. D.h. die Adresse eines Objekts oder Felds wird in einer Variablen gespeichert, Methoden übergeben, zugewiesen usw. Die Standardtypen werden immer als Wert ***(by value)*** behandelt.

In Java gibt es daher keine "Adresse-von"- oder Dereferenzierungs-Operatoren.

Zwei Referenztypen können *dasselbe* Objekt meinen. Ein Objekt wird mit `new` erzeugt.

Es gibt in Java keine **Zeiger** ***(Pointer)***, keine Zeigerarithmetik, keine Berechnung von Objektgrößen und Manipulation von Speicheradressen.

Der Defaultwert für alle Referenztypen ist `null` und bedeutet, dass kein Objekt referenziert wird. `null` ist ein Schlüsselwort von Java.

2.5.1 Zuweisung von Referenztypen

Die Zuweisung von Referenztypen

```
a = b;
```

bedeutet nicht eine Kopie, sondern bedeutet, dass `a` jetzt auf dasselbe Objekt wie `b` zeigt (oder dasselbe Objekt wie `b` ist). Das Objekt, auf das `a` vorher gezeigt hat, wird eventuell von einer anderen Referenzvariablen referenziert oder fällt früher oder später in die Hände des Garbage-Collectors, der den Speicher des Objekts deallokiert, wenn das Objekt nicht mehr referenziert wird.

Ein Kopie wird gemacht (Klasse sollte die Schnittstelle `Cloneable` implementieren, d.h. die Methode `clone ()` überschreiben.) mit:

```
a = b.clone ();
```

Folgender Code demonstriert das angegebene Verhalten:

```
public class   Zei implements Cloneable
   {
   public char c;

   public   Zei (char c) { this.c = c; }

   public boolean equals (Object z)
      {
      return  c == ( (Zei)z).c;
      }

   public Object  clone ()
      {
      Object   kopie = null;
      try
         {
         kopie = super.clone ();
         }
         catch (CloneNotSupportedException e)
            {
            e.printStackTrace ();
            }
      return kopie;
      }
   }



public class   Zuweisung
   {
   public static void   main (String[] argument)
      {
      char  a = 'X';
      char  b = a;
      System.out.print ("b = a         : a ist ");
      System.out.println (a + "    b ist " + b);
      a = 'Y';       // b bleibt 'X'
      System.out.print ("a = 'Y'       : a ist ");
      System.out.println (a + "    b ist " + b);
      System.out.println ();

      Zei   d = new Zei ('X');
      Zei   e = new Zei ('Y');
      System.out.print ("nach init     : d.c ist ");
```

```
      System.out.println (d.c + "  e.c ist " + e.c);
      e = d;
      System.out.print ("e = d         : d.c ist ");
      System.out.println (d.c + "  e.c ist " + e.c);
      d.c = 'Y';        // e.c wird auch 'Y'
      System.out.print ("d.c = 'Y'     : d.c ist ");
      System.out.println (d.c + "  e.c ist " + e.c);
      e = (Zei)(d.clone ());
      System.out.print ("e = d.clone (): d.c ist ");
      System.out.println (d.c + "  e.c ist " + e.c);
      d.c = 'X';        // e.c wird nicht auch 'Y'
      System.out.print ("d.c = 'X'     : d.c ist ");
      System.out.println (d.c + "  e.c ist " + e.c);
      }
   }
```

Das Resultat von

```
$ java Zuweisung
```

ist:

```
b = a         : a ist X    b ist X
a = 'Y'       : a ist Y    b ist X

nach init     : d.c ist X  e.c ist Y
e = d         : d.c ist X  e.c ist X
d.c = 'Y'     : d.c ist Y  e.c ist Y
e = d.clone (): d.c ist Y  e.c ist Y
d.c = 'X'     : d.c ist X  e.c ist Y
```

2.5.2 Vergleich von Referenztypen

Der Operator "`==`" prüft, ob zwei Referenztypen auf dasselbe Objekt zeigen. Um zu prüfen, ob zwei verschiedene Objekte inhaltlich gleich sind, muss man eine spezielle Methode `equals` definieren.

Folgender Code zeigt das:

```
public class   Vergleich
   {
   public static void   main (String[] argument)
      {
      Zei   d = new Zei ('X');
      Zei   e = new Zei ('Y');
```

```
      System.out.println ("nach init      d == e : " + (d == e));
      e = d;
      System.out.println ("e = d          d == e : " + (d == e));
      e = (Zei)(d.clone ());
      System.out.println ("e = d.clone () d == e : " + (d == e));
      System.out.println ("e.equals (d)          : " + e.equals (d));
      }
   }
```

Das Resultat von `java Vergleich` ist:

```
nach init      d == e : false
e = d          d == e : true
e = d.clone () d == e : false
e.equals (d)          : true
```

2.5.3 Erzeugung von Objekten

Die Deklaration eines Referenztyps erzeugt noch nicht ein Objekt, sondern nur eine Variable, die eine Referenz auf ein Objekt enthalten kann. Sie wird mit dem Defaultwert `null` belegt.

```
Zei   c;
```

`c` enthält `null`. Ein Objekt wird durch Aufruf von `new` gefolgt von einem Klassennamen und einer Argumentenliste erzeugt. Die Argumente werden einem passenden Konstruktor übergeben:

```
c = new Zei ('X');
String   s;
s = new String ("Guten Tag!");
```

`"Guten Tag!"` selbst ist eine Referenz auf ein String-Objekt mit dem Inhalt "Guten Tag!". Mit `new String ("Guten Tag!")` wurde ein neues String-Objekt gemacht mit dem Inhalt "Guten Tag!" gemacht. Das ist *nicht* äquivalent zur Zuweisung

```
s = "Guten Tag!";
```

Denn hier referiert `s` dasselbe String-Objekt, das `"Guten Tag!"` referiert.

Java benutzt ***garbage collection***, um den Speicher von nicht mehr benötigten Objekten automatisch freizugeben.

2.6 Felder

Felder sind auch Referenztypen und werden dynamisch mit `new` erzeugt. Mit

```
byte[]   f1 = new byte[20];
```

wird ein Feld `f1` von 20 `bytes` `f1[0]` bis `f1[19]` angelegt. Für jedes Feldelement wird Speicher für ein `byte` angelegt. Sie werden mit dem Default-Wert `0` belegt.

(Die C-Syntax

```
byte  f1[] = new byte[20];
```

ist auch möglich, wird aber von uns nicht verwendet.)

Bei Referenztypen wie z.B. der Klasse `Zei` ist die Semantik anders als etwa in C++, aber konsequenter. Mit

```
Zei[] f2 = new Zei[20];
```

wird ein Feld `f2` von 20 Variablen des Typs `Zei` deklariert. Für die einzelnen Variablen wird *kein* Speicher angelegt. Die Elemente des Feldes `f2` werden mit `null` initialisiert.

Es können auch statische Initialisatoren verwendet werden:

```
int[] f3 = {27, 47, 67, 114};
String[] s = {"Gu", "ten", " ", "Tag"};
algemein:
Klasse[] f = {new Klasse (...), new Klasse (...),
   new SubKlasse (...), ... };
```

Die Elemente in geschweiften Klammern können beliebige (auch nicht konstante) Ausdrücke sein, die in dem betreffenden Typ resultieren. Dieser statische Initialisator kann auch mit einem `new` verwendet werden:

```
int[] f4 = new int[] {27, 47, 56, 67}
```

Die rechte Seite ist ein Beispiel für ein sogenanntes **anonymes Feld** ***(anonymous array)***.

Ein mehrdimensionales Feld wird mit

```
short[][]   m1 = new short[47][114];
```

angelegt. Mehrdimensionale Felder sind als Felder von Feldern implementiert, so dass `m1` ein Feld von `47` Feldern des Typs `short[]` der jeweiligen Länge `114` `short` ist.

Bei einem mehrdimensionalen Feld müssen nur die ersten `n` (`n >=1`) spezifiziert werden.

```
long[][][][]   m2 = new long[36][][][];
```

```
long[][][][]   m3 = new long[36][12][][];
long[][][][]   m4 = new long[36][][12][];    // Fehler
long[][][][]   m5 = new long[][17][12][];    // Fehler
```

Damit können Matrizen mit verschiedenen Zeilenlängen, z.B. dreieckige Matrizen angelegt werden:

```
int[][]  dreieck = new int[17][];
for (int i = 0; i < dreieck.length; i++)
   {
   dreieck[i] = new int[i + 1];
   }
```

Hier haben wir ein *nur lesbares* Feld `length` verwendet, das alle Feldtypen anbieten und das die Anzahl der Feldelemente enthält.

Der Zugriff auf Feldelemente erfolgt mit der in C üblichen Syntax:

```
m1[27][67]
```

gibt das Element in der 28-ten Zeile und 68-ten Spalte zurück.

Verschachtelte Initialisierung ist auch möglich:

```
int[][]  m6 = { {5, 2}, {7, 9, 1, 6, 7}, {5, 3, 9} };
```

Natürlich bleibt es einem freigestellt, mehrdimensionale Felder als eindimensionale Felder anzulegen und auf die Elemente mittels der Rechteckformel zuzugreifen.

Bei einem Index außerhalb der Grenzen wird die Exception

```
ArrayIndexOutOfBoundsException
```

geworfen.

Ein Feld kann einer Variablen vom Typ `Object` zugewiesen werden. Die Methoden der Klasse `Object` können auf Felder angewendet werden.

2.7 Modifikatoren

In diesem Abschnitt sollen noch einmal die **Modifikatoren** ***(modifier)*** zusammengefasst werden.

- `final` : Kann auf Klassen, Methoden und Variable angewendet werden. Eine `final` Klasse kann keine Subklassen haben. Eine `final` Methode kann nicht überschrieben werden. Eine `final` Variable ist eine Konstante.

- `native` : Kann auf Methoden angewendet werden und bedeutet, dass die Methode in einer anderen Sprache – meist plattformabhängig – implementiert ist. Anstatt eines Körpers hat sie ein Semikolon.

- `synchronized` : Kann auf Klassen- und Instanzmethoden angewendet werden und bedeutet, dass die Methode nicht threadsicher ist. Der Java-Interpreter setzt ein Lock auf die entsprechende Klasse oder Instanz, bevor die Methode ausgeführt wird.

- `transient` : Kann auf Variable angewendet werden, die nicht zum **dauerhaften *(persistent)*** Zustand eines Objekts gehören. Transiente Felder werden nicht mit dem Objekt serialisiert.

- `volatile` : Kann auf Variable angewendet werden und bedeutet, dass die Variable ihren Wert asynchron ändern kann *(memory mapped hardware register)* und dass der Compiler mit dieser Variablen keine Optimierungen (z.B. Speichern in Registern) versuchen sollte. Bei jeder Verwendung sollte die Variable vom Speicher gelesen werden.

2.8 Zeichenketten

Für Zeichenketten gibt es die Klasse `java.lang.String` von nicht `\0`-terminierten Zeichenketten. Für sie ist der Operator "`+`" als Konkatenierung definiert. Zeichen zwischen doppelten Anführungsstrichen (`"Hallo"`) werden automatisch als ein `String`-Objekt behandelt.

Man kann `String`-Objekte nicht verändern, d.h. es ist kein Schreib-Zugriff auf die Komponenten möglich. Dazu muss man mit dem `String`-Objekt ein `StringBuffer`-Objekt erzeugen, dessen Inhalt manipuliert werden kann.

Einige der wichtigsten Methoden sind:

```
int   length ()
char  charAt (int i)
boolean  equals (Object s2)
int   compareTo (String s2)
            // Gibt negative Zahl zurück, wenn s2 lexikalisch
            // größer ist,
            // positive Zahl, wenn s2 kleiner ist,
            // 0 bei Gleichheit.

String   substring (int von)
String   substring (int von, int bis)
            // Das Zeichen am Index bis wird nicht
            // mit in den Unterstring aufgenommen.

String   replace (char alt, char neu)
            // Alle Zeichen alt werden durch neu ersetzt.
```

Die Klasse `String` hat eine Methode `intern ()`, die einen String in einem Pool von eindeutigen Strings sucht und entweder eine Referenz auf den gefundenen String zurückgibt oder, falls er nicht gefunden wird, diesen String dem Pool eindeutiger Strings hinzufügt.

Diese Operation kann man bei der Erzeugung eines Strings oder auch später machen:

```
String   t = new String ("irgendetwas");
t == "irgendetwas" ergibt false.
String   s = new String ("irgendetwas").intern ();
s == "irgendetwas" ergibt true.
t = t.intern ();
t == "irgendetwas" ergibt nun auch true.
```

Stringliterale und String-wertige konstante Ausdrücke gehören immer zum Pool. Damit ist der Ausdruck

```
"abc" + 47 == "abc" + 47
```

immer `true`.

Weitere Methoden von `String` siehe Anhang A.

2.9 Klasse `ArrayList`

Die Klasse `java.util.ArrayList` implementiert ein Feld von Objekten, dessen Größe bei Bedarf mitwächst. Ein Objekt dieser Klasse kann jede Art von Objekten enthalten, aber keine Standardtypen. Die wichtigsten Methoden sind:

```
public   ArrayList ();
public int  size ();
public void add (Object ob);
public Object  get (int i);
public Object  remove (int i);
public Enumeration   elements ();
public Object[]   toArray (Object[] ob);
```

Die Methode `toArray` ist sehr nützlich zur Verwandlung von `ArrayList`-Objekten in Felder:

```
ArrayList   v;
// ---
Klasse[] k = new Klasse[v.size];
v.toArray (k);
```

Das geht allerdings nur gut, wenn die Elemente von `v` vom Typ `Klasse` sind.

Weitere Methoden von `ArrayList` siehe Anhang A.

2.10 Klasse `Vector`

Die Klasse `java.util.Vector` implementiert ebenfalls ein Feld von Objekten, dessen Größe bei Bedarf mitwächst. Allerdings sind die Methoden hier im Gegensatz zu `ArrayList` `synchronized`, was Performanz kostet. Obwohl man deswegen im allgemeinen eher `ArrayList` verwenden sollte, wird die Klasse `Vector` hier vorgestellt, da sie vor Java 2 die Behälterklasse schlechthin war und daher in zahllosen Programmen verwendet wird.

Ein Objekt der Klasse `Vector` kann jede Art von Objekten enthalten, aber keine Standardtypen. Die wichtigsten Methoden sind:

```
public   Vector ();
public int  size ();
public synchronized void   add (Object ob);
public synchronized Object get (int i);
public synchronized boolean   remove (Object ob);
public synchronized Object remove (int i);
public synchronized Enumeration  elements ();
public synchronized int copyInto (Object[] ob);
```

Oft werden noch folgende ältere Formen verwendet:

```
public synchronized void   addElement (Object ob);
public synchronized Object elementAt (int i);
public synchronized boolean   removeElement (Object ob);
public synchronized void   removeElementAt (int i);
```

Die Methode `copyInto` ist sehr nützlich zur Verwandlung von `Vector`-Objekten in Felder:

```
Vector   v;
// ---
Klasse[] k = new Klasse[v.size];
v.copyInto (k);
```

Das geht allerdings nur gut, wenn die Elemente von `v` vom Typ `Klasse` sind.

Weitere Methoden von `Vector` siehe Anhang A.

2.11 Klasse `Arrays`

Die Klasse `java.util.Arrays` implementiert verschiedene `static` Methoden um Felder wie z.B. sortieren oder suchen. Insbesondere gibt es eine Methode `asList`, die aus einem Feld eine Liste

macht, so dass man das Feld mit bequemen Listen-Operationen manipulieren kann. Es folgt ein Beispiel dazu:

```
import   java.util.*;

public class   FeldAlsListe
   {
   public static void   main (String[] argument)
      {
      // Definition eines Feldes:
      String[] a = new String[] { "A", "B" };

      // Verwandlung des Feldes in eine Liste:
      List<String>   b = new ArrayList<String> (Arrays.asList (a));

      // Manipulation der Liste:
      b.add ("C");

      // Rückverwandlung der Liste in ein Feld:
      a = b.toArray (new String[0]);

      // Ausdruck des Feldes:
      for (String s : a) System.out.print (s + " ");
      System.out.println ();
      }
   }
```

2.12 Aufzählungstyp `enum`

Jeder Aufzählungstyp

```
enum Geschlecht {Masculinum, Femininum, Neutrum};
```

leitet sich konzeptionell von der Klasse

```
java.lang.Enum
```

ab. Die dort definierten Methoden können auf ein Objekt

z.B. `Geschlecht.Neutrum.methode (...)`

oder bei `static` Methoden auf die Klasse

z.B. `Geschlecht.methode (...)`

angewendet werden.

Die Verwendung des Aufzählungstyps `enum` sei an folgendem Beispiel demonstriert:

```
import   java.util.*;

public class   EnumDemo
   {
   enum Wochentag {Montag, Dienstag, Mittwoch, Donnerstag, Freitag,
      Samstag, Sonntag};

   public static void   main (String[] arg)
      {
      ArrayList<Wochentag> wtliste = new ArrayList<Wochentag> ();
      wtliste.add (Wochentag.Montag);
      wtliste.add (Wochentag.Mittwoch);
      wtliste.add (Wochentag.Freitag);
      wtliste.add (Wochentag.Sonntag);
      System.out.println ("Wir arbeiten nur an den Tagen:");
      for (Wochentag wt : wtliste)
         {
         System.out.println ("\t" + wt);
         }
      System.out.println ();
      System.out.println ("Als Argumente wurden gelesen:");
      Wochentag   next;
      for (String s : arg)
         {
         next = null;
         try
            {
            next = Wochentag.valueOf (Wochentag.class, s);
            }
            catch (Exception e) { e.printStackTrace (); }
         System.out.println ("Gelesener String: \"" + s + "\" Wochentag: "
            + next);
         }
      System.out.println ();
      int   i = 0;
      do {
         next = wtliste.get (i);
         switch (next)
            {
            case Montag:
            case Dienstag:
            case Mittwoch:
            case Donnerstag:
            case Freitag: System.out.println (next + " ist ein Arbeitstag.");
               break;
```

```
            case Samstag: System.out.println ("Am " + next + " wird nur manchmal gearbeitet.");
               break;
            case Sonntag: System.out.println ("Am " + next + " wird ganz selten gearbeitet.");
               break;
            }
         i++;
         } while (next != Wochentag.Sonntag);

      }
   }
```

2.13 Übungen

Übung Umgebungsvariable: Schreiben Sie ein Programm, das den Wert von Umgebungsvariablen ausgibt. Die Umgebungsvariable soll als Argument übergeben werden.

Übung Verwendung anderer Programme: Schreiben Sie ein Programm "Fragenbeantworter" in einer anderen Sprache als Java (z.B. C), das jede Frage (von Stdin kommend) "`fragentext?`" mit "`Natürlich fragentext.`" beantwortet und nach Stdout schickt. Fehlerausgaben dieses Programmes sollen auf die Fehlerausgabe weitergeleitet werden. Das Java-Programm soll eine Frage von Stdin einlesen und diese Frage an das Programm "Fragenbeantworter" weiterleiten. Die Antwort soll empfangen werden und nach Stdout geschrieben werden.

Kapitel 3

Operatoren

In diesem Kapitel werden wir die von Java zur Verfügung gestellten Operatoren betrachten, mit denen im wesentlichen nur **Standardtypen** ***(primitive type)*** verändert und verknüpft werden können.

Die Überladung von Operatoren ist in Java *nicht* möglich. Davon gibt es eine Ausnahme: Der binäre Operator "`+`" ist auch für Objekte der Klasse `String` definiert und bedeutet dort Konkatenierung.

3.1 Unäre, binäre und ternäre Operatoren

Operatoren können aus ein bis drei Zeichen bestehen, z.B. `+` oder `++` . *Unäre* Operatoren wirken auf *ein* Datenelement. Sie sind *rechts*-assoziativ, d.h. `-++a` wird als `-(++a)` nicht als `++(-a)` interpretiert.

Die *binären* Operatoren verknüpfen *zwei* Datenelemente und sind alle – mit Ausnahme der Zuweisungsoperatoren – *links*-assoziativ.

`a + b + c + d` bedeutet `((a + b) + c) + d` .

Aber `a = b = c = d = 1` bedeutet `a = (b = (c = (d = 1)))` .

`a *= b += c = d -= 1` bedeutet `a *= (b += (c = (d -= 1)))` .

Es gibt einen einzigen *ternären* Operator (`x ? y : z`), wobei `x` ein `boolean` Ausdruck ist. Das Resultat ist `y`, wenn `x` `true` ist, sonst `z`. `y` und `z` müssen typkompatibel sein.

Welche Komponente bei binären Operatoren zuerst ausgewertet wird, ist unbestimmt:

```
int   a = 0;
(a *= 2) == ++a;
```

Nach diesen Statements kann `a` entweder gleich `1` oder `2` sein. Nur die Operatoren "`&&`" und "`||`" garantieren, dass der linke Ausdruck zuerst ausgewertet wird.

3.2 Operatoren und ihre Hierarchie

In der folgenden Tabelle sind alle Operatoren mit abnehmender Präzedenz aufgeführt. Operatoren in einem Kasten haben gleiche Präzedenz. Die Präzedenzregeln sind so gemacht, dass sie weitgehend "natürlich" funktionieren.

Subskript ("`[]`"), Funktionsaufruf ("`()`"), `new` und Namensauflösung ("`.`") werden in Java nicht als Operatoren betrachtet. Der Kommaoperator ist kein erlaubter Operator mit Ausnahme im `for`-Schleifenkopf.

Einen Operator oder eine Methode `sizeof` gibt es nicht.

`a++` innerhalb eines Ausdrucks bedeutet, dass der Ausdruck erst mit dem ursprünglichen Wert von `a` ausgewertet wird und dann `a` um `1` erhöht wird. Bei `++a` innerhalb eines Ausdrucks wird `a` erst um `1` erhöht, und dann der Ausdruck mit dem erhöhten Wert von `a` ausgewertet. Entsprechendes gilt für den Operator "`--`" . Moderne Compiler sind allerdings so gut, dass man auf diese manchmal schwer verständliche Schreibweise zugunsten einer ausführlicheren Schreibweise verzichten kann. Sie ist eigentlich nur noch praktisch bei `for`-Schleifen (`for (i = 0; i < 5; i++)`). Dort ist es übrigens gleichgültig, ob `i++` oder `++i` verwendet wird.

Die Division "`/`" wird bei ganzzahligen Größen ganzzahlig durchgeführt.
`11 / 3` ergibt `3` .
`11 mod 3` muss als `11 % 3` geschrieben werden und ergibt `2` .

`a << b` bedeutet, dass `a` um `b` Bit nach links (nicht zirkular) verschoben wird. Dabei werden die rechts entstehenden Bit mit Nullen belegt. Entsprechendes gilt für die Rechtsverschiebung. Bei negativem Wert werden Einsen von links nachgeschoben. Wenn Nullen nachgeschoben werden sollen, dann muss der Operator `>>>` verwendet werden. Bei den Schiebeoperatoren werden bei `int` (`long`) die 5 (6) LSB des rechten Ausdrucks verwendet.

Die Operatoren vom Typ `a *= b` sind Abkürzungen für `a = a * b` .

Bei den logischen Operatoren "`&`" und "`|`" werden immer beide Seiten ausgewertet, d.h. Nebeneffekte werden berücksichtigt. Bei den logischen Operatoren "`&&`" und "`||`" wird eventuell nur die linke Seite ausgewertet, wenn damit das Resultat klar ist.

Wenn bei ganzzahligen Operanden ein Operand vom Typ `long` ist, dann wird die Operation als 64-Bit-Operation durchgeführt, sonst als 32-Bit-Operation. D.h. die Ergebnisse sind entweder vom Typ `int` oder `long`.

Mit dem Operator `instanceof` kann der Typ eines Ausdrucks verglichen werden, wenn beide Seiten Referenztypen sind (d.h. Klassen, Schnittstellen oder Felder). Dieser Operator gibt auch `true` zurück, wenn rechts eine Superklasse des Typs des linken Ausdrucks steht oder wenn eine Schnittstelle rechts steht, die irgendwo in der Klassenhierarchie des Typs des linken Ausdrucks implementiert wird.

3.3 Übungen

Übungen zur Operatorensyntax:

```
byte  a;
```

`++`	Nachinkrementierung	*Variable* `++`
`--`	Nachdekrementierung	*Variable* `--`
`++`	Vorinkrementierung	`++` *Variable*
`--`	Vordekrementierung	`--` *Variable*
`~`	Komplement (bitweise)	`~` *Ausdr*
`!`	logisches Nicht	`! boolean` *Ausdr*
`-`	unäres Minus	`-` *Ausdr*
`+`	unäres Plus	`+` *Ausdr*
`()`	Typwandlung	`(` *Typ*`)` *Ausdr*
`*`	Multiplikation	*Ausdr* `*` *Ausdr*
`/`	Division	*Ausdr* `/` *Ausdr*
`%`	Modulo	*Ausdr* `%` *Ausdr*
`+`	Addition	*Ausdr* `+` *Ausdr*
`-`	Subtraktion	*Ausdr* `-` *Ausdr*
`<<`	Linksverschiebung (bitweise)	*Ausdr* `<<` *Ausdr*
`>>`	Rechtsverschiebung (bitweise) mit Vorzeichen	*Ausdr* `>>` *Ausdr*
`>>>`	Rechtsverschiebung (bitweise) mit Nullen ohne Vorzeichen	*Ausdr* `>>>` *Ausdr*
`<`	kleiner als	*Ausdr* `<` *Ausdr*
`<=`	kleiner oder gleich als	*Ausdr* `<=` *Ausdr*
`>`	größer als	*Ausdr* `>` *Ausdr*
`>=`	größer oder gleich als	*Ausdr* `>=` *Ausdr*
instanceof	Typvergleich	*Ausdr* `instanceof` *Typ*
`==`	gleich	*Ausdr* `==` *Ausdr*
`!=`	ungleich	*Ausdr* `!=` *Ausdr*
`&`	bitweises Und	*Ausdr* `&` *Ausdr*
`&`	logisches Und	`boolean` *Ausdr*
`^`	bitweises exklusives Oder	*Ausdr* `^` *Ausdr*
`^`	logisches exklusives Oder	`boolean` *Ausdr*
`|`	bitweises inklusives Oder	*Ausdr* `|` *Ausdr*
`|`	logisches inklusives Oder	`boolean` *Ausdr*
`&&`	logisches Und	`boolean` *Ausdr*
`||`	logisches inklusives Oder	`boolean` *Ausdr*
`? :`	bedingter Ausdruck (ternär)	
`=`	einfache Zuweisung	*Variable* `=` *Ausdr*
`*=`	Multiplikation und Zuweisung	*Variable* `*=` *Ausdr*
`/=`	Division und Zuweisung	*Variable* `/=` *Ausdr*
`%=`	Modulo und Zuweisung	*Variable* `%=` *Ausdr*
`+=`	Addition und Zuweisung	*Variable* `+=` *Ausdr*
`-=`	Subtraktion und Zuweisung	*Variable* `-=` *Ausdr*
`<<=`	Linksverschiebung und Zuweisung	*Variable* `<<=` *Ausdr*
`>>=`	Rechtsverschiebung und Zuweisung	*Variable* `>>=` *Ausdr*
`>>>=`	Rechtsverschiebung und Zuweisung	*Variable* `>>>=` *Ausdr*
`&=`	Und und Zuweisung	*Variable* `&=` *Ausdr*
`^=`	excl. Oder und Zuweisung	*Variable* `^=` *Ausdr*
`|=`	incl. Oder und Zuweisung	*Variable* `|=` *Ausdr*

```
byte  b;
byte  c;
a = 5;
b = a++;
```
Was ist a? Was ist b?

```
b = ~a;
```
Was ist b?

```
b = !a
```
Was ist b? Geht das?

```
b = a << 3
```
Was ist b?

```
a = -7
b = a >> 2
```
Was ist b?
```
b = a >>> 2
```
Was ist b?

```
a = 6; b = 5;
```
Was ist a == b?
Was ist c = a & b?
Was ist c = a ^ b?
Was ist c = a | b?
Was ist c = a && b? Geht das?
Was ist c = a || b? Geht das?
Was ist a-- != b ? a : b?

Kapitel 4

Kontrollstrukturen

In diesem Kapitel behandeln wir nur die in anderen Sprachen üblichen Kontrollstrukturen. Auf Ausnahmebehandlung und Synchronisation von parallelen Abläufen wird in eigenen Kapiteln eingegangen.

4.1 Anweisungen

Die einfachste **Anweisung** ***(statement)*** ist eine leere Anweisung und besteht nur aus einem Semikolon:

```
;
```

Die leere Anweisung kann dann nützlich sein, wenn die Syntax eine Anweisung erfordert, wo man eigentlich keine benötigt. Die nächst komplexere Anweisung besteht aus einem **Ausdruck** ***(expression)*** und einem Semikolon:

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

Ein **Block** ***(block)*** ist eine möglicherweise leere Liste von Anweisungen zwischen geschweiften Klammern:

```
{
int   b = 0;
a = b + c;
b++;
}
```

Die Art der Einrückungen und Unterteilung in Zeilen spielt keine Rolle. Oben gezeigtes Beispiel ist äquivalent zu

```
{int b=0;a=b+c;b++;}
```

Ein Block ist eine Anweisung. Mit einem Block kann man mehrere Anweisungen als eine einzige Anweisung behandeln. Der **Geltungsbereich *(scope)*** eines im Block definierten Namens erstreckt sich vom Deklarationspunkt bis zum Ende des Blocks.

4.2 Bedingte oder `if`-Anweisung

Die bedingte Anweisung besteht aus dem Schlüsselwort `if` gefolgt von einem Bedingungsausdruck in runden Klammern, einer Anweisung und eventuell einem `else` mit Anweisung.

```
if (a > 0) b = c / a;
```

oder

```
if (a > 0)
   {
   b = c / a;
   c++;
   }
else
   {
   System.out.println ("a nicht positiv!");
   b = 0;
   }
```

Der Bedingungsausdruck muss ein Resultat vom Typ `boolean` liefern. Wenn er `true` liefert, dann wird in die Anweisung hinter dem `if`, sonst in die Anweisung hinter dem `else` verzweigt.

4.3 Fallunterscheidung (`switch`-Anweisung)

Die Fallunterscheidung testet gegen eine Menge von Konstanten:

```
char c;
// ---
switch (c)
   {
   case 'a':
      x = xa;
      break;
   case 'b':
   case 'c': x = xb; break;
   default:
      x = 0;
      break;
   }
```

Der Ausdruck `(c)` muss vom Typ `char`, `byte`, `short`, `int` oder `Enum` sein.

Die `case`-Konstanten oder `case`-Konstanten-Ausdrücke müssen alle verschieden und von einem nach `int` castbaren Typ oder `Enum` sein. Die `default`-Alternative wird genommen, wenn keine andere passt. Eine `default`-Alternative muss nicht gegeben werden. Die `break`-Statements werden benötigt, damit das `switch`-Statement nach Abarbeitung einer Alternative verlassen wird. Sonst würden alle folgenden Alternativen auch abgearbeitet werden. Die verschiedenen `case`-Marken dienen nur als Einsprungspunkte, von wo an der Programmfluss weitergeht. Das `switch`-Statement ist ein übersichtliches Goto!

4.4 `while`-Schleife

Die `while`-Schleife besteht aus dem Schlüsselwort `while` gefolgt von einem Bedingungsausdruck vom Typ `boolean` in runden Klammern und einer Anweisung:

```
a = 10;
while (a > 0)
   {
   b = c / a;
   a--;
   }
```

Der Bedingungsausdruck in der `while`-Anweisung wird ausgewertet. Solange der Ausdruck `true` ist, wird die Anweisung hinter dem `while` immer wieder durchgeführt. Mit dem Schlüsselwort `do` kann man die auszuführende Anweisung auch vor das `while` setzen:

```
a = 10;
do {
   b = c / a;
   a--;
   } while (a > 0);
```

Die Bedingung wird nach Durchführung der Anweisung abgeprüft.

4.5 `for`-Schleife

Die `for`-Schleife

```
for (int i = 0; i < 10; i++)
   {
   a[i] = i;
   b[i] = i * i;
   }
```

ist äquivalent zu

```
{
int i = 0;
while (i < 10)
   {
   a[i] = i;
   b[i] = i * i;
   i++;
   }
}
```

aber lesbarer, da die Schleifenkontrolle lokalisiert ist. Zu bemerken ist, dass der Zähler `i` am Ende der Schleife hochgezählt wird, gleichgültig, ob man `i++` oder `++i` schreibt.

Wenn die Variable `i` im Schleifenkopf definiert wird, dann gehört sie zum Scope der `for`-Schleife, so dass `i` nach der `for`-Schleife nicht mehr zur Verfügung steht.

4.5.1 Enhanced `for`-Loop

Die Verwaltung des Laufindexes ist sehr lästig und kann eigentlich leicht automatisiert werden. Daher gibt es alternativ folgende `foreach`-Syntax:

```
for (<Typ>  <Laufvariable> : <Collection oder Feld>)
```

```
Klasse[] k = new Klasse[10];
for (int i = 0; i < k.length; i++)
   {
   k[i] = new Klasse ();
   }

// foreach-Loop:

for (Klasse x :k)
   {
   x.methode ();
   }
```

Wir gehen davon aus, dass in der Klasse `Klasse` eine Methode `methode` definiert ist. Die letzte `for`-Schleife ist äquivalent zu:

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

`k` kann ein Feld oder eine Collection sein.

Man kann keine Zuweisungen an die Laufvaribale machen. `x = ...` wird vom Compiler zwar toleriert, aber die Zuweisung bleibt wirkungslos. Das ist insbesondere bei Feldern von primitiven Datentypen eine Fehlerquelle.

4.6 Sprünge

Es gibt keine Goto-Anweisung. (Dennoch ist `goto` in Java ein reserviertes Wort.) Mit den Sprunganweisungen `break` und `continue` kann eine umgebende Anweisung verlassen werden.

Mit

```
break
```

oder

`break` *Sprungmarke*

wird an das Ende der direkt umgebenden bzw. der mit *Sprungmarke* bezeichneten Anweisung gesprungen.

Mit

```
continue
```

oder

`continue` *Sprungmarke*

wird sofort mit dem neuen Schleifendurchlauf der direkt umgebenden bzw. der mit *Sprungmarke* bezeichneten `for`- oder `while`-Anweisung begonnen. Es ist zu bemerken, dass bei einer `while`-Schleife und `continue` ein Laufindex nur dann verändert wird, wenn das entsprechende Statement schon vor dem `continue` ausgeführt wurde.

```
int   a = 0;
hierher: for (int i = 0; i < 10; i++)
   {
   a++;
   for (int j = 0; j < 20; j++)
      {
      a++;
      a++;
      if (a == 5) continue;
                     // Es geht gleich mit dem nächsten j weiter.

      if (a == 6) break;
                     // Es geht gleich nach for j weiter.

      if (a == 7) continue hierher;
                     // Es geht gleich mit dem nächsten i weiter.

      if (a == 8) break hierher;
                     // Es geht gleich nach for i weiter.
      a++;
      a++;
      }
   // nach for j
   a++;
   }
// nach for i
```

4.7 Wertrückgabe

Mit der `return` Anweisung wird die Kontrolle wieder an die aufrufende Methode zurückgegeben, wobei ein Ausdruck des Resultattyps der aufgerufenen Methode zurückzugeben ist. (Bei `void` als Resultattyp muss das Argument fehlen. Ein `return` ist nicht notwendig.)

4.8 Methoden, Funktionen, Operationen

Eine **Methode** oder **Funktion** oder **Operation** ist ein Teil eines Programms, der über einen Namen (Methodenname, Funktionsname, Operationsname) beliebig oft aufgerufen werden kann. Eine Methode oder Funktion kann in Java nur innerhalb einer Klasse deklariert werden. Außerdem muss sie dort auch definiert d.h. implementiert werden. Die Methoden**definition** hat folgende Form:

Modifikatoren Rückgabetyp Methodenname (Kommaliste von Argumenten)
{
Implementation
}

Die erste Zeile wird auch Methoden**signatur** oder Methoden**spezifikation** genannt.

z.B.:

```
public double  methode (int a, double x, char z)
   {
   double   y;
   y = a * (z - '0');
   return x * y;
   }
```

Eine Methode darf nicht innerhalb einer anderen Methode definiert werden. Die Liste von Argumenten muss Typangaben und Namen für die Argumente enthalten. Mit Rückgabetyp ist der Typ des Rückgabewerts gemeint.

Grundsätzlich werden alle Argumente immer als Wert übergeben, so dass keine durch die Methode veränderte Argumente zurückgegeben werden können. Nur der Rückgabewert ist durch die Methode zu verändern. Wenn variable Argumente benötigt werden, dann muss man Referenzdatentypen übergeben, die dann in der Methode verändert werden. Ein Referenzdatentyp wird bei der Übergabe an die Methode nicht kopiert. Auch wenn er als Resultat zurückgegeben wird, wird er nicht kopiert.

Normalerweise haben verschiedene Methoden verschiedene Namen. Aber wenn Methoden ähnliche Aufgaben haben, dann kann es nützlich sein, den gleichen Namen zu verwenden. In Java ist es erlaubt, Methoden zu **überladen**, d.h. denselben Namen öfter zu verwenden, sofern Anzahl oder Typ der Argumente unterschiedlich sind. Welche Methode zu verwenden ist, kann der Compiler mit Hilfe der Typen der Argumente feststellen.

Die *Signatur* oder *Spezifikation* einer Methode wird durch den Methodennamen, die Anzahl und den Typ der Argumente festgelegt. Methoden mit unterschiedlicher Signatur sind verschiedene Methoden. Der Rückgabetyp einer Methode gehört *nicht* zur Signatur.

Der *Typ* einer Methode wird bestimmt durch Rückgabetyp und Anzahl und Typ der Argumente. Der Methodenname spielt für den Typ der Methode *keine* Rolle. Das hat in Java allerdings keine Bedeutung, da es keine Methodenzeiger oder Methoden-Referenzdatentypen gibt.

Methoden können in Java **keine Defaultargumente** haben.

Die übergebenen Argumente gehören zum Scope der Methode. Ihre Namen sind nur in der Methode bekannt und der dafür angelegte Speicher besteht nur so lang, wie die Methode läuft. In der Methode mit `new` angelegte Objekte werden nach Beendigung der Methode nicht mehr referenziert und fallen bei Gelegenheit in die Hände des Garbage-Kollektors mit Ausnahme eventuell eines Objekts, das als Funktionswert zurückgegeben wird und wird einem Refenzddatentyp zugewiesen wird.

Auf die Modifikatoren von Methoden sind wir im Kapitel "Datentypen" schon zusammenfassend eingegangen.

Argumente variabler Länge: Das letzte Argument einer Methode kann variable Länge habe, indem man den Typ des letzten Arguments um drei Punkte `...` erweitert:

```
public static int min (int a, int... b)
   {
   int   min = a;
   for (int i : b)
      {
      if (i < min) min = i;
      }
   return min;
```

Folgende Aufrufe sind möglich:

```
min (5)
min (7, 4)
min (7, 4, 12, 3, 6)
min (7, new int[] {4, 12, 3, 6})
```

Die Signatur der Methode ist äquivalent zu:

```
public static int min (int a, int[] b)
```

(Aber hier wäre nur der letztere Aufruf möglich.)

Häufig wird diese Syntax für Object-Argumente verwendet:

```
public void methode (Object... o)
```

4.9 Assertions – Zusicherungen

Für Debug-Zwecke kann es nützlich sein, Programmierannahmen gelegentlich zu überprüfen. Wenn die Annahme als boolscher Ausdruck `assertion` eventuell mit einer Fehlermeldung `errorcode` formuliert werden kann, dann sieht die Syntax folgendermaßen aus:

```
assert assertion;
```
oder
```
assert assertion : errorcode;
```

Falls `assertion` `false` ergibt, wird ein `AssertionError` geworfen mit `errorcode` als Meldung. Da es ein `Error` ist, muss der Fehler nicht abgefangen werden.

Assertions müssen allerdings mit den Optionen `-ea:Klasse` oder `-ea:Paket` aktiviert werden:

```
java -ea:KlasseMitAssertion KlasseMitAssertion
```

Beispiel:

```
public class KlasseMitAssertion
   {
   public static void   main (String[] argument)
      {
      assert 5 == 5;
      try { assert 4 == 5;}
         catch (AssertionError e) { e.printStackTrace (); }
      try { assert 6 == 7 : "6 == 7"; }
         catch (AssertionError e) { e.printStackTrace (); }
      }
   }
```

4.10 Übungen

Übung Kontrollstrukturen: Die Java-Klasse `Produkt` zur Berechnung des Produkts von zwei Zahlen soll folgendermaßen geändert und ergänzt werden:

1. Es sollen nur Integer eingelesen werden.
2. Der Benutzer soll angeben, welche Verknüpfung zu machen ist (`- + * / ^ %`). Fehleingaben sollten abgefangen werden. (Für die Potenz gibt es die Methode `Math.pow (double x, double y)`.)
3. Das Programm soll solange laufen, bis der Benutzer als Verknüpfungszeichen `~` eingibt.
4. Bei Verknüpfung `/` soll der Nenner auf Null geprüft werden. Gegebenenfalls soll eine Fehlermeldung ausgegeben werden.

Kapitel 5

Die Java-Klasse

Die Klasse ist ein vom Benutzer definierter Datentyp. Sie besteht aus Datenelementen möglicherweise verschiedenen Typs und einer Anzahl Funktionen, mit denen diese Daten manipuliert werden können.

Das Klassenkonzept hat folgende Leistungsmerkmale:

- Bildung neuer Typen, die den Bedürfnissen der ProgrammierIn Daten zu repräsentieren genügen.
- Kontrolle des Zugangs zu Daten: Datenstrukturen können vor dem direkten Zugriff des Benutzers geschützt werden **(Datenabstraktion, Datenkapselung**, ***information hiding)***. Details der Implementation können von der Benutzeroberfläche des Typs getrennt werden.
- Initialisierung und Aufgabe von Objekten kann für den Benutzer transparent erfolgen.
- Vermeidung von Code-Wiederholungen durch Vererbungsmechanismen.
- Bildung von typunabhängigen Datenstrukturen.

5.1 Syntax

Die Klassendefinition besteht aus einem **Klassenkopf *(class head)***, der sich aus dem Schlüsselwort `class` und einem Klassennamen zusammensetzt, und einem **Klassenkörper *(class body)***, der in geschweiften Klammern steht.

```
public class Klassenname
   {
   }
```

Konventionsgemäß beginnen Klassennamen mit einem Großbuchstaben.

Dem Modell der **Datenabstraktion** oder **Datenkapselung** entspricht es, alle Daten `private` zu deklarieren und sogenannte **Zugriffsfunktionen** ***(access functions)***, mit denen die Daten manipuliert werden, `public` zu deklarieren. Die in einer Klasse definierten Funktionen heißen **Methoden** oder **Elementfunktionen** ***(method, member function)***.

In Java können wir das Sichtbarkeits-Niveau mit den weiteren Modifikatoren `protected` und *default* (ohne Modifikator) in Beziehung auf Klassen im selben Package und erbende Klassen genauer einstellen (vgl. Kapitel Datentypen).

Warum will man Datenkapselung? Ein wichtiger Grund ist, die Objekte der Klasse gegen zufälligen oder absichtlichen Missbrauch zu schützen. Eine Klasse enthält häufig Variable, die voneinander abhängen und die in einem konsistenten Zustand erhalten werden müssen. Wenn man einer ProgrammiererIn erlaubt, diese Variablen direkt zu manipulieren, dann kann sie eine Variable ändern ohne die in Beziehung stehenden auch zu ändern, wodurch das Objekt in einen inkonsistenten Zustand gerät. Wenn sie stattdessen eine Methode aufrufen muss, um die Variable zu ändern, dann kann die Methode dafür sorgen, dass die anderen Variablen entsprechend geändert werden. Datenkapselung reduziert die Möglichkeiten der – eventuell unerlaubten – Datenmanipulation drastisch.

Beispiel Klausurnote:

Als Beispiel wollen wir uns die Klasse `Klausurnote` anschauen.

Eine Klausurnote ist ein so komplexes Objekt, dass wir es mit einem elementaren Typ nicht adäquat darstellen können. Klausurnoten können nur Werte zwischen 1,0 und 5,0 annehmen und sie müssen auf Zehntel gerundet sein. Ferner gibt es für eine Klausurnote eine verbale Darstellung (sehr gut, gut usw). Außerdem wollen wir noch den Namen des Faches mitverwalten. Damit all dies gewährleistet ist, bilden wir einen neuen Typ `Klausurnote`, indem wir eine entsprechende Klasse definieren:

```
public class Klausurnote
   {
   private String  fach;
   private char[]  note = new char[3];

   public void setFach (String Fach) { }
   public int  setNote (String Note) { }
   public String  getNumerisch () { }
   public String  getVerbal () { }
   public void drucke () { }
   }
```

Als Datenmodell für die Klausurnote wählen wir einen `String` für den Fachnamen und ein Feld von drei `char` für die Note, um das deutsche Dezimalkomma darstellen zu können. Als Interface dieser Klasse bieten wir fünf Funktionen (Methoden) an, mit denen der Benutzer der Klasse die Note belegen (`setFach`, `setNote`), lesen (`getNumerisch`, `getVerbal`) und auf dem Bildschirm ausgeben kann (`drucke`).

Das Hauptprogramm packen wir in eine andere Klasse. Es sieht folgendermaßen aus:

```
public class HauptKlausurnote
   {
   public static void   main (String[] argument)
      {
      Klausurnote bio = new Klausurnote ();
      bio.setFach ("Biologie");
      bio.setNote ("2,3");
      bio.drucke ();
      }
   }
```

Es folgt die vollständige Klasse `Klausurnote` mit den implementierten Methoden:

```
public class Klausurnote
   {
   private String fach;
   private char[] note = new char[3];

   public void setFach (String Fach)
      {
      fach = Fach;
      }

   public int  setNote (String Note)
      {
      int n = Note.length ();
      if ((n > 3) || (n == 0)
         || (n > 1 && Note.charAt (1) != ',')
         )
         {
         System.out.print ("Note \"" + Note);
         System.out.println ("\" ist keine gültige Note!");
         return -1;
         }
      switch (Note.charAt (0))
         {
         case '5':
            if (n == 3 && Note.charAt (2) != '0')
               {
               System.out.print ("Note \"" + Note);
               System.out.println ("\" ist keine gültige Note!");
               return -1;
               }
         case '4': case '3': case '2': case '1':
            note[0] = Note.charAt (0);
            note[1] = ',';
            if (n < 3) note[2] = '0';
```

```
            else if (Note.charAt (2) >= '0' && Note.charAt (2) <= '9')
               note[2] = Note.charAt (2);
            else
               {
               System.out.print ("Note \"" + Note);
               System.out.println ("\" ist keine gültige Note!");
               return -1;
               }
            return 0;
         default:
            System.out.print ("Note \"" + Note);
            System.out.println ("\" ist keine gültige Note!");
            return -1;
         }
      }

   public String  getNumerisch ()
      {
      return new String (note);
      }

   public String  getVerbal ()
      {
      double   x = zahl ();
      if (x < 0.99) return "ungültig";
      else if ( x < 1.51) return "sehr gut";
      else if ( x < 2.51) return "gut";
      else if ( x < 3.51) return "befriedigend";
      else if ( x < 4.01) return "ausreichend";
      else return "nicht ausreichend";
      }

   public void drucke ()
      {
      System.out.print ("Die Note ");
      System.out.print (fach);
      System.out.print (" ist ");
      System.out.print (getVerbal ());
      System.out.print (" (");
      System.out.print (getNumerisch ());
      System.out.println (").");
      }

   private double zahl ()
      {
      double   x = note[0] - '0';
      x = x + (note[2] - '0') / 10.0;
      return x;
      }
```

}

Bemerkung 1: Eine Elementfunktion kann andere Elemente der Klasse ohne Dereferenzierung ansprechen. Z.B. `note` kann direkt benutzt werden. Doch wie wäre so etwas zu dereferenzieren? Mit der Variablen `this` wird immer das gerade vorliegende Objekt referenziert. `this` kann immer zur Klärung von Namenskonflikten oder auch nur zur Verdeutlichung des Codes benutzt werden, z.B.:

```
this.note[1]
```

`this` wird auch benötigt, wenn eine Methode das vorliegende Objekt als Resultat zurückgibt.

Bemerkung 2: Ein Objekt, d.h. seine Elementfunktionen können auf die privaten Daten eines anderen Objekts *derselben* Klasse zugreifen.

Bemerkung 3: Eine – allerdings etwas umständliche – "Java-like" Implementation der Methode `zahl ()` wäre:

```
private double zahl ()
   {
   StringBuffer   s = new StringBuffer (new String (note));
   s.setCharAt (1, '.');
   double   x = 0.0;
   try
      {
      x = Double.valueOf (new String (s)).doubleValue ();
      }
      catch (NumberFormatException e)
         {
         System.err.println ("Fehler in Klausurnote.zahl (): " + e);
         }
   return x;
   }
```

5.2 Konstruktoren

Wenn wir ein Objekt der Klasse `Klausurnote` anlegen, dann wird es – bisher jedenfalls – nur defaultmäßig initialisiert. Die Initialisierung von Datentypen sollte i.a. einerseits nicht vergessen werden, andererseits höchstens ein einziges Mal erfolgen.

Man möchte im wesentlichen zwei Dinge garantieren:

- Automatische, einmalige Initialisierung von Variablen
- Automatische Allokierung von Speicher

Mit den normalen Zugriffs-Methoden kann dies nicht garantiert werden. Daher gibt es für Klassen spezielle Initialisierungsfunktionen, die sogenannten **Konstruktoren** ***(constructor)***. Ein Konstruktor wird noch nicht bei der Deklaration einer Referenz aufgerufen, sondern erst beim Anlegen von Speicher mit `new` (Allokieren oder Erzeugen eines neuen Objekts). Der Konstruktor erzwingt die automatische Durchführung einer Methode – nämlich sich selbst – beim Erzeugen eines Objekts.

Der Konstruktor ist eine Funktion, die *denselben* Namen wie die Klasse hat, *keinen* Rückgabewert hat und nur beim Speicheranlegen für ein Objekt der Klasse oder innerhalb von Konstruktoren derselben Klasse aufgerufen wird. Der Rückgabewert eines Konstruktors ist implizit die neu erzeugte Instanz der Klasse. Falls man ausversehen einen Rückgabetyp angibt, wird das als Definition einer normalen Methode verstanden, was zu schwer entdeckbaren Fehlern führen kann.

Eine Klasse kann mehrere Konstruktoren haben, die zwar denselben Namen tragen, sich aber durch die Anzahl und Art der Argumente unterscheiden, d.h. unterschiedliche Signatur haben. Entsprechend können auch alle anderen Methoden einer Klasse **überladen** ***(overload)*** werden.

```
public class Klassenname
   {
   public   Klassenname (int i, int j) {} // Konstruktor
   public   Klassenname (double x) {}     // Konstruktor
   public   Klassenname () {}             // Konstruktor
   }
```

In unserem Notenbeispiel wollen wir bei Initialisierung eines Notenobjekts diesem Objekt entweder die zwar ungültige Note "`0,0`" und den Namen "N.N." zuordnen oder eine vom Benutzer bei der Definition des Notenobjekts angegebene korrekte Note und Fach zuordnen. Dazu definieren wir drei Konstruktoren:

```
public class   Klausurnote
   {
   public   Klausurnote (String Note, String Fach)
      {
      if (setNote (Note) == -1)
         {
         note[0] = '0'; note[1] = ','; note[2] = '0';
         }
      setFach (Fach);
      }

   public   Klausurnote (String Note)
      {
      this (Note, "N.N.");
      }

   public   Klausurnote ()
      {
      this ("");
      }
```

```
   // ---
   }
```

Bemerkung 1: Mit dem Schlüsselwort `this ()` werden andere Konstruktoren derselben Klasse aufgerufen. Gegenüber C++ ist das eine sehr praktische Möglichkeit, um Codewiederholungen zu vermeiden.

Bemerkung 2: Der Konstruktoraufruf `this` kann nur in einem Konstruktor erfolgen und muss dort als erstes Statement erscheinen.

Bemerkung 3: Der Konstruktor hat in Java nicht die große Bedeutung wie in C++, da viele Initialisierungen bei der Definition einer Variablen möglich sind. Außerdem ist die Speicherverwaltung viel einfacher.

Die Definition von Objekten sieht nun folgendermaßen aus:

```
public class HauptKlausurnote
   {
   public static void   main (String[] argument)
      {
      Klausurnote bio = new Klausurnote ();
      bio.setFach ("Biologie");
      bio.setNote ("2,3");
      Klausurnote phy = new Klausurnote ("1,8");
      Klausurnote che = new Klausurnote ("3,1", "Chemie");
      }
   }
```

Die verschiedenen überladenen Konstruktoren sollten sich ähnlich verhalten. Sie sollten *konsistente* Objekte kreieren. Ansonsten müssten komplizierte Unterscheidungen bei anderen Elementfunktionen getroffen werden.

Ein Konstruktor ohne Argumente heißt *Default-Konstruktor*. Der Default-Konstruktor wird vom Compiler *nur* dann automatisch erzeugt, wenn es keine vom Klassen-Implementor definierte Konstruktoren gibt.

Bemerkung 4: Ein Konstruktor kann nur nach `new` oder mit `this ()` in einem anderen Konstruktor aufgerufen werden.

5.3 Instanz-Initialisatoren

Eine weitere, elegante Möglichkeit zur Initialisierung sind sogenannte **Instanz-Initialisatoren**. Das sind namenlose Code-Blöcke, die an beliebiger Stelle und beliebig oft in der Klasse auftreten können. Sie werden in der Reihenfolge ihres Auftretens nach dem Superklassen-Konstruktor, aber vor allen anderen Konstruktoren abgearbeitet. Damit kann man z.B. Objektfelder an der Stelle ihres Auftretens initialisieren, was oft die Lesbarkeit des Codes erhöht.

5.4 Aufgabe von Objekten

Um die Deallokierung von Speicher oder Objekten muss sich der Java-Programmierer nicht kümmern! Java benutzt ***garbage collection***, um Objekte zu deallokieren, die nicht mehr benötigt werden. Der **Java-Garbage-Kollektor** (GC) läuft als Thread mit niedriger Priorität und tut daher seine Arbeit, wenn sonst nichts zu tun ist. Nur wenn Speicherplatz fehlt, läuft dieser Thread mit höchster Priorität.

Wenn ein Block verlassen wird, dann werden alle in dem Block erzeugten Objekte zur Aufgabe freigegeben. Aber es kann vorkommen, dass man – insbesondere große – Objekte schon früher zur Aufgabe freigeben möchte. Das kann man dem Garbage-Kollektor durch die Zuweisung von `null` anzeigen:

```
int[] riesig = new int[1000000];
// benutzt riesig
// riesig wird nicht mehr benötigt
riesig = null;
```

Mit dem Aufruf

```
System.gc ();
```

kann man den GC zwingen, nicht referenzierten Speicherplatz freizugeben.

Bevor ein Objekt vom Garbage-Kollektor deallokiert wird, wird die Methode `finalize ()` aufgerufen. Diese Methode spielt in Java ungefähr, aber wirklich nur sehr "ungefähr" die Rolle eines C++ Destruktors. `finalize ()` wird verwendet, um sogenannte "persistente" Resourcen freizugeben wie z.B. Dateien oder Sockets, die dem Betriebssystem zurückgegeben werden müssen, wenn sie nicht mehr gebraucht werden.

Bemerkung 1: Der Java-Interpreter kann verlassen werden, ohne dass der Garbage-Kollektor die Möglichkeit hat, alle Objekte zu deallokieren. Damit würden auch die entsprechenden `finalize`-Methoden *nicht* aufgerufen werden. In diesen Fällen gibt normalerweise das Betriebssystem belegte Resourcen frei.

Bemerkung 2: Es ist nicht spezifiziert, wann und in welcher Reihenfolge der Garbage-Kollektor die Objekte deallokiert, d.h. wann und in welcher Reihenfolge die `finalize`-Methoden aufgerufen werden.

Bemerkung 3: Eine `finalize`-Methode kann das Objekt wieder zum Leben erwecken, indem `this` wieder einer Variablen zugewiesen wird. In solch einem Fall wird `finalize` nicht wieder aufgerufen, wenn das Objekt schließlich deallokiert werden kann.

Bemerkung 4: `finalize`-Methoden können Exceptions werfen. Wenn diese Exceptions nicht gefangen werden, werden sie ignoriert, d.h. es kommt nicht zum Absturz.

Bemerkung 5: Ab JDK 1.2 wird `finalize` nicht direkt vom GC, sondern von einem niedrigprioren Thread aufgerufen. Mit der Methode `System.runFinalization ()` kann man die JVM auffordern, `finalize ()` für alle vom GC als nicht mehr referenziert festgestellte Objekte aufzurufen.

Bemerkung 6: Es ist i.a. sehr schwierig, `finalize ()` zu programmieren. In den meisten Fällen kommt man mit einer Kombination von `finally`-Blöcken und expliziten `close`-Methoden aus.
Regel: Vermeide `finalize`.

Für unser Beispiel `Klausurnote` schreiben wir zur Demonstration eine `finalize`-Methode.

```
protected void finalize () throws Throwable
   {
   System.out.print ("Die Note ");
   System.out.print (fach);
   System.out.print (" (");
   System.out.print (getVerbal ());
   System.out.print (" (");
   System.out.print (getNumerisch ());
   System.out.println ")) wird gleich deallokiert.");
   super.finalize ();
   }
```

Die Metode `finalize` muss `protected` sein, da sie höchstens dasselbe Sichtbarkeitsniveau (d.h. nicht sichtbarer) wie in `Object` haben darf.

Die Aktivität des GC kann man mit der Option `-verbosegc` beobachten:

```
java -verbosegc Speicherfresser
```

Dabei ist `Speicherfresser` folgendes kleine Programm:

```
public class   Speicherfresser
   {
   public static void   main (String[] arg)
      {
      int   i = 0;
      while (true)
         {
         i++;
         System.err.println ("Iteration " + i);
         double[] d = new double[10000];
         for (int j = 0; j < d.length; j++) d[j] = Math.random ();
         }
      }
   }
```

5.5 Klassenvariable und -methoden

Eine Klasse ist ein Typ und jede Instanz einer Klasse hat eine eigene Kopie der Datenelemente der Klasse. Aber es kann sein, dass manche Klassen so implementiert werden sollen, dass alle Instanzen einer Klasse dasselbe Datenelement benutzen. Das können z.B. gemeinsame Zähler etwa für die Anzahl der Objekte dieser Klasse sein oder generell Daten, auf die jedes Objekt

zugreift und die sich mit der Zeit ändern, sodass es nicht möglich wäre, eine Konstante daraus zu machen. Um solche Daten als Klassenmitglieder zu verwalten gibt es das Schlüsselwort `static`. Statische Datenelemente heißen auch **Klassenvariable** ***(class variable)***.

Z.B. verwalten wir in der Klasse `Klausurnote` einen Zähler
`anzKlausurnoten`, der die Anzahl der Instanzen dieser Klasse mitzählt.

```
public class Klausurnote
   {
   // ---
   static int  anzKlausurnoten = 0;
   // ---
   }
```

Da es in Java keine globale Variablen gibt, ist das eine wichtige Möglichkeit globale Variable zu emulieren.

Durch die `static`-Deklaration wird erreicht, dass `anzKlausurnoten` für alle Objekte der Klasse `Klausurnote` nur einmal angelegt wird. Innerhalb des Scopes einer Klasse kann ein statisches Datenmitglied wie jedes andere Element direkt mit seinem Namen angesprochen werden. Außerhalb des Scopes einer Klasse gibt es zwei Möglichkeiten, sofern das statische Element sichtbar ist:

```
Klausurnote  a = new Klausurnote ();
a.anzKlausurnoten = 5;           // Zugriff über ein Objekt
Klausurnote.anzKlausurnoten = 6; // Zugriff über Klassenname
```

Auch Elementfunktionen können als `static` deklariert werden (**Klassenmethoden**, ***class methods)***. Das macht dann Sinn, wenn die Funktion nur auf statische Daten zugreift, sodass sie auch ohne Referenz auf ein Objekt der Klasse aufgerufen werden kann. Für sie ist `this` nicht definiert.

```
public class Klausurnote
   {
   // ---
   static void druckAnzKlausurnoten () {}
   // ---
   }
```

Es gibt zwei Klassen im API, die nur statische Methoden definieren:
`java.lang.Math` arbeitet nur mit Standardtypen, die keine Objekte sind.
`java.lang.System` bietet Systemfunktionen, für die es kein sinnvolles Objekt gibt.

In der Klasse Klausurnote definieren wir noch zwei Überladungen einer Methode `besser`, die uns die bessere von zwei Noten zurückgibt. Eine der beiden Überladungen ist statisch.

```
public class Klausurnote
   {
   // ---
   public Klausurnote   besser (Klausurnote b)
      {
      if (zahl () < b.zahl ()) return this;
      else return b;
      }

   public static Klausurnote  besser (Klausurnote a, Klausurnote b)
      {
      if (a.zahl () < b.zahl ()) return a;
      else return b;
      }
   // ---
   }
```

Im Anwendungsprogramm hat man dann die Alternativen:

```
Klausurnote a = new Klausurnote ("2,7", "Bio");
Klausurnote b = new Klausurnote ("3,1", "Che");
Klausurnote c;
c = a.besser (b);
// oder
c = Klausurnote.besser (a, b);
// oder
c = c.besser (a, b);
```

Klassenvariable werden initialisiert, wenn die Klasse das erste Mal geladen wird. Instanzvariable werden initialisiert, wenn ein Objekt der Klasse erzeugt wird.

Wenn die Initialisierung von Klassenvariablen komplizierter ist als eine einfache Zuweisung, dann kann ein **statischer Initialisator** geschrieben werden, der beim Laden der Klasse aufgerufen wird. Syntaktisch besteht der statische Initialisator aus dem Schlüsselwort `static` und einem Block.

```
public class Klausurnote
   {
   // ---
   static int[] feld;
   static
      {
      feld = new int[20];
      for (int i = 0; i < 20; i++)
         {
         feld[i] = i;
         }
      }
   // ---
   }
```

Eine Klasse kann beliebig viele statische Initialisatoren haben, die vom Compiler in der vorgegebenen Reihenfolge zu einem einzigen Initialisator zusammengefasst werden, der beim Laden der Klasse ein einziges Mal ausgeführt wird. `native` Methoden werden typischerweise in einem Initialisator mit den Methoden `System.load ()` oder `System.loadLibrary ()` geladen.

Statische Methoden können von Erben nicht überschrieben werden.

5.6 Übungen

Übung Methoden Klausurnote:

1. Schreiben Sie für die Klasse `Klausurnote` einen Konstruktor, der ein `double` und einen `String` als Argumente zur Initialisierung der Note mit Fach nimmt, und wenden Sie ihn an. (Dieser Konstruktor soll auch Noten wie `1.67` vertragen, die zu `1.6` geschnitten werden.)

2. Schreiben Sie für die Klasse `Klausurnote` einen Konstruktor, der nur ein `double` als Argument zur Initialisierung der Note nimmt, und wenden Sie ihn an. (Rufen Sie dabei Ihren Konstruktor auf.)

3. Schreiben Sie für die Klasse `Klausurnote` die Methode `plus`, mit der zwei Klausurnoten addiert werden, wobei das geschnittene Mittel gebildet wird ($3,3 + 3,8 = 3,5$).

   Anwendung:

   ```
   Klausurnote a, b, c;
   // Objekterzeugung
   c = a.plus (b);
   ```

4. Schreiben Sie für die Klasse `Klausurnote` die `static`-Funktion `plus`, mit der zwei Klausurnoten addiert werden, wobei das geschnittene Mittel gebildet wird.

   Anwendung:

```
Klausurnote a, b, c;
// Objekterzeugung
c = Klausurnote.plus (a, b);
```

Nur für C++ -Programmierer: Kann diese Funktion auch so

```
c = Klausurnote.plus (1.6, 2.8);
```

angewendet werden? Ausprobieren!

5. Schreiben Sie die Methode `plus2` so, dass das aufrufende Objekt das Resultat der Addition ist und dass das Resultat zurückgegeben wird.

   ```
   d = c.plus2 (a, b)
   ```

6. Zählen Sie die instanziierten Klausurnotenobjekte mit:

   (a) Alle jemals angelegten Objekte.

   (b) Nur die, die gerade noch angelegt sind.

   (c) Schreiben Sie die statische Methode, die die Zählerstände auf dem Bildschirm ausgibt.

7. Ändern Sie die Datenrepräsentation für die Note. Stellen Sie sie mit einem Byte dar. Das gibt zwar eine größere Änderung der Klasse `Klausurnote`, aber der Code für das Anwendungsprogramm (`main`) sollte unverändert bleiben.

Übung Rechteck:

1. Schreiben Sie eine Klasse `Rechteck` mit einem Anwendungsprogramm `main` und den `get`- und `set`-Methoden `laenge`, `breite`, `flaeche`, `umfang` und Konstruktoren. Wählen Sie als Datenrepräsentation Länge und Breite des Rechtecks.

2. Schreiben Sie eine Methode `zeige`, die alle Daten des Rechtecks in einer Zeile darstellt.

3. Schreiben Sie die Methode `equals` für `Rechteck`.

4. Kopieren Sie das Programm in einen anderen File. Ändern Sie jetzt im neuen Programm die Datenrepräsentation: Anstatt der Länge und Breite sollen jetzt Fläche und Umfang verwendet werden. Führen Sie die notwendigen Korrekturen an den Methoden durch. Die Länge a und die Breite b eines Rechtecks errechnen sich aus der Fläche f und dem Umfang u folgendermaßen:

$$b = \frac{u - \sqrt{u^2 - 16f}}{4}$$

$$a = \frac{u - 2b}{2}$$

Übung Konten:

1. Schreiben Sie eine Klasse `Konto`, die einen Kontostand `ktostd` und den Namen eines Kontoinhabers `ktoinh` verwaltet.

   Folgende Methoden bzw Konstruktoren sollen angeboten werden:

   - Das Konto kann nur mit einem positiven Betrag eröffnet werden.
   - `void ktoStand ()` :
     Diese Methode gibt den Namen des Kontoinhabers und den Kontostand in einer Zeile aus.
   - `void einzahlen (double betrag)` :
     Diese Methode nimmt als Argument einen Betrag und aktualisiert das Konto, indem der Betrag addiert wird. Der eingelesene Betrag muß positiv sein.
   - `void abheben (double betrag)` :
     Diese Methode nimmt als Argument einen Betrag und aktualisiert das Konto, indem der Betrag subtrahiert wird. Der eingelesene Betrag muß positiv sein.

2. **Anwendungsprogramm:**
   Schreiben Sie ein Programm, das ein Konto mit einem Namen und dem Anfangsbetrag von DM 17 anlegt. Anschließend wird der Kontostand ausgegeben. Dann folgt eine Einzahlung mit anschließender Anzeige des Kontostands. Schließlich wird noch ein Betrag abgehoben.

3. **Kontonummer:**
   Verwalten Sie eine Kontonummer. Wenn ein neues Konto angelegt wird, soll die Kontonummer größer sein als alle vorhergehenden Kontonummern. Verwalten Sie alle Kontonummern in einem Feld.

Kapitel 6

Vererbung

Durch das Konzept der Klasse bietet Java schon sehr mächtige Möglichkeiten. Der Vererbungsmechanismus bringt nun eine elegante Möglichkeit, Code wiederzuverwenden. Durch Vererbung kann die Code-Redundanz wesentlich verringert werden.

Mit den heutzutage zur Verfügung stehenden Editoren ist die Wiederverwendung einmal geschriebenen Codes scheinbar kein Problem, da beliebige Mengen Text beliebig oft kopiert werden können. Damit handelt man sich aber ein furchtbares Redundanzproblem ein. Häufig müssen Programmteile leicht modifiziert werden, oder der Code muss an neue Gegebenheiten angepasst werden, was dann Änderungen an sehr vielen Stellen in einem oder vielen Programmen nach sich zieht.

Der Vererbungs- und Ableitungsmechanismus für Klassen ist eine Möglichkeit, Strukturen und Funktionen wiederzuverwenden in Strukturen, die sich nur leicht voneinander unterscheiden, insbesondere bei Strukturen, die in einer "Ist-ein" Beziehung oder Erweiterungs-Beziehung stehen.

Ein Angestellter *ist eine* Person. Eine Dampflokomotive *ist eine* Lokomotive. Eine Lokomotive *ist ein* Fahrzeug. Ein Girokonto *ist ein* Konto. Ein Quadrat *ist ein* Rechteck. Das letzte Beispiel ist problematisch, da ein Quadrat ein Rechteck nicht immer substituieren kann. Denn wie soll eine Methode, die nur die Länge des Rechtecks ändert, auf ein Quadrat angewendet werden?

Der Vererbungsmechanismus in Java dient folgenden komplementären Zielen:

- Erweiterung oder Spezialisierung: Erzeugung neuer Typen durch Anfügen von zusätzlichen Eigenschaften an alte Typen (Angestellter – Person)
- Reduktion auf Gemeinsamkeiten oder Generalisierung: Definition einer gemeinsamen Schnittstelle für verschiedene Typen

6.1 Syntax des Vererbungsmechanismus

Die Syntax für den Vererbungsmechanismus ist einfach. `A` sei eine Klasse, von der die Klasse `B` erben soll. Dann sieht die Definition von `B` folgendermaßen aus:

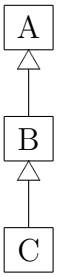
```
public class B extends A
   {
   // ---
   }
```

`A` heißt **Basisklasse**, **Obertyp** oder **Superklasse**, von der `B` *abgeleitet* wird oder von der `B` *erbt*. `B` heißt **Untertyp** oder **Subklasse**. Ein Objekt von `B` enthält ein Objekt von `A` als *Subobjekt*. Graphisch wird dies dargestellt durch:

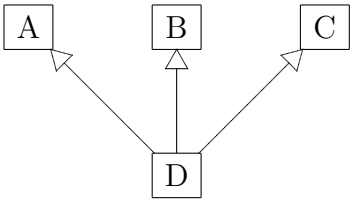
`B` kann wiederum Basisklasse sein:

```
class C extends B { //---}; ,
```

so dass *Klassenhierarchien* erstellt werden können.



Wenn sich eine Klasse von mehreren Klassen ableitet, dann spricht man von **Mehrfachvererbung *(multiple inheritance)***:



Mehrfachvererbung ist in Java **nicht** möglich. Aber Mehrfachvererbung ist ein problematisches Konzept. Wir werden sehen wie man unter Verwendung von **Schnittstellen** die unproblematischen Anwendungen von Mehrfachvererbung in Java implementieren kann.

Die Sichtbarkeit von Klassenelementen bei Vererbung wurde schon im Kapitel "Datentypen" im Abschnitt "Zugriffsregeln" besprochen. Wenn die Zugriffsregeln es erlauben, stehen der abgeleiteten Klasse alle Elemente der Basisklasse zur Verfügung, so als ob diese Elemente in der abgeleiteten Klasse selbst definiert wären.

Von einer `final` deklarierten Klasse können keine Klassen abgeleitet werden. Ein Klasse kann aus Performanzgründen `final` deklariert werden, weil dann der Compiler *dynamic method lookup* vermeiden kann. Sicherheit kann ein weiterer Grund sein, um andere Programmierer zu hindern, sicheren oder zuverlässigen Code zu überschreiben. Insbesondere ist die Klasse `String` und sind die Standardtyp-Klassen `Integer`, `Double` usw. aus Performanzgründen `final` deklariert.

In Java hat jede Klasse – ausgenommen die Klasse `Object` – automatisch mindestens eine Superklasse, nämlich die Klasse `Object`, wenn sie von keiner anderen Superklasse abgeleitet wurde. Jede Klassenhierarchie beginnt in Java mit der Klasse `Object`.

Die Methoden, die in `Object` definiert sind, können von jedem Java-Objekt aufgerufen werden, z.B.

```
public final Class   getClass ();
public String  toString ();
public boolean equals (Object obj);
protected native Object clone ()
   throws CloneNotSupportedException;
protected void finalize () throws Throwable;
```

6.2 Polymorphismus

Die wichtigste Eigenschaft des Ableitungsmechanismus ist, dass jedes Objekt einer abgeleiteten Klasse auch ein Objekt der Basisklasse ist. Folgender Code ist daher möglich:

```
class B extends A {}
// ---

B  b = new B ();
A  a = b;
```

`a` ist nun dasselbe Objekt wie `b` mit der Einschränkung, dass über die Variable `a` nur die in `A` definierten Elemente angesprochen werden können. Wenn aber eine Methode von `A` in `B` überschrieben wurde, dann wird nicht die Methode von `A`, sondern die Methode von `B` verwendet. Das bedeutet, dass sich eine Variable vom Typ `A` anders als normale `A`-Objekte verhält, wenn ihr ein Objekt einer abgeleiteten Klasse zugewiesen wurde. Dieses Verhalten heißt **Polymorphismus *(polymorphisme)*** und ist die Grundlage objekt-orientierten Programmierens.

Folgendes Beispiel soll das demonstrieren:

```
public class   Polymorphismus
```

```
   {
   public static void   main (String[] argument)
      {
      A  a = new A ();
      B  b = new B ();
      a.f ("a.f ()");
      a = b;
      a.f ("a.f ()");
      // a.g ("a.g ()");   // Fehler
      b.g ("b.g ()");
      }
   }

class A
   {
   void  f (String s)
      {
      System.out.println (s + ": f kommt von A.");
      }
   }

class B extends A
   {
   void  f (String s)
      {
      System.out.println (s + ": f kommt von B.");
      }

   void  g (String s)
      {
      System.out.println (s + ": g kommt von B.");
      }
   }
```

Voraussetzung ist, dass die überschriebene Methode im Rückgabetyp, Namen und in der Argumentenliste mit der Methode der Basisklasse übereinstimmt. Methoden, die gleichen Namen haben, aber unterschiedliche Argumentenliste sind *überladen*. (Auch beim Überladen müssen die Rückgabetypen übereinstimmen.) Die Rückgabetypen können in einer Vererbungshierarchie stehen, d.h. der Rückgabetyp kann in einer abgeleiteten Klasse ein Subtyp des Rückgabetyps in der Superklasse sein ***covariant return types***.

Eine `final`-deklarierte Methode kann in einer abgeleiteten Klasse nicht überschrieben werden.

Alle `static` und alle `private` Methoden – wie auch alle Methoden einer `final` Klasse – sind implizit `final`.

Da der Compiler nicht wissen kann, welches Objekt einer Variablen vom Typ einer Basisklasse

zugewiesen wird, muss zur Laufzeit dynamisch festgestellt werden, welche Methode zu verwenden ist ***(dynamic method lookup)***.

Konzeptionell kann man sich das so vorstellen, dass ein Referenz-Datentyp einen **statischen** und einen **dynamischen** Typ hat. Der statische Typ wird bei der Deklaration oder durch einen Cast festgelegt. Der statische Typ bestimmt, auf welche Datenelemente zugegriffen werden kann. Der dynamische Typ wird durch das Objekt bestimmt, auf den ein Referenz-Datentyp zeigt. Der Polymorphismus sorgt dafür, dass die Methoden dieses Objekts angezogen werden.

Bemerkung: Die überschreibende Methode kann keine geringere Sichtbarkeit haben als die überschriebene Methode. Sie kann aber eine höhere Sichtbarkeit haben. Allerdings verhält sie sich nur dann polymorph, wenn die überschriebene Methode nicht `private` deklariert war.

6.3 Zugriff auf verdeckte Variable und Methoden

Wenn in einer abgeleiteten Klasse Elemente mit demselben Namen redeklariert werden, dann sind die Elemente der Basisklasse verdeckt. Ein Zugriff darauf ist möglich entweder durch einen Cast auf die Basisklasse (bei Variablen und nicht überschriebenen Methoden) oder mit `super.` . Als Beispiel betrachten wir folgende Vererbungs-Hierarchie:

```
class A { int z = 1; void f () {}}
class B extends A { int z = 2; void f () {}}
class C extends B
   {
   int   z = 3;
   void  f () {}

   public void zugriff ()
      {
      z = 5;               // z von C
      f ();                // f von C
      this.z = 5;          // z von C
      this.f ();           // f von C
      ((B) this).z = 5;    // z von B
      super.z = 5;         // z von B
      ((B) this).f ();     // f von C
      super.f ();          // f von B
      ((A) this).z = 5;    // z von A
      super.super.z = 5;   // Fehler
      ((A) this).f ();     // f von C
      super.super.f ();    // Fehler
      }
   }
```

6.4 Konstruktoren

Die Konstruktoren der Basisklassen werden automatisch aufgerufen, wobei die Reihenfolge in der Vererbungshierarchie von oben nach unten geht. Wenn Argumente an einen Konstruktor einer Basisklasse zu übergeben sind, dann muss der Konstruktor der Basisklasse mit `super (`*Argumente*`)` als *erste* Anweisung im Konstruktor der abgeleiteten Klasse explizit aufgerufen werden. Die Syntax lautet z.B.

```
class A
   {
   private int a;
   public   A (int k) { a = k; }
   }


class B extends A
   {
   public   B ()
      {
      super (17);
      // ---
      }
   }
```

`super` darf nur in Konstruktoren verwendet werden. Wenn `super` nicht explizit aufgerufen wird und auch nicht mit `this ()` ein anderer Konstruktor derselben Klasse aufgerufen wird, dann generiert der Compiler einen Aufruf des Defaultkonstruktors der Basisklasse `super ();`. Wenn die Basisklasse keinen Defaultkonstruktor hat, dann gibt das einen Übersetzungsfehler. `this ()` und `super ()` können nicht gleichzeitig in einem Konstruktor aufgerufen werden.

Wenn eine Klasse keinen Konstruktor definiert, dann generiert der Compiler einen Defaultkonstruktor, der nur die Anweisung `super ();` enthält.

Die Konstruktoren werden in einer Vererbungshierachie automatisch so verkettet, dass die Objekte ganz oben in der Hierarchie zuerst initialisiert werden. Bei den `finalize`-Methoden gibt es *nicht* diese Entsprechung. D.h. der Programmierer muss selbst dafür sorgen, dass die `finalize`-Methode der Basisklasse in der eigenen `finalize`-Methode aufgerufen wird, sinnvollerweise als letztes Statement:

```
protected void finalize () throws Throwable
   {
   // ---
   super.finalize ();
   }
```

Bemerkung: Konstruktoren können nicht vererbt werden (leider!).

6.5 Initialisierungsreihenfolge

Beim Erzeugen eines Objekts mit `new` werden die verschiedenen Initialisierungen in folgender Reihenfolge durchgeführt:

1. Es wird für alle Datenelemente der ganzen Vererbungshierarchie Speicher angelegt. Die Datenelemente werden mit ihren Defaultwerten initialisiert, z.B. `boolean` mit `false`, `int` mit `0` und Referenztypen mit `null`.
2. Es werden für jede Klasse in der Vererbungshierarchie – beginnend bei der obersten Superklasse –
   (a) zuerst die Instanzinitialisatoren in der Reihenfolge ihres Auftretens durchgeführt. Eine Definition
   ```
   int k = 5;
   ```
   ist auch ein Instanzinitialisator und wird daher erst jetzt ausgeführt.
   (b) Dann wird ein Konstruktor ausgeführt, der allerdings andere Konstruktoren derselben Klasse aufrufen kann.

Wenn in den Initialisatoren oder Konstruktoren polymorph überschriebene Methoden aufgerufen werden, dann werden die dem Typ des Objekts entsprechenden überschriebenen Formen genommen (im Gegensatz zu C++). D.h., wenn ein Instanzinitialisator der Superklasse eine in der Subklasse überschriebene Methode aufruft, dann wird die überschriebene Methode verwendet. Dabei kann es sein, dass Datenelemente mit ihren Defaultwerten verwendet werden, weil ihr Initialisator noch nicht ausgeführt worden ist.

Folgendes Beispiel demonstriert das.

```
public class   ABinit
   {
   public static void   main (String[] arg)
      {
      System.out.println ("new A ():");
      new A ();
      System.out.println ();
      System.out.println ("new B ():");
      new B ();
      }
   }

class A
   {
   {
   f ("im Initialisator A vor Varzuweisung");
   }
   String   a = "Aha";
   {
   f ("im Initialisator A nach Varzuweisung");
```

```
   }
   void  f (String s)
      {
      System.out.println ("   " + s + ":  f kommt von A  a = " + a);
      }
   public   A ()
      {
      System.out.println ("   Konstruktor A");
      f ("im Konstruktor A");
      }
   }

class B extends A
   {
   {
   f ("im Initialisator B vor Varzuweisung");
   }
   String   b = "Bha";
   {
   f ("im Initialisator B nach Varzuweisung");
   }
   void  f (String s)
      {
      System.out.println ("   " + s + ":  f kommt von B a = "
         + a + "  b = " + b);
      }
   public   B ()
      {
      super ();
      System.out.println ("   Konstruktor B");
      f ("im Konstruktor B");
      }
   }
```

Mit dem Resultat:

```
new A ():
   im Initialisator A vor Varzuweisung:  f kommt von A  a = null
   im Initialisator A nach Varzuweisung:  f kommt von A  a = Aha
   Konstruktor A
   im Konstruktor A:  f kommt von A  a = Aha

new B ():
   im Initialisator A vor Varzuweisung:  f kommt von B a = null  b = null
   im Initialisator A nach Varzuweisung:  f kommt von B a = Aha  b = null
   Konstruktor A
   im Konstruktor A:  f kommt von B a = Aha  b = null
```

```
im Initialisator B vor Varzuweisung:  f kommt von B a = Aha  b = null
im Initialisator B nach Varzuweisung:  f kommt von B a = Aha  b = Bha
Konstruktor B
im Konstruktor B:  f kommt von B a = Aha  b = Bha
```

6.6 Abstrakte Klassen und Schnittstellen

6.6.1 Abstrakte Klassen

In einer Übung zu diesem Kapitel wollen wir die Klasse `Konto` zu `Girokonto`, `Sparkonto` und `Supersparkonto` erweitern. Dann wird es so sein, dass es für die Klasse `Konto` selbst eigentlich keine vernünftigen Objekte gibt. Man kann halt nicht ein "allgemeines Konto" anlegen.

Java bietet die Möglichkeit, solche allgemeinen Klassen, die zwar die Methoden anbieten, um Objekte zu manipulieren, für die es aber keine sinnvollen Objekte gibt, als **abstrakte *(abstract)*** Klassen zu definieren.

Syntaxregeln:

- Durch den Modifikator `abstract` wird eine Klasse abstrakt.

  ```
  public abstract class AbtrakteKlasse {}
  ```

  Von einer abstrakten Klasse können keine Objekte angelegt werden.

- Durch den Modifikator `abstract` wird eine Methode abstrakt. Eine abtrakte Methode hat anstatt einer Implementation ein Semikolon.

  ```
  public abstract void methode ();
  ```

  Eine Klasse mit einer abtrakten Methode ist automatisch abstrakt und muss auch so deklariert werden.

- Von einer Subklasse einer abstrakten Klasse können Objekte angelegt werden, wenn die Subklasse *alle* abstrakten Methoden der abstrakten Basisklasse implementiert. Sonst ist sie selbst abstrakt und muss so deklariert werden.

- Der Modifikator `abstract` kann nicht mit den Modifikatoren `private`, `final`, `native` oder `synchronized` kombiniert werden.

Das Beispiel `Rechteck` aus den Übungen der Einleitung könnte man so modifizieren, dass das Rechteck von einer abstrakten Klasse `ZweiDimGebilde` erbt, die die abstrakten Methoden `flaeche ()` und `umfang ()` zur Verfügung stellt. Wenn man nun fordert, dass `Rechteck`, `Kreis` usw. von dieser Klasse erben, dann erzwingt man die Implementation dieser beiden Methoden. Man kann sich also darauf verlassen, dass diese beiden Methoden bei jedem zweidimensionalen Gebilde zur Verfügung stehen.

```
public abstract class   ZweiDimGebilde
```

```
   {
   public String  name;

   public   ZweiDimGebilde (String name)
      {
      this.name = name;
      }

   public abstract double  flaeche ();
   public abstract double  umfang ();

   public static void   main (String[] argument)
      {
      ZweiDimGebilde[]  zdg = new ZweiDimGebilde[2];
      zdg[0] = new Kreis ("Rundes", 12.0);
      zdg[1] = new Rechteck ("Eckiges", 17.0, 9.5);
      for (int i = 0; i < zdg.length; i++)
         {
         System.out.print (zdg[i].name + " : ");
         System.out.print ("Fläche = ");
         System.out.print (zdg[i].flaeche ());
         System.out.print ("   Umfang = ");
         System.out.println (zdg[i].umfang ());
         }
      }

   }

class Kreis extends ZweiDimGebilde
   {
   private double radius;

   public   Kreis (String name, double radius)
      {
      super (name);
      this.radius = radius;
      }

   public double  flaeche ()
      {
      return java.lang.Math.PI * radius * radius;
      }

   public double  umfang ()
      {
      return 2 * java.lang.Math.PI * radius;
      }
   }
```

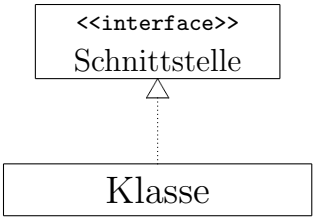
```
class Rechteck extends ZweiDimGebilde
   {
   private double laenge;
   private double breite;

   public   Rechteck (String name, double x, double y)
      {
      super (name);
      if (x > y) { laenge = x; breite = y; }
      else { laenge = y; breite = x; }
      }

   public double  flaeche ()
      {
      return laenge * breite;
      }

   public double  umfang ()
      {
      return 2 * laenge + 2 * breite;
      }
   }
```

Zu bemerken ist, dass Referenzen vom Typ einer abstrakten Klasse Objekte der Subklassen zugewiesen werden können. D.h. Methoden abstrakter Klassen können *polymorph* verwendet werden.

6.6.2 Schnittstellen

Wir wollen das oben genannte Beispiel erweitern. Die Darstellung der zweidimensionalen Gebilde hatten wir im Hauptprogramm durch eine vierzeilige Botschaft geleistet. Jetzt wollen wir spezielle Klassen `DarstRechteck` und `DarstKreis` definieren, die für ein Rechteck bzw einen Kreis eine Darstellungsmethode `zeige ()` anbieten. Dazu sollten diese Klassen von einer abstrakten Klasse `DarstGebilde` erben, die die Methode `zeige ()` anbietet. Da sie außerdem von `Rechteck` bzw von `Kreis` erben sollten, läge hier ein Fall von Mehrfachvererbung vor, der in Java so nicht zu realisieren ist.

In Java haben wir aber die Möglichkeit `DarstGebilde` als **Schnittstelle** ***(interface)*** zu definieren:




```
public interface  DarstGebilde
   {
   void  zeige ();
   }
```

Eine Schnittstelle kann wie eine Klasse `public` oder ***default*** deklariert werden. Eine Schnittstelle ist implizit `abstract`. Alle Methoden einer Schnittstelle sind implizit `abstract` *und* `public`. (Man darf aber auch die Modifikatoren `abstract` und `public` verwenden.) Andere Sichtbarkeitsstufen als `public` sind in Schnittstellen *nicht* einstellbar. In einer Schnittstelle dürfen Klassen-Konstanten definiert werden, d.h. alle Variablen haben implizit die Modifikatoren `public static final`. Die Definition von `static` Methoden ist nicht erlaubt.

Die Klasse `DarstKreis` erbt nun die Klasse `Kreis` und implementiert die Schnittstelle `DarstGebilde`.

```
public class   DarstGebildeMain
   {
   public static void   main (String[] argument)
      {
      DarstGebilde[] zdg = new DarstGebilde[2];
      zdg[0] = new DarstKreis ("Rundes", 12.0);
      zdg[1] = new DarstRechteck ("Eckiges", 17.0, 9.5);
      for (int i = 0; i < zdg.length; i++)
         {
         zdg[i].zeige ();
         }
      }
   }

interface   DarstGebilde
   {
   void  zeige ();
   }

class DarstKreis extends Kreis implements DarstGebilde
   {
   public   DarstKreis  (String name, double radius)
      {
      super (name, radius);
      }
```

```
   public void zeige ()
      {
      System.out.print ("Kreis " + name + " : ");
      System.out.print ("Fläche = ");
      System.out.print (flaeche ());
      System.out.print ("   Umfang = ");
      System.out.println (umfang ());
      }
   }


class DarstRechteck extends Rechteck implements DarstGebilde
   {
   public   DarstRechteck  (String name, double x, double y)
      {
      super (name, x, y);
      }

   public void zeige ()
      {
      System.out.print ("Rechteck " + name + " : ");
      System.out.print ("Fläche = ");
      System.out.print (flaeche ());
      System.out.print ("   Umfang = ");
      System.out.println (umfang ());
      }
   }
```

Eine Klasse, die eine Schnittstelle implementiert, muss *alle* Methoden der Schnittstelle implementieren. Jedes Objekt einer Klasse, die eine Schnittstelle implementiert, ist auch eine Instanz der Schnittstelle, d.h Schnittstellen sind auch Datentypen, obwohl sie nicht instanziiert werden können.

Eine Klasse kann beliebig viele Schnittstellen implementieren. Die zu implementierenden Schnittstellen werden – durch Komma getrennt – nach dem Schlüsselwort `implements` genannt.

Konstanten, die in einer Schnittstelle definiert wurden, können in der implementierenden Klasse und deren Subklassen ohne Referenz auf den Namen der Schnittstelle verwendet werden.

Eine Schnittstelle kann Sub-Schnittstellen haben, d.h. man kann Vererbungshierarchien bauen:

```
  public interface S1 extends S01, S02, S03
     {
     // eigene Methoden
     }
```

Hier ist Mehrfachvererbung möglich! Eine Klasse, die solch eine Schnittstelle implementiert, muss natürlich alle Methoden von `S1`, `S01`, `S02` und `S03` implementieren.

Methoden vom Typ einer Schnittstelle können polymorph verwendet werden.

Eine sogenannte **Marker**-Schnittstelle ist eine leere Schnittstelle, die, wenn sie eine Klasse implementiert, anzeigt, dass die Klasse etwas kann oder erlaubt ist, etwas zu tun. Beispiele sind die Schnittstellen `Cloneable` und `Serializable`.

6.7 Klasse `java.lang.Class`

Für jede Java-Klasse und Java-Schnittstelle wird ein Objekt der Klasse `Class` verwaltet. Dieses Objekt enthält Informationen über den Namen der Klasse, die Superklasse, die implementierten Schnittstellen und den Klassenlader.

Es gibt insbesondere die Methoden `forName` und `newInstance`, mit denen das `Class`-Objekt zu einem Klassennamen gefunden werden kann, mit dem dann Objekte dieser Klasse angelegt werden können:

```
Class c  = Class.forName ("Klausurnote");
Klausurnote k = (Klausurnote)c.newInstance ();
```

Damit ist es möglich, Klassen zu schreiben, die ähnlich wie Templates in C++ verwendet werden können.

Anstatt der Methode `forName` kann auch folgende Syntax verwendet werden:

```
Class c = Klausurnote.class;
```

Außerdem ist über ein Objekt der Klasse das Klassenobjekt zugänglich:

```
Klausurnote k = new Klausurnote ();
Class c = k.getClass ();
```

Für die Standardtypen und Felder gibt es auch Klassenobjekte:

```
Class ic = Integer.TYPE; // oder: = int.class
Class c = int[].class;
oder
int[] v = new int[10];
Class c = v.getClass ();
```

Über das Klassenobjekt ist auch das Superklassenobjekt zugänglich:

```
Class c  = Class.forName ("Klausurnote");
Class sc = c.getSuperclass ();
```

Im folgenden Beispiel kann man als Argument den Namen einer Klasse übergeben, die die Schnittstelle S implementiert. Es wird dann ein Objekt dieser Klasse angelegt und eine Methode der Klasse aufgerufen.

```
public class   ClassBsp
   {
   public static void   main (String[] argument)
      {
      try
         {
         Class c = Class.forName (argument[0]);
         S  s = (S) c.newInstance ();
         s.zeige ();
         }
      catch (ClassNotFoundException e)
         {
         System.err.println ("ClassBsp.main: " + e);
         }
      catch (InstantiationException e)
         {
         System.err.println ("ClassBsp.main: " + e);
         }
      catch (IllegalAccessException e)
         {
         System.err.println ("ClassBsp.main: " + e);
         }
      }
   }

interface   S
   {
   void  zeige ();
   }

class A implements S
   {
   public void zeige ()
      {
      System.out.println ("Es grüßt Sie A!");
      }
   }

class B implements S
   {
```

```
   public void zeige ()
      {
      System.out.println ("Es grüßt Sie B ganz herzlich!");
      }
   }
```

6.8 Implementation von `clone`

Die Methode

```
protected native Object clone ()
```

der Klasse `Object` macht nur eine "flache" Kopie des Objekts. Das bedeutet, dass nur die Standardtypen kopiert werden, Referenztypen aber nicht. Wenn eine "tiefe" Kopie benötigt wird, muss der Programmierer dafür sorgen, dass die Referenztypen kopiert werden. Allgemein sollte `clone` folgendermaßen für eine Klasse `A`, die eventuell von anderen Klassen (insbesondere natürlich von `Object`) erbt, implementiert werden:

```
public class   A implements Cloneable
   {
   private int intDatenElement;
   private String stringDatenElement;
   private B   cloneableDatenElement;

   public Object  clone ()
      {
      A  kopie = null;
      try
         {
         kopie = (A) super.clone ();
            // flache Kopie
         kopie.stringDatenElement = new String (stringDatenElement);
            // tiefe Kopie
         kopie.cloneableDatenElement = (B) cloneableDatenElement.clone ();
            // tiefe Kopie
         }
         catch (CloneNotSupportedException e)
            {
            e.printStackTrace ();
            }
      return kopie;
      }
   }
```

`Cloneable` muss implementiert werden, damit die `clone`-Methode von `Object` aufgerufen werden kann. In der Klasse `Vector` ist `clone` definiert, wobei die Objekte nicht geklont werden. Bei Feldern muss man das Klonen von Hand machen.

6.9 Implementation von objektorientiertem Design in Java

In diesem Abschnitt zeigen wir, wie das Resultat einer objektorientierten Analyse bzw Design in Java implementiert werden kann.

6.9.1 "Ist-ein" – Beziehung

Die "Ist-ein"-Beziehung (`is-a`) oder Erweiterungs-Beziehung wird implementiert durch Vererbung.

```
class B extends A
   {
   //---
   }
```

`B` ist ein `A`. Alles, was für `A` zutrifft, trifft auch für `B` zu. Die Objekte von `B` sind eine Teilmenge der Objekte von `A`. Objekte von `B` können Objekte von `A` voll ersetzen *(Substitutionsprinzip von Liskov)*.

`private` Methoden von `A` sind Methoden, die *nicht* geerbt werden.

`final` Methoden von `A` sind Methoden, deren Schnittstelle *und* deren Implementation geerbt werden. Sie können nicht überschrieben werden.

`abstract` Methoden von `A` sind Methoden, deren Schnittstelle nur geerbt wird. Sie müssen implementiert werden.

Bei allen anderen Funktionen wird die Schnittstelle geerbt. Außerdem können sie neu implementiert (überschrieben) werden.

6.9.2 "Ist-fast-ein" – Beziehung

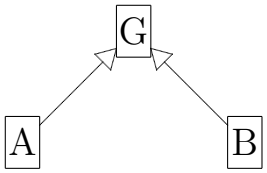
Die Beziehung "Ist-fast-ein" (`is-like-a`)

```
B is-like-an A
```

ist dadurch charakterisiert, dass die Teilmengen-Beziehung der "Ist-ein"-Beziehung insofern nicht gilt, als die Objekte der links stehende Klasse `B` *nicht* vollständig Objekte der Klasse `A` subtituieren können. Die Klasse `B` ist häufig eingeschränkt (Quadrat ist ein eingeschränktes Rechteck), bietet eventuell auch zusätzliche Methoden an.

Bei dieser Beziehung sollte man die Gemeinsamkeiten von `A` und `B` in einer abstrakten Klasse, etwa `G`, zusammenfassen, von der `A` und `B` erben.

```
abstract class G
   {
   // abstrakte und implementierte Methoden
   // für Gemeinsamkeiten von A und B,
   // Definition gemeinsamer Daten von A und B
   }

class A extends G
   {
   // implementiert abstrakte Methoden von G,
   // fügt eigenes Verhalten und Daten hinzu
   }

class B extends G
   {
   // implementiert abstrakte Methoden von G,
   // fügt eigenes Verhalten und Daten hinzu
   }
```



6.9.3 "Hat-ein" – Beziehung

Die "Hat-ein"-Beziehung (`B` hat ein `A`) wird auch als "Ist-implementiert-mit"-Beziehung, ***Layering, Containment, Embedding,*** **Einbettung** bezeichnet (`has-a`). Ein Objekt vom Typ `A` ist Komponente von einem Objekt vom Typ `B`. `B` heißt auch Komponentengruppe.

Die "Hat-ein" Beziehung wird implementiert, indem ein Objekt oder ein Feld von Objekten von `A` ein Datenelement von `B` wird.

```
class A {---}

class B
   {
   private A   a;
   private A[] a = new A[10];
   }
```

Da die Komponenten (`A`) einer Komponentengruppe (`B`) ganz zu `B` gehören, müssen sie bei einer Kopie eines Objekts vom Typ `B` auch vollständig kopiert werden. Die Methode `clone` muss daher so implementiert werden, dass bezüglich der Komponenten der "Hat-ein"-Beziehung eine "tiefe" Kopie (Verwendung von `clone ()` oder Erzeugung neuer Objekte mit `new`) gemacht wird.

6.9.4 "Benutzt-ein" – Beziehung

Wenn Objekte einer Klasse `B` die Dienste von Objekten einer anderen Klasse `A` benutzen und die Objekte von `A` nicht als Teile betrachtet werden können, spricht man von einer "Benutzt-ein" – Beziehung (`uses-a`).

Implementieren kann man dies, indem man ein Objekt der Klasse `A` als Datenelement in der Klasse `B` führt.

```
class A {---}

class B
   {
   private A   a;
   }
```

Es ist auch möglich, dass eine Methode von `B` sich ein Objekt der Klasse `A` kreiert und dann dessen Methoden verwendet.

Diese Beziehung ist in Java nur bezüglich des Kopierens `clone` unterschiedlich zur "Hat-ein"-Beziehung zu behandeln. Die Methode `clone` muss so implementiert werden, dass bezüglich der Komponenten der "Benutzt-ein"-Beziehung eine "flache" Kopie (Verwendung des Zuweisungsoperators "`=`" bei den "benutzten" Objekten) gemacht wird.

6.9.5 Andere Beziehungen

Beziehungen, die nicht in eine der oben genannten Kategorien passen, werden in den meisten Fällen – insbesondere, wenn die Beziehung Eigenschaften und Verhalten hat, – durch Definition einer eigenen Klasse implementiert. Diese Klasse enthält dann die Partner der Beziehung als Datenelemente.

Zusätzlich können – aus Effizienzgründen – Objekte oder Felder von Objekten der Beziehung in den Partnerklassen verwaltet werden.

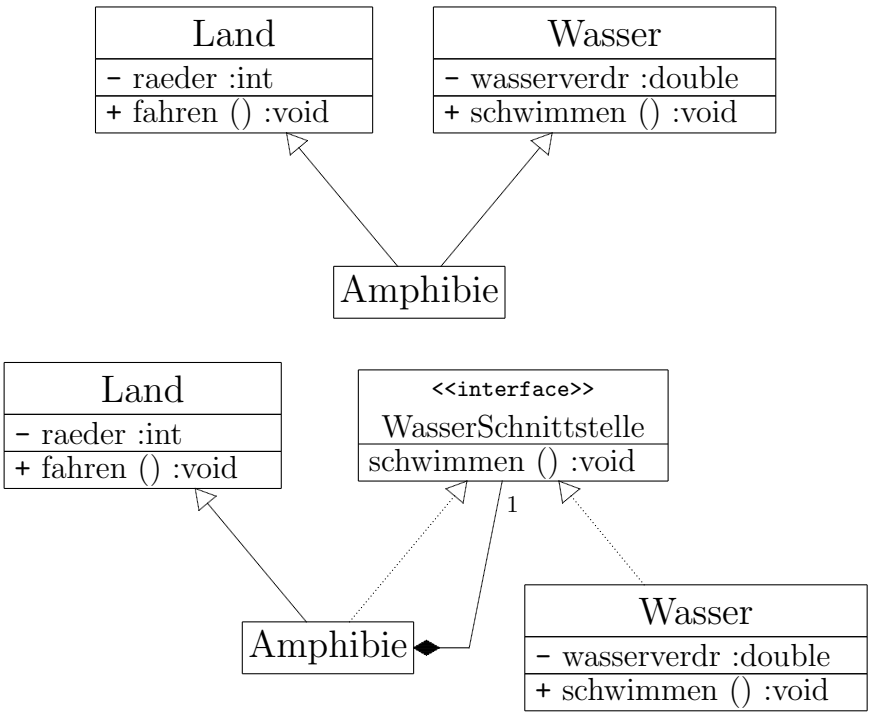
6.9.6 Botschaften

In der Objektorientierung spielen Botschaften *messages* an Objekte von anderen Objekten eine große Rolle. In Java wird das Senden einer Botschaft an ein anderes Objekt dadurch implementiert, dass eine Methode des die Botschaft empfangenden Objekts aufgerufen wird.

6.9.7 Mehrfachvererbung

Java kennt keine Mehrfachvererbung. Sie kann nur so implementiert werden, dass höchstens eine der Basisklassen geerbt wird. Für die anderen Basisklassen müssen Schnittstellen definiert werden. Die Mehrfacherbin hat von diesen nicht geerbten Basisklassen je ein Datenelement und implementiert mit diesen Datenelementen die Schnittstellen, wobei zu empfehlen ist, dass die Basisklassen die entsprechenden Schnittstellen auch implementieren.

Beispiel:




```
public class   Mehrfachvererbung
   {
   public static void   main (String[] argument)
      {
      Amphibie a = new Amphibie (4, 1.5);
      a.fahren ();
      a.schwimmen ();
      }
   }

class Land
   {
   private int raeder;

   public   Land (int raeder)
```

```
      {
      this.raeder = raeder;
      }

   public void fahren ()
      {
      System.out.print ("Landfahrzeug: Ich fahre mit ");
      System.out.print (raeder);
      System.out.println (" Rädern.");
      }
   }

interface   WasserSchnittstelle
   {
   void schwimmen ();
   }

class Wasser implements WasserSchnittstelle
   {
   private double wasserverdr;

   public   Wasser (double wasserverdr)
      {
      this.wasserverdr = wasserverdr;
      }

   public void schwimmen ()
      {
      System.out.print ("Wasserfahrzeug: Ich schwimme mit ");
      System.out.print (wasserverdr);
      System.out.println (" cbm Wasserverdrängung.");
      }
   }

class Amphibie extends Land implements WasserSchnittstelle
   {
   private WasserSchnittstelle   wasser;

   public   Amphibie (int raeder, double wasserverdr)
      {
      super (raeder);
      wasser = new Wasser (wasserverdr);
      }

   public void schwimmen ()
      {
      wasser.schwimmen ();
      }
   }
```

Bemerkung: Diese Art der Anwendung von Mehrfachvererbung wird auch in C++ empfohlen, wo man die Schnittstellen als abstrakte Klassen ohne Datenelemente realisieren kann.

6.10 Übungen

Übung Klausurnote: Überschreiben Sie die Methode `toString ()` der Klasse `Object` für die Klasse Klausurnote und wenden Sie sie an, indem sie eine Klausurnote einfach mit `println` ausgeben. (Die Methode sollte ungefähr das machen, was die Methode `drucke` tut.

Übung Konten: Programmieren Sie die Vererbungshierarchie Konto, Girokonto, Sparkonto, Supersparkonto.

Übung ClassBsp: Ergänzen Sie in dem Beispiel `ClassBsp` eine Klasse "`C`".

Übung Rechteck: Arbeiten Sie die "Ist-fast-ein"-Beziehung Rechteck – Quadrat aus.

Übung Reihenfolge der Initialisierung: Was ist das Resultat der Ausführung des folgenden Programms:

```
class A
   {
   int   a = f ();
   int   f () { return 1; }
   }

class B extends A
   {
   int   b = f ();
   int   f () { return 2; }
   }

public class   ABinit2
   {
   public static void   main (String[] arg)
      {
      A  aA = new A ();
      System.out.println ("aA.a = " + aA.a);
      B  aB = new B ();
      System.out.println ("aB.a = " + aB.a);
      System.out.println ("aB.b = " + aB.b);
      }
   }
```

Was ist das Resultat der Ausführung des folgenden Programms:

```
class A
   {
   int   a = f ();
   int   f () { return 1; }
   }

class B extends A
   {
   int   b = 47;
   int   f () { return b; }
   }

public class   ABinit3
   {
   public static void   main (String[] arg)
      {
      A  aA = new A ();
      System.out.println ("aA.a = " + aA.a);
      B  aB = new B ();
      System.out.println ("aB.a = " + aB.a);
      System.out.println ("aB.b = " + aB.b);
      }
   }
```

Kapitel 7

Ausnahmebehandlung

Eine **Ausnahme** ***(exception)*** ist ein Signal, das irgendeine besondere Situation anzeigt, wie z.B. dass ein Fehler eingetreten ist. Ein Signal **Werfen** ***(throw)*** bedeutet, das Signal zu geben. Ein Signal **Fangen** ***(catch)*** bedeutet, die entsprechende Ausnahmesituation zu **behandeln** ***(handle)***.

Dabei spielen die Schlüsselwörter `try`, `throw`, `throws`, `catch` und `finally` eine Rolle.

7.1 Syntax

Die Syntax ist folgendermaßen:

```
// ---
try
   {
   // ---
   throw objekt;
   // ---
   }
catch (Typ_des_Objekts e)
   {
   // ---
   }
catch (Typ_von_anderen_Objekten e)
   {
   // ---
   }
finally
   {
   // ---
   }
```

Im Code des `try`-Blocks können Exceptions (`throw`) und abnormale Abbrüche (`break`, `continue`, `return`) auftreten. Der Code des `try`-Blockes wird an den genannten Stellen abgebrochen.

Nach einem `try`-Block können keine oder mehrere `catch`-Blöcke auftreten, die die verschiedenen Exceptions behandeln. Die `catch`-Klausel wird mit einem Argument deklariert. Dieses Argument muss vom Typ `Throwable` oder einer Subklasse davon sein. Wenn eine Exception geworfen wird, dann wird die erste `catch`-Klausel aufgerufen, deren Argumenttyp vom Typ des geworfenen Objekts oder ein Obertyp (Superklasse) davon ist. Das Argument ist nur im `catch`-Block gültig und referenziert das geworfene Objekt.

Gibt es keinen passenden `catch`-Block, dann wird das Objekt weitergeworfen in der Hoffnung, das ein übergeordneter `try-catch`-Block ein `catch` dafür anbietet. Wenn das geworfene Objekt nie gefangen wird, dann propagiert das Objekt schließlich bis zur `main`-Methode und veranlasst den Interpreter zum Drucken einer Fehlermeldung und Abbruch.

Die `finally`-Klausel wird zum Aufräumen verwendet (Schließen von Dateien, Freigeben von Resourcen). Ihr Code wird auf jeden Fall durchgeführt, wenn der `try`-Block betreten wurde, aber erst nach einem eventuellen `catch`-Block.

Der Code der `finally`-Klausel wird abgearbeitet

1. normalerweise nach Abarbeitung des `try`-Blocks.
2. nach Werfen einer Exception und Abarbeitung des `catch`-Blocks.
3. nach Werfen einer Exception, für die es hier kein `catch` gibt.
4. nach einem `break`, `continue` oder `return` im `try`-Block.

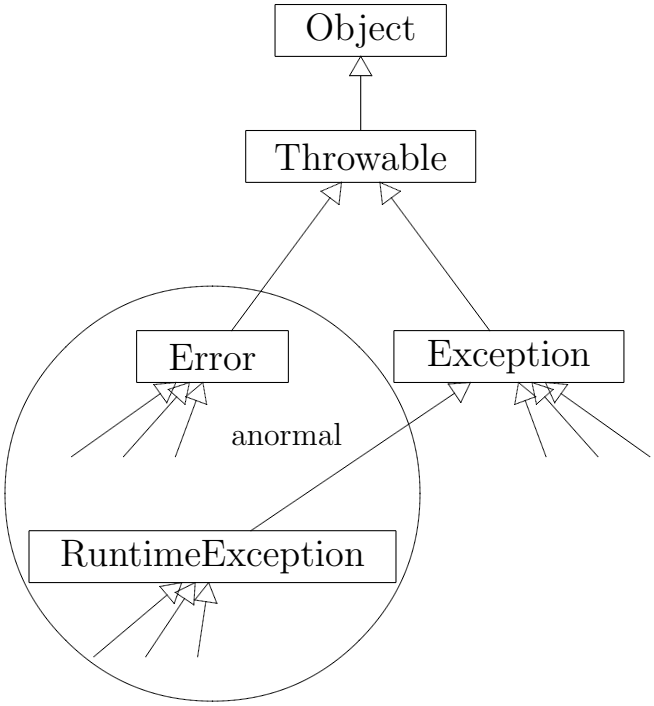
Wenn der `finally`-Block selbst ein `throw`, `break`, `continue` oder `return` macht, dann wird die laufende Abbruchs-Aktion abgebrochen und diese neue Abbruchs-Aktion weiterverfolgt.

7.2 Ausnahme-Objekte

Klassen von Ausnahme-Objekten müssen von `java.lang.Throwable` erben. Es gibt zwei standardisierte Subklassen:

- `java.lang.Error` : Diese Fehler betreffen Probleme des Bindens im Zusammenhang mit dem dynamischem Laden von Code oder Probleme der virtuellen Maschine (Speicherplatz). Von diesen Fehlern kann sich ein Programm normalerweise nicht erholen. Diese Fehler sollten daher nicht gefangen werden.
- `java.lang.Exception` : Von diesen Fehlern kann sich ein Programm meistens erholen. Sie sollten daher gefangen und behandelt werden (z.B. `java.io.EOFException` oder `java.lang.ArrayAccessOutOfBounds`). Diese Klassen enthalten meistens eine Diagnose-Methode und Status-Methoden oder -Variablen, die im `catch`-Block verwendet werden können.

"Nicht-normale" Exceptions sind vom Typ `java.lang.Error` und `java.lang.Exception.RuntimeException` und ihren jeweiligen Subklassen. Diese Exceptions müssen nicht behandelt oder deklariert werden (siehe unten).



7.3 Deklaration von Exceptions

Jede Methode muss alle "normalen" ***(normal, checked)*** Exceptions, die eventuell von ihr geworfen werden,

- entweder in einem `try-catch`-Konstrukt behandeln
- oder deklarieren.

Die Deklaration erfolgt mit der Klausel `throws` *Exceptiontyp*, die zwischen die Argumentenliste und die Implementation plaziert wird:

```
public void leseMethode () throws IOException, MeineException
   {
   // --- es gibt kein catch für IOException
   // --- es gibt kein catch für MeineException
   }
```

Es genügt, wenn die spezifizierte Exception-Klasse eine Superklasse der Klasse des eigentlich geworfenen Objekts ist.

Da fast jede Methode "nicht-normale" Exceptions werfen kann, wäre es zu mühsam, diese immer spezifizieren oder behandeln zu müssen, da sie ohnehin normalerweise nicht behandelt werden können. Daher muss man sie nicht deklarieren.

Im Zweifelsfall wird der Compiler sagen, welche Exceptions zu deklarieren oder zu behandeln sind.

Man kann Fehler oder Ausnahmen gruppieren, indem man die Fehler bzw Ausnahmen von einem allgemeinen Fehler- oder Ausnahmetyp ableitet.

Ein `catch (Throwable e) {}` fängt alle Fehler auf.

Wenn man eigene Fehlerklassen von `Error` ableitet, dann muss man diese Fehler nicht deklarieren und auch nicht fangen.

Wenn eine Methode in der abgeleiteten Klasse überschrieben wird, dann kann die überschreibende Methode nicht mehr Exceptions werfen als die überschriebene Methode. Sie kann aber weniger Exceptions werfen. Diese Regel betrifft nur die normalen Exceptions. Für nicht-normale Exceptions gibt es keine Einschränkung.

Wenn man eine Exception einer `throws`-Klausel hinzufügt, dann ändert sich auch nicht die Signatur der Methode.

D.h. weder durch Überschreiben noch durch Überladen kann man das nachträgliche Ändern der `throws`-Klausel maskieren. Da gibt nur die Möglichkeit, entweder einen neuen Namen für die Methode zu finden oder die Exception zur nicht-normalen Exception zu machen. Beides ist aber unbefriedigend. Deswegen neigen Designer dazu, eine Exception-Superklasse für ein ganzes Paket zu definieren. Die öffentlichen Methoden deklarieren dann nur die Superklasse. Es können aber Subklassen-Objekte geworfen werden.

7.4 Basisklasse `Throwable`

Alle werfbaren Objekte müssen sich von der Klasse `Throwable` ableiten, die unter anderem folgende interessante Methoden anbietet:

- `printStackTrace ()` und `printStackTrace (PrintWriter s)` geben einen Stack-Trace aus (nach `System.err` bzw `s`), der zeigt, wo der Fehler aufgetreten ist.
- `fillInStackTrace ()` erweitert den Stack-Trace, wenn eine Exception gefangen wird, teilweise behandelt wird und dann weitergeworfen wird. Beim Weiterwerfen wird man dann anstatt

  ```
  throw e;
  throw (Ausnahmetyp)(e.fillInStackTrace ());
  ```

  verwenden.

7.5 Best Practices

- Arbeite mit einer möglichst spezifischen Exception. Wenn z.B. eine Methode nur eine `FileNotFoundException` werfen kann, dann sollte man nicht nur eine `IOException` deklarieren, da man sonst den aufrufenden Kontext zwingt alle möglichen `IOException`s zu behandeln. Allerdings sollte man nicht zu spezifisch werden, insbesondere wenn dadurch Implementations-Details veröffentlicht werden.

- Vermeide leere `catch`-Blöcke. Ein `e.printStackTrace ()` ist eine bequeme und auch nützliche Fehlerbehandlungsart. Wenn diese Fehlerausgabe stört, weil der `catch`-Block unter normalen Umständen (z.B. Warteschleife) immer wieder angelaufen wird, dann sollte der `catch`-Block mindestens einen Kommentar enthalten, der erklärt, warum er ignoriert wird.
- Javadoc kennt das `@throws`-Tag, das den Namen der geworfenen Exception und auch einen Kommentar enthalten sollte, damit die Information über den Inhalt der Methoden-Deklaration hinausgeht.

```
/**
*  @throws ClassNotFoundException
*     Klasse SoUndSo wurde nicht gefunden.
**/
public Class   ladeKlasse (String name)
   throws   ClassNotFoundException
   {
   // ...
   }
```

- `RuntimeException`s müssen nicht deklariert und abgefangen werden. Es ist **nicht** Best Practice, `RuntimeException`s in einem `try-catch`-Block zu behandeln. Es ist besser den Fehlerfall durch geeigneten Code abzufangen.
- Anstatt eigene Fehlerklassen zu schreiben sollte man versuchen, die vom JSDK zur Verfügung gestellten Klassen zu verwenden, da das die Code-Basis reduziert und der Benutzer des Codes diese meist schon kennt.

7.6 Beispiel

```
public class   Ausnahme
   {
   public static void   main (String[] argument) throws A1
      // mögliche Argumente: A1, A2, A3, A31, A4, A5
      // E, break, return, continue
      {
      if (argument.length > 0)
         {
         String   fehler = argument[0];
         Ausnahme a = new Ausnahme ();
         a.f (fehler);
         }
      }

   void  f (String fehler) throws A1
      {
      try
         {
         bem ("f: try Anfang");
```

```
      int   j = 1;   // normalerweise nur einmal durch while
      if (fehler.equals ("continue")) j = 0;
      while (j < 2)
         {
         bem ("f: while Anfang j: " + j);
         if (j == 0 && fehler.equals ("continue")) {j++; continue;}
         if (fehler.equals ("break")) break;
         if (fehler.equals ("return")) return;
         ff (fehler);
         bem ("f: while Ende   j: " + j);
         j++;
         }
      bem ("f: try Ende");
      }
   catch (A2 e)
      {
      bem ("f: catch (A2)");
      e.diagnose ();
      }
   catch (A4 e)
      {
      bem ("f: catch (A4)");
      e.diagnose ();
      }
   catch (E e)
      {
      bem ("f: catch (E)");
      e.diagnose ();
      }
   finally
      {
      bem ("f: finally");
      }
   bem ("f: nach finally");
   }

void  ff (String fehler) throws A1, A2, A4
   {
   try
      {
      bem ("ff: try Anfang");
      fff (fehler);
      bem ("ff: try Ende");
      }
   catch (A3 e)
      {
      bem ("ff: catch (A3)");
      e.printStackTrace ();
      e.diagnose ();
```

```
        }
     catch (A5 e)
        {
        bem ("ff: catch (A5)");
        e.printStackTrace ();
        e.diagnose ();
        }
     finally
        {
        bem ("ff: finally");
        if (fehler.equals ("A4")) throw new A4 ();
        }
     bem ("ff: nach finally");
     }

  void  fff (String fehler) throws A1, A2, A3, A4, A5
     {
     try
        {
        bem ("fff: try Anfang");
        if (fehler.equals ("A1")) throw new A1 ();
        if (fehler.equals ("A2")) throw new A2 ();
        if (fehler.equals ("A3")) throw new A3 ();
        if (fehler.equals ("A4")) throw new A4 ();
        if (fehler.equals ("A5")) throw new A5 ();
        if (fehler.equals ("A31")) throw new A31 ();
        if (fehler.equals ("E")) throw new E ("E");
        bem ("fff: try Ende");
        }
     catch (A31 e)
        {
        bem ("fff: catch (A31)");
        e.diagnose ();
        }
     catch (A3 e)
        {
        bem ("fff: catch (A3)");
        e.diagnose ();
        throw e;
        }
     catch (A5 e)
        {
        bem ("fff: catch (A5)");
        e.diagnose ();
        throw (A5)(e.fillInStackTrace ());
        }
     finally
        {
        bem ("fff: finally");
```

```
         }
      bem ("fff: nach finally");
      }

   static void bem (String bemerkung)
      {
      System.out.println (bemerkung);
      }
   }

class A extends Throwable
   {
   A (String diag)
      {
      this.diag = diag;
      }

   void diagnose ()
      {
      Ausnahme.bem ("Ausnahme " + diag + " wurde geworfen.");
      }

   private String diag;
   }

class A1 extends A { A1 () { super ("A1"); } }

class A2 extends A { A2 () { super ("A2"); } }

class A3 extends A
   {
   A3 () { super ("A3"); }
   A3 (String diag) { super (diag); }
   }

class A4 extends A { A4 () { super ("A4"); } }

class A5 extends A { A5 () { super ("A5"); } }

class A31 extends A3 { A31 () { super ("A31"); } }

class E extends Error
   {
   E (String diag)
      {
      this.diag = diag;
      }

   void diagnose ()
```

```
    {
    Ausnahme.bem ("Ausnahme " + diag + " wurde geworfen.");
    }

private String diag;
}
```

7.7 Ausnahmen in Initialisatoren

Wenn ein Instanz-Initialisator eine normale (checked) Ausnahme wirft und wenn diese dort nicht behandelt wird, dann muss jeder Konstruktor in seiner `throws`-Klausel mindestens diese Ausnahme deklarieren. In dem Fall muss auch mindestens ein Konstruktor implementiert werden.

Ein `static` Initialisator darf keine normalen Ausnahmen werfen. Falls doch, müssen diese dort auch behandelt werden.

7.8 Übungen

Übung Ausnahme:

1. Übersetzen Sie das Programm `Ausnahme.java` und lassen Sie es mit verschiedenen Argumenten (`A1`, `A2`, `A3`, `A31`, `A4`, `E`, `break`, `return`, `continue`) laufen. Versuchen Sie zu verstehen, was passiert.
2. Definieren Sie eine neue Ausnahme und bauen Sie diese in das Programm ein.
3. Ergänzen Sie Ihr Programm `Produkt.java` so, dass bei einem negativen Resultat eine besondere Meldung kommt. Verwenden Sie dabei Ausnahmebehandlung.
4. Schreiben Sie eine eigene Fehlerklasse, von der ein Objekt in der Klasse `Schmier` dann geworfen wird, wenn die Anzahl der Striche größer als tausend ist.

Übung `setNote`:

Die Methode `setNote` der Klasse `Klausurnote` ist noch im C-Stil geschrieben, indem sie einen Zahlencode im Fehlerfall zurückgibt. Schreibe diese Methode so um, dass sie im Fehlerfall das Objekt einer selbstgeschriebenen Fehlerklasse wirft.

Kapitel 8

Innere Klassen

Ab JDK 1.1 unterstützt Java nicht nur ***top-level*** Klassen, die Mitglieder von Packages sein müssen, sondern auch **innere Klassen** ***(inner class)***. Das sind Klassen, die innerhalb einer Klasse definiert werden.

Nützliche Anwendungen sind sogenannte Adapter-Klassen und Schonung des Namenraums. Man kann Block-Struktur mit Klassen-Struktur kombinieren.

Innere Klassen können definiert werden als

- Elemente von Klassen oder
- innerhalb von Blöcken oder
- – anonym – innerhalb von Ausdrücken oder
- als `static` innere Klasse oder Schnittstelle.

Außer bei `static` inneren Klasse ist ein innerer Klassenname nicht außerhalb des Definitionsbereichs verwendbar. Der Code von inneren Klassen kann Namen des umgebenden Scopes verwenden. So wie nicht-statische Methoden immer ihre aktuelle Instanz `this` kennen, kennt der Code einer inneren Klasse immer die äußere Instanz. D.h., wenn ein Objekt einer inneren Klasse erzeugt wird, muss immer klar sein, was das äußere Objekt ist. Daher in unserem Beispiel die Syntax:

```
ak.new InnereKlasse ();
```

Bei der anonym definierten Klasse folgt einem `new`-Ausdruck der Klassenkörper der anonymen Klasse. Sie leitet sich ab von der im `new`-Ausdruck genannten Klasse oder, wenn das eine Schnittstelle ist (siehe Kapitel Vererbung), von `Object`.

Beispiel:

```
public class   AeussereKlasse
   {
```

```
public static void   main (String[] argument)
   {
   AeussereKlasse ak = new AeussereKlasse ();
   ak.f ();
   InnereKlasse   ik = ak.new InnereKlasse ();
   ik.f ();
   ak.ff ();
   ak.fff ();
   }

void  f ()
   {
   System.out.println ("Kommt von der äußeren Klasse.");
   InnereKlasse   k = new  InnereKlasse ();
   k.f ();
   }

class InnereKlasse
   {
   void  f ()
      {
      System.out.print ("Kommt von der inneren Klasse, ");
      System.out.println ("die als Element definiert ist.");
      System.out.println ("   Das innere Objekt ist " + this + ",");
      System.out.println ("   das äußere Objekt ist "
         + AeussereKlasse.this + ".");
      }
   }

void  ff ()
   {
   class InBlockKlasse
      {
      void  f ()
         {
         System.out.print ("Kommt von der inneren Klasse, ");
         System.out.println ("die im Block definiert ist.");
         }
      }
   new InBlockKlasse ().f ();
   }


interface   KlasseOderSchnittstelle { void f (); }

void  fff ()
   {
   (new KlasseOderSchnittstelle ()
      {
```

```
        public void f ()
           {
           System.out.print ("Kommt von der inneren Klasse, ");
           System.out.println ("die im Block anonym definiert ist.");
           }
        }
     ).f ();
     }

  }
```

Eine innere Klasse darf nicht denselben Namen haben wie die enthaltende Klasse, (was nicht für Datenelemente und Methoden gilt.)

Nicht-statische innere Klassen dürfen keine statischen Elemente enthalten.

Klassen, die innerhalb eines Blocks definiert werden, dürfen nur `final` definierte Variable des Blocks verwenden.

Anonyme Klassen haben keinen eigenen Konstruktor, da sie keinen Namen haben. Sie übernehmen die Konstruktoren der Superklassse. Eventuelle Argumente werden automatisch an den Konstruktor der Superklasse übergeben. Ansonsten können Instanz-Initialisatoren verwendet werden.

Statische innere Klassen können wie Toplevel-Klassen verwendet werden, wobei der Name der äußeren Klasse mit Punktnotation vorangestellt werden muss. Sie beziehen sich nicht auf ein äußeres Objekt.

Für innere Klassen können alle Sichtbarkeitsstufen (`public`, `protected`, *default*, `private`) verwendet werden. Die Compiler berücksichtigen das zwar, aber die meisten JVMs nicht, da innere Klassen letztlich in ganz normale Klassen übersetzt werden, für die es ja nur die beiden Stufen `public` und *default* gibt. Daher sollte man nur diese beiden Stufen verwenden.

Innere Klassen haben **ohne Rücksicht auf die Sichtbarkeit** auf alle Elemente der äußeren Klasse Zugriff. Dies gilt auch umgekehrt! D.h. die äußere Klasse hat Zugriff auf alle Elemente der inneren Klasse unabhängig von deren Sichtbarkeit. Das gilt auch zwischen inneren Klassen einer gemeinsamen äußeren Klasse.

8.1 Anwendungen

8.1.1 Behandlung von Ereignissen

Die Erstellung von Klassen zur Behandlung von GUI-Ereignissen ist ein Beispiel für die Verwendung von anonymen Klassen. Das Ereignismodell des AWT verlangt, dass für jedes empfangene Ereignis ein Empfänger angegeben wird, der ein Objekt einer Klasse ist, die eine `Listener`-Schnittstelle implementiert. Da diese Klassen sonst nicht weiter verwendet werden, wäre es hier lästig, sich immer wieder neue Namen oder Dummy-Namen auszudenken.

Wir schreiben also das Beispiel `Schmier.java` aus der Einleitung folgendermaßen um:

```
// Schmier.java
import   java.applet.*;
import   java.awt.*;
import   java.awt.event.*;
public class   Schmier extends Applet
   {
   private int altesX = 0;
   private int altesY = 0;
   private Color  aktuelleFarbe = Color.black;
   private Button loeschKnopf;
   private Choice farbWahl;
   private Button endeKnopf;

   public void init ()
      {
      this.setBackground (Color.white);

      loeschKnopf = new Button ("Löschen");
      loeschKnopf.addActionListener
         (
         new ActionListener ()
            {
            public void actionPerformed (ActionEvent event)
               {
               Graphics g = Schmier.this.getGraphics ();
               Rectangle   r = Schmier.this.getBounds ();
               g.setColor (Schmier.this.getBackground ());
               g.fillRect (r.x, r.y, r.width, r.height);
               }
            }
         );
      loeschKnopf.setForeground (Color.black);
      loeschKnopf.setBackground (Color.lightGray);
      this.add (loeschKnopf);

      farbWahl = new Choice ();
      farbWahl.addItemListener
         (
         new ItemListener ()
            {
            public void itemStateChanged (ItemEvent e)
               {
               if (e.getItem ().equals ("Schwarz"))
                  aktuelleFarbe = Color.black;
               else if (e.getItem ().equals ("Rot"))
                  aktuelleFarbe = Color.red;
               else if (e.getItem ().equals ("Gelb"))
                  aktuelleFarbe = Color.yellow;
               else if (e.getItem ().equals ("Grün"))
                  aktuelleFarbe = Color.green;
               else if (e.getItem ().equals ("Blau"))
                  aktuelleFarbe = Color.blue;
               }

            }
         );
      farbWahl.addItem ("Schwarz");
      farbWahl.addItem ("Rot");
      farbWahl.addItem ("Gelb");
      farbWahl.addItem ("Grün");
      farbWahl.addItem ("Blau");
      farbWahl.setForeground (Color.black);
      farbWahl.setBackground (Color.lightGray);
      this.add (new Label ("Farbe: "));
      this.add (farbWahl);
      this.addMouseMotionListener
         (
         new MouseMotionListener ()
            {
            public void mouseDragged (MouseEvent e)
               {
               Graphics g = Schmier.this.getGraphics ();
               g.setColor (aktuelleFarbe);
```

```
            g.drawLine (altesX, altesY, e.getX (), e.getY ());
            altesX = e.getX ();
            altesY = e.getY ();
            }
         public void mouseMoved (MouseEvent e)
            {
            altesX = e.getX ();
            altesY = e.getY ();
            }
         }
      );
   }
public static void   main (String[] argument)
   {
   Frame f = new Frame ("Schmier");
   Schmier  s = new Schmier ();
   s.init ();
   s.endeKnopf = new Button ("Ende");
   s.endeKnopf.addActionListener
      (
      new ActionListener ()
         {
         public void actionPerformed (ActionEvent event)
            {
            System.exit (0);  // Nur bei Applikation erlaubt,
                              // nicht erlaubt bei Applet
            }
         }
      );
   s.endeKnopf.setForeground (Color.black);
   s.endeKnopf.setBackground (Color.lightGray);
   s.add (s.endeKnopf);
   f.add ("Center", s);
   f.pack ();
   f.setSize (600, 400);
   f.setVisible (true);
   s.start ();
   }
}
```

Für jedes Ereignis wird ein anonymes Objekt einer anonymen Listener-Klasse erzeugt und der Ereignis-erzeugenden Komponente gegeben. Damit wird z.B. die Fallunterscheidung in der Methode `actionPerformed` überflüssig.

8.1.2 `Callable`

Die generische Schnittstelle `Callable`

```
package  java.util.concurrent;

public interface  Callable<V>
   {
   V  call () throws Exception;
   }
```

deklariert eine Methode `call ()`, die ein `V` zurückgibt. Diese Schnittstelle muss durch eine Klasse – ein sogenannter Funktor – realisiert werden, deren Objekte als "Funktionsargumente" an Methoden übergeben werden können. Das implementierte `call ()` ist dann die übergebene Funktion. Die realisierende Klasse bleibt dabei typischerweise anonym.

Folgendes Beispiel soll das verdeutlichen:

```
import   java.util.concurrent.*;
public class   Rechner
   {
   public static class  Resultat
      {
      public String  aufgabe;
      public int  ergebnis;
      public   Resultat (String aufgabe, int ergebnis)
         {
         this.aufgabe = aufgabe;
         this.ergebnis = ergebnis;
         }
      }
   public int  eingabe;
   public   Rechner (int eingabe)
      {
      this.eingabe = eingabe;
      }
   public void berechneUndZeige (Callable<Resultat> funktor)
      {
      try
         {
         Resultat resultat = funktor.call ();
         System.out.println ("Das Ergebnis der Berechnung der "
            + resultat.aufgabe + eingabe
            + " ergibt " + resultat.ergebnis + "!");
         }
         catch (Exception e) { e.printStackTrace (); }
      }
   static Rechner rechner;
   public static void   main (String[] arg)
      {
      rechner = new Rechner (7);
      rechner.berechneUndZeige
         (
         new Callable<Resultat> ()
            {
            public Resultat   call ()
               {
               int   sum = 0;
               for (int i = 0; i <= rechner.eingabe; i++)
                  {
                  sum = sum + i;
                  }
               return new Resultat ("Summe von 0 bis ", sum);
               }
            }
         );
      rechner.berechneUndZeige
         (
         new Callable<Resultat> ()
            {
            public Resultat   call ()
               {
               int   fak = 1;
               for (int i = 1; i <= rechner.eingabe; i++)
                  {
                  fak = fak * i;
                  }
```

```
            return new Resultat ("Fakultät von ", fak);
            }
        }
    );
  }
}
```

8.2 Übungen

Übung Schmier:

Kopieren Sie Ihre Version der Klasse `Schmier` und eventuell dazugehöriger Klassen in ein neues Verzeichnis. Schreiben Sie Ihre Schmierklasse unter Verwendung von anonymen Ereignisbehandlern um.

Kapitel 9

Threads

Java bietet die Möglichkeit der Parallel-Programmierung, indem man besondere Methoden als eigene **Threads** ***(thread)*** laufen lassen kann. Bei einem Ein-Prozessor-System sind das quasi-parallel ablaufende Steuerflüsse innerhalb eines Programms. Ein Scheduler simuliert die Parallelität, wobei es verschiedene Strategien gibt.

Man unterscheidet gemeinhin **Prozesse** und **Threads**. Prozesse haben einen eigenen Adressraum (Speicherbereich) und können nur über Betriebssystem-Mittel kommunizieren. Threads haben keinen eigenen Adressraum und können z.B. auch über gemeinsame Variablen kommunizieren. Threads werden auch als **Leichtgewichts-Prozesse** ***(light weight process)*** bezeichnet. Als übergeordneter Begriff ist **Task** zu sehen, der die Ausführung eines **Programms** bezeichnet. Ein Programm ist statisch und kann als Task mehrmals und auch parallel ausgeführt werden.

Der Java-Scheduler lässt einen aktiven Thread so lange laufen, bis er entweder von einem Thread mit höherer Priorität unterbrochen wird, oder bis er selbst die Kontrolle abgibt, weil er durch Ein- oder Ausgabe blockiert wird oder weil er ein `yield`, `sleep` oder `wait` macht ***(priority based preemptive scheduling without time slicing)***. Sind zu einem Zeitpunkt mehrere Threads mit höchster Priorität bereit zu laufen, dann beginnt der Scheduler mit dem Thread, der am längsten gewartet hat.

Man muss aber die Gegebenheiten des zugrundeliegenden Betriebssystems beachten: evtl. gibt es doch Zeitscheiben oder weniger (bzw. mehr) Prioritätsstufen. Windows hat zum Beispiel nur sieben Prioritätsstufen und unter Linux steht uns nur eine – also keine! – Prioritätsstufe zur Verfügung.

Der Programmierer sollte sich nicht auf ein Verhalten des Schedulers verlassen. Er sollte weder "Gerechtigkeit noch nachvollziehbare Logik" vom Scheduler erwarten [24].

9.1 Einzelner Thread

Zur Erzeugung eines Threads muss man eine Klasse schreiben, die entweder von der Klasse `Thread` erbt oder die die Schnittstelle `Runnable` realisiert. Dabei wird die Methode `run ()` mit dem eigentlichen Thread-Code implementiert.

Die den Thread startende Methode `start ()` stellt die Klasse `Thread` zur Verfügung. Sie ruft im wesentlichen die Methode `run ()` auf, d.h sie lässt den Code der Methode `run ()` als Thread laufen. Die Schnittstelle `Runnable` deklariert die Methode `run ()`. Die Methode `run ()` enthält die eigentliche Aktivität des Threads.

Mit einem Objekt vom Typ der Schnittstelle `Runnable` kann ein Objekt vom Typ `Thread` initialisiert werden, so dass die beiden Möglichkeiten, eigenen Code als Thread laufen zu lassen, nun folgendermaßen aussehen:

1. Die eigene Klasse, etwa `MeinThread1`, erbt von `Thread` und überschreibt die Methode `run ()` mit eigenem Code:

```
public class   MeinThread1 extends Thread
   {
   public void run ()
      {
      // Eigentlicher Threadcode
      }
   }
```

   Im Anwendungscode wird der eigene Thread erzeugt und durch Aufruf von `start ()` gestartet:

```
MeinThread1 p = new MeinThread1 ();
p.start ();
```

2. Die eigene Klasse, etwa `MeinThread2`, realisiert `Runnable` und implementiert die Methode `run ()` mit eigenem Code:

```
public class   MeinThread2 implements Runnable
   {
   public void run ()
      {
      // Eigentlicher Threadcode
      }
   }
```

   Im Anwendungscode wird ein Thread mit einem Objekt der eigenen Klasse initialisiert und durch Aufruf von `start ()` gestartet. Dabei wird aber das `run ()` der eigenen Klasse als Thread gestartet:

```
MeinThread2 ob = new MeinThread2 ();
Thread   p = new Thread (ob);
p.start ();
```

Diese Variante muss man wählen, wenn man sonst mehrfach erben müsste.

Die beiden folgenden Beispiele zeigen jeweils eine primitive Ausarbeitung dieser Konzepte.

Erben von `Thread`:

```
public class   ErbtThread
   {
   public static void   main (String[] argument)
      {
      MeinThread1 herr = new MeinThread1 ("Herr");
      herr.start ();

      MeinThread1 knecht = new MeinThread1 ("Knecht");
      knecht.start ();
      }
   }

class MeinThread1 extends Thread
   {
   private String name;

   public   MeinThread1 (String name)
      {
      this.name = name;
      }

   public void run ()
      {
      while (true)
         {
         System.out.println ("Thread " + name + " läuft.");
         //yield ();
         }
      }
   }
```

Implementieren von `Runnable`:

```
public class   ImpleRunnable
   {
   public static void   main (String[] argument)
      {
      MeinThread2 herr = new MeinThread2 ("Herr");
      Thread   herrTh = new Thread (herr);
      herrTh.start ();
```

```
      MeinThread2 knecht = new MeinThread2 ("Knecht");
      Thread   knechtTh = new Thread (knecht);
      knechtTh.start ();
      }
   }

class MeinThread2 implements Runnable
   {
   private String name;

   public   MeinThread2 (String name)
      {
      this.name = name;
      }

   public void run ()
      {
      while (true)
         {
         System.out.println ("Thread " + name + " läuft.");
         //Thread.yield ();
         }
      }
   }
```

9.2 Methoden der Klasse `Thread`

`start ()`: Mit `start ()` wird ein Thread gestartet. Der Thread läuft solange, bis die `run`-Methode an ihr natürliches Ende gelangt ist. Ein Neustart ist nur möglich, indem man das Thread-Objekt neu erzeugt und wieder startet. D.h. ein Thread-Objekt kann nur ein einziges Mal als Thread gestartet werden.

`Thread.sleep ()`: Mit `sleep (long ms)` bzw. `sleep (long ms, int nanos)` wird der Thread `ms` Millisekunden und eventuell `nanos` Nanosekunden angehalten. (Bei `sleep (0)` wird der Thread nicht angehalten. Negative Werte sind illegal.)

`Thread.yield ()`: Mit `yield ()` wird die Kontrolle an einen lauffähigen Thread gleicher Priorität abgegeben. `yield ()` und `sleep (...)` sind `static` Methoden. Diese kann ein Thread also nur für sich selbst aufrufen.

Mit `yield ()` kann ein "kooperatives" Verhalten oder Multitasking unter Threads gleicher Priorität programmiert werden.

`join ()`: `p.join ()` wartet solange, bis der Thread `p` zu Ende gekommen ist. Mit `p.join (long ms)` bzw. `p.join (long ms, int nanos)` wird höchstens `ms` Millisekun-

den und eventuell `nanos` Nanosekunden gewartet. (Bei `p.join (0)` wird unendlich lange gewartet.)

`interrupt ():` Mit `interrupt ()` wird ein Unterbrechungsflag für die Task gesetzt (auf `true`), das mit `isInterrupted ()` oder `Thread.interrupted ()` abgefragt werden kann, wobei nur die statische Methode `Thread.interrupted ()` das Flag wieder zurücksetzt (auf `false`).

`sleep`, `join` und `wait` werden mit `interrupt ()` abgebrochen, wobei eine `InterruptedException` geworfen und gleichzeitig das Flag zurückgesetzt wird. Wenn das `interrupt ()` vor erreichen von `sleep`, `join` und `wait` erhalten wird, dann wird es bei Erreichen dieser Methoden wirksam, wobei die `InterruptedException` geworfen wird, es sei denn, dass vorher `Thread.interrupted ()` aufgerufen wurde.

`join ()` und `interrupt ()` sind die einzigen direkten Inter-Thread-Kommunikations-Möglichkeiten (indirekt auch `notify ()` und `notifyAll ()`).

`stop (), suspend (), resume ():` Die Methoden `stop`, `suspend` und `resume` gibt es nicht mehr, weil sie sehr fehlerträchtig sind. Daher stellt sich die Frage, wie ein Thread zu unterbrechen bzw. abzubrechen ist. Für das Unterbrechen bieten die Methoden `sleep`, `join` und `wait` genügend Möglichkeiten. Für den Abbruch wird folgendes empfohlen: Der Thread sollte regelmäßig eine Variable (für prompte Reaktion vorzugsweise als `volatile` deklariert) überprüfen. Wenn die Variable anzeigt, dass der Thread stoppen sollte, dann sollte sie an ihr natürliches oder an ein vernünftiges Ende laufen.

`Name:` Die Klasse `Thread` hat das Attribut `Name` mit den Methoden `setName (String name)` und `getName ()`. (Daher wäre in dem Beispiel oben die Verwaltung des Namens überflüssig gewesen, wenn man von `Thread` erbt.)

`isAlive ():` Mit der Methode `isAlive ()` kann überprüft werden, ob ein Thread noch läuft bzw. überhaupt gestartet wurde.

`Thread.currentThread ():` Gibt eine Referenz auf den gerade laufenden Thread zurück.

9.3 Gruppen von Threads

Jeder Thread gehört einer Threadgruppe an. Jede Threadgruppe, außer der "Wurzel"-Threadgruppe, gehört einer Threadgruppe an.

Mit der Gruppierung von Threads kann man Steuermethoden wie z.B. `interrupt ()` nicht nur für einzelne Threads, sondern für ganze Gruppen von Threads aufrufen. Ferner kann man Gruppenhierarchien bilden.

Eine Thread-Gruppe kann eine maximale Priorität haben.

Mit den Konstruktoren

```
ThreadGroup (String name) und
ThreadGroup (ThreadGroup gruppe, String name)
```

können Threadgruppen mit einem Namen angelegt werden und gegebenenfalls einer anderen Threadgruppe untergeordnet werden. (Default ist die Wurzel.) Ein Thread kann mit den Konstruktoren

`Thread (ThreadGroup gruppe, String name)` oder
`Thread (ThreadGroup gruppe, Runnable objekt, String name)`

einer Gruppe zugeordnet werden.

9.4 Priorität

Mit

```
public final void setPriority (int prioritaet)
```

kann die Priorität eines Threads gesetzt werden. Je höher die Zahl, desto höher ist die Priorität. Sie ist einstellbar von `Thread.MIN_PRIORITY` bis `Thread.MAX_PRIORITY`. Ein Thread wird defaultmäßig mit der Priorität des ihn anlegenden Prozesses ausgestattet. Default ist `Thread.NORM_PRIORITY`.

9.5 Synchronisation

Java garantiert, dass die meisten primitiven Operationen atomar (unteilbar) durchgeführt werden. Das sind z.B. Operationen, zu denen der Zugriff auf die Standardtypen (außer `long` und `double`) und auch die Referenztypen gehört (gemeint sind z.B. Zuweisungen an einen Referenztyp selbst).

Zum Schutz von "kritischen Bereichen" stellt Java ein Monitorkonzept zur Verfügung. Jedes Objekt mit kritischem Bereich bekommt zur Laufzeit einen Monitor. Kritische Bereiche sind eine oder mehrere Methoden oder Anweisungsblöcke, die von nicht mehr als einem Thread gleichzeitig durchlaufen werden dürfen.

Das Grundprinzip ist, dass individuelle Code-Blöcke über den Konstrukt

```
synchronized (irgendeinObjekt) { irgendeinCode }
```

synchronisiert werden können.

Eine `synchronized` Methode ist eine Methode, deren gesamte Implementation als ein `synchronized` Block

```
synchronized (this) { Methodencode }
```

zu verstehen ist. Das Monitor-Objekt ist das Objekt, wofür die Methode aufgerufen wird. Bei `static` Metoden wird das zur Klasse gehörige Klassenobjekt verwendet:

```
synchronized (this.getClass ()) { staticMethodencode }
```

Synchronisation ist so implementiert, dass für jedes Objekt (inklusive Klassenobjekten) ein nicht zugängliches Lock verwaltet wird, das im wesentlichen ein Zähler ist. Wenn der Zähler beim Betreten eines über das Objekt synchronisierten Blocks oder einer synchronisierten Methode *nicht* Null ist, dann wird der Thread blockiert (suspendiert). Er wird in eine Warteschlange für das Objekt gestellt ***(Entry Set)***, bis der Zähler Null ist. Wenn ein synchronisierter Bereich betreten wird, dann wird der Zähler um Eins erhöht. Wenn ein synchronisierter Bereich verlassen wird, dann wird der Zähler um Eins erniedrigt, (auch wenn der Bereich über eine Exception verlassen wird.)

Ein Thread, der ein Lock auf ein Objekt hat, kann mehrere kritische Bereiche oder Methoden desselben Objekts benutzen, wobei der Zähler beim Betreten des Bereichs hochgesetzt, beim Verlassen heruntergesetzt wird.

Nicht-synchronisierte Methoden oder Bereiche können von anderen Threads verwendet werden, auch wenn ein Thread ein Lock auf das Objekt hat. Insbesondere können alle Daten eines eventuell gelockten Objekts benutzt werden, sofern sie zugänglich (sichtbar) sind.

Der Modifikator `synchronized` wird nicht automatisch vererbt. Wenn also eine geerbte Methode auch in der abgeleiteten Klasse `synchronized` sein soll, so muss das angegeben werden. Bei Schnittstellen kann `synchronized` nicht angegeben werden, aber die Methoden können `synchronized` implementiert werden.

9.5.1 `synchronized`

Syntaktisch gibt es folgende Möglichkeiten:

1. `synchronized` als Modifikator einer Methode: Bevor die Methode von einem Thread für ein Objekt abgearbeitet wird, wird das Objekt von dem betreffenden Thread zum ausschließlichen Gebrauch von synchronisiertem Code des Objekts reserviert. Dabei wird der Thread eventuell so lange blockiert, bis der synchronisierte Code des Objekts nicht mehr von anderen Threads benötigt wird.

2. `synchronized (`*Ausdruck*`)` *Block* : Der Ausdruck muss in einem Objekt oder Feld resultieren. Bevor ein Thread den Block betritt, wird versucht, ein Lock auf das Resultat des Ausdrucks zum ausschließlichen Gebrauch synchronisierten Codes zu bekommen. Dabei wird der Thread eventuell so lange blockiert, bis der synchronisierte Code nicht mehr von anderen Threads benötigt wird.

3. `synchronized (`*Ausdruck*`.getClass ())` *Block* : Der Ausdruck muss in einem Objekt resultieren, dessen Klasse ermittelt wird. Damit werden `synchronized` Klassenmethoden geschützt. Bevor ein Thread den Block betritt, werden alle `synchronized` Klassenmethoden der betreffenden Klasse oder über die Klasse synchronisierte Code-Stücke zum ausschließlichen Gebrauch reserviert.

4. Konstruktoren können nicht `synchronized` definiert werden. Das ergibt einen Compilezeit-Fehler.

Mit `synchronized` werden Codestücke definiert, die *atomar* oder *unteilbar* bezüglich eines Objekts durchzuführen sind. Das Problem des gegenseitigen Ausschlusses ist damit schon gelöst und wesentlich eleganter gelöst als etwa mit Semaphoren.

Bemerkung: Das Problem der Prioritäts-Inversion – ein hochpriorer Thread läuft effektiv mit niedrigerer Priorität, wenn er versucht, ein Lock zu bekommen, das ein niedriger-priorer Thread hat – wird von vielen JVMs durch Prioritäts-Vererbung gelöst. Das ist aber nicht verpflichtend für eine JVM.

9.5.2 `wait`, `notify` und `notifyAll`

Die Methoden `wait ()`, `wait (long ms)`, `wait (long ms, int nanos)`, `notify ()` und `notifyAll ()` sind Methoden der Klasse `Object` und stehen daher für jedes Objekt zur Verfügung. Diese Methoden können nur in kritischen (`synchronized`) Bereichen des Objekts aufgerufen werden, d.h. wenn für das Objekt, wofür sie aufgerufen werden, ein Lock erhalten wurde (d.h. – wie man auch sagt – der Monitor des Objekts betreten wurde).

Wenn ein Thread für irgendein Objekt `x` die Methode `wait`

```
x.wait ();
```

aufruft, dann wird er in eine dem Objekt `x` zugeordnete `wait`-Warteschlange gestellt ***(Wait Set)***. Außerdem wird der kritische Bereich freigegeben, d.h. der Thread gibt sein Lock auf das Objekt `x` ab (d.h. der Monitor wird verlassen).

Wenn ein anderer Thread

```
x.notify ();
```

aufruft, dann wird *ein* Thread aus der `wait`-Warteschlange von `x` entfernt und wieder lauffähig. Dieser Thread muss sich aber nun um ein Lock für das Objekt `x` bemühen, um den kritischen Bereich weiter zu durchlaufen. Er muss also mindestens solange warten, bis der notifizierende Thread das Objekt freigegeben hat (d.h. den Monitor verlassen hat).
(Bemerkung: Mit der `Thread`-Instanzmethode `interrupt ()` kann ein spezifischer Thread aus der `wait`-Warteschlange geholt werden, wobei allerdings die `InterruptedException` geworfen wird.)

Wenn

```
x.notifyAll ();
```

aufgerufen wird, dann werden *alle* Threads aus der `wait`-Warteschlange von `x` entfernt und wieder lauffähig. Diese Threads müssen sich aber nun um ein Lock für das Objekt `x` bemühen, um den kritischen Bereich weiter zu durchlaufen. Ein Objekt mit kritischen Bereichen hat also konzeptionell *zwei* Warteschlangen:

- eine für den Monitor (d.h. die kritischen Bereiche) ***(Entry Set)***
- und eine für das `wait` ***(Wait Set)***.

Das Monitor-Konzept kann man auch unter folgenden Gesichtspunkten betrachten:

Ressourcen-Sicht:
Die Operationen

`"beantrage Ressource"` und `"gib Ressource frei"`
(bzw. `"beantrage Lock"` und `"gib Lock frei"`)
entsprechen in Java dem Paar:
`"synchronized (x) {"` und `"}"`

Nur der Thread, der ein Lock beantragt und erhalten hat, kann es auch wieder freigeben. Bekommt der Thread das Lock nicht, muss er warten (und nur dann).

"`x`" ist hierbei als Ressource zu verstehen.

Ereignis-Sicht:
Die Operationen

`"warte auf Ereignis"` und `"sende Ereignis"`
(bzw. `"warte auf Event"` und `"sende Event"`)
entsprechen in Java dem Paar:
`"x.wait ()"` und `"x.notify ()"` (oder `"x.notifyAll ()"`)

`"x"` ist dabei als Ereignis ***(Event)*** zu verstehen. Ein Thread, der auf ein Ereignis wartet, muss auf jeden Fall warten. Er kann das Ereignis nur von einem anderen Thread bekommen.

Bemerkung: In Java sind diese beiden Sichten insofern etwas vermischt, als man die Ressource `x` benötigt, um die Ereignis-Operationen `wait` oder `notify/All` durchführen zu können.

9.6 Beispiele

9.6.1 Erzeuger-Verbraucher-Problem

Beim Erzeuger-Verbraucher-Problem ***(producer-consumer-problem)*** generiert der Erzeuger ein Objekt (z.B. einen Satz von xyz-Koordinaten für die Position eines Roboterarms) und stellt es dem Verbraucher (z.B. dem Roboter) zur Verfügung, der das Objekt dann verarbeitet (z.B. sich an die angegebene Position bewegt). Es kommt hierbei darauf an, dass jedes erzeugte Objekt auch verbraucht wird und dass kein Objekt zweimal verbraucht wird.

Für die Übergabe des Objekts programmieren wir einen Puffer, der die Methoden

```
void  legeAb (Object x)
```
und
```
Object   entnehme ()
```

zur Verfügung stellt. Der Erzeuger soll `legeAb` für das erzeugte Objekt aufrufen. Der Verbraucher soll sich mit `entnehme` das Objekt holen. Diese Methoden sind so programmiert, dass, falls das alte Objekt noch nicht entnommen wurde, kein neues Objekt abgelegt werden kann und die Erzeuger-Task warten muss. Umgekehrt muss die Verbraucher-Task warten, wenn kein neues Objekt vorliegt.

```
public class   Puffer
   {
   private boolean   belegbar = true;
   private Object transferObjekt;
```

```
   public synchronized void   legeAb (Object x)
      {
      while (!belegbar)
         {
         try
            {
            wait ();
            }
            catch (InterruptedException e) { e.printStackTrace (); }
         }
      transferObjekt = x;
      belegbar = false;
      notify ();
      System.out.println (transferObjekt + " wurde abgelegt von "
         + Thread.currentThread ().getName () + ".");
      }
   public synchronized Object entnehme ()
      {
      while (belegbar)
         {
         try
            {
            wait ();
            }
            catch (InterruptedException e) { e.printStackTrace (); }
         }
      belegbar = true;
      notify ();
      System.out.println ("          " + transferObjekt
         + " wurde entnommen von "
            + Thread.currentThread ().getName () + ".");
      return transferObjekt;
      }
   }  // end class Puffer
```

```
public class   Erzeuger extends Thread
   {
   static java.util.Random zufall = new java.util.Random ();
   static class   Position
      // Objekte dieser Klasse werden transferiert.
      {
      int x = zufall.nextInt (9);
      int y = zufall.nextInt (9);
      int z = zufall.nextInt (9);
      public String  toString ()
         {
         String   s = super.toString ();
         s = s.substring (s.indexOf ('@'));
         return "Position (" + s
            + ")[x = " + x + "  y = " + y + "  z = " + z + "]";
         }
      }  // end class Erzeuger.Position
   public static void   warte (String botschaft)
      {
      System.out.println (botschaft);
      try
         {
         Thread.sleep (zufall.nextInt (500));
         }
         catch (InterruptedException e) { e.printStackTrace (); }
      }
   private Puffer puffer;
   public   Erzeuger (String name, Puffer puffer)
      {
      setName ("Erzeuger " + name);
```

```
      this.puffer = puffer;
      }
   public void run ()
      {
      while (true)
         {
         Object   x = new Position ();
         warte (x + " wurde erzeugt von " + getName () + ".");
         puffer.legeAb (x);
         }
      }
   }  // end class Erzeuger
```

```
public class   Verbraucher extends Thread
   {
   private Puffer puffer;
   public   Verbraucher (String name, Puffer puffer)
      {
      setName ("Verbraucher " + name);
      this.puffer = puffer;
      }
   public void run ()
      {
      while (true)
         {
         Object   x = puffer.entnehme ();
         Erzeuger.warte ("          " + x + " wird verbraucht von " + getName () + ".");
         }
      }
   }  // end class Verbraucher
```

```
public class   ErzeugerVerbraucher
   {
   public static void   main (String[] arg)
      {
      Puffer   puffer = new Puffer ();
      Erzeuger erzeuger = new Erzeuger ("Oskar", puffer);
      Verbraucher verbraucher = new Verbraucher ("Katrin", puffer);
      erzeuger.start ();
      verbraucher.start ();
      }
   }  // end class ErzeugerVerbraucher
```

Was passiert aber, wenn viele Erzeuger und Verbraucher denselben Puffer verwenden:

```
public class   VieleErzeugerVerbraucher
   {
   public static void   main (String[] arg)
      {
      Puffer   puffer = new Puffer ();
      for (int i = 0; i < 10; i++)
         {
         Erzeuger erzeuger = new Erzeuger ("" + (i + 1), puffer);
         Verbraucher verbraucher = new Verbraucher ("" + (i + 1), puffer);
         erzeuger.start ();
         verbraucher.start ();
         }
      }
```

```
} // end class VieleErzeugerVerbraucher
```

Wenn man das lang genug laufen lässt, dann kommt es zu einer Verklemmung, da eventuell alle Erzeuger in der Warteschlange sind, und das letzte `notify` eines Verbrauchers gerade wieder nur einen Verbraucher weckt.

Dieses Problem kann gelöst werden, indem man

```
notify ()
```

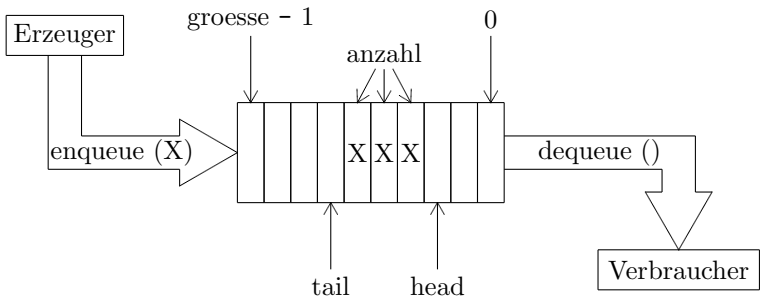
durch

```
notifyAll ()
```

ersetzt. Im Zweifelsfall ist es sicherer, `notifyAll` zu verwenden!

9.6.2 Beispiel: Ringpuffer

Eine allgemeinere Lösung des Erzeuger-Verbraucher-Problems bietet der Ringpuffer (Zirkularpuffer, Kanal, Pipe, Queue), der hier für Objekte implementiert ist. Durch einen Ringpuffer werden Erzeuger und Verbraucher zeitlich entkoppelt.



Bemerkung: Das Erzeuger-Verbraucher-Muster ist ein wichtiges nebenläufiges Entwurfs-Muster, etwa auch unter der Sicht, dass der Erzeuger Arbeitseinheiten definiert, die vom Verbraucher zu erledigen sind. Arbeiten werden zeitlich verschoben. Dadurch entstehen weniger Code-Abhängigkeiten. Ferner wird das Arbeits-Management *(workload management)* vereinfacht.

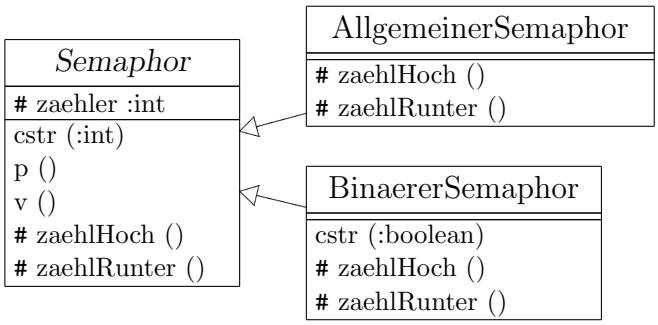
Die Klasse `RingPuffer` bietet zwei Methoden an: `enqueue (Object ob)` und `dequeue ()`.

```
import   java.util.*;
public class   RingPuffer
   {
   private Object[]  f;
   private int groesse;
   private int anzahl = 0;
```

```
   private int tail = 0;
   private int head = -1;
   public   RingPuffer (int groesse)
      {
      this.groesse = groesse;
      f = new Object[groesse];
      }
   synchronized public void   enqueue (Object ob)
       {
       while (anzahl >= groesse)
         try
            {
            wait ();
            }
            catch (InterruptedException e)
               {
               System.err.println ("Fehler in RingPuffer.enqueue:");
               System.err.println ("   " + e);
               }
       f[tail] = ob;
       anzahl = anzahl + 1;
       tail = tail + 1;
       if (tail == groesse) tail = 0;
       notifyAll ();
       }
   synchronized public Object dequeue ()
      {
      while (anzahl == 0)
         try
            {
            wait ();
            }
            catch (InterruptedException e)
               {
               System.err.println ("Fehler in RingPuffer.dequeue:");
               System.err.println ("   " + e);
               }
      anzahl = anzahl - 1;
      head = head + 1;
      if (head == groesse) head = 0;
      notifyAll ();
      return f[head];
      }
   public static void   main (String [] argument)
      {
      RingPuffer  rp = new RingPuffer (3);
      for (int i = 0; i < 5; i++)
         {
         new Erzeuger (rp, "E" + (i + 1), 7).start ();
         new Verbraucher (rp, "V" + (i + 1), 7).start ();
         }
      }
   Random   zufall = new Random ();
   }
class Erzeuger extends Thread
   {
   private RingPuffer   rp;
   private String name;
   private int wieoft;
   public   Erzeuger (RingPuffer rp, String name, int wieoft)
      {
      this.rp = rp;
      this.name = name;
      this.wieoft = wieoft;
      }
   public void run ()
      {
      for (int i = 0; i < wieoft; i++)
```

```
        {
        rp.enqueue (name + "." + (i + 1));
        System.out.println (name + ": " + (i + 1));
        try
           {
           sleep ((int)(rp.zufall.nextDouble () * 10000));
           }
           catch (InterruptedException e)
              {
              System.err.println ("Fehler in Erzeuger.run:");
              System.err.println ("   " + e);
              }
        }
     }
  }
class Verbraucher extends Thread
  {
  private RingPuffer   rp;
  private String name;
  private int wieoft;
  public   Verbraucher (RingPuffer rp, String name, int wieoft)
     {
     this.rp = rp;
     this.name = name;
     this.wieoft = wieoft;
     }
  public void run ()
     {
     for (int i = 0; i < wieoft; i++)
        {
        Object   s = rp.dequeue ();
        System.out.println (name + ": " + s);
        try
           {
           sleep ((int)(rp.zufall.nextDouble () * 10000));
           }
           catch (InterruptedException e)
              {
              System.err.println ("Fehler in Verbraucher.run:");
              System.err.println ("   " + e);
              }
        }
     }
  }
```

9.6.3 Beispiel: Emulation von Semaphoren



```
public abstract class   Semaphor
   {
   public   Semaphor (int zaehler)
      {
      if (zaehler < 0) this.zaehler = 0;
      else this.zaehler = zaehler;
      }
   public synchronized void   p ()
      {
      while (zaehler == 0)
         {
         try { wait (); }
         catch (InterruptedException e) {}
         }
      zaehlRunter ();
      }
   public synchronized void   v ()
      {
      zaehlHoch ();
      notifyAll ();  // notify () würde hier auch genügen.
      }
   protected int   zaehler;
   protected abstract void  zaehlHoch ();
   protected abstract void  zaehlRunter ();
   }
```

```
public class   AllgemeinerSemaphor extends Semaphor
   {
   public   AllgemeinerSemaphor (int zaehler)
      {
      super (zaehler);
      }
   protected void  zaehlHoch ()
      {
      zaehler++;
      }
   protected void  zaehlRunter ()
      {
      zaehler--;
      if (zaehler < 0) zaehler = 0;
      }
   }
```

```
public class   BinaererSemaphor extends Semaphor
   {
   public   BinaererSemaphor (boolean frei)
      {
      super (frei ? 1 : 0);
      }
   protected void  zaehlHoch ()
      {
      zaehler = 1;
      }
   protected void  zaehlRunter ()
      {
      zaehler = 0;
      }
   }
```

Als Anwendung schreiben wir ein primitives Programm zum automatischen Schweißen mit einem Roboterarm. Folgende Aufgaben sind hintereinander durchzuführen:

uB: (unkritische) Berechnung der nächstens Armposition.

kP: (kritische) Positionierung des Arms.

uV: (unkritische) Vorbereitung des Schweißwerkzeugs.

kS: (kritisches) Schweißen.

uB kann parallel zu uV und kS durchgeführt werden. uV kann parallel zu uB und kP durchgeführt werden. kP und kS dürfen nicht gleichzeitig und kP muss vor kS durchgeführt werden. Um möglichst effektiv zu arbeiten, wollen wir alle parallel durchführbaren Aktivitäten auch parallel durchführen.

Das Programm dazu ist:

```
import java.util.*;
public class   SchweissRoboter
   {
   BinaererSemaphor  s1 = new BinaererSemaphor (true);
   BinaererSemaphor  s2 = new BinaererSemaphor (false);
   public static void   main (String[] argument)
      {
      SchweissRoboter   r = new SchweissRoboter ();
      new ArmThread (r).start ();
      new SchweissThread (r).start ();
      }
   public void uB (int n) { operation ("uB" + n); }
   public void kP (int n) { operation ("kP" + n); }
   public void uV (int n) { operation ("uV" + n); }
   public void kS (int n) { operation ("kS" + n); }
   private static Random   rand = new Random ();
   private static void  warte (Random rand)
      {
      try
         {
         Thread.sleep ((long)(Math.abs (rand.nextInt ()) % 100));
         }
         catch (InterruptedException e) {}
      }
   private void   operation (String opName)
      // Simuliert die Operationen des Roboters durch Bildschirmausgaben
      // mit unterschiedlich vielen Zeilen.
      {
      if (Thread.currentThread ().getName ().equals ("SchweissThread"))
         opName = "                  " + opName;
      int   j = rand.nextInt ();
      j = j < 0 ? -j : j;
      j = j % 4 + 1;
      System.out.println (opName + " Anfang"); warte (rand);
      for (int i = 0; i < j; i++)
         {
         System.out.println (opName + " ......"); warte (rand);
         }
      System.out.println (opName + " Ende. "); warte (rand);
      }
```

```
   }
class ArmThread extends Thread
   {
   private SchweissRoboter r;
   public   ArmThread (SchweissRoboter r)
      {
      this.r= r;
      setName ("ArmThread");
      }
   public void run ()
      {
      int   n = 0;
      while (true)
         {
         n++;
         r.uB (n);
         r.s1.p ();
         r.kP (n);
         r.s2.v ();
         }
      }
   }
class SchweissThread extends Thread
   {
   private SchweissRoboter r;
   public   SchweissThread (SchweissRoboter r)
      {
      this.r= r;
      setName ("SchweissThread");
      }
   public void run ()
      {
      int   n = 0;
      while (true)
         {
         n++;
         r.uV (n);
         r.s2.p ();
         r.kS (n);
         r.s1.v ();
         }
      }
   }
```

9.7 Dämonen

Der Java-Interpreter wird erst verlassen, wenn alle Threads aufgehört haben zu laufen. Allerdings nimmt der Java-Interpreter dabei keine Rücksicht auf sogenannte **Dämonen**. Ein Dämon ***(daemon)*** ist ein Thread, der immer im Hintergrund läuft und nie aufhört zu laufen. Der Java-Interpreter wird also verlassen, wenn nur noch Dämonen laufen.

Mit der Methode `setDaemon (true)` wird ein Thread zum Daemon gemacht. Die Methode kann nur *vor dem Start* eines Threads aufgerufen werden.

Mit `isDaemon ()` kann festgestellt werden, ob ein Thread ein Dämon ist.

Alle Threads, die von einem Dämon erzeugt werden, sind ebenfalls Dämonen.

Dämonen sollten sparsam verwendet werden, da sie ja, ohne die Möglichkeit von Aufräumarbeiten, rücksichtslos bei Abschalten der JVM abgebrochen werden. Z.B. kann ein Log-Service-Dämon die letzten Nachrichten evtl. nicht mehr in einen File schreiben.

9.8 Übungen

9.8.1 `start/run`-Methode

Erklären Sie den Unterschied zwischen

```
MeinThread1 p = new MeinThread1 ();
p.start ();
```

und

```
MeinThread1 p = new MeinThread1 ();
p.run ();
```

9.8.2 Threads

Verändern Sie das Herr-Knecht-Beispiel so, dass das Hauptprogramm den Knecht nach 6 Sekunden stoppt. Der Herr soll seine while-Schleife nur 1000 mal durchlaufen. Das Hauptprogramm soll dann warten, bis der Herr zu Ende gelaufen ist.

9.8.3 Klasse `GroupTree`

Schreiben Sie eine Klasse `GroupTree`, deren `static` Methode `dumpAll ()` alle Threads und ihre Threadgruppen listet. (Hinweis: Die Klasse `ThreadGroup` bietet hier geeignete Methoden an.)

9.8.4 Sanduhr

In dem unten angegebenen Beispiel dauert die Abarbeitung der Methode `dauertLange ()` etwa 5 bis 10 Sekunden. Aufgabe ist es, diese Zeit zu überbrücken, indem jede halbe Sekunde ein Punkt nach Standard-Out geschrieben wird.

```
public class   Sanduhr
   {
   public static void   main (String[] arg)
      {
      Sanduhr  su  = new Sanduhr ();
      su.dauertLange ();
      }
```

```
public void dauertLange ()
   {
   System.out.println ("Achtung! Das dauert jetzt ein bisschen.");
   try
      {
      Thread.sleep ((long)(Math.random ()*5000 + 5000));
      }
      catch (InterruptedException e) {}
   System.out.println ("Endlich! Es ist vorbei.");
   }
}
```

9.8.5 Konten-Schieberei

1. Schreiben Sie eine ganz primitive Klasse `Konto`, die nur einen Kontostand verwaltet und einfache `get`-und `set`-Methoden zur Verfügung stellt. Der Konstruktor soll einen Anfangs-Kontostand übernehmen.

2. Schreiben Sie ein Klasse `Transaktion`, die als Thread laufen kann. Der Konstruktor übernimmt ein Feld von Konten, einen Betrag und eine Anzahl Transaktionen, die durchzuführen sind.

   Der eigentliche Thread-Code soll folgendes tun: Aus dem Feld von Konten sollen zwei Konten zufällig herausgesucht werden. Und es soll der im Konstruktor angegebene Betrag von dem einen zum anderen Konto übertragen werden. Das soll so oft passieren, wie im Konstruktor angegeben wurde. Damit das gewünschte Ergebnis erreicht wird, muss an vielen Stellen eine zufällige Anzahl von `yield ()` eingestreut werden.

3. Schreiben Sie eine Klasse `KontenSchieberei`, die nur aus einem `main`-Programm besteht, das folgendes tut:

   (a) Ein Feld von etwa 10 Konten wird angelegt.

   (b) Die Summe aller Konten wird bestimmt und ausgegeben.

   (c) Es werden jetzt etwa 100 Transaktions-Objekte angelegt und gestartet.

   (d) Es wird gewartet, bis alle Transaktionen zu Ende gelaufen sind.

   (e) Die Summe aller Konten wird bestimmt und ausgegeben.

4. Als Ergebnis wird erwartet, dass die Transaktionen sich gegenseitig stören und daher die Gesamtsumme nicht mehr stimmt. Wie kann man das reparieren?

9.8.6 Join-Beispiel

Im folgenden Beispiel wird ein Feld `feld` willkürlich mit `true` oder `false` belegt.

Der unten programmierte Thread `Zaehler` bestimmt in einem Bereich des Feldes von `anfang` bis `ende` die Anzahl der wahren und falschen Werte.

Es sollen nun mehrere Threads angelegt werden, die jeweils für einen Teil des Feldes für das Zählen verantwortlich sind. Das ganze Feld soll abgedeckt werden. Die Threads sollen gestartet werden. Das Hauptprogramm soll warten, bis alle Threads zu Ende gekommen sind, und dann das Resultat ausgeben.

```
import   java.util.*;
public class   JoinBeispiel
   {
   public static void   main (String[] arg)
      {
      int   groesse = 10000;
      boolean[]   feld = new boolean[groesse];
      for (int i = 0; i < groesse; i++)
         {
         feld[i] = Math.random () > 0.5 ? true : false;
         }
      ArrayList<Zaehler>   th = new ArrayList<Zaehler> ();
      // Aufgabe:
      // ...
      // Threads anlegen und in ArrayList th verwalten.
      // ...
      // Threads starten.
      // ...
      // Auf das Ende der Threads warten.
      // Ende der Aufgabe.
      int   tresultat = 0;
      int   fresultat = 0;
      for (Zaehler t :th)
         {
         tresultat = tresultat + t.tresultat;
         fresultat = fresultat + t.fresultat;
         }
      System.out.println ("Von " + groesse + " Werten sind " + tresultat
         + " true.");
      System.out.println ("Von " + groesse + " Werten sind " + fresultat
         + " false.");
      System.out.println ("Kontrolle: " + groesse + " = " + tresultat
         + " + " + fresultat);
      System.out.println ();
      }
   }
class Zaehler
   extends  Thread
   {
   boolean[]   feld;
   int   anfang;
   int   ende;
   int   tresultat;
   int   fresultat;
   public   Zaehler (boolean[] feld, int anfang, int ende)
      {
      this.feld = feld;
      this.anfang = anfang;
      this.ende = ende;
      }
   public void run ()
      {
      tresultat = 0;
      fresultat = 0;
      for (int i = anfang; i <= ende; i++)
         {
         if (feld[i]) tresultat = tresultat + 1;
         else fresultat = fresultat + 1;
         try
            {
            Thread.sleep ((long) (Math.random () * 10));
            }
            catch (InterruptedException e) { e.printStackTrace (); }
         }
      }
   }
```

9.8.7 Witz

In einer Stadt von n Personen erfindet eine Person einen Witz und erzählt ihn anderen Personen, die ihn nach folgenden Regeln weitererzählen:

1. Die Personen treffen zufällig aufeinander.
2. Wird der Witz einer Person erzählt, die ihn noch nicht kennt, dann erzählt auch sie den Witz weiter.
3. Wird der Witz einer Person erzählt, die ihn schon kennt, dann hören beide auf, den Witz weiterzuerzählen.

Wieviel Personen erfahren von dem Witz? Lösen Sie das Problem durch Simulation, indem Sie für jede Person ein Objekt anlegen, das als Witz-verbreitender Thread laufen kann.

9.8.8 Schweiß-Roboter mit Semaphoren

Bringen Sie das folgende Schweiss-Roboter-Programm zum Laufen, indem Sie eine Semaphor-Klasse erstellen und die beiden Threads für das Positionieren und Schweißen erstellen.

```
import java.util.*;
public class   SchweissRoboter
   {
   public static void   main (String[] argument)
      {
      SchweissRoboter   r = new SchweissRoboter ();
      new ArmThread (r).start ();
      new SchweissThread (r).start ();
      }
   public void uB (int n) { operation ("uB" + n); }
   public void kP (int n) { operation ("kP" + n); }
   public void uV (int n) { operation ("uV" + n); }
   public void kS (int n) { operation ("kS" + n); }
   private static Random   rand = new Random ();
   private void   operation (String opName)
      // Simuliert die Operationen des Roboters durch Bildschirmausgaben
      // mit unterschiedlich vielen Zeilen.
      {
      if (Thread.currentThread ().getName ().equals ("SchweissThread"))
         opName = "                  " + opName;
      int   j = rand.nextInt ();
      j = j < 0 ? -j : j;
      j = j % 4 + 1;
      System.out.println (opName + " Anfang"); Thread.yield ();
      for (int i = 0; i < j; i++)
         {
         System.out.println (opName + " ......"); Thread.yield ();
         }
      System.out.println (opName + " Ende. "); Thread.yield ();
      }
   }
```

9.8.9 Schweiß-Roboter ohne Semaphore

Programmieren Sie dieses Beispiel unter Verwendung des Monitor-Konzepts von Java, d.h. ohne Verwendung von Semaphoren.

Hinweis: Schreiben Sie für die kritischen Roboter-Operationen eigene Methoden (in der Klasse `SchweissRoboter`), wobei diese Methoden `synchronized` sind und mit geigneten `wait` und `notify` zu versehen sind. Mit einem Boolean wird verwaltet, welche Operation gerade dran ist.

9.8.10 Timeout für Ringpuffer-Methoden

1. Schreiben Sie für die Methoden `enqueue (...)` und `dequeue ()` überladene Versionen, bei denen man ein Timeout angeben kann.
2. Priorisierter Zugriff auf Ringpuffer: Die Methoden `enqueue (...)` und `dequeue ()` sollen bezüglich einander priorisierbar sein. D.h. wenn `enqueue (...)` Priorität über `dequeue ()` hat, dann wird garantiert, dass bei durch gegenseitigen Ausschluss blockierten Tasks, eine `enqueue`-Task Vorrang hat.

9.8.11 Problem der speisenden Philosophen

Ein berühmtes Synchronisations-Problem ist das Problem der n Philosophen, die an einem runden Tisch sitzen und zyklisch immer wieder denken und speisen. Für jeden Philosophen ist *eine* Gabel ausgelegt. Um aber speisen zu können, benötigt ein Philosoph auch die Gabel seines rechten Nachbarn. Das bedeutet, dass nicht alle Philosophen gleichzeitig speisen können. (Die Philosphen wechseln nicht ihre Plätze.) Es gilt nun zu verhindern, dass zwei Philosophen dieselbe Gabel greifen. Natürlich darf auch keine Verklemmung auftreten.

1. Lösen Sie dieses Problem mit dem Monitor-Konzept von Java.
2. Lösen Sie dieses Problem mit einfachen Semaphoren.
3. Lösen Sie dieses Problem mit Semaphorgruppen.

9.8.12 Stoppuhr

Entwickeln Sie eine Stoppuhr.

9.8.13 Barkeeper

Der Barkeeper schenkt an die Kunden Whiskey oder Wodka aus. Wenn er gerade die Whiskeyflasche in der Hand hat, dann schenkt er Wiskey aus, solange es Kunden gibt, die Whiskey möchten. Wenn niemand mehr Whiskey möchte, dann wartet er noch 2 Sekunden, ob nicht doch noch einer Whiskey möchte. Wenn das der Fall ist, wird auch noch der Whiskeykunde bedient und eventuell noch weitere plötzlich aufgetauchte Whiskeykunden.

Ebenso verhält es sich mit dem Wodka.

Wenn niemand mehr Whiskey oder Wodka möchte, dann spült er ein paar Gläser. Danach schaut er wieder, ob es Kunden gibt.

Jedesmal, wenn er eingeschenkt hat oder Gläser gespült hat, atmet er tief durch.

Programmieren Sie die unten zur Verfügung gestellte Klasse `Barkeeper` so, dass er sich wie oben beschrieben verhält.

Die Klasse `Kunde` sollte dabei nicht verändert werden.

```
import   java.util.*;
public class   BarkeeperHaupt
   // Implementation über Algorithmic State Machine
   {
   public static void   main (String[] argument)
      {
      Barkeeper   bk = new Barkeeper ("Bartender");
      Kunde k1 = new Kunde ("Boris", "          ", bk);
      Kunde k2 = new Kunde ("Fedor", "                    ", bk);
      Kunde k3 = new Kunde ("Ivano", "                              ", bk);
      }
   public static void   jemandTutWas (String offset, String jemand, String was)
      {
      pausiere ();
      System.out.println (offset + jemand + " " + was);
      }
   private static java.util.Random  r = new java.util.Random ();
   public static int zufall () { return BarkeeperHaupt.zufall (6); }
   public static int zufall (int n)
      {
      return Math.abs (r.nextInt ()) % n + 1;
      }
   public static void   pausiere ()
      {
      try
         {
         Thread.sleep (BarkeeperHaupt.zufall (200));
         }
         catch (InterruptedException e) { }
      }
   }  // end BarkeeperHaupt

class Barkeeper
   {
   private String name;
   public   Barkeeper (String name)
      {
      this.name = name;
      }
   public void whiskey (String name, String offset)
      {
      BarkeeperHaupt.jemandTutWas ("", this.name, "nimmt Whiskeyflasche.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "schenkt " + name + " Whiskey ein.");
      BarkeeperHaupt.jemandTutWas (offset, name, "bekommt Whiskey.");
      BarkeeperHaupt.jemandTutWas (offset, name, "hört Prost Whiskey.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "schaut nach Whiskeykunden.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "stellt Flasche Whiskeyflasche ab.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "spült paar Gläser.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "atmet tief durch.");
      }
   public void wodka (String name, String offset)
      {
      BarkeeperHaupt.jemandTutWas ("", this.name, "nimmt Wodkaflasche.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "schenkt " + name + " Wodka ein.");
```

```
      BarkeeperHaupt.jemandTutWas (offset, name, "bekommt Wodka.");
      BarkeeperHaupt.jemandTutWas (offset, name, "hört Prost Wodka.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "schaut nach Wodkakunden.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "stellt Flasche Wodkaflasche ab.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "spült paar Gläser.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "atmet tief durch.");
      }

   }  // end Barkeeper

class Kunde extends Thread
   {
   private String name;
   private String offset;
   private Barkeeper bk;
   public   Kunde (String name, String offset, Barkeeper bk)
      {
      this.name = name;
      this.offset = offset;
      this.bk = bk;
      this.start ();
      }
   int n = 0;
   public void run ()
      {
      java.util.Random  r = new java.util.Random ();
      BarkeeperHaupt.jemandTutWas (offset, name, "kommt.");
      while (true)
         {
         n++;
         for (int i = 0; i < BarkeeperHaupt.zufall (3); i++)
            {
            BarkeeperHaupt.jemandTutWas (offset, name + n, "wählt.");
            }
         switch (BarkeeperHaupt.zufall (4))
            {
            case 1:
               BarkeeperHaupt.jemandTutWas (offset, name + n, "bestellt Whiskey.");
               bk.whiskey (name + n, offset);
               break;
            case 2:
               BarkeeperHaupt.jemandTutWas (offset, name + n, "bestellt Wodka.");
               bk.wodka (name + n, offset);
               break;
            case 3:
               BarkeeperHaupt.jemandTutWas (offset, name + n, "bestellt Wodka und Whiskey.");
               Thread   t1 = new Thread ()
                  {
                  public void run ()
                     {
                     bk.wodka (name + n, offset);
                     }
                  };
               t1.start ();
               bk.whiskey (name + n, offset);
               try { t1.join (); } catch (InterruptedException e) {}
               break;
            case 4:
               BarkeeperHaupt.jemandTutWas (offset, name + n, "bestellt Whiskey und Wodka.");
               Thread   t2 = new Thread ()
                  {
                  public void run ()
                     {
                     bk.whiskey (name + n, offset);
                     }
                  };
               t2.start ();
               bk.wodka (name + n, offset);
               try { t2.join (); } catch (InterruptedException e) {}
               break;
            }
```

```
        BarkeeperHaupt.jemandTutWas (offset, name + n, "hat genug.");
        BarkeeperHaupt.jemandTutWas (offset, name + n, "geht.");
        BarkeeperHaupt.jemandTutWas (offset, name + n, "kommt wieder.");
        }
    }
  }
```

Kapitel 10

Streams

Ein- und Ausgabe gehören zu jedem Programm. Java verwendet dafür das Konzept des **Datenstroms** ***(data stream)*** und stellt dafür in dem Paket `java.io` verschiedene Klassen zur Verfügung.

Ein Datenstrom steht für einen definierten Transport von Daten von einer **Datenquelle (Datenerzeuger**, ***source)*** zu einer **Datensenke (Datenverbraucher**, ***sink)***. Bei einem **Eingabestrom** ***(input stream)*** liest das Programm die von einem äußeren Erzeuger transportierten Daten ein. Bei einem **Ausgabestrom** ***(output stream)*** gibt das Programm die Daten zum Transport an einen äußeren Verbraucher aus.

In Java ist die transportierte Einheit ein Byte. Dementsprechend werden mit den grundlegenden Methoden nur Byte gelesen oder geschrieben.

Alle Ströme unterstützen die Methode `close ()`, die man aufrufen sollte, wenn ein Strom nicht mehr benötigt wird, damit eventuell beanspruchte Betriebssystem-Resourcen frühzeitig freigegeben werden.

10.1 Ein- und Ausgabeströme

Alle Eingabeströme bzw. Ausgabeströme (ausgenommen `RandomAccessFile`) leiten sich von den abstrakten Stream-Basis-Klassen `InputStream` bzw. `OutputStream` ab.

Ein- und Ausgabeströme verhalten sich normalerweise reziprok, indem es zu den meisten Eingabeströmen einen entsprechenden Ausgabestrom gibt mit reziproken `read`- bzw. `write`-Methoden, die so gemacht sind, dass, wenn ein Erzeuger eine `write`-Methode aufruft, ein Verbraucher mit der entsprehenden `read`-Methode genau dieselben Daten lesen kann.

10.1.1 Klasse `InputStream`

Mit der Methode `int read ()` wird ein einziges Byte aus dem Eingabestrom entfernt und als `int` zurückgegeben! Dieses `int` enthält die Bitkombination des gelesenen Byte. Wenn das Resultat als `byte` weiterverwendet werden soll, muss es gecasted werden, um Warnungen zu vermeiden.

```
byte  b = (byte) (s.read ());
```

Wenn kein Byte gelesen werden konnte, wird `-1` zurückgegeben.

Mit der Methode `int read (byte[] b)` werden maximal `b.length` Byte gelesen und in das Feld `b` geschrieben. Zurückgegeben wird die Anzahl der gelesenen Byte, was wichtig ist, wenn das Ende des Stroms vorher erreicht wird.

Mit `int read (byte[] b, int von, int anz)` werden maximal `anz` Byte gelesen und an der entsprechenden Stelle von `b` beginnend mit `b[von]` gespeichert.

Alle `read`-Methoden warten ***(block)***, bis ihre Einleseforderung befriedigt ist oder das Ende des Stroms erreicht ist. Mit der Methode

```
int available ()
```

kann festgestellt werden, wieviele Byte gerade lesbar sind.

Mit `long skip (long n)` kann man `n` Bytes überspringen. Gibt tatsächlich gesprungene Bytes zurück.

Das Objekt `System.in` (Standard-Eingabe) ist vom Typ einer Klasse, die sich von `InputStream` ableitet.

10.1.2 Klasse `OutputStream`

Die drei Methoden

```
void  write (int b); // Schreibt ein Byte!
void  write (byte[] b);
void  write (byte[] b, int von, int anz);
```

schreiben Byte in den Ausgabestrom und warten ***(block)*** bis alles geschrieben ist.

Mit `flush ()` wird eventuell erreicht, dass die Daten schneller ihren Verbraucher erreichen, z.B. irgendwelche Puffer geleert werden und die Daten z.B. wirklich auf dem Bildschirm ausgegeben werden.

10.2 Byte-Felder

Mit den Klassen `ByteArrayInputStream` und `ByteArrayOutputStream` kann von Byte-Feldern gelesen bzw. geschrieben werden. Beide Klassen erben von den jeweiligen Stream-Basis-Klassen, womit deren Methoden zur Verfügung stehen.

Objekte der Klasse `ByteArrayInputStream` werden mit einem Byte-Feld oder einem Teil davon initialisiert.

```
byte[]   feld = new byte[256];
// fülle feld mit sinnvollen Daten
ByteArrayInputStream s1 = new ByteArrayInputStream (feld);
// oder
InputStream s2 = new ByteArrayInputStream (feld, 47, 127);
```

Die Methode `reset ()` setzt den Lesezeiger immer an den Anfang des Streams zurück.

Bei der Klasse `ByteArrayOutputStream` sind die geschriebenen Daten als Byte-Feld über die Methode

```
public synchronized byte[] toByteArray ()
```

als Byte-Feld und über die Methoden

```
public String toString () oder
public String toString (int oberesByte)
```

als String zugänglich, wobei das obere Byte eines String-Chars mit `oberesByte` belegt wird.

Objekte der Klasse `ByteArrayOutputStream` können ohne Parameter oder mit einem `int` initialisiert werden, das die Größe des Streams vorgibt. Die Methode `reset ()` setzt den Stream auf den Anfang zurück. Alle bis dahin geschriebenen Daten werden verworfen. Die Methode `int size ()` gibt die Anzahl der geschriebenen Bytes zurück.

10.3 Datei-Ströme

Mit den Klassen `FileInputStream` und `FileOutputStream` können Dateien zum Lesen und Schreiben als Streams geöffnet werden. Ihre Konstruktoren nehmen als Argument entweder einen

`String` (Dateinamen) oder
ein Objekt vom Typ `File`
oder ein Objekt vom Typ `FileDescriptor`.

- `FileInputStream (String dateiname)`
- `FileInputStream (File datei)`
- `FileInputStream (FileDescriptor fd)`
- `FileOutputStream` hat entsprechende Konstruktoren.

Auf die Typen `File` und `FileDescriptor` gehen wir in einem späteren Abschnitt ein. Der übergebene String muss einen Dateinamen mit eventueller Pfadspezifikation enthalten, wobei die Unix-Syntax vewendet wird, z.B. `"/home/klaus/projekte/daten"`.

Mit `getFD ()` (Klasse `File`) bekommt man das `FileDescriptor`-Objekt.

10.4 Filter-Klassen

Die Klassen `FilterInputStream` und `FilterOutputStream` erben von den Stream-Basis-Klassen *und* werden mit einer Streamklasse initialisiert. Die geerbten Methoden werden so überschrieben, dass sie die entsprechenden Methoden der übergebenen Streamklasse aufrufen. Da dies noch nicht viel Sinn macht, können von diesen Klassen außerhalb des Pakets `java.io` keine Objekte angelegt werden, da der Konstruktor `protected` ist.

Von Subklassen allerdings können Objekte angelegt werden. Diese Subklassen können die Methoden sinnvoll überschreiben. Folgendes Beispiel soll das demonstrieren. Wir wollen folgende Filter schreiben:

1. Filter, das jedes Leerzeichen durch ein "e" ersetzt.
2. Filter, das jedes "e" entfernt.

Diese Filter werden nacheinander auf einen `ByteArrayInputStream` angewendet.

```
import java.io.*;
public class   Filter1
   {
   static final int  Blank = ' ';
   static final int  KleinE = 'e';
   public static void   main (String[] argument)
      {
      String   text = "Das Wetter ist heute besonders schön.";
      byte[]   feld = new byte[text.length ()];
      for (int i = 0; i < feld.length; i++)
         feld[i] = (byte)text.charAt (i);
      InputStream s = new ByteArrayInputStream (feld);
      gibStreamAus (s);
      InputStream s1 = new  FilterBlankZuKleinE (s);
      gibStreamAus (s1);
      InputStream s2 = new  FilterEntferntKleinE (s);
      gibStreamAus (s2);
      InputStream s3 = new  FilterBlankZuKleinE (s2);
      gibStreamAus (s3);
      InputStream s4 = new  FilterEntferntKleinE (s1);
      gibStreamAus (s4);
      }
   static void gibStreamAus (InputStream s)
      {
      int   c;
      try
         {
         while ( (c = s.read ()) != -1) System.out.write (c);
         s.reset (); // damit s wieder verwendet werden kann.
         }
      catch (IOException e) {}
      System.out.write ('\n');
      }
   }
class FilterBlankZuKleinE extends FilterInputStream
   {
   public   FilterBlankZuKleinE (InputStream s)
      {
      super (s);
      }
```

```
   public int  read () throws IOException
      {
      int   c = super.read ();
      if (c == Filter1.Blank) return Filter1.KleinE;
      else return c;
      }
   public int  read (byte[] b, int off, int len) throws IOException
      {
      int   c = super.read (b, off, len);
      for (int i = 0; i < c + off; i++)
         {
         if (b[i] == Filter1.Blank) b[i] = Filter1.KleinE;
         }
      return c;
      }
   }
class FilterEntferntKleinE extends FilterInputStream
   {
   public   FilterEntferntKleinE (InputStream s)
      {
      super (s);
      }
   public int  read () throws IOException
      {
      int   c = super.read ();
      while (c == Filter1.KleinE) c = super.read ();
      return c;
      }
   public int  read (byte[] b, int off, int len) throws IOException
      {
      int   c = super.read (b, off, len);
      for (int i = off; i < c + off; i++)
         {
         if (b[i] == Filter1.KleinE)
            {
            for (int j = i; j < c - 1 + off; j++)
               {
               b[j] = b[j + 1];
               }
            int c2 = super.read ();
            if (c2 != -1)
               {
               b[c - 1 + off] = (byte) c2;
               }
            else
               {
               c = c - 1;
               }
            }
         }
      return c;
      }
   }
```

Dies Programm liefert als Resultat:

```
Das Wetter ist heute besonders schön.
DaseWettereisteheuteebesonderseschön.
Das Wttr ist hut bsondrs schön.
DaseWttreistehutebsondrseschön.
DasWttristhutbsondrsschön.
```

Diese Filter-Streams können beliebig verschachtelt werden.

Bemerkung: `FilterInputStream` leitet die `read`-Methoden standardmäßig an den angeschlossenen `InputStream` weiter. Eine `FilterInputStream`-Klasse sollte daher die Methoden

```
public int  read () throws IOException
und
public int  read (byte[] b, int off, int len) throws IOException
```

überschreiben. Die Methode

```
public int  read (byte[] b) throws IOException
```

ist in `FilterInputStream` mit einer der beiden anderen Methoden implementiert und muss daher nicht extra implementiert werden.

Von `FilterInputStream` bzw. von `FilterOutputStream` leiten sich folgende Stream-Klassen ab:

10.4.1 `BufferedInputStream` und `BufferedOutputStream`

Diese Streams ermöglichen gepufferte Ein- und Ausgabe. Ausgabe erfolgt, wenn `flush ()` aufgerufen wird oder wenn der Puffer voll ist. Die Größe des Puffers kann im Konstruktor angegeben werden.

```
public   BufferedInputStream (InputStream s);
public   BufferedInputStream (InputStream s, int groesse);
public   BufferedOutputStream (OutputStream s);
public   BufferedOutputStream (OutputStream s, int groesse);
```

`BufferedInputStream` ist die einzige Klasse, die `mark` und `reset` korrekt implementiert. Mit

```
public synchronized void   mark (int leseGrenze)
```

kann eine Marke im Stream gesetzt werden, auf die mit der Methode

```
public synchronized void   reset () throws IOException
```

sicher zurückgesprungen werden kann, wenn höchsten `leseGrenze` weitere Byte eingelesen wurden.

Damit und durch Effizienzsteigerungen bei der Ein- und Ausgabe sind die gepufferten Streams wertvolle Hilfsmittel.

10.4.2 DataInputStream und DataOutputStream

Diese Streams implementieren die Schnittstelle `DataInput` bzw. `DataOutput`, die Methoden zur Ein- bzw. Ausgabe der *Bit-Sequenzen* (d.h. keine formatierte oder textuelle Ausgabe als Zeichenkette) von Standardtypen definieren.

Die Methoden von `DataInputStream` werfen eine `EOFException`, wenn das Ende des Streams erreicht ist. Damit wird die Abfrage auf Dateiende wesentlich eleganter:

```
try
   {
   while (true)
      {
      int   i = s.readInt ();
      // verarbeite i
      }
   }
catch (EOFException e)
   {
   // Stream-Ende wurde erreicht.
   // Da muss man nichts tun.
   }
catch (IOException e)
   {
   // Eingabefehler ist aufgetreten.
   // Da muss man irgendetwas machen.
   }
```

10.4.3 LineNumberInputStream

Mit dieser Klasse kann jederzeit die aktuelle Zeilennummer ermittelt oder gesetzt werden, was für Compiler und Editoren ein wichtiges Hilfsmittel ist.

10.4.4 PushbackInputStream

Mit dieser Klasse kann das zuletzt gelesene Zeichen wieder in den Stream zurückgegeben werden.

10.5 PipedInputStream und PipedOutputStream

Mit diesen Streams lässt sich sehr einfach eine Pipe-Verbindung zwischen zwei Threads aufbauen:

```
PipedInputStream  ein;
PipedOutputStream aus;
aus = new PipedOutputStream ();
```

```
ein = new PipedInputStream ();
aus.connect (ein);
```

oder

```
PipedInputStream  ein = new PipedInputStream ();
PipedOutputStream aus = new PipedOutputStream (ein);
```

oder

```
PipedOutputStream aus = new PipedOutputStream ();
PipedInputStream  ein = new PipedInputStream (aus);
```

Ein Thread kann nun nach `ein` schreiben, während ein anderer Thread von `aus` liest.

Es ist nicht möglich einen Erzeuger (`aus`) mit mehreren Verbrauchern (`ein1`, `ein2` usw) zu verknüpfen.

10.6 SequenceInputStream

Mit dieser Klasse kann man zwei oder mehrere Streams hintereinanderhängen, so dass man die Streams als einen Stream behandeln kann. Die beiden Konstruktoren sind:

```
public   SequenceInputStream (InputStream s1, InputStream s2);
public   SequenceInputStream (Enumeration e);
```

Mit dem letzteren Konstruktor ist die Konkatenierung eines ganzen Vektors von Streams möglich:

```
InputStream s1, s2, s3;
// ---
Vector   v = new Vector ();
v.addElement (s1);
v.addElement (s2);
v.addElement (s3);
SequenceInputStream  s = new SequenceInputStream (v.elements ());
```

Die Methode `elements ()` der Klasse `Vector`, gibt die Elemente eines `Vector`-Objekts als `Enumeration` zurück.

10.7 StringBufferInputStream

Diese Klasse ist ähnlich zu `ByteArrayInputStream`, nur dass die Objekte mit einem `String` initialisiert werden. Besser ist dafür die Klasse `StringReader` (siehe unten).

10.8 RandomAccessFile

Diese Klasse implementiert `DataInput` und `DataOutput` und wird verwendet, um an beliebiger Stelle in einer Datei zu lesen oder zu schreiben. Sie hat zwei Konstruktoren

```
public   RandomAccessFile (String dateiname, String modus);
public   RandomAccessFile (File datei, String modus);
```

`modus` kann entweder `"r"` oder `"rw"` sein.

Mit der Methode

```
public long length ()
```

bekommt man die Länge der Datei. Mit

```
public void seek (long pos)
```

kann man den Lese- oder Schreibzeiger auf eine Position `pos` in der Datei setzen.

10.9 File

Objekte der Klasse `File` repräsentieren Dateien oder Verzeichnisse. Es werden die typischen Betriebssystem-Funktionen eines Datei-Managers als Methoden zur Verfügung gestellt.

10.10 char-Ströme

Über die Klassen `InputStream` und `OutputStream` werden `byte`-Ströme unterstützt. Seit JDK1.1 werden durch die Basisklassen

`Reader` und `Writer`

auch 16-Bit Unicode `char`-Streams unterstützt. Sie unterstützen etwa dieselben Operationen wie die `byte`-Streams. Die Namen der abgeleiteten Klassen ergeben sich durch Austausch von `InputStream` mit `Reader` und `OutputStream` mit `Writer`.

Zwei Klassen bilden eine Brücke zwischen diesen beiden Strömen:

`InputStreamReader`: Liest `byte` und liefert `char`
`OutputStreamWriter`: Schreibt `byte` und liest `char`

`char`-Ströme sind potentiell effizienter wegen gepuffertem Lesen und Schreiben und besserem Locking-Schema.

Folgende Tabelle zeigt die `char`-Stream-Klassen (Einrückung bedeutet Ableitung):

```
Reader                    Writer
  BufferedReader            BufferedWriter
    LineNumberReader
  CharArrayReader           CharArrayWriter
  InputStreamReader         OutputStreamWriter
    FileReader                FileWriter
  FilterReader               FilterWriter
    PushbackReader
  PipedReader               PipedWriter
  StringReader               StringWriter
                            PrintWriter
```

Eine sehr nützliche Methode von `BufferedReader` ist:

```
public String  readLine () throws IOException;
```

10.10.1 `PrintStream` und `PrintWriter`

Die Klassen `PrintStream` und `PrintWriter` bieten mit ihren `print`- und `println`-Methoden die Möglichkeit zur textuellen Ausgabe von Standard- *und* Referenztypen. Dabei werden die Objekte in einen `String` konvertiert, indem die Methode `toString ()` aufgerufen wird. Wenn eine Klasse die Methode `toString ()` irgendwie sinnvoll implementiert, kann jedes Objekt textuell ausgegeben werden.

Die Objekte `System.out` (Standard-Ausgabe) und `System.err` (Standard-Fehler-Ausgabe) sind vom Typ `PrintStream`.

`PrintStream` benutzt die Default-Codierung des jeweiligen Hosts, indem es einen `OutputStreamWriter` inkorporiert, durch den alle `char` gehen, um `byte` für die Ausgabe zu produzieren. Die `println`-Methode benutzt daher auch den Default-Zeilen-Terminator, der durch die System-Eigenschaft `line.separator` definiert wird. `PrintStream` ist bequem zur Ausgabe von Debugging-Information.

Für eine textuelle Ausgabe, die sich nicht auf irgendeine Default-Codierung verlässt, sollte die Klasse `PrintWriter` verwendet werden.

Die textuellen Repräsentationen der Standardtypen genügen normalerweise nicht den Ansprüchen einer formatierten Ausgabe. Das API von Java bietet darüberhinaus keine weiteren Möglichkeiten ?! Das muss der Anwender offenbar selbst tun, indem er die Klassen der Standardtypen erweitert und insbesondere die Methode `toString ()` überschreibt, so dass sie die gewünschte Formatierung leistet.

10.11 Übungen

Übung zur Klasse `Klausurnote`:

1. Überschreibe die Methode `toString` so, dass die Note mit `print` als zum Beispiel
   `Chemie befriedigend (2,8)`
   ausgegeben wird.
2. Schreib die Methode `zahl ()` "Java-like".

Kapitel 11

Netzwerk-Programmierung

In dem Paket `java.net` sind Klassen definiert, die eine mächtige Infrastruktur für Netzwerk-Programmierung bilden. Viele dieser recht komplizierten Klassen werden in normalen Anwendungen nicht benötigt. Wir beschränken uns hier auf die üblicherweise benötigten Klassen.

11.1 URL

Die Klasse `URL` repräsentiert einen ***Uniform Resource Locator*** des Internets. Mit dieser Klasse kann man bequem Objekte herunterladen, oder man kann einen Stream zum Lesen einer Datei öffnen.

Mit dem folgenden Beispiel kann unter Verwendung der Methode `getContent ()` ein Text oder ein Gif-Bild von einer URL geladen und dargestellt werden.

```
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import java.net.*;
import java.util.*;

public class   URLinhalt extends Canvas
   {
   public static void   main (String[] argument)
      {
      URLinhalt   ui = new URLinhalt ();
      switch (argument.length)
         {
         case 2:
            try
               {
               switch (ui.arg0 = Integer.parseInt (argument[0]))
                  {
```

```
                case 1:
                   System.out.println (ui.holeText (argument[1]));
                   break;
                case 2:
                   Frame f = new Frame ("GIF-Bild");

                   ui.bild = ui.holeBild (argument[1], ui);
                   System.out.println ("Höhe: " + ui.bild.getHeight (f)
                      + " Breite :" + ui.bild.getWidth (f));
                   f.add ("Center", ui);
                   f.setSize (200, 200);
                   f.setVisible (true);
                   break;
                default: throw new NumberFormatException ();
                }
             break;
             }
          catch (NumberFormatException e)
             {
             System.err.println ("URLinhalt.main: " + e);
             }
          catch (MalformedURLException e)
             {
             System.err.println ("URLinhalt.main: " + e);
             }
          catch (IOException e)
             {
             System.err.println ("URLinhalt.main: " + e);
             }
       default:
          System.err.println ("Benutzung: 2 Argumente");
          System.err.println ("   1): 1 oder 2 für Text oder Gif-Bild");
          System.err.println ("   2): URL-Adresse");
          break;
       }
    }

 public String  holeText (String urlAdresse)
    throws MalformedURLException, IOException
    {
    URL   url = new URL (urlAdresse);
    printURLinfo (url);
    System.out.println ("File:    " + url.getFile ());
    System.out.println ("Host:    " + url.getHost ());
    System.out.println ("Port:    " + url.getPort ());
    System.out.println ("protocol:" + url.getProtocol ());
    System.out.println ("Ref :    " + url.getRef ());
    return (String)(url.getContent ());
    }
```

```
public Image   holeBild (String urlAdresse, Component c)
   throws MalformedURLException, IOException
   {
   URL   url = new URL (urlAdresse);
   printURLinfo (url);
   System.out.println ("File:    " + url.getFile ());
   System.out.println ("Host:    " + url.getHost ());
   System.out.println ("Port:    " + url.getPort ());
   System.out.println ("protocol:" + url.getProtocol ());
   System.out.println ("Ref :    " + url.getRef ());
   return c.createImage ((ImageProducer)(url.getContent ()));
   }

public void printURLinfo (URL url)
   {
   try
      {
      URLConnection  uc = url.openConnection ();
      System.err.println (url.toExternalForm () + ":");
      System.err.println
         ("  Inhaltstyp     : " + uc.getContentType ());
      System.err.println
         ("  Länge          : " + uc.getContentLength ());
      System.err.println ("  Datum          : "
         + new Date (uc.getLastModified ()));
      System.err.println
         ("  Expiration     : " + uc.getExpiration ());
      System.err.println
         ("  Verschlüsselung: " + uc.getContentEncoding ());

      if (arg0 == 1)
         {
         System.err.println ("Erste fünf Zeilen:");
         BufferedReader ein =
            new BufferedReader (
            new InputStreamReader (uc.getInputStream ()));
         for (int i = 1; i <= 5; i++)
            {
            String   zeile = ein.readLine ();
            if (zeile == null) break;
            System.err.println ("Zeile " + i + ": " + zeile);
            }
         }
      }
   catch (IOException e)
      {
      System.err.println ("URLinhalt.printURLinfo: " + e);
      }
```

```
      }

   public void paint (Graphics g)
      {
      g.drawImage (bild, 0, 0, this);
      }

   Image bild;
   int   arg0 = 0;
   }
```

Dieses Programm kann man z.B. anwenden mit:

```
java URLinhalt 1 http://www.ba-stuttgart.de/~kfg/dummy.java
```

oder

```
java URLinhalt 2 http://www.ba-stuttgart.de/~kfg/images/kfg.gif
```

oder

```
java URLinhalt 2 file:/var/home/kfg/public-html/images/kfg.gif
```

Die Methode `getContent` benutzt einen Content-Handler. Der Java-Content-Handler kann nur `gif` oder `plain` Dateien verarbeiten. `plain` Dateien müssen die Erweiterung `.java` haben.

Als Applet:

```
import java.awt.*;
import java.applet.*;
import java.net.*;
import java.io.*;
import java.util.*;

public class   URLApplet extends Applet
   {
   Image bild;
   String[] text;
   Exception e1;
   Exception e2;
   URL   url1;
   URL   url2;

   public void init ()
      {
      System.err.println ("init: 1*"); System.err.flush ();
      showStatus ("init: 1*");
      try
         {
         // 1.)
```

```
         bild = getImage (url1 = URLApplet.class.getResource ("kfg.gif"));
            // Das funktioniert nicht, sollte aber. kfg.gif ist in jar-File
            // Funktioniert mit Explorer auf MacIntosh
            // Funktioniert nicht mit Netscape Communicator 4.79

         // 2.)
         //bild = getImage (getDocumentBase (), "kfg.gif");
            // Das funktionier nicht, kfg.gif ist in jar-File
            // Das muß auch nicht funktionieren.

         // 3.)
         //bild = getImage (getDocumentBase (), "images/kfg.gif");
            // Das funktioniert auch mit Netscape 4.79.
            // kfg.gif ist im Directory images
/*
         // Andere Versuche:
         bild = getImage (
            new URL (
               getCodeBase (), "kfg.gif"));
//             "http://" + getCodeBase ().getHost () + "~kfg/kfg.gif"));
//             "kfg.gif"));
//             "http://www.ba-stuttgart.de/~kfg/images/kfg.gif"));
//             "http://www.ba-stuttgart.de/~kfg/kfg.gif"));
//             "file:/home/blackhole.itstaff/kfg/public-html/images/kfg.gif"));
*/
         }
      catch (Exception e1)
         {
         System.err.println ("init: " + e1);
         }
      try
         {
         BufferedReader br = new BufferedReader (new InputStreamReader (
            URLApplet.class.getResourceAsStream ("kfg.txt")));
         url2 = URLApplet.class.getResource ("kfg.txt");
         Vector   v = new Vector ();
         String   s;
         while ((s = br.readLine ()) != null) v.addElement (s);
         text = new String[v.size ()];
         for (int i = 0; i < v.size (); i++) text[i] = (String) v.elementAt (i);
         }
      catch (Exception e2)
         {
         System.err.println ("init: " + e2);
         }
      System.err.println ("init: 2*"); System.err.flush ();
      showStatus ("init: 2*");
      }
```

```
   public void paint (Graphics g)
      {
      super.paint (g);
      g.drawString ("Das Applet läuft erstmal. Jetzt müßte der Text kommen 2:",
         10, 10);
      if (text != null)
         for (int i = 0; i < text.length; i++)
            g.drawString (text[i], 20, 10 + 10 * (i + 1));
      else g.drawString ("text ist null " + e2, 10, 20);
      if (bild != null)
         g.drawImage (bild, 20, 40 + 10 * (text.length + 1),
            bild.getWidth (this),
            bild.getHeight (this),
            this);
      else g.drawString ("bild ist null " + e1 + " URL1: " + url1
         + " URL2: " + url2, 10, 30);
      }

   }
```

Das nächste Beispiel öffnet mit `openStream ()` einen Stream, mit dem man von der URL lesen kann:

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

public class   URLstrom
   {
   public static void   main (String[] argument)
      {
      switch (argument.length)
         {
         case 1:
            try
               {
               URL   url = new URL (argument[0]);
               InputStream ein = url.openStream ();
               int   i;
               while ((i = ein.read ()) != -1)
                  System.out.write (i);
               break;
               }
            catch (MalformedURLException e)
               {
               System.err.println ("URLstrom.main: " + e);
               }
```

```
            catch (IOException e)
               {
               System.err.println ("URLstrom.main: " + e);
               }
         default:
            System.err.println ("Benutzung: 1 Argument");
            System.err.println ("  URL-Adresse von Text-Datei");
            break;
         }
      }
   }
```

Einen Stream zum Schreiben kann man mit einem Objekt der Klasse `URL` *nicht* öffnen.

Anwendung:
`java URLstrom http://www.ba-stuttgart.de/~kfg/dummy.plain`
`java URLstrom file:/var/home/kfg/public-html/dummy.plain`

11.2 URLconnection

Mit dieser Klasse kann man einen Stream zum Lesen oder Schreiben einer Datei öffnen. Dabei wird mit der `URL`-Methode `openConnection ()` ein Objekt der Klasse `URLconnection` erzeugt, mit dem man schreiben und lesen kann. Im folgenden Beispiel kopieren wir eine Datei.

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

public class   Kopieren
   {
   public static void   main (String[] argument)
      {
      switch (argument.length)
         {
         case 2:
            try
               {
               URL   url1 = new URL (argument[0]);
               URLConnection  verb1 = url1.openConnection ();
               verb1.setDoInput (true);
               InputStream ein = verb1.getInputStream ();

               URL   url2 = new URL (argument[1]);
               URLConnection  verb2 = url2.openConnection ();
               verb2.setDoOutput (true);
               OutputStream   aus = verb2.getOutputStream ();
```

```
                int   i;
                while ((i = ein.read ()) != -1)
                   {
                   System.out.write (i);
                   aus.write (i);
                   }
                break;
                }
             catch (MalformedURLException e)
                {
                System.err.println ("Kopieren.main: " + e);
                }
             catch (UnknownServiceException e)
                {
                System.err.println ("Kopieren.main: " + e);
                }
             catch (IOException e)
                {
                System.err.println ("Kopieren.main: " + e);
                }
          default:
             System.err.println ("Benutzung: 2 Argumente");
             System.err.println ("   1): URL-Adresse von Lese-Datei");
             System.err.println ("   2): URL-Adresse von Schreib-Datei");
             break;
          }
       }
   }
```

Die Klasse `URLconnection` bietet sehr viele Methoden an, um Informationen über eine Datei zu ermitteln.

Anwendung:

```
java URLstrom http://www.ba-stuttgart.de/~kfg/dummy.plain
   http://www.ba-stuttgart.de/~kfg/dummy
```

11.3 Kommunikation über Datagramme

Das Verschicken von Datagrammen ***UDP – universal ("unreliable") datagram protocol*** ist schnell, aber unzuverlässig, da nicht garantiert ist, dass ein Datagramm ankommt. Ferner muss die Reihenfolge der abgeschickten Datagramme nicht der Empfangsreihenfolge entsprechen. Datagramme sind nützlich, wenn die Kommunikation nicht einem Stream-Modell entspricht.

Java bietet für diese Kommunikation zwei Klassen an: `DatagramPacket` und `DatagramSocket`.

Das erste Beispiel zeigt, wie ein Datagramm versendet wird. Dazu wird ein `DatagramPacket` mit den zu versendenen Daten, der Länge der Daten, dem Empfänger und dessen Port erzeugt.

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

public class   DatagrammSenden
   {
   public static final int DEFAULTPORT = 6010;

   public static void   main (String[] argument)
      {
      int   entfernterPort = DEFAULTPORT;

      switch (argument.length)
         {
         case 2:
            try { entfernterPort = Integer.parseInt (argument[1]); }
            catch (NumberFormatException e) { }
         case 1:
            try
               {
               InetAddress adresse =InetAddress.getByName (argument[0]);
               byte[]   botschaft =
                  {(byte) 'G', (byte) 'u', (byte) 't', (byte) 'e', (byte) 'n',
                  (byte) ' ', (byte) 'T', (byte) 'a', (byte) 'g'};
               DatagramPacket paket =
                  new DatagramPacket (botschaft, botschaft.length,
                     adresse, entfernterPort);
               DatagramSocket socket = new DatagramSocket ();
               System.out.println ("Adresse: " + adresse);
               System.out.println ("Gesendete Botschaft: "
                  + new String (botschaft));
               System.out.flush ();
               socket.send (paket);
               }
            catch (UnknownHostException e)
               {
               System.err.println ("DatagrammSenden.main: " + e);
               }
            catch (IOException e)
               {
               System.err.println ("DatagrammSenden.main: " + e);
               }
            break;
         default:
            System.err.println ("Benutzung: 1 Argument");
```

```
            System.err.println ("   1): Empfängername");
            break;
         }
      }
   }
```

Zum Empfangen eines Datagramms wird ein `DatagramSocket` zum Empfangen erzeugt, indem ihm eine Portnummer mitgegeben wird. Das empfangene Datagramm enthält Botschaft, Adresse und Port des Absenders. Die Botschaft wird auf die Größe des zur Verfügung gestellten Puffers gekürzt.

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

public class   DatagrammEmpfangen
   {
   public static final int DEFAULTPORT = 6010;

   public static void   main (String[] argument)
      {
      int   lokalerPort = DEFAULTPORT;
      switch (argument.length)
         {
         case 1:
            try
               {
               lokalerPort = Integer.parseInt (argument[0]);
               }
            catch (NumberFormatException e) { }
         default: ;
         }
      try
         {
         byte[]   botschaft = new byte[256];
         DatagramPacket paket =
            new DatagramPacket (botschaft, botschaft.length);
         DatagramSocket socket = new DatagramSocket (lokalerPort);
         socket.receive (paket);
         String   s = new String (botschaft, 0, paket.getLength ());
         System.out.println ("UDPEmpfang von "
            + paket.getAddress ().getHostName ()
            + ":"
            + paket.getPort ()
            + " Botschaft: "
            + s);
         }
```

```
    catch (IOException e)
       {
       System.err.println ("DatagrammEmpfangen.main: " + e);
       }
    }
}
```

Für Broadcast- oder Multicast-Anwendungen gibt es die Klasse `MulticastSocket` mit den entsprechenden Methoden, auf die hier nicht näher eingegangen wird.

11.4 Client/Server

Mehrere Hosts senden als Clients (Kunden) einem Server (Dienstleister) Botschaften oder Dienstanforderungen, die ein Host als Server empfängt und entsprechende Antworten an den Client zurückschickt.

Für eine zuverlässige ***(TCP – reliable)*** Verbindung zwischen Client und Server wird die Klasse `Socket` verwendet.

11.4.1 Server

Mit der Klasse `ServerSocket` wird ein Objekt erzeugt, das mit der Methode `accept ()` auf eine Verbindungsanforderung eines Clients wartet, die auf dem im Konstruktor spezifizierten Port – optional innerhalb einer gewissen Zeit – erwartet wird. Die Methode `accept ()` akzeptiert die angeforderte Verbindung, wobei ein Objekt der Klasse `Socket` für die weitere Kommunikation mit dem Client zurückgegeben wird. Ist ein `ServerSocket`-Objekt einmal erzeugt, erwartet er immer weitere Verbindungen, die mit `accept` akzeptiert werden können.

Die Klasse `ServerSocket` hat drei Konstruktoren:

`ServerSocket (int lokalerPort)`
`ServerSocket (int lokalerPort, int anzVA)`
(`anzVA` ist maximale Anzahl der anstehenden (noch nicht akzeptierten) Verbindungsanforderungen.)
`ServerSocket (int lokalerPort, int anzVA, InetAddress lokaleAdresse)`
(Bei Hosts mit mehr als einer Internetadresse kann die Empfangsadresse spezifiziert werden.)

Der Server ist typischerweise ***multithreaded***. Das Server-Objekt selbst ist ein Thread. Seine `run`-Methode läuft immer und wartet auf Verbindungsanforderungen durch Clients. Wenn eine Verbindung zustandekommt, erzeugt der Server einen neuen Thread `Verbindung`, der die Kommunikation mit dem Client über die neue Verbindung abwickelt und die eigentlichen Dienste des Servers zur Verfügung stellt.

In unserem Beispiel erhält der Server einen Text (eine "Frage?"), die er mit "Natürlich Frage." beantwortet.

Die Serverklasse heißt bei uns `Dienst` und wird wie folgt implementiert. Um den Überblick nicht zu verlieren zeigen wir den den Code zunächst ohne Fehler- und Eingabebehandlung:

```
public class   Dienst extends Thread
   {
   public static void   main (String[] argument)
      {
      int   lokalerPort = Integer.parseInt (argument[0]);
      new Dienst (lokalerPort);
      }

   protected ServerSocket  dienstSocket;

   public   Dienst (int lokalerPort)
      {
      dienstSocket = new ServerSocket (lokalerPort);
      this.start ();
      }

   public void run ()
      {
      while (true)
         {
         Socket   kundenSocket = dienstSocket.accept ();
         Verbindung  v = new Verbindung (kundenSocket);
         }
      }
   }

class Verbindung extends Thread
   {
   protected Socket  kundenSocket;
   protected BufferedReader   ein;
   protected PrintWriter   aus;

   public   Verbindung (Socket kundenSocket)
      {
      this.kundenSocket = kundenSocket;
      ein = new BufferedReader (
            new InputStreamReader (
            kundenSocket.getInputStream ()));
      aus = new PrintWriter (
            new OutputStreamWriter (
            kundenSocket.getOutputStream ()));
      this.start ();
      }

   public void run ()
```

```
      {
      String   zeile;
      while ((zeile = ein.readLine ()) != null)
         {
         aus.print ("Natürlich ");
         aus.println (zeile.replace ('?', '.'));
         aus.flush ();
         if (zeile.equals ("quit")) break;
         }
      }
   }
```

Nach Übersetzen der Files mit

```
$ javac Dienst.java
$ javac Kunde.java
$ javac KundeApplet.java
```

wird in einem Fenster der Dienst gestartet mit

```
$ java Dienst
```

und in einem anderen Fenster oder auf einem anderen Rechner werden der Kunde oder beliebig viele Kunden gestartet mit:

```
$ java Kunde <Rechnername bzw Internet-Nummer des Dienst-Rechners>
```

Vollständiger Code:

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

public class   Dienst extends Thread
   {
   protected ServerSocket  dienstSocket;

   public static void   main (String[] argument)
      {
      int   lokalerPort = 0;
      if (argument.length == 1)
         {
```

```
         try { lokalerPort = Integer.parseInt (argument[0]); }
         catch (NumberFormatException e) { lokalerPort = 0; }
         }
      new Dienst (lokalerPort);
      }

   public   Dienst (int lokalerPort)
      {
      if (lokalerPort == 0) lokalerPort = KundeApplet.DEFAULTPORT;

      try { dienstSocket = new ServerSocket (lokalerPort); }
      catch (IOException e) { fehler (e, "Erzeugung ServerSocket: "); }
      System.out.println ("Server: erwartet auf Port " + lokalerPort);
      this.start ();
      }

   public void run ()
      {
      try
         {
         while (true)
            {
            Socket   kundenSocket = dienstSocket.accept ();
            Verbindung  v = new Verbindung (kundenSocket);
            }
         }
      catch (IOException e) { fehler (e, "ServerSocket.accept: "); }
      }

   public static void   fehler (Exception e, String botschaft)
      {
      System.err.println ("Fehler bei " + botschaft + e);
      System.exit (1);
      }

   }

class Verbindung extends Thread
   {
   public   Verbindung (Socket kundenSocket)
      {
      this.kundenSocket = kundenSocket;
      try
         {
         ein = new BufferedReader (
            new InputStreamReader (kundenSocket.getInputStream ()));
         aus = new PrintWriter (
            new OutputStreamWriter (kundenSocket.getOutputStream ()));
         }
```

```
      catch (IOException e)
         {
         try { kundenSocket.close (); }
         catch (IOException e2)
            { Dienst.fehler (e2, "Schließen kundenSocket: "); }
         Dienst.fehler (e, "Erzeugung Socket-Streams: ");
         return;
         }
      this.start ();
      }

   public void run ()
      {
      String   zeile;
      try
         {
         while ((zeile = ein.readLine ()) != null)
            {
            System.out.println ("Kunde (" + kundenSocket + ") fragt: " + zeile);
            aus.print ("Natürlich ");
            aus.println (zeile.replace ('?', '.'));
            aus.flush ();
            if (zeile.equals ("quit")) break;
            }
         }
      catch (IOException e) { Dienst.fehler (e, "Verbindung.run: "); }
      finally
         {
         try { kundenSocket.close (); }
         catch (IOException e2)
            { Dienst.fehler (e2, "Verbindung.run.finally: "); }
         }
      }

   protected Socket  kundenSocket;
   protected BufferedReader   ein;
   protected PrintWriter   aus;
   }
```

11.4.2 Client

Die Client-Klasse stellt ein `main` zur Verfügung, das als Argumente den Host-Namen und optional den Port des Servers nimmt. Der Client erzeugt einen Socket, der die Verbindung zum Server aufbaut. Unser Client-Beispiel liest eine Frage von `stdin` und schickt sie dem Server zur

Beantwortung. Unsere Client-Klasse heißt konventionsgemäß `Kunde`.

Code ohne Fehler- und Eingabebehandlung:

```
public class   Kunde
   {
   public static void   main (String[] argument)
      {
      int   entfernterPort = Integer.parseInt (argument[1]);
      Socket   s = new Socket (argument[0], entfernterPort);
      BufferedReader sein =
         new BufferedReader (
         new InputStreamReader (s.getInputStream ()));
      PrintWriter saus =
         new PrintWriter (
         new OutputStreamWriter (s.getOutputStream ()));
      BufferedReader eingabe =
         new BufferedReader (
         new InputStreamReader (System.in));

      while (true)
         {
         System.out.println ("Eingabe der Frage:");
         System.out.print ("> ");
         System.out.flush ();
         String   frage = eingabe.readLine ();
         saus.println (frage); saus.flush ();

         System.out.println ("Antwort:");
         String   antwort = sein.readLine ();
         if (antwort == null) break;
         System.out.println (antwort);
         if (frage.equals ("quit")) break;
         }
      }
   }
```

Vollständiger Code:

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

public class   Kunde
   {
   public static void   main (String[] argument)
      {
      int   entfernterPort = KundeApplet.DEFAULTPORT;
      Socket   s = null;
```

```
switch (argument.length)
   {
   case 2:
      try { entfernterPort = Integer.parseInt (argument[1]); }
      catch (NumberFormatException e)
         { entfernterPort = KundeApplet.DEFAULTPORT; }
   case 1:
      try
         {
         s = new Socket (argument[0], entfernterPort);
         System.out.println ("Verbunden mit "
            + s.getInetAddress ()
            + ":"
            + s.getPort ());

         BufferedReader sein =
            new BufferedReader (
            new InputStreamReader (s.getInputStream ()));
         PrintWriter saus =
            new PrintWriter (
            new OutputStreamWriter (s.getOutputStream ()));
         BufferedReader eingabe =
            new BufferedReader (
            new InputStreamReader (System.in));

         while (true)
            {
            System.out.println ("Eingabe der Frage:");
            System.out.print ("> ");
            System.out.flush ();
            String   frage = eingabe.readLine ();
            saus.println (frage); saus.flush ();

            System.out.println ("Antwort:");
            String   antwort = sein.readLine ();
            if (antwort == null) break;
            System.out.println (antwort);
            if (frage.equals ("quit")) break;
            }
         break;
         }
      catch (UnknownHostException e)
         { Dienst.fehler (e, "Socket (): "); }
      catch (IOException e)
         { Dienst.fehler (e, "Kunde.main.try: "); }
      finally
         {
         try { if (s != null) s.close (); }
         catch (IOException e2)
```

```
                { Dienst.fehler (e2, "Kunde.finally: "); }
             }
          default:
             System.err.println ("Benutzung: 1 optional 2 Argumente");
             System.err.println ("   1): Hostadresse des Servers");
             System.err.println ("   2): Port des Servers");
             break;
          }
       }
   }
```

11.4.3 Applet-Client

Der folgende Client hat eine graphische Benutzeroberfläche. Er sendet die Eingabe von einer `TextField`-Komponente an den Server und stellt die Antwort in einer `TextArea`-Komponente dar.

Dieser Client benutzt einen Thread `Empfaenger` zum Lesen der vom Server geschickten Daten und zum Darstellen der Daten.

Code ohne Fehler- und Eingabebehandlung:

```
public class   KundeApplet extends Applet
   implements  ActionListener
   {
   int   entfernterPort = DEFAULTPORT;
   PrintWriter saus;
   BufferedReader sein;
   Socket   s;
   TextField   textFeld;
   TextArea textBereich;
   Empfaenger  empfaenger;

   public void init ()
      {
      entfernterPort = Integer.parseInt (getParameter ("Port"));
      s = new Socket (this.getCodeBase ().getHost (),
            entfernterPort);

      sein = new BufferedReader (
         new InputStreamReader (s.getInputStream ()));
      saus = new PrintWriter (
         new OutputStreamWriter (s.getOutputStream ()));
      this.setLayout (new BorderLayout ());
      Panel panel = new Panel ();
```

```
      Label eingabe = new Label ("Eingabe der Frage: ", Label.LEFT);
      panel.add (eingabe);
      textFeld = new TextField (40);
      textFeld.addActionListener (this);
      panel.add (textFeld);

      this.add ("North", panel);
      textBereich = new TextArea ();
      this.add ("Center", textBereich);

      empfaenger = new Empfaenger (this, sein, textBereich);
      }

   public void actionPerformed (ActionEvent e)
      {
      if (e.getSource () == textFeld)
         {
         saus.println (e.getActionCommand ()); saus.flush ();
         textFeld.setText ("");
         }
      }
   }

class Empfaenger extends Thread
   {
   Applet   applet;
   BufferedReader ein;
   TextArea aus;

   public   Empfaenger
      (
      Applet   applet,
      BufferedReader sein,
      TextArea textBereich
      )
      {
      this.applet = applet;
      ein = sein;
      aus = textBereich;
      this.start ();
      }

   public void run ()
      {
      String   zeile;
      while ((zeile = ein.readLine ()) != null)
         {
         aus.setText (zeile);
         if (zeile.endsWith ("quit"))
```

```
            {
            applet.stop ();
            return;
            }
         }
      }
   }
```

Vollständiger Code:

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

public class   KundeApplet extends Applet
   implements  ActionListener
   {
   public static final int DEFAULTPORT = 6547;

   public void init ()
      {
      try
         {
         int   entfernterPort = 0;
         try
            {
            entfernterPort = Integer.parseInt (getParameter ("Port"));
            }
         catch (NumberFormatException e)  { entfernterPort = 0; }
         if (entfernterPort != 0) this.entfernterPort = entfernterPort;
         else this.entfernterPort = DEFAULTPORT;

         s = new Socket (this.getCodeBase ().getHost (),
            this.entfernterPort);

         sein = new BufferedReader (
            new InputStreamReader (s.getInputStream ()));
         saus = new PrintWriter (
            new OutputStreamWriter (s.getOutputStream ()));
         this.setLayout (new BorderLayout ());
         Panel panel = new Panel ();
         Label eingabe = new Label ("Eingabe der Frage: ", Label.LEFT);
         panel.add (eingabe);
         textFeld = new TextField (40);
         textFeld.addActionListener (this);
         panel.add (textFeld);
```

```
         this.add ("North", panel);
         textBereich = new TextArea ();
         this.add ("Center", textBereich);

         empfaenger = new Empfaenger (this, sein, textBereich);

         this.showStatus ("Verbunden mit "
            + s.getInetAddress ().getHostName ()
            + ":"
            + s.getPort ());
         }
         catch (UnknownHostException e)
            { this.showStatus ("KundeApplet.init: " + e); }
         catch (IOException e)
            { this.showStatus ("KundeApplet.init: " + e); }
      }

   public void actionPerformed (ActionEvent e)
      {
      if (e.getSource () == textFeld)
         {
         saus.println (e.getActionCommand ()); saus.flush ();
         textFeld.setText ("");
         }
      }

   public String[][] getParameterInfo ()
      {
      String [][] parameterInformation =
         {
         // Parametername, Parametertyp, Parameterbeschreibung
         {"Port", "int", "Port des Servers (Default "
            + DEFAULTPORT + ")"}
         };
      return parameterInformation;
      }

   public String  getAppletInfo ()
      {
      return "Kunden-Applet";
      }

   int   entfernterPort = DEFAULTPORT;
   PrintWriter saus;
   BufferedReader sein;
   Socket   s;
   TextField   textFeld;
   TextArea textBereich;
```

```
   Empfaenger  empfaenger;
   }

class Empfaenger extends Thread
   {
   public   Empfaenger
      (
      Applet   applet,
      BufferedReader sein,
      TextArea textBereich
      )
      {
      this.applet = applet;
      ein = sein;
      aus = textBereich;
      this.start ();
      }

   public void run ()
      {
      String   zeile;
      try
         {
         while ((zeile = ein.readLine ()) != null)
            {
            aus.setText (zeile);
            if (zeile.endsWith ("quit"))
               {
               applet.stop ();
               return;
               }
            }
         }
      catch (IOException e) { aus.setText ("Lesefehler " + e); }
      finally { aus.setText ("Verbindung vom Server geschlossen."); }
      }

   Applet   applet;
   BufferedReader ein;
   TextArea aus;
   }
```

Die HTML-Datei dazu sieht folgendermaßen aus:

```
<HTML>
<HEAD>
<TITLE>Client-Applet</TITLE>
```

```
</HEAD>
<BODY>
Wenn Sie einen Java-fähigen Browser benutzen,
dann schauen Sie das unten angegebenen Applet an.
Ansonsten haben Sie Pech gehabt.
<P>
<APPLET code="KundeApplet.class" width=500 height=300>
<PARAM name="Port" value="6547">
</APPLET>
</BODY>
</HTML>
```

11.5 Übungen

11.5.1 Übung Client/Server

Modifizieren Sie das Client/Server-Beispiel so, dass der Client mehrzeilige Fragen stellen kann, die mit einem Fragezeichen enden. Der Server kann ebenfalls mehrzeilig antworten. Seine Antwort endet mit einem Ausrufezeichen.

11.5.2 Übung Filetransfer

Schreiben Sie ein Programm, mit dem eine Datei übertragen werden kann.

11.5.3 Übung Verwaltung einer Warteschlange

In Finanzämter oder ähnlichen Einrichtungen muss man hin- und wieder warten, bis man bedient wird. Die Warteschlangen werden über eine Anzeige verwaltet. Als Kunde muss man eine Nummer ziehen und dann warten, bis auf der Anzeige die gezogene Nummer mit einem zugehörigen Bearbeitungsplatz erscheint.

Programmieren Sie einen Server, der alle Nummern und Bearbeitungsplätze verwaltet. Die Bearbeitungsplätze selbst sind Clients dieses Servers. Das Ziehen einer Nummer kann ebenfalls als ein Client, ein "Kundenclient" realisiert werden.

Kapitel 12

Graphische Benutzeroberflächen

Zur Programmierung von graphischen Benutzeroberflächen bietet Java das **AWT** ***(abstract windowing toolkit)*** und seit Java 2 **Swing** an.

Mit Swing hat Java einen neuen Satz von GUI-Komponenten, die vollständiger, flexibler und portabler als die AWT-Komponenten sind. Insbesondere hängt Swing nicht mehr von den nativen Komponenten der jeweiligen Plattform ab. Daher werden wir im folgenden nur noch den für Swing benötigten Anteil von AWT behandeln. Von AWT werden die Layout-Manager und das Ereignismodell übernommen.

Swing bietet die Möglichkeit unterschiedliche "Look-and-Feels" einzustellen (PLAF, ***Pluggable Look-and-Feel***, Metal, Windows, Motif und eigene LFs). Ferner macht Swing eine deutliche Unterscheidung zwischen den Daten einer Komponente ***(Model)***, der Darstellung einer Komponente ***(View)*** und der Ereignisbehandlung ***(Controller)***. Damit benutzt Swing eine *model-view-controller*-Architektur (MVC) als das fundamentale Design-Prinzip hinter jeder Komponente.

Swing baut auf AWT auf. Insbesondere werden das Ereignismodell und die Layout-Manager verwendet. Ansonsten sollte man nur noch Swing-Komponenten verwenden. Auf keinen Fall sollte man in Behältern Swing- und AWT-Komponenten mischen. Eine AWT-Komponente darf nicht in einem Swing-Behälter verwendet werden.

Die meisten Swing-Komponenten sind ***lightweight***. Sie hängen nicht von nativen Peers ab, um sich darzustellen. Sie benutzen Graphik-Primitive der Klasse `Graphics`. Damit werden die Komponenten schneller und weniger Speicher-intensiv.

Ab Version Java 2 (JDK 1.2) sind die Java Foundation Classe (JFC) in das JDK integriert. Es genügt aber schon JDK 1.1.5, um Swing zu verwenden. Zu den JFC gehören alle Pakete `java.*` und `javax.*` , insbesondere das Paket `javax.swing` . Es genügt JDK 1.1.5, um Swing zu verwenden.

Bei einer GUI-Entwicklung geht man typischerweise so vor, dass man von einem **Behälter** ***(container)*** ausgeht und diesen mit **Komponenten** ***(component)*** füllt. Manche Komponenten können wiederum als Behälter verwendet werden, sodass ein hierarchischer Aufbau möglich ist.

Bei Anwendungsprogrammen geht man normalerweise von einem Objekt der Toplevel-Klasse `JFrame` aus, die ein ikonisierbares, in der Größe veränderbares Fenster mit Titel repräsentiert und einen GUI-Komponentenbehälter zur Verfügung stellt.

Bei Applets erbt man von der Toplevel-Klasse `JApplet`, die auch einen GUI-Komponentenbehälter zur Verfügung stellt.

Innerhalb eines Behälters werden die Komponenten mit Hilfe von verschiedenen **Layout-Managern** angeordnet. Wir beginnen zunächst mit der Beschreibung der verschiedenen Layout-Manager.

12.1 Layout-Manager

Die Anordnung der Komponenten einer Benutzeroberfläche wird bestimmt durch die Reihenfolge, in der sie einem Behälter zugeordnet werden, und durch den verwendeten Layout-Manager.

Es gibt fünf verschiedene, von AWT übernommene Layout-Manager: `FlowLayout`, `GridLayout`, `GridBagLayout`, `BorderLayout` und `CardLayout`. Swing bringt dann noch die Layout-Manager `BoxLayout`, `OverlayLayout`, `ScrollPaneLayout` und `ViewportLayout` mit. Die Swing-Layout-Manager werden meistens in Zusammenhang mit speziellen Behältern verwendet. Daher werden wir auf diese hier nicht näher eingehen.

Mit der Methode `setLayout` kann einem Behälter ein Layout gegeben werden:

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

12.1.1 `FlowLayout`

Das einfachste Layout ist das `FlowLayout`, bei dem die Komponenten zeilenweise angeordnet werden. Passt eine Komponente nicht mehr in eine Zeile, dann wird eine neue Zeile aufgemacht. Defaultmäßig wird jede Zeile zentriert. Bei der Erzeugung eines `FlowLayout`s kann die Bündigkeit als Parameter des Konstruktors angegeben werden (`FlowLayout.LEFT`, `FlowLayout.RIGHT` oder `FlowLayout.CENTER`).

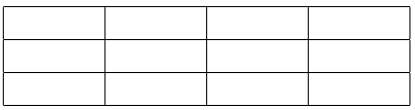
Als weitere Parameter des Konstruktors können horizontale und vertikale Abstände zwischen den Komponenten in Pixeln angegeben werden (Default ist hier 5 Pixel.):

```
setLayout (new FlowLayout (FlowLayout.LEFT, 7, 9));
```

12.1.2 `GridLayout`

Die Fläche des Behälters wird in gleichgroße Zellen aufgeteilt in Abhängigkeit von der im Konstruktor angegebenen Anzahl Zeilen und Spalten.

```
setLayout (new GridLayout (3, 4));
```

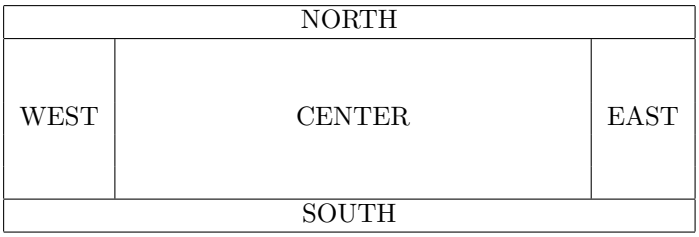


Als weitere Parameter des Konstruktors können horizontale und vertikale Abstände zwischen den Komponenten in Pixeln angegeben werden (Default ist hier 0 Pixel.):

```
setLayout (new GridLayout (3, 4, 7, 9));
```

12.1.3 BorderLayout

Ein `BorderLayout` hat fünf Bereiche:



Mit der zweiparametrigen `add`-Methode

```
add (new JButton ("Nord"), BorderLayout.NORTH);
```

werden Komponenten in die angegebenen Bereiche gesetzt.

Als optionale Parameter des Konstruktors können horizontale und vertikale Abstände zwischen den Komponenten in Pixeln angegeben werden (Default ist hier 0 Pixel.):

```
setLayout (new BorderLayout (7, 9));
```

12.1.4 CardLayout

Mit dem `CardLayout` kann man mehrere Komponenten an einem Ort plazieren, wobei immer nur eine Komponente sichtbar ist. Jede Komponente bekommt einen `String` als Namen. Dann wird wieder die zweiparametrige `add`-Methode verwendet. Folgendes Code-Fragment soll das demonstrieren:

```
JPanel   karten = new JPanel ();
CardLayout  kartenLayout = new CardLayout ();
karten.setLayout (kartenLayout);

JPanel   karte1 = new JPanel ();
String   nameKarte1 = "Knöpfe";
// Fülle karte1 mit Knöpfen oder ähnlichem
karten.add (karte1, nameKarte1);

JPanel   karte2 = new JPanel ();
String   nameKarte2 = "Textfelder";
// Fülle karte2 mit Textfeldern oder ähnlichem
karten.add (karte2, nameKarte2);
```

Für die Auswahl der Karte, die gezeigt werden soll, gibt es fünf Möglichkeiten:

```
kartenLayout.first (karten);
kartenLayout.next (karten);
kartenLayout.previous (karten);
kartenLayout.last (karten);
kartenLayout.show (karten, nameKarte2);
```

12.1.5 GridBagLayout

Das `GridBagLayout` bietet die meisten Möglichkeiten Komponenten anzuordnen. Das Layout legt Zeilen und Spalten an, wobei Komponenten mehrere Zeilen oder Spalten einnehmen können. Die Zellengröße richtet sich nach der **bevorzugten** ***(preferred)*** Größe der Komponenten. Den Zellen können Gewichte gegeben werden, die beim "Kampf" der Komponenten um den zur Verfügung stehenden Platz berücksichtigt werden. Für diese und andere Angaben wird ein `GridBagConstraints`-Objekt definiert.

Die Vorgehensweise ist exemplarisch folgende:

```
JPanel   sack = new JPanel ();
sack.setLayout (new GridBagLayout ());

GridBagConstraints   c = new GridBagConstraints ();

// Für jede Komponente komponente, die zum JPanel sack addiert wird,
// werden die Parameter des GridBagConstraints c gesetzt.
// z.B.
c.gridx = 3;
c.gridy = 4;
// usw

sack.add (komponente, c);
```

Ohne auf Einzelheiten einzugehen, seien die einstellbaren Parameter von `GridBagConstraints` genannt:

`gridx, gridy:` Spezifikation der Spalten und Zeilennummer der Zelle beginnend bei 0 oder `RELATIVE` (bezüglich zuletzt addierter Komponente).

`gridwidth, gridheight:` Anzahl Spalten und Zeilen, die die Zelle einnimmt.

`fill:` Angabe der Art, wie die Komponente die Zelle ausfüllen soll (`NONE` (Default), `HORIZONTAL`, `VERTICAL`, `BOTH`).

`ipadx, ipady:` Die Minimalgröße der Komponente wird um `2 * ipadx` bzw `2 * ipady` Pixel vergrößert.

`insets:` Spezifikation des externen Padding durch Angabe eines `Insets`-Objekt.

`anchor:` Spezifikation des Ortes innerhalb der Zelle (`CENTER` (Default), `NORTH`, `NORTHEAST` usw).

`weightx, weighty:` Mit den Gewichten zwischen `0.0` (Default!) und `1.0` wird spezifiziert, wie übriger Platz verteilt wird, wobei `0.0` das höchste Gewicht hat.

12.1.6 `BoxLayout`

Dieses Layout wird normalerweise mit dem Behälter `Box` verwendet und eignet sich für eine horizontale oder vertikale Anordnung von z.B. "Werkzeugen" (Toolbar).

12.1.7 `OverlayLayout`

Mit diesem Layout kann man Komponenten aufeinanderlegen unter Berücksichtigung eines bestimmten Versatzes.

12.1.8 `ViewportLayout`

Diese Layout sollte nur mit der Komponente `JViewport` verwendet werden, mit der man einen Ausschnitt einer großen Komponente betrachten kann.

12.1.9 `ScrollPaneLayout`

Dieser Layout-Manager gibt der Klasse `JScrollPane` die verschiedenen Steuerungsmöglichkeiten.

12.1.10 Bemerkungen

1. Mit Inset-Werten, d.h. Objekten der Klasse `Insets` können bei *eigenen* Behälter-Klassen Ränder definiert werden, indem die Methode `getInsets ()` überschrieben wird. Diese Ränder sind Bereiche, die von den Layout-Managern nicht benutzt werden. Beispiel:

```
public class   meinBehaelter extends JPanel
   {
   // ...
   public Insets  getInsets ()
      {
      return new Insets (17, 8, 20, 5);
            // oben, links, unten, rechts
      }
   // ...
   }
```

2. Man kann die Kontrolle über die Plazierung von Komponenten ganz selbst übernehmen, indem man den Layout-Manger durch

```
setLayout (null)
```

ausschaltet. Es wird aber dringend davon abgeraten.

3. Mit der Methode `invalidate ()` wird ein Behälter als `invalid` gekennzeichnet. Das System macht das z.B. automatisch, wenn eine Komponente addiert oder entfernt wurde, d.h. wenn das Layout neu zu berechnen ist.

   Mit der Methode `validate ()` wird das Layout eines Behälters neu berechnet, der als `invalid` gekennzeichnet wurde. Dabei wird eine Methode `dolayout ()` aufgerufen. `validate` wird auch auf alle Behälter-Komponenten angewendet.

   Es ist nicht trivial `validate` zu überschreiben.

   Vom Programmierer sollte `validate` aufgerufen werden, wenn eine Komponente zu einem schon sichtbaren Behälter addiert wurde.

4. Ein Komponenten-Objekt kann nur zu *einem* Behälter gehören. Addition zu einem anderen Behälter entfernt die Komponente automatisch aus dem ersten Behälter. Eine Komponente kann nur ein Elter haben. Damit ist garantiert, dass die Behälterstruktur immer ein Baum (kein Graph) ist. Bäume sind leichter zu verstehen und zu berechnen.

5. Mit der fundamentalen Methode

   ```
   void setBounds (int x, int y, int breite, int hoehe)
   void setBounds (Rectangle r)
   ```

   der abstrakten Klasse `Component` wird Ort und Größe einer Komponente festgelegt. Die Koordinaten sind absolute Bildschirmkoordinaten in Pixeln. Der Ursprung ist die linke obere Ecke des Elter-Behälters oder des Bildschirms.

   Zur Bequemlichkeit gibt es noch die beiden Methoden:

   ```
   void setLocation (int x, int y)
   void setLocation (Point p)
   void setSize (int breite, int hoehe)
   void setSize (Dimension d)
   ```

   Informationen über die Bildschirmgröße und -auflösung erhält man mit

   ```
   Dimension getScreenSize ()// in Pixel
   int getScreenResolution () // in Pixel pro Inch
   ```

   der abstrakten Klasse `java.awt.Toolkit`. Die statische Methode

   ```
   Toolkit.getDefaultToolkit ()
   ```

   liefert das aktuelle Objekt vom Typ `Toolkit`.

   Mit diesen Mitteln ist es möglich, Graphik zentimetergenau darzustellen.

12.2 Swing GUI-Komponenten

Jede Swing-Komponente ist ein Java-Bean. Daher verwenden wir in der Diskussion der Klassen die Java-Bean-Konvention, indem wir z.B. eine Klasseneigenschaft `background` folgendermaßen beschreiben:

```
background {Color, get, set, Color.black}
```

Das bedeutet: Der Defaultwert, falls überhaupt angegeben, ist `Color.black` und es gibt die Methoden:

```
public Color  getBackground ()
public void   setBackground (Color farbe)
```

In diesem Kapitel können umöglich alle Details der Swing-Komponenten dargestellt werden. Inzwischen gibt es sehr viel Literatur über Swing. Wir verweisen hier auf das Buch von Eckstein, Loy und Wood [5].

Die Namen der Swing-GUI-Komponenten-Klassen beginnen mit `J`. Alle außer vier Toplevel-Klassen leiten sich von der abstrakten Klasse `JComponent` ab, die sich wiederum von der AWT-Klasse `Container` ableitet. Damit haben die Swing-Klassen sehr viel AWT-Funktionalität geerbt. Bemerkenswert ist, dass jede Komponente ein Komponenten-Behälter ist.

Die Folgende Tabelle zeigt die Swing-Klassen, wobei eine Einrückung Ableitung bedeutet. Geklammerte Klassen sind abstrakt.

```
Object
   (Component)
      (Container)
         Panel
            Applet
               JApplet
         Window
            JWindow
            Frame
               JFrame
            Dialog
               JDialog
         (JComponent)
            JComboBox
            JLabel
            JList
            JMenuBar
            JPanel
            JPopupMenu
            JScrollBar
            JScrollPane
            JTable
            JTree
            JInternalFrame
            JOptionPane
            JProgressBar
            JRootPane
            JSeparator
            JSlider
```

```
JSplitPane
JTabbedPane
JToolBar
JToolTip
JViewport
JColorChooser
JTextComponent
   JTextArea
   JTextField
      JPasswordField
   JEditorPane
      JTextPane
JFileChooser
JLayeredPane
   JDesktopPane
(AbstractButton)
   JToggleButton
      JCheckBox
      JRadioButton
   JButton
   JMenuItem
      JMenu
      JRadioButtonMenuItem
      JCheckBoxMenuItem
```

12.3 AWT-Erbe

Die Swing-Klassen erben von `Container` und `Component` und haben daher auch folgende Eigenschaften geerbt:

```
background        {Color, get, set}
colorModel        {ColorModel, get}
component         {Component, get (indexed)}
componentCount    {int, get}
components        {Component[], get}
cursor            {Cursor, get, set, Cursor.DEFAULT_CURSOR}
enabled           {boolean, is, set, true}
font              {Font, get, set}
foreground        {Color, get, set}
insets            {Insets, get, Insets (0, 0, 0, 0)}
layout            {LayoutManager, get, set, BorderLayout ()}
locale            {Locale, get, set}
location          {Point, get, set}
locationOnScreen  {Point, get, set}
name              {String, get, set, ""}
parent            {Container, get, set, null}
```

```
size              {Dimension, get, set}
showing           {boolean, is, true}
valid             {boolean, is}
visible           {boolean, is, set, true}
```

Die meisten Eigenschaften sind selbsterklärend. Nur `showing`, `valid` und `visible` müssen erklärt werden. Die meisten typischen Behälter-Klassen (z.B. `JPanel`, `JFrame`) sind Default-mäßig nicht `visible`. Die meisten anderen Komponenten (z.B. `JButton`) sind Default-mäßig `visible`. Wenn man nun einen `JButton` an ein nicht sichtbares `JFrame` addiert, dann ist der `JButton` zwar `visible`, aber nicht `showing` und erscheint daher erst, wenn auch der Behälter `visible` wird.

Wenn die Eigenschaft `valid` `false` ist, dann muss die Komponente durch den Layout-Manager wieder eingepasst werden (Aufruf von `validate ()` für den Behälter). Diese Eigenschaft wird nicht immer automatisch für eine Komponente gesetzt. Daher muss u.U. für die Komponente `invalidate ()` aufgerufen werden.

12.4 Toplevel-Komponenten

Die Toplevel-Komponenten sind `JWindow`, `JFrame`, `JDialog` und `JApplet`. Applikationsprogramme gehen typischerweise von einem `JFrame` aus, Applets erben von der Klasse `JApplet`.

12.4.1 Einstieg

Die Toplevel-Komponenten sind zwar wie alle Komponenten auch Behälter, sollten aber nicht als solche verwendet werden.

Stattdessen holt man sich von einer Toplevel-Komponente mit der Methode

```
Container   getContentPane ()
```

einen Behälter, den man dann mit anderen Komponenten füllen kann.

```
JFrame   jf = new JFrame ("Titel");
Container   bjf = jf.getContentPane ();

bjf.add (...);
   // Baue Oberfläche durch
   // Addition von Komponenten auf.

jf.setVisible (true);
   // Jetzt erst erscheint das GUI

// Wenn das GUI nicht mehr gebraucht wird,
// dann kann man es mit:
jf.setVisible (false);
jf.dispose ();
```

```
        // verschwinden lassen.
```

Als Beispiel zeigen wir ein kleines GUI-Programm, mit dem das Look-and-Feel einstellen kann. (Möglicherweise werden auf der vorliegenden Maschine nicht alle Look-and-Feels unterstützt.)

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

public class   LookAndFeel extends JPanel
   {
   public   LookAndFeel (Component komponente)
      {
      setBackground (Color.gray);

      JButton  metal = new JButton ("Metal");
      metal.setBackground (Color.lightGray);
      metal.addActionListener (
         new LookAndFeelListener (
            "javax.swing.plaf.metal.MetalLookAndFeel",
            komponente));
      add (metal);

      JButton  motif = new JButton ("Motif");
      motif.setBackground (Color.lightGray);
      motif.addActionListener (
         new LookAndFeelListener (
            "com.sun.java.swing.plaf.motif.MotifLookAndFeel",
            komponente));
      add (motif);

      JButton  windows = new JButton ("Windows");
      windows.setBackground (Color.lightGray);
      windows.addActionListener (
         new LookAndFeelListener (
            "com.sun.java.swing.plaf.windows.WindowsLookAndFeel",
            komponente));
      add (windows);

      JButton  mac = new JButton ("Macintosh");
      mac.setBackground (Color.lightGray);
      mac.addActionListener (
         new LookAndFeelListener (
            "com.sun.java.swing.plaf.mac.MacLookAndFeel",
            komponente));
      add (mac);
      }
```

```
   public static void   main (String[] argument)
      {
      JFrame   jf = new JFrame ("Look-and-Feel Test");
      jf.setSize (400, 200);
      Container   bjf = jf.getContentPane ();
      bjf.add (new LookAndFeel (jf), BorderLayout.NORTH);
      jf.setVisible (true);
      }
   }

class LookAndFeelListener
   implements  ActionListener
   {
   private String lfName;
   private Component komponente;

   public   LookAndFeelListener (String lfName, Component komponente)
      {
      this.lfName = lfName;
      this.komponente = komponente;
      }

   public void actionPerformed (ActionEvent e)
      {
      try
         {
         UIManager.setLookAndFeel (lfName);
         SwingUtilities.updateComponentTreeUI (komponente);
         }
         catch (Exception ex)
            {
            ex.printStackTrace ();
            }
      }
   }
```

12.4.2 Einzelheiten

Die Toplevel-Klassen sind Behälter und enthalten eine einzige Komponente vom Typ `JRootPane`. Mit Hilfe dieser Komponente implementieren sie die Schnittstelle `RootPaneContainer`, die folgende Methoden hat:

```
public Container  getContentPane ();
public void setContentPane (Container c);
```

```
public Component  getGlassPane ();
public void setGlassPane (Component c);

public JLayeredPane  getLayeredPane ();
public void setLayeredPane (JLayeredPane c);

public JRootPane  getRootPane ();
```

Außerdem werden von `JFrame` und `JApplet` auch die Methoden

```
public JMenuBar   getJMenuBar ();
public void setJMenuBar (JMenuBar c);
```

unterstützt.

Damit sind die Behälter-Komponenten der Klasse `JRootPane` zugänglich. Diese besteht nämlich aus einer

GlassPane

und – darunter – einer

LayeredPane.

Die *GlassPane* ist defaultmäßig ein Objekt vom Typ `JPanel`. Die Komponenten der *GlassPane* erscheinen über allen anderen Komponenten. Die Komponenten der *GlassPane* sind daher normalerweise nicht opaque, d.h sie sind durchsichtig oder unsichtbar. Da alle Komponenten von `JRootPane` z.B. Maus-Ereignisse empfangen können, kann die *GlassPane* verwendet werden, um Maus-Ereignisse abzufangen, indem Komponenten überdeckt werden.

Die *LayeredPane* ist vom Typ `JLayeredPane` und enthält zwei Komponenten, die vom Typ `JMenuBar` (Default `null`) und `JPanel` sind. Letztere Komponente bekommt man mit der Methode `getContentPane ()`. Sie wird normalerweise für den Aufbau einer GUI verwendet.

..... *GlassPane*
_____ _____ *LayeredPane*
JMenuBar und *ContentPane*

Die Defaultwerte von `getContentPane ()` und `getGlassPane ()` sind Objekte vom Typ `JPanel`. Mit den `set`-Methoden kann man andere Behälter oder Komponenten setzen. Z.B.

```
JScrollPane rollbar  = new JScrollPane ();
JFrame   jf = new JFrame ("");
jf.setContentPane (rollbar);
```

12.5 Demonstrationsprogramm

Mit Swing wird ein Demonstrationsprogramm mitgeliefert, das alle Komponenten zeigt. Dazu muss man das Verzeichnis `SwingSet` finden und von dort aus die Demo mit

```
$ java SwingSet
```

starten. Unter JDK 1.1 muss man allerdings zuvor den `CLASSPATH` um das Verzeichnis `SwingSet` und den den `jar`File `SwingSet.jar` ergänzen.

Der Quellcode ist im Verzeichnis `.../SwingSet/src` zu finden.

12.6 Informations-Dialog

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

public class   InfoDialog extends JDialog
   implements  ActionListener
   {
   private JButton   knopf;

   public static void   main (String[] args)
      {
      InfoDialog  d = new InfoDialog (null, "Informations-Dialog Demo",
         "Das ist ein Beispiel für eine\n" +
         "graphische Benutzeroberfläche.\n");
      d.setVisible (true);
      }

   public   InfoDialog (Frame elter, String titel, String botschaft)
      {
      super (elter, titel, false);
      Container   c = getContentPane ();
      c.setLayout (new BorderLayout (15,15));
      MehrzeilenLabel   label = new MehrzeilenLabel (botschaft);
      c.add (label, BorderLayout.CENTER);

      knopf = new JButton ("Weg damit");
      knopf.addActionListener (this);
      JPanel   p = new JPanel ();
      p.setLayout (new FlowLayout (FlowLayout.CENTER, 15, 15));
      p.add (knopf);
      c.add (p, BorderLayout.SOUTH);
```

```
      this.pack ();
      }

   public void actionPerformed (ActionEvent e)
      {
      if (e.getSource () == knopf)
         {
         this.setVisible (false);
         this.dispose ();
         }
      }

   }
```

12.6.1 MehrzeilenLabel

Wir haben hier eine Klasse `MehrzeilenLabel` verwendet, die ein mehrzeiliges Label repräsentiert. Ihr Code ist:

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

public class   MehrzeilenLabel extends JPanel
   {
   public static final int LINKSBUENDIG = 0;
   public static final int ZENTRIERT = 1;
   public static final int RECHTSBUENDIG = 2;

   protected String[] zeile;
   protected int  anzZeilen;
   protected int  randBreite;       // linker und rechter Rand
   protected int  randHoehe;        // oberer und unterer Rand
   protected int  zeilenHoehe;
   protected int  zeilenOffset;     // Font-Höhe über Basislinie
   protected int[]   zeilenLaenge;  // für jede Zeile
   protected int  maxZeilenLaenge;  // maxZeilenBreite >= zeilenBreite[i]
   protected int  buendigkeit = LINKSBUENDIG;

   public   MehrzeilenLabel
      (
      String   label,
      int   randBreite,
      int   randHoehe,
      int   buendigkeit
      )
```

```
    {
    neuesLabel (label);
    this.randBreite = randBreite;
    this.randHoehe = randHoehe;
    this.buendigkeit = buendigkeit;
    }

public   MehrzeilenLabel
    (
    String   label,
    int   randBreite,
    int   randHoehe
    )
    {
    this (label, randBreite, randHoehe, LINKSBUENDIG);
    }

public   MehrzeilenLabel
    (
    String   label,
    int   buendigkeit
    )
    {
    this (label, 10, 10, buendigkeit);
    }

public   MehrzeilenLabel
    (
    String   label
    )
    {
    this (label, 10, 10, LINKSBUENDIG);
    }

protected void neuesLabel (String label)
    // macht aus dem String label ein Zeilen-Feld zeile
    {
    StringTokenizer   t = new StringTokenizer (label, "\n");
    anzZeilen = t.countTokens ();
    zeile = new String [anzZeilen];
    zeilenLaenge = new int[anzZeilen];
    for (int i = 0; i < anzZeilen; i++) zeile[i] = t.nextToken ();
    }

protected void messen ()
    // bestimmt Font-Größen und Zeilenlängen
    {
    FontMetrics fm = this.getFontMetrics (this.getFont ());
    if (fm == null) return;
```

```
      else
         {
         zeilenHoehe = fm.getHeight ();
         zeilenOffset = fm.getAscent ();
         maxZeilenLaenge = 0;
         for (int i = 0; i < anzZeilen; i++)
            {
            zeilenLaenge[i] = fm.stringWidth (zeile[i]);
            if (zeilenLaenge[i] > maxZeilenLaenge)
               maxZeilenLaenge = zeilenLaenge[i];
            }
         }
      }

   public void setLabel (String label)
      {
      neuesLabel (label);
      messen ();
      repaint ();
      }

   public void setFont (Font f)
      {
      super.setFont (f);
      messen ();
      repaint ();
      }

   public void setForeground (Color c)
      {
      super.setForeground (c);
      repaint ();
      }

   public void setBuendigkeit (int buendigkeit)
      {
      this.buendigkeit = buendigkeit;
      repaint ();
      }

   public void setRandBreite (int randBreite)
      {
      this.randBreite = randBreite;
      repaint ();
      }

   public void setRandHoehe (int randHoehe)
      {
      this.randHoehe = randHoehe;
```

```
      repaint ();
      }

   public int  getBuendigkeit ()
      {
      return  buendigkeit;
      }

   public int  getRandBreite ()
      {
      return  randBreite;
      }

   public int  getRandHoehe ()
      {
      return  randHoehe;
      }

   public void addNotify ()
      {
      super.addNotify ();
      messen ();
      }

   public Dimension  getPreferredSize ()
      {
      return new Dimension (maxZeilenLaenge + 2 * randBreite,
         anzZeilen * zeilenHoehe + 2 * randHoehe);
      }

   public Dimension  getMinimumSize ()
      {
      return new Dimension (maxZeilenLaenge,
         anzZeilen * zeilenHoehe);
      }

   public void paint (Graphics g)
      {
      super.paint (g);
      int   x;
      int   y;
      Dimension   d = this.getSize ();
      y = zeilenOffset + (d.height - anzZeilen * zeilenHoehe) / 2;
      for (int i = 0; i < anzZeilen; i++, y += zeilenHoehe)
         {
         switch (buendigkeit)
            {
            case LINKSBUENDIG:   x = randBreite; break;
            case ZENTRIERT:
```

```
            default: x = (d.width - zeilenLaenge[i]) / 2; break;
            case RECHTSBUENDIG:
               x = d.width - randBreite - zeilenLaenge[i]; break;
            }
         g.drawString (zeile[i], x, y);
         }
      }

   }
```

12.6.2 JaNeinDialog

Der Ja-Nein-Dialog ermöglicht dem Anwender die Wahl zwischen drei Antworten, typischerweise "Ja", "Nein", "Abbruch".

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

public class   JaNeinDialog extends JDialog
   implements  ActionListener
   {
   protected JButton ja = null;
   protected JButton nein = null;
   protected JButton abbruch = null;
   protected MehrzeilenLabel  label;

   public   JaNeinDialog
      (
      JFrame   elter,
      String   titel,
      String   botschaft,
      String   jaLabel,
      String   neinLabel,
      String   abbruchLabel
      )
      {
      super (elter, titel, true);
      Container   c = getContentPane ();
      c.setLayout (new BorderLayout (15,15));
      label = new MehrzeilenLabel (botschaft, 20, 20);
      c.add (label, BorderLayout.CENTER);
      JPanel   p = new JPanel ();
      p.setLayout (new FlowLayout (FlowLayout.CENTER, 15, 15));
      if (jaLabel != null)
```

```
        {
        p.add (ja = new JButton (jaLabel));
        ja.addActionListener (this);
        }
     if (neinLabel != null)
        {
        p.add (nein = new JButton (neinLabel));
        nein.addActionListener (this);
        }
     if (abbruchLabel != null)
        {
        p.add (abbruch = new JButton (abbruchLabel));
        abbruch.addActionListener (this);
        }
     c.add (p, BorderLayout.SOUTH);
     this.pack ();
     }

  protected void jaMethode () {}
  protected void neinMethode () {}
  protected void abbruchMethode () {}

  public void actionPerformed (ActionEvent e)
     {
     if (e.getSource () == ja)
        {
        this.setVisible (false); this.dispose ();
        jaMethode ();
        }
     else if (e.getSource () == nein)
        {
        this.setVisible (false); this.dispose ();
        neinMethode ();
        }
     else if (e.getSource () == abbruch)
        {
        this.setVisible (false); this.dispose ();
        abbruchMethode ();
        }
     }

  }
```

Das folgende Beispiel ist ein spezieller Ja-Nein-Dialog, der zum Beenden eines Programms oder Programmteils aufgerufen wird.

```
import javax.swing.JFrame;
```

```
import javax.swing.text.JTextComponent;

public class   WirklichEndeDialog extends JaNeinDialog
   {
   JTextComponent zustand;

   public   WirklichEndeDialog (JFrame elter, JTextComponent zustand)
      {
      super (elter, "Wirklich Ende?", "Wirklich Ende?",
         "Ja", "Nein", null);
         this.zustand = zustand;
      }

   public void jaMethode ()
      {
      System.exit (0);
      }

   public void neinMethode ()
      {
      if (zustand != null) zustand.setText ("Ende wurde abgebrochen.");
      }

   }
```

12.7 Test für alle GUI-Komponenten

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

public class   AlleKomponenten extends JFrame
   {
   JMenuBar menueLeiste;
   JMenu datei;
   JMenu hilfe;
   JMenuItem   oeffnen;
   JMenuItem   beenden;
   JMenuItem   ueber;
   JButton  weiter;
   JButton  abbrechen;
   JComboBox   auswahl;
   ButtonGroup ankreuzGruppe;
   JRadioButton[] ankreuz;
```

```
JList liste;
JTextField  textFeld;
JTextArea   textBereich;
JScrollPane textBereichScroller;
RollbaresSchmieren   schmier;
JFileChooser   dateiDialog;
JPanel   linkeSeite;
GridBagLayout  gitterSack = new GridBagLayout ();
JPanel   rechteSeite;
JPanel   knopfPanel;
Color grau = new Color (175, 175, 175);

public static void   main (String[] args)
   {
   JFrame   f = new AlleKomponenten ("GUI-Demo");
   f.pack ();
   f.setVisible (true);
   }

public   AlleKomponenten (String titel)
   {
   super (titel);

   Container   thisC = getContentPane ();

   menueLeiste = new JMenuBar ();
   setJMenuBar (menueLeiste);

   JPanel jp = new JPanel ();
   jp.setBackground (grau);
   thisC.add (jp, BorderLayout.CENTER);

   datei = new JMenu ("Datei");
   oeffnen = new JMenuItem ("Öffnen");
   datei.add (oeffnen);
   beenden = new JMenuItem ("Beenden");
   datei.add (beenden);
   menueLeiste.add (datei);

   hilfe = new JMenu ("Hilfe");
   ueber = new JMenuItem ("Über");
   hilfe.add (ueber);
   menueLeiste.add (hilfe);
   // menueLeiste.setHelpMenu (hilfe); // not yet implemented!

   weiter = new JButton ("Weiter");
   abbrechen = new JButton ("Abbrechen");

   auswahl = new JComboBox ();
```

```
auswahl.addItem ("Klee");
auswahl.addItem ("Kandinsky");
auswahl.addItem ("Schiele");

ankreuzGruppe = new ButtonGroup ();
ankreuz = new JRadioButton[3];
ankreuz[0] = new JRadioButton ("Schubert", false);
ankreuz[1] = new JRadioButton ("Chopin", true);
ankreuz[2] = new JRadioButton ("Bach", false);

for (int i = 0; i < ankreuz.length; i++)
   ankreuzGruppe.add (ankreuz[i]);

Vector   dramatiker = new Vector ();
dramatiker.addElement ("Eugene O'Neill");
dramatiker.addElement ("Max Anderson");
dramatiker.addElement ("Bert Brecht");
dramatiker.addElement ("Walter Jens");
dramatiker.addElement ("Woody Allen");
dramatiker.addElement ("Neil Simon");
dramatiker.addElement ("Horvath");
liste = new JList (dramatiker);

textFeld = new JTextField (15);
textBereich = new JTextArea (6, 40);
textBereich.setEditable (false);
textBereichScroller = new JScrollPane (textBereich);

schmier = new RollbaresSchmieren ();

dateiDialog = new JFileChooser ();

linkeSeite = new JPanel ();
linkeSeite.setBackground (grau);
linkeSeite.setLayout (gitterSack);

einpassen (linkeSeite, new JLabel ("Name:"), 0, 0, 1, 1, 10, 0, 0, 0);
einpassen (linkeSeite, textFeld, 0, 1, 1, 1);
einpassen (linkeSeite, new JLabel ("Maler:"),
   0, 2, 1, 1, 10, 0, 0, 0);
einpassen (linkeSeite, auswahl, 0, 3, 1, 1);
einpassen (linkeSeite, new JLabel ("Lieblingskomponist:"),
   0, 4, 1, 1, 10, 0, 0, 0);
einpassen (linkeSeite, ankreuz[0], 0, 5, 1, 1);
einpassen (linkeSeite, ankreuz[1], 0, 6, 1, 1);
einpassen (linkeSeite, ankreuz[2], 0, 7, 1, 1);
einpassen (linkeSeite, new JLabel ("Dramatiker:"),
   0, 8, 1, 1, 10, 0, 0, 0);
einpassen (linkeSeite, liste, 0, 9, 1, 3,
```

```
      GridBagConstraints.VERTICAL, GridBagConstraints.NORTHWEST,
      0.0, 1.0, 0, 0, 0, 0);

   rechteSeite = new JPanel ();
   rechteSeite.setBackground (grau);
   rechteSeite.setLayout (gitterSack);

   einpassen (rechteSeite, new JLabel ("Botschaften"), 0, 0, 1, 1);
   einpassen (rechteSeite, textBereichScroller, 0, 1, 1, 3,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.NORTH,
      1.0, 0.0, 0, 0, 0, 0);
   einpassen (rechteSeite, new JLabel ("Schmiererei"), 0, 4, 1, 1,
      10, 0, 0, 0);
   einpassen (rechteSeite, schmier, 0, 5, 1, 5,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER,
      1.0, 1.0, 0, 0, 0, 0);

   knopfPanel = new JPanel ();
   knopfPanel.setBackground (grau);
   knopfPanel.setLayout (gitterSack);
   einpassen (knopfPanel, weiter, 0, 0, 1, 1,
      GridBagConstraints.NONE, GridBagConstraints.CENTER,
      0.3, 0.0, 0, 0, 0, 0);
   einpassen (knopfPanel, abbrechen, 1, 0, 1, 1,
      GridBagConstraints.NONE, GridBagConstraints.CENTER,
      0.3, 0.0, 0, 0, 0, 0);

   jp.setLayout (gitterSack);
   einpassen (jp, linkeSeite, 0, 0, 1, 1,
      GridBagConstraints.VERTICAL, GridBagConstraints.NORTHWEST,
      0.0, 1.0, 10, 10, 5, 5);
   einpassen (jp, rechteSeite, 1, 0, 1, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER,
      1.0, 1.0, 10, 10, 5, 10);
   einpassen (jp, knopfPanel, 0, 1, 2, 1,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER,
      1.0, 0.0, 5, 0, 0, 0);
   }

public void einpassen
   (
   Container   behaelter,
   JComponent  komponente,
   int   x,
   int   y,
   int   breite,
   int   hoehe,
   int   fuellung,
   int   anker,
```

```
      double   wichtungX,
      double   wichtungY,
      int   oben,
      int   links,
      int   unten,
      int   rechts
      )
      {
      GridBagConstraints   c = new GridBagConstraints ();
      c.gridx = x;
      c.gridy = y;
      c.gridwidth = breite;
      c.gridheight = hoehe;
      c.fill = fuellung;
      c.anchor = anker;
      c.weightx = wichtungX;
      c.weighty = wichtungY;
      if (oben + unten + links + rechts > 0)
         c.insets = new Insets (oben, links, unten, rechts);

      ( (GridBagLayout)behaelter.getLayout ()).setConstraints (komponente, c);
      behaelter.add (komponente);
      }

   public void einpassen
      (
      Container   behaelter,
      JComponent  komponente,
      int   x,
      int   y,
      int   breite,
      int   hoehe
      )
      {
      einpassen (behaelter, komponente, x, y, breite, hoehe,
         GridBagConstraints.NONE, GridBagConstraints.NORTHWEST,
         0.0, 0.0, 0, 0, 0, 0);
      }

   public void einpassen
      (
      Container   behaelter,
      JComponent  komponente,
      int   x,
      int   y,
      int   breite,
      int   hoehe,
      int   oben,
      int   links,
```

```
      int   unten,
      int   rechts
      )
      {
      einpassen (behaelter, komponente, x, y, breite, hoehe,
         GridBagConstraints.NONE, GridBagConstraints.NORTHWEST,
         0.0, 0.0, oben, links, unten, rechts);
      }

   }
```

12.7.1 Rollbares Gekritzel

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

public class   RollbaresSchmieren extends JPanel
   implements  MouseMotionListener, AdjustmentListener
   {
   JPanel   leinwand = new JPanel ();
   JScrollBar  hbar = new JScrollBar (JScrollBar.HORIZONTAL);
   JScrollBar  vbar = new JScrollBar (JScrollBar.VERTICAL);
   java.util.Vector  linie = new java.util.Vector (100, 100);
   int   offsetX;
   int   offsetY;
   int   altesX;
   int   altesY;
   int   leinwandBreite;
   int   leinwandHoehe;

   public   RollbaresSchmieren ()
      {
      this.setLayout (new BorderLayout (0, 0));
      this.add (leinwand, BorderLayout.CENTER);
      this.add (hbar, BorderLayout.SOUTH);
      this.add (vbar, BorderLayout.EAST);
      leinwand.addMouseMotionListener (this);
      hbar.addAdjustmentListener (this);
      vbar.addAdjustmentListener (this);
      }

   public void paint (Graphics g)
      {
      super.paint (g);
```

```
      Zeile z;
      Graphics leinwandG = leinwand.getGraphics ();
      for (int i = 0; i < linie.size (); i++)
         {
         z = (Zeile)linie.elementAt (i);
         leinwandG.drawLine (z.x1 - offsetX, z.y1 - offsetY,
            z.x2 - offsetX, z.y2 - offsetY);
         }
      }

   public void mouseDragged (MouseEvent e)
      {
      Graphics g = leinwand.getGraphics ();
      g.drawLine (altesX, altesY, e.getX (), e.getY ());
      linie.addElement (new Zeile (altesX + offsetX, altesY + offsetY,
         e.getX () + offsetX, e.getY () + offsetY));
      altesX = e.getX ();
      altesY = e.getY ();
      }

   public void mouseMoved (MouseEvent e)
      {
      altesX = e.getX ();
      altesY = e.getY ();
      }

   public void adjustmentValueChanged (AdjustmentEvent e)
      {
      if (e.getSource () == hbar)
         {
         offsetX = e.getValue ();
         this.update (leinwand.getGraphics ());
         }
      else if (e.getSource () == vbar)
         {
         offsetY = e.getValue ();
         this.update (leinwand.getGraphics ());
         }
      }


   public synchronized void   setBounds (int x, int y, int width, int height)
      {
      super.setBounds (x, y, width, height);
      Graphics g = leinwand.getGraphics ();
      if (g != null)
         {
         Dimension   hbarGroesse = hbar.getSize ();
         Dimension   vbarGroesse = vbar.getSize ();
```

```
         leinwandBreite = width - vbarGroesse.width;
         leinwandHoehe = height - hbarGroesse.height;
         hbar.setValues (offsetX, leinwandBreite, 0, 1000 - leinwandBreite);
         vbar.setValues (offsetY, leinwandHoehe, 0, 1000 - leinwandHoehe);
         hbar.setBlockIncrement (leinwandBreite / 2);
         vbar.setBlockIncrement (leinwandHoehe / 2);
         this.update (g);
         }
      }

   }

class Zeile
   {
   public   Zeile (int x1, int y1, int x2, int y2)
      {
      this.x1 = x1;
      this.y1 = y1;
      this.x2 = x2;
      this.y2 = y2;
      }

   public int  x1;
   public int  x2;
   public int  y1;
   public int  y2;
   }
```

12.8 Interaktive Ereignisse

12.8.1 Ereignis-Modell

In der Version JDK1.0 basierte das Ereignis-Modell auf Vererbung. Der API-Anwender hat GUI-Komponenten vererbt und deren Methoden `action` oder `handleEvent` für seine Zwecke überschrieben. Wenn die Methoden `true` zurückgaben, war das Ereignis verbraucht und wurde nicht weiter – d.h. von einem Behälter der Komponente – behandelt. Der Anwender hat entweder für jede Komponente eine Subklasse geschrieben, die die Ereignisse behandelt hat, oder er hat einen geeigneten Behälter der Komponenten definiert, der alle Ereignisse behandelt hat. Damit hat man entweder sehr viele Klassen geschaffen oder musste in den Ereignisbehandlungsmethoden komplizierte Unterscheidungen treffen. Auf weitere Nachteile dieses Modells – insbesondere bei komplizierten Anwendungen – soll hier nicht eingegangen werden.

In JDK1.1 wird ein Ereignis-Modell eingeführt, das auf Delegation basiert und folgende Vorteile bietet:

- Es ist leicht zu lernen.
- Saubere Trennung zwischen Anwendungs- und GUI-Code.
- Robuster Ereignis-Behandlungs-Code, "bei dem der Compiler schon viel sieht".
- Gestaltungsmöglichkeiten für Ereignis-Flüsse bzw -Weiterleitung.

Ereignistypen gehören zu einer Klassen-Hierarchie, die bei der Klasse `java.util.EventObject` beginnt. Die AWT-Ereignisse leiten sich von der Subklasse `java.awt.AWTEvent` ab.

Ein Ereignis wird von einer ***Source*** erzeugt und an einen ***Listener*** weitergeleitet, indem eine Methode des Listeners mit einem Objekt einer Ereignisklasse als Argument aufgerufen wird. Ein Listener ist ein Objekt, das eine spezifische ***EventListener*** Schnittstelle implementiert. Diese EventListener Schnittstellen leiten sich ab von `java.util.EventListener`. Ein EventListener deklariert Methoden, die von der Ereignis-Source aufgerufen werden, wenn ein Ereignis eingetreten ist.

Eine Ereignis-Source ist ein Objekt, das Ereignisse "feuert". Die Art der Ereignisse, die gefeuert werden können, sind durch die angebotenen

```
Source.add<Ereignistyp>Listener (<Ereignistyp>Listener);
```

definiert. Mit diesen Methoden werden Listener für spezifische Ereignisse registriert.

`<Ereignistyp>` ist z.B. `Action`, `Mouse`, `MouseMotion`, `Item`.

In einem Anwendunsgprogramm ist eine GUI-Komponente typischerweise eine Source. Der Listener ist typischerweise ein Adapter-Objekt, das die entsprechende Listener-Schnittstelle implementiert und den Anwendungs-Code repräsentiert. Dort kann das Ereignis auch weitergereicht werden, indem entsprechende Listener-Methoden aufgerufen werden. Der Listener könnte auch wieder eine GUI-Komponente sein.

Da Schnittstellen immer komplett implementiert werden müssen, dies aber für viele Anwendungen unnötig ist, werden für die sogenannten ***low-level*** Listener **Adapter**-Klassen angeboten, die geerbt werden können und wo dann nur die Methoden implementiert werden müssen, die man für die eigene Anwendung benötigt. Für die sogenannten ***semantic*** Listener werden keine Adapter-Klassen angeboten, da diese Schnittstellen nur eine Methode deklarieren, deren "Nicht-Implementation" keinen Sinn machen würde.

Die Daten der Ereignis-Klassen sind komplett gekapselt und nur über `set-` und `get`-Methoden zugänglich. Wenn es eine `set`-Methode gibt, d.h. der Listener kann das Ereignis verändern, dann wird das Ereignis *kopiert*, damit jeder Listener eine eigene modifizierbare Kopie des Ereignisses hat.

Da unterschiedliche Ereignisse Objekte derselben Klasse sein können, wird ein Ereignis manchmal durch eine ID-Nummer repräsentiert. Die Klasse `MouseEvent` repräsentiert z.B. die verschiedenen Maus-Zustände, die sich durch eine ID-Nummer unterscheiden. Wenn eigene Ereignisklassen geschrieben werden und ID-Nummern verwendet werden, dann sollten diese größer als

```
java.awt.AWTEvent.RESERVED_ID_MAX
```

gewählt werden.

Noch einige Details: Ein Ereignis wird zunächst an die "erzeugende" Komponente ausgeliefert und dort vorverarbeitet, bevor es an die Ereignis-Listener weitergereicht wird. Wenn man eigene AWT-Komponenten über eine Subklasse schreibt, dann kann man die Ereignisse zunächst abfangen, bevor sie an den Listener gehen, indem die Methode

```
processEvent (AWTEvent e)
```

der Klasse `Component` überschrieben wird. Denn diese Methode bekommt das Ereignis zuerst.

`processEvent` übergibt normalerweise das Ereignis an eine klassenspezifische Methode

```
process<X>Event (AWTEvent e)
```

(z.B. `processFocusEvent` oder `processActionEvent`). `processEvent` entspricht etwa dem `handleEvent` aus JDK1.0.

Bei dieser Art von low-level Ereignisprogrammierung muss man der Komponente noch mitteilen, dass sie Ereignisse abfangen soll, indem die Methode

```
enableEvents (long Ereignismaske)
```

aufgerufen wird. Typischerweise sollten nicht behandelte Ereignisse mit

```
super.processEvent (Ereignis);
oder
super.process<X>Event (Ereignis);
```

an die Superklasse durchgereicht werden.

Bemerkung: Offenbar werden an manche Komponenten (z.B. `JPanel`) `KeyEvent`s nicht ausgeliefert, auch wenn `KeyListener`s registriert sind. Dann hilft eventuell für solch eine Komponente der Aufruf:

```
aJpanel.setFocusable (true);
```

12.8.2 Ereignis-Beispiel

Mit dem folgenden Applet können vom Benutzer erzeugte Ereignisse (Maus- und Tastatur-Ereignisse) getestet werden. Das Programm benutzt die Methode `paint (Graphics g)`, um das Ergebnis des Tests auszugeben.

Wenn ein Mausknopf gedrückt wurde, dann wird der Modifikator `BUTTON1_MASK` bzw `BUTTON2_MASK` bzw `BUTTON3_MASK` gesetzt. Bei Systemen mit zwei oder drei Mausknöpfen wird zusätzlich beim rechten Mausknopf der `<AltGr>`- (`META_MASK`) oder der Meta-Tastatur-Modifikator, beim mittleren Mausknopf der `<Alt>`-Tastatur-Modifikator (`ALT_MASK`) gesetzt. Diese Masken sind in der Klasse `InputEvent` definiert.

Ereignis-Testprogramm:

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

public class   EreignisTester extends JApplet
//public class EreignisTester extends JPanel
   // geht mit Keys nur, wenn setFocusable (true)

   implements  MouseListener, MouseMotionListener,
               FocusListener, KeyListener
   {
   String   botschaft = new String ("");

   public static void   main (String[] argument)
      {
      JFrame f = new JFrame ("EreignisTester");
      Container   c = f.getContentPane ();

      EreignisTester  s = new EreignisTester ();
      s.init ();
      c.add (s, BorderLayout.CENTER);
      f.pack ();
      f.setSize (800, 400);
      f.setVisible (true);
      s.start (); // streichen bei JPanel
      }

   public void init ()
      {
      this.addMouseListener (this);
      this.addMouseMotionListener (this);
      this.addFocusListener (this);

      //this.setFocusable (true);   // notwendig für JPanel
      this.addKeyListener (this);
      }

   public void paint (Graphics g)
      {
      super.paint (g);
      g.drawString ("Mause und Tippe in diesem Fenster!", 20, 20);
      g.drawString (botschaft, 20, 50);
      }

   public void mousePressed (MouseEvent e)
      {
```

```
    botschaft =  modusTaste (e.getModifiers ())
         + " mousePressed wurde aufgerufen (" + e + ").";
    repaint ();
    }

  public void mouseReleased (MouseEvent e)
    {
    botschaft = modusTaste (e.getModifiers ())
        + " mouseReleased wurde aufgerufen. (" + e + ").";
    repaint ();
    }

  public void mouseEntered (MouseEvent e)
    {
    botschaft = modusTaste (e.getModifiers ())
         + " mouseEntered wurde aufgerufen (" + e + ").";
    repaint ();
    }

  public void mouseExited (MouseEvent e)
    {
    botschaft = modusTaste (e.getModifiers ())
         + " mouseExited wurde aufgerufen (" + e + ").";
    repaint ();
    }

  public void mouseClicked (MouseEvent e)
    {
    botschaft = modusTaste (e.getModifiers ())
         + " mouseClicked wurde aufgerufen (" + e + ").";
    repaint ();
    }

  public void mouseDragged (MouseEvent e)
    {
    botschaft = modusTaste (e.getModifiers ())
         + " mouseDragged wurde aufgerufen (" + e + ").";
    repaint ();
    }

  public void mouseMoved (MouseEvent e)
    {
    botschaft = modusTaste (e.getModifiers ())
         + " mouseMoved wurde aufgerufen (" + e + ").";
    repaint ();
    }

  public void focusGained (FocusEvent e)
    {
```

```
      botschaft = "focusGained wurde aufgerufen (" + e + ").";
      repaint ();
      }

   public void focusLost (FocusEvent e)
      {
      botschaft = "focusLost wurde aufgerufen (" + e + ").";
      repaint ();
      }

   public void keyPressed (KeyEvent e)
      {
      botschaft = modusTaste (e.getModifiers ())
         + e.getKeyText (e.getKeyCode ())
         + " keyPressed wurde aufgerufen (" + e + ").";
      repaint ();
      }

   public void keyReleased (KeyEvent e)
      {
      botschaft = modusTaste (e.getModifiers ())
         + e.getKeyChar ()
         + " keyReleased wurde aufgerufen (" + e + ").";
      repaint ();
      }

   public void keyTyped (KeyEvent e)
      {
      botschaft = modusTaste (e.getModifiers ())
         + e.getKeyChar ()
         + " keyTyped wurde aufgerufen (" + e + ").";
      repaint ();
      }

   private String modusTaste (int kode)
      {
      String   s = "";
      if (kode == 0) return s;
      if ( (kode & InputEvent.SHIFT_MASK) != 0) s += "<Shift>";
      if ( (kode & InputEvent.CTRL_MASK) != 0) s += "<Strg>";
      if ( (kode & InputEvent.META_MASK) != 0) s += "<AltGr>";
      if ( (kode & InputEvent.ALT_MASK) != 0) s += "<Alt>";
      if ( (kode & InputEvent.BUTTON1_MASK) != 0) s += "<Button1>";
      if ( (kode & InputEvent.BUTTON2_MASK) != 0) s += "<Button2>";
      if ( (kode & InputEvent.BUTTON3_MASK) != 0) s += "<Button3>";
      return s;
      }

   }
```

12.9 Test für einige GUI-Ereignisse

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

public class   AlleEreignisse extends AlleKomponenten
   implements  ActionListener, ItemListener, FocusListener,
      ListSelectionListener
   {
   public static void   main (String[] args)
      {
      JFrame   f = new AlleEreignisse ("AlleEreignisse-Demo");
      f.pack ();
      f.setVisible (true);
      }

   public   AlleEreignisse (String titel)
      {
      super (titel);
      textFeld.addActionListener (this);
      liste.addListSelectionListener (this);
      weiter.addActionListener (this);
      abbrechen.addActionListener (this);
      oeffnen.addActionListener (this);
      beenden.addActionListener (this);
      ueber.addActionListener (this);
      auswahl.addItemListener (this);
      ankreuz[0].addItemListener (this);
      ankreuz[1].addItemListener (this);
      ankreuz[2].addItemListener (this);
      this.addFocusListener (this);
      }

   public void actionPerformed (ActionEvent e)
      {
      if (e.getSource () == textFeld)
         {
         textBereich.append ("Ihr Name ist: " + e.getActionCommand () + "\n");
         }
      else if (e.getSource () == weiter)
         {
```

```
        textBereich.append ("Weiter-Knopf wurde angeklickt.\n");
        }
     else if (e.getSource () == abbrechen)
        {
        textBereich.append ("Abbrechen-Knopf wurde angeklickt.\n");
        }
     else if (e.getSource () == oeffnen)
        {
        textBereich.append ("Sie wählten Öffnen.\n");
        //dateiDialog.setVisible (true);
        dateiDialog.showOpenDialog (this);
        textBereich.append ("Sie wählten Datei: "
           + dateiDialog.getSelectedFile ().getName () + "\n");
        }
     else if (e.getSource () == beenden)
        {
        JaNeinDialog   d = new WirklichEndeDialog (this, textBereich);
        d.setVisible (true);
        }
     else if (e.getSource () == ueber)
        {
        InfoDialog  d = new InfoDialog (this,
           "Über GUI Demo",
           "Diese Demo zeigt einige GUI-Elemente.\n");
        d.setVisible (true);
        textBereich.append ("Sie wählten Über.\n");
        }
     }

  public void itemStateChanged (ItemEvent e)
     {
     if (e.getStateChange () == ItemEvent.SELECTED)
        {
        if (e.getSource () == auswahl)
           {
           textBereich.append ("Der gewählte Maler ist: " +
              e.getItem () + "\n");
           }
        else if (
           e.getSource () == ankreuz[0]
           || e.getSource () == ankreuz[1]
           || e.getSource () == ankreuz[2]
           )
           {
           textBereich.append ("Ihr Lieblingskomponist ist: " +
              ((AbstractButton) e.getItem ()).getText () + "\n");
           }
        }
     }
```

```
public void valueChanged (ListSelectionEvent e)
   {
   if (!e.getValueIsAdjusting ())
      {
      if (e.getSource () == liste)
         {
         Object[] p = liste.getSelectedValues ();
         textBereich.append ("Sie haben ausgewählt: \n");
         for (int i = 0; i < p.length; i++)
            {
            textBereich.append ("   " + p[i] + "\n");
            }
         }
      }
   }

public void focusGained (FocusEvent e)
   {
   repaint ();
   }

public void focusLost (FocusEvent e)
   {
   }
}
```

12.10 Übungen

Übung GUI: Schreiben Sie das Programm `Produkt` mit einer graphischen Benutzeroberfläche.

Kapitel 13

Applets

In diesem Kapitel werden wir an Hand von einfachen Beispielen zeigen, wie Applets zu programmieren sind. Die Applets können aus einer HTML-Datei über einen Web-Browser oder mit dem Programm `appletviewer` gestartet werden. Zunächst zeigen wir zwei einfache Beispiele und fassen dann die Grundlagen zusammen.

Als Literatur zu diesem Thema ist Flanagan [10] und Horstmann/Cornell [18] [19] zu empfehlen. Letztere gehen sehr kritisch auf die Verwendung von Applets ein. Demnach sollte man die Verwendung von Applets möglichst vermeiden.

13.1 "Hello-World"-Applet

Der folgende Code zeigt das "Hello-World"-Programm als Applet:

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

public class   GutenTagApplet1 extends JApplet
   {
   public void paint (Graphics g)
      {
      super.paint (g);
      g.drawString ("Guten Tag", 127, 47);
      }
   }
```

Die zugehörige HTML-Datei muß folgenden Code enthalten:

```
<APPLET code="GutenTagApplet1.class" width=500 height=300>
```

```
</APPLET>
```

Die Zahlenangaben sind hier in Pixel.

Mit `import javax.swing.*;` wird der Name der Klasse `JApplet` bekannt gemacht. Eine Klasse, die als Applet laufen soll, muß von der Klasse `JApplet` erben.

`import java.awt.*;` wird für die Graphik benötigt.

Die Methode `paint` mit der Signatur

```
public void paint (Graphics g)
```

wird von einem Applet-Viewer oder Web-Browser aufgerufen. Dabei wird ein Objekt `g` der Klasse `Graphics` übergeben, das von der Klasse `JApplet` angelegt wurde. Ein Applet, das irgendwie eine Graphik-Ausgabe macht, muß im allgemeinen die Methode `paint` überschreiben.

Besser ist es, eine neue (innere) Klasse zu definieren, die einen `JComponent`-Behälter – typischerweise `JPanel` – erweitert, und dort die Methode

```
paintComponent (Graphics g)
```

mit dem Graphik-Code zu überschreiben. Auch hier ist wichtig, daß das `paintComponent` für die Superklasse aufgerufen wird:

```
JPanel   jp = new JPanel ()
   {
   public void paintComponent (Graphics g)
      {
      super.paintComponent (g);
      ...
      ...
      }
   }
```

Für das Graphik-Objekt `g` wird die Methode `drawString` aufgerufen, mit der eine Zeichenkette am Pixel (`x = 127`, `y = 47`) (links oben ist `(0, 0)`) bezogen auf die linke untere Ecke der Zeichenkette ausgegeben wird.

13.2 "Hello-World"-Applet – komplizierter

Das folgende "Hello-World"-Applet ist etwas aufwendiger gestaltet. Hierbei lernen wir einige wenige Graphikmöglichkeiten von Java kennen.

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

public class   GutenTagApplet2 extends JApplet
```

```
{
protected Font zeichenSatz;
protected int  zeichenGröße = 72;
protected Color   textFarbe = Color.yellow;
private int ovalx = 20;
private int ovaly = 20;
private int ovalb = 430;
private int ovalh = 200;
private int arcb = 30;
private int arch = 30;

public void init ()
   {
   zeichenSatz = new Font ("sanscerif", Font.BOLD, zeichenGröße);
   }

public void paint (Graphics g)
   {
   super.paint (g);

   g.setColor (Color.blue);
   g.fillRoundRect (ovalx, ovaly, ovalb, ovalh, arcb, arch);

   g.setColor (Color.magenta);
   for (int i = 0; i < 6; i++)
      g.drawRoundRect (
         ovalx - i, ovaly - i,
         ovalb + 2 * i, ovalh + 2 * i, arcb, arch);

   g.setColor (textFarbe);
   g.setFont (zeichenSatz);
   g.drawString ("Guten Tag", ovalx + 25,
      ovaly + ovalh / 2 + zeichenGröße / 2);
   }

}
```

Die Methode `init` mit der Signatur

```
public void init ()
```

wird genau einmal vom Browser aufgerufen, nachdem das Applet-Objekt erzeugt wurde. Wir verwenden hier diese Methode, um einen Zeichensatz anzulegen, der im `java.awt`-Paket angeboten wird.

Während `init` nur einmal im Leben eines Applets aufgerufen wird, wird die Methode `paint` häufig aufgerufen, nämlich immer dann, wenn das Applet neu gezeichnet werden muß.

In der Methode `paint` wird zunächst ein mit Blau gefülltes Oval mit linker oberer Ecke (`ovalx`, `ovaly`), der Breite `ovalb` und der Höhe `ovalh` (in Pixel) gezeichnet.

Den dicken Magenta-Rand erzeugen dann vier versetzte leere Ovale in Magenta.

Schließlich setzen wir die Farbe auf Gelb, ändern den Zeichensatz und zeichnen die Zeichenkette.

13.3 Grundlagen

- Ein Applet ist eine Subklasse der Swing-Klasse `JApplet`, die sich wiederum ableitet von der AWT-KLasse `Applet`.
- Mit der Methode `getContentPane ()` bekommt man einen Behälter, der mit Swing-GUI-Komponenten gefüllt werden kann. Der Default-Layout-Manager ist `BorderLayout`.
- Der Lebenszyklus eines Applets wird durch die vier Methoden `init ()`, `start ()`, `stop ()` und `destroy ()` bestimmt.
  - `init ()`: Diese Methode wird von einem Browser oder Appletviewer nur aufgerufen, wenn das Applet geladen ***(load)*** oder wiedergeladen ***(reload)*** wird. In dieser Methode werden alle (einmaligen) Initialisierungen programmiert. Ihr Inhalt ist typischerweise der Inhalt eines Konstruktors. `init ()` wird pro Applet-Objekt nur ein einzigesmal aufgerufen. Trotzdem sollte stattdessen *kein* Konstruktor geschrieben werden, da der Konstruktor eines Applets häufig zu früh, d.h. ehe die volle Umgebung des Applets erzeugt ist, aufgerufen wird. Insbesondere funktionieren die Bild-Lade-Methoden in einem Konstruktor nicht.

    Am Ende der Methode sollte man alle Aktivitäten, d.h. Threads des Applets erzeugen und starten, aber dann sofort wieder anhalten (siehe `stop ()`).
  - `start ()`: Diese Methode wird aufgerufen, wenn das Applet geladen und initialisiert (nach `init ()`) ist oder wenn die Browser-Seite, die das Applet enthält, wieder besucht wird. Die Methode `start` sollte so überschrieben werden, daß sie die in `stop` eventuell vorübergehend freigegebenen Betriebsmittel wieder anfordert und/oder daß sie angehaltene Threads wieder weiterlaufen läßt.
  - `stop ()`: Diese Methode wird aufgerufen, wenn die Seite des Applets verlassen wird. Ein Applet, das `start` überschreibt, sollte auch `stop` überschreiben, um eventuell gestartete Threads anzuhalten oder Betriebsmittel vorübergehend freizugeben. Da die Thread-Methoden `suspend` und `resume` nicht mehr verwendet werden sollen (`deprecated`), muß man zum Anhalten der Threads ein Flag setzen, das in regelmäßigen Abständen von den Threads geprüft wird (siehe Beispiel `Aktiv`).
  - `destroy ()`: Die Methode `destroy` wird vor dem Entladen des Applets aufgerufen. Davor aber wird immer die Methode `stop` aufgerufen. Mit `destroy` können vor dem Entladen des Applets eventuell verwendete Resourcen (endgültig) freigegeben werden. Außerdem sollten für alle in `init` gestarteten Threads Flags gesetzt, die die Threads an ihr natürliches Ende laufen lassen. Dieses Thread-Ende kann mit einem `join` abgewartet werden.
- **Laden** eines Applets: Wenn ein Applet geladen wird, dann werden
  - eine Instanz der Klasse des Applets erzeugt,

  - die Methode `init` aufgerufen,
  - die Methode `start` aufgerufen.

- Beim **Verlassen der Seite** des Applets oder beim **Ikonifizieren** des Applets wird die Methode `stop` aufgerufen. Daher hat man hier die Möglichkeit, alle Aktivitäten anzuhalten.
- Beim **Wiederbesuch der Seite** oder **Restaurieren** des Applets wird die Methode `start` aufgerufen, die nun die Aktivitäten fortsetzen kann.
- Beim **Entladen** eines Applets wird `stop` und `destroy` aufgerufen.
- Beim **Wiederladen** des Applets werden nacheinander `stop`, `destroy`, `init` und `start` aufgerufen.
- `repaint ()`, `update (Graphics g)` und `paint (Graphics g)`:
  Das sind drei Methoden, die in der abstrakten Klasse `Component` definiert sind, die von der Klasse `Applet` und ihren Subklassen geerbt werden und die im Prinzip alle von den Subklassen von `Applet` überschrieben werden können, wobei normalerweise nur `paint` (bzw besser: `paintComponent` einer Komponente des Applets) überschrieben wird, das in `Component` leer definiert ist.

  - `repaint`:
    * `public void repaint ()`
    * `public void repaint (long ms)`
    *
      ```
      public void repaint (int x, int y,
                           int width, int height)
      ```
    * `public void repaint (long ms, int x, int y,`
      `int width, int height)`

    Wenn diese Methode aufgerufen wird, dann wird – evtl. innerhalb `ms` Millisekunden – die Wiederherstellung einer spezifizierbaren rechteckigen Fläche oder der gesamten Fläche der Komponente "eingeplant". Das AWT-System wird die Komponente wiederherstellen, indem die Methode `update` aufgerufen wird.

    `repaint` kann vom Programmierer aufgerufen werden.

  - `public void update (Graphics g)`:
    Diese Methode wird als Folge eines `repaint`-Aufrufs aufgerufen. Sie ist defaultmäßig so implementiert, daß die Komponente mit der Hintergrundfarbe gefüllt wird und dann `paint` aufgerufen wird. Es folgt die Default-Implementation:

    ```
    public void update (Graphics g)
       {
       g.setColor (getBackground ());
       g.fillRect (0, 0, width, height);
       g.setColor (getForeground ());
       paint (g);
       }
    ```

    Diese Methode wird oft überschrieben, da es häufig nicht nötig ist, die ganze Fläche noch einmal zu zeichnen. Mit der `Graphics`-Methode `Rectangle getClipBounds ()`

kann die wiederherzustellende Fläche bestimmt werden. (Diese Methode liefert `null`, wenn die ganze sichtbare Fläche gemeint ist.)

Auch wenn die eigene Implementation von `update` die Methode `paint` *nicht* aufruft, sondern selbst den Graphik-Code enthält, dann sollte trotzdem `paint` implementiert werden, (indem es z.B. `update` aufruft), da das AWT-System `paint` aufruft (siehe unten).

- `public void paint (Graphics g)`:
  Diese Methode wird automatisch vom AWT-System aufgerufen, wenn die Komponente durch andere Fenster überdeckt war und wiederhergestellt werden muß.

  In diese Methode gehört der eigentliche Graphik-Code. Im allgemeinen muß sie so überschrieben werden, daß `super.paint (...)` aufgerufen wird.
- `public void paintComponent (Graphics g)`:
  Diese Methode ist `paint` vorzuziehen. Leider ist sie nicht in `JApplet` definiert und kann daher nichtbeinfach überschrieben werden. Man muß also extra eine Klasse definieren, die z.B. von `JPanel` erbt, und muß dann dort `paintComponent` überschreiben. Ein Objekt der Klasse muß dann dem Behälter von `JApplet` hinzugefügt werden. Auch `paintComponent` muß so überschrieben werden, daß `super.paintComponent (...)` aufgerufen wird.

Mit dem folgenden Programm können die Phasen eines Applets verfolgt werden. (Um unendliche Rekursion zu vermeiden, wurde hier der Graphik-Code in die Methode `addierePhase` gelegt.)

```
import   java.awt.Graphics;
import   javax.swing.JApplet;

public class   Lebenszyklus extends JApplet
   {
   public void init ()
      {
      puffer = new StringBuffer ();
      addierePhase ("init ");
      }

   public void start ()
      {
      addierePhase ("start ");
      }

   public void stop ()
      {
      addierePhase ("stop ");
      }

   public void destroy ()
      {
      addierePhase ("destroy ");
      }
```

```
public void repaint ()
   {
   addierePhase ("repaint ");
   super.repaint ();
   }

public void repaint (long ms)
   {
   addierePhase ("repaint(ms) ");
   super.repaint (ms);
   }

public void repaint (int x, int y, int width, int height)
   {
   addierePhase ("repaint(x) ");
   super.repaint (x, y, width, height);
   }

public void repaint (long ms, int x, int y, int width, int height)
   {
   addierePhase ("repaint(ms,x) ");
   super.repaint (ms, x, y, width, height);
   }

public void update (Graphics g)
   {
   addierePhase ("update ");
   super.update (g);
   }

public void paint (Graphics g)
   {
   super.paint (g);
   addierePhase ("paint ");
   super.paint (g);
   }

void  addierePhase (String phase)
   {
   System.out.println (phase);
   puffer.append (phase);
   Graphics g = getGraphics ();
   g.drawRect (5, 5, getSize ().width - 11, getSize ().height - 11);
   g.drawString (puffer.toString (), 10, 25);
   }

StringBuffer   puffer;
}
```

13.4 Applet-Parameter

Man kann einem Applet Zeichenketten (`String`) als Parameter mitgeben. Unserem "Guten Tag"-Applet wollen wir die Farbe des Textes und die Größe des Textes als Parameter mitgeben.

Das angegebene Beispiel führt die Methode `getParameter` und noch zwei weitere Methoden ein:

- `public String getParameter (String parameter)` : Sie gibt den Wert des Parameters mit Namen `parameter` als `String` zurück.
- `public String[][] getParameterInfo ()` : Web-Browsers und Applet-Viewer rufen eventuell diese Methode auf, um dem Benutzer die möglichen Parameter anzuzeigen.
- `public String getAppletInfo ()` : Diese Methode kann von einer "Über"-Dialogbox verwendet werden und sollte von jedem Applet implementiert werden.

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

public class   GutenTagApplet3 extends GutenTagApplet2
   {
   public void init ()
      {
      String   s = this.getParameter ("Zeichengröße");
      try { zeichenGröße = Integer.parseInt (s); }
      catch (NumberFormatException e) { zeichenGröße = 72; }
      super.init ();

      s = this.getParameter ("Textfarbe");
      if (s.equals ("Gelb")) textFarbe = Color.yellow;
      else if (s.equals ("Rot")) textFarbe = Color.red;
      else if (s.equals ("Grün")) textFarbe = Color.green;
      else if (s.equals ("Blau")) textFarbe = Color.blue;
      else if (s.equals ("Schwarz")) textFarbe = Color.black;
      else if (s.equals ("Weiß")) textFarbe = Color.white;
      }

   public String[][] getParameterInfo ()
      {
      String [][] parameterInformation =
         {
         // Parametername, Parametertyp, Parameterbeschreibung
         {"Zeichengröße", "int", "Größe des Zeichensatzes in Pixel"},
         {"Textfarbe", "Gelb, Rot usw", "Farbe der Schrift"}
         };
```

```
      return parameterInformation;
      }

   public String  getAppletInfo ()
      {
      return "GutenTag-Programm vom 14.5.1997";
      }

   }
```

Die HTML-Datei enthält dazu folgenden Code:

```
<APPLET code="GutenTagApplet3.class" width=500 height=300>
<PARAM name="Zeichengröße" value="36">
<PARAM name="Textfarbe" value="Grün">
</APPLET>
```

13.5 HTML-Datei

Im APPLET-*tag* sind elf Attribute spezifizierbar. Drei Attribute (`code` oder `object`, `width`, `heigth`) müssen angegeben werden, die übrigen sind optional.

Die Attribute haben folgende Bedeutung:

`codebase`: Basis-URL des Applets. Wenn `codebase` nicht spezifiziert wird, dann wird als `codebase` das Verzeichnis angenommen, in dem die HTML-Datei steht. `codebase` kann absolut oder relativ zum HTML-Datei-Verzeichnis angegeben werden.
`codebase="http://www.ba-stuttgart.de/~kfg/meineApplets"`
oder `codebase="meineApplets"`

`code`: Name der Appletklassen-Datei relativ zur `codebase` (mit oder) ohne Extension `.class`. Dieses Attribut wird auf jeden Fall benötigt. Hier dürfen auf keinen Fall irgendwelche Pfad-Angaben erscheinen. Die Verwendung der Extension `.class` ist offenbar etwas sicherer.

`object`: Anstatt von `code` kann mit `object` eine Datei angegeben werden, in der sich ein serialisiertes Applet-Objekt befindet (siehe Kapitel Serialisierung).

`archive`: Hier können Java-Archive spezifiziert werden, die vor dem Start des Applets geladen werden. Die Applet-Klasse selbst kann in einem der Archive enthalten sein.

`width` **und** `heigth`: Breite und Höhe des Applet in Pixeln

`alt`: Spezifiert Text der ausgegeben werden soll, wenn der Browser zwar das APPLET-tag versteht (APPLET-fähig ist), aber keine Java-Applets laufen lassen kann. Ein Browser, der schon das APPLET-tag nicht versteht, überliest das APPLET-tag und die PARAM-tags, stellt aber sonstiges HTML-Material in der APPLET-Klammer dar. Ein APPLET-fähiger Browser dagegen ignoriert zusätzliches HTML-Material in der APPLET-Klammer.

name: Dem Applet kann ein Name gegeben werden, über den mit anderen Applets auf derselben Web-Seite kommuniziert werden kann. Angenommen der Name eines Applets sei `"Oskar"` und der der Appletklasse sei `OskarApplet`, die eine Methode `kontakt ()` anbietet, dann kann ein zweites Applets mit folgendem Code-Fragment die Kommunikation mit Oskar aufnehmen:

```
AppletContext  kontext = getAppletContect ();
OskarApplet oskar = (OskarApplet) kontext.getApplet ("Oskar");
oskar.kontakt ();
```

Die Namen der Applets können natürlich auch als Applet-Parameter übergeben werden.

align: Spezifiziert die Ausrichtung des Applets. Dieselben Werte wie beim IMG-tag sind möglich: `left`, `right`, `top`, `texttop`, `middle`, `absmiddle`, `baseline`, `bottom`, `absbottom`.

vspace und hspace: Spezifieren in Pixeln einen Rand um das Applet.

Das Attribut `code` gibt die Klasse an, die das Applet enthält. Außerdem benötigt ein Applet häufig noch andere Klassen. Wir wollen noch einmal genauer auf die Frage eingehen, woher diese Klassen geladen werden?

Die Browser gehen in folgender Reihenfolge vor, bis die gewünschte Klasse gefunden ist:

1. Es wird auf der Machine gesucht, wo der Browser läuft. Dabei wird der `CLASSPATH` des Benutzers verwendet, der den Browser benutzt.
2. Es wird in den `jar`- oder `zip`-Archiven gesucht, die beim Attribute `archive` angegeben sind.
3. Es wird relativ zu dem beim Attribute `codebase` angegebenen Verzeichnis gesucht.
4. Es wird relativ zur *document-base*, d.h. dem Verzeichnis, wo die HTML-Datei steht, gesucht.

Wenn `codebase` angegeben ist, dann wird offenbar nur dort nach Klassen gesucht. Auch die `.jar`-Archive müssen im `codebase`-Verzeichnis sein.

Wenn ein Applet außer Klassendateien auch andere Dateien (Text oder Bild) benötigt, dann können die auch in einem `jar`-File untergebracht werden. Sie können dann über irgendeine Klasse im `jar`-File geholt werden:

```
URL   url = IrgendeineKlasse.class.getResource ("Dateiname");
```

Mit der URL kann man dann einen Stream öffen, um die Datei dann zu lesen, oder man gibt die URL der Methdoe `getImage (URL url)`, um eine Graphik einzulesen.

Das Öffnen des Streams ist auch in einem Schritt möglich mit:

```
InputStream ein = IrgendeineKlasse.class.getResourceAsStream ("Dateiname");
```

Diese Möglichkeiten können mit folgendem Beispiel getestet werden (Siehe auch Horstmann und Cornell [18] Seite 571):

```
import java.awt.*;
import java.applet.*;
import java.net.*;
import java.io.*;
import java.util.*;

public class   URLApplet extends Applet
   {
   Image bild;
   String[] text;
   Exception e1;
   Exception e2;
   URL   url1;
   URL   url2;

   public void init ()
      {
      System.err.println ("init: 1*"); System.err.flush ();
      showStatus ("init: 1*");
      try
         {
         // 1.)
         bild = getImage (url1 = URLApplet.class.getResource ("kfg.gif"));
            // Das funktioniert nicht, sollte aber. kfg.gif ist in jar-File
            // Funktioniert mit Explorer auf MacIntosh
            // Funktioniert nicht mit Netscape Communicator 4.79

         // 2.)
         //bild = getImage (getDocumentBase (), "kfg.gif");
            // Das funktionier nicht, kfg.gif ist in jar-File
            // Das muß auch nicht funktionieren.

         // 3.)
         //bild = getImage (getDocumentBase (), "images/kfg.gif");
            // Das funktioniert auch mit Netscape 4.79.
            // kfg.gif ist im Directory images
/*
         // Andere Versuche:
         bild = getImage (
            new URL (
               getCodeBase (), "kfg.gif"));
//             "http://" + getCodeBase ().getHost () + "~kfg/kfg.gif"));
//             "kfg.gif"));
//             "http://www.ba-stuttgart.de/~kfg/images/kfg.gif"));
//             "http://www.ba-stuttgart.de/~kfg/kfg.gif"));
//             "file:/home/blackhole.itstaff/kfg/public-html/images/kfg.gif"));
*/
```

```
      }
   catch (Exception e1)
      {
      System.err.println ("init: " + e1);
      }
   try
      {
      BufferedReader br = new BufferedReader (new InputStreamReader (
         URLApplet.class.getResourceAsStream ("kfg.txt")));
      url2 = URLApplet.class.getResource ("kfg.txt");
      Vector   v = new Vector ();
      String   s;
      while ((s = br.readLine ()) != null) v.addElement (s);
      text = new String[v.size ()];
      for (int i = 0; i < v.size (); i++) text[i] = (String) v.elementAt (i);
      }
   catch (Exception e2)
      {
      System.err.println ("init: " + e2);
      }
   System.err.println ("init: 2*"); System.err.flush ();
   showStatus ("init: 2*");
   }

public void paint (Graphics g)
   {
   super.paint (g);
   g.drawString ("Das Applet läuft erstmal. Jetzt müßte der Text kommen 2:",
      10, 10);
   if (text != null)
      for (int i = 0; i < text.length; i++)
         g.drawString (text[i], 20, 10 + 10 * (i + 1));
   else g.drawString ("text ist null " + e2, 10, 20);
   if (bild != null)
      g.drawImage (bild, 20, 40 + 10 * (text.length + 1),
         bild.getWidth (this),
         bild.getHeight (this),
         this);
   else g.drawString ("bild ist null " + e1 + " URL1: " + url1
      + " URL2: " + url2, 10, 30);
   }

}
```

13.6 Interaktives Applet

Das Beispiel "Schmier" aus dem Einleitungs-Kapitel ist ein Beispiel für ein interaktives Applet.

13.7 Applet mit Aktivität

Das folgende Applet "tut etwas von selbst". Ein gefüllter Kreis bewegt sich von einem zufällig gewählten Punkt zum nächsten zufällig gewählten Punkt. Die Bewegung erfolgt in mehreren Teilschritten.

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

public class   Aktiv extends JApplet
   {
   private Container behälter;

   private int kreisDurchmesser = 10;
   private Color  kreisFarbe = Color.red;
   private Point  kreisZentrum;
   ArrayList   kreise;
   private int teilSchritte = 10;
   private int zeitIntervall = 500;

   Bewegung bewegung;

   class Bewegung extends Thread
      {
      boolean  angehalten = false;
      boolean  beendet = false;

      public void run ()
         {
         weiter: while (true)
            {
            yield ();
            Point p = getZufall ();
            int   dx = (p.x - kreisZentrum.x) / teilSchritte;
            int   dy = (p.y - kreisZentrum.y) / teilSchritte;
            for (int i = 0; i < teilSchritte; i++)
               {
               if (beendet) break weiter;

               try
                  {
                  sleep (zeitIntervall);
```

```
                }
                catch (InterruptedException e)
                   {
                   }

             if (beendet) break weiter;

             while (angehalten)
                {
                try
                   {
                   sleep (10000);
                   }
                   catch (InterruptedException e)
                      {
                      angehalten  = false;
                      }
                }

             if (beendet) break weiter;

             kreisZentrum = new Point (
                kreisZentrum.x + dx,
                kreisZentrum.y + dy);

             kreise.add (
                new Oval (
                   kreisZentrum,
                   kreisDurchmesser,
                   kreisDurchmesser,
                   kreisFarbe));

             repaint (
                kreisZentrum.x - kreisDurchmesser / 2,
                kreisZentrum.y - kreisDurchmesser / 2,
                kreisDurchmesser, kreisDurchmesser);
             }
          }  // weiter
       }
    }

 public void init ()
    {
    behälter = getContentPane ();
    behälter.setBackground (Color.gray);
    kreise = new ArrayList ();
    kreisZentrum = new Point (0, 0);
    kreise.add (
       new Oval (
```

```
            kreisZentrum,
            kreisDurchmesser, kreisDurchmesser, kreisFarbe));

      bewegung = new Bewegung ();
      stop ();
      bewegung.start ();
      }

   public void start ()
      {
      bewegung.angehalten = false;
      bewegung.beendet = false;
      bewegung.interrupt ();        // Um aus eventuellem Schlaf zu wecken.
      }

   public void stop ()
      {
      bewegung.angehalten = true;
      }

   public void destroy ()
      {
      bewegung.beendet = true;
      bewegung.interrupt ();
      try
         {
         bewegung.join ();          // Wartet darauf, daß Bewegung
                                    // sich beendet.
         }
         catch (InterruptedException e)
            {
            e.printStackTrace ();
            }
      }

   public void paint (Graphics g)
      {
      super.paint (g);
      for (int i = 0; i < kreise.size (); i++)
         {
         ((Oval) kreise.get (i)).male (g);
         }
      }

   private Random zufall = new Random ();

   private Point  getZufall ()
      {
      return new Point
```

```
         (
         Math.abs (zufall.nextInt ()) % behälter.getSize ().width,
         Math.abs (zufall.nextInt ()) % behälter.getSize ().height
         );
      }

   }

class Oval
   {
   Point zentrum;
   int   breite;
   int   höhe;
   Color farbe;

   public   Oval (Point zentrum, int breite, int höhe, Color farbe)
      {
      this.zentrum = zentrum;
      this.breite = breite;
      this.höhe = höhe;
      this.farbe = farbe;
      }

   public void male (Graphics g)
      {
      g.setColor (farbe);
      g.fillOval (zentrum.x - breite / 2, zentrum.y - höhe / 2,
         breite, höhe);
      }
   }
```

Die Bewegung wird in einem eigenen Thread `bewegung` der von `Thread` erbenden Klasse `Bewegung` gemacht, indem die Graphik um die gefüllten Kreise erweitert wird und nach jedem Kreis die Methode `repaint` für das den neuen Kreis umschreibende Rechteck aufgerufen wird.

Die Bewegung wird mit dem Flag `angehalten` angehalten und mit dem Flag `beendet` endgültig gestoppt.

13.8 Übungen

Übung Kreise: Schreiben Sie ein Applet, das eine Anzahl Kreise in verschiedenen Farben dort zeichnet, wo mit dem linken Mausknopf geklickt wird. Wo mit dem rechten Mausknopf geklickt wird, soll ein Text erscheinen. Die Größe der Kreise und der Text sollen als Parameter übergeben werden.

Kapitel 14

Datenbankanbindung – JDBC

14.1 Einführung

JDBC^{TM} ***(Java Database Connectivity)*** ist eine Plattform-unabhängige Schnittstelle zwischen Java und Datenbanken, die SQL verstehen. Es ist ein Java-API zur Ausführung von SQL-Anweisungen. Damit ist JDBC ein ***low-level*** API, womit dann ***high-level*** oder benutzerfreundliche Oberflächen geschrieben werden können.

Von JDBC aus gibt es im wesentlichen zwei Entwicklungsrichtungen:

- ***embedded*** SQL für Java: SQL-Anweisungen können mit Java-Code gemischt werden. Java-Variable können in SQL-Anweisungen verwendet werden. Ein Produkt dieser Art ist SQLJ.
- direkte Abbildung von Tabellen als Java-Klassen: Jede Zeile einer Tabelle wird zur Instanz der Klasse der Tabelle ***(object/relational mapping)***. Jede Spalte entspricht einem Attribut der Klasse. Der Programmierer kann dann direkt mit den Java-Objekten und ihren Attributen arbeiten. Die benötigten SQL-Anweisungen werden automatisch generiert.

Das ODBC API *(Open DataBase Connectivity)* von Microsoft gibt es für beinahe jede Datenbank. Eine direkte Übersetzung von ODBC in ein Java-API ist nicht wünschenswert, da ODBC z.B. reichlichen Gebrauch von Pointern macht. Daher gibt es eine JDBC-ODBC-Bridge, die eine Übersetzung von ODBC in JDBC ist. Ferner ist ODBC schwierig zu lernen, indem einfache und komplexe Konzepte vermischt werden. Z.B. ist für einfache Anfragen die Angabe komplizierter Optionen notwendig. JDBC ist viel leichter zu lernen als ODBC. Beide Schnittstellen basieren auf X/Open SQL CLI (Call Level Interface). Die JDBC-ODBC-Bridge wird mit dem Java 2 SDK mitgeliefert.

`ADO.NET` bietet allerdings ähnlich bequeme Mechanismen, um auf Datenquellen zuzugreifen. Statt eine gemeinsame Schnittstelle für alle datenbanken anzubieten, werden für jedes Datenbanksystem spezielle Klassen angeboten.

JDBC unterstützt einstufige und zweistufige Modelle **(2-Tier**, ***two-tier***, **3-Tier**, ***three-tier models***, allgemein **N-Tier-Modelle oder Architekturen)**. Im einstufigen Modell kommuniziert ein Applet oder eine Anwendung direkt mit dem Datenbank-Management-System, das auf

einer anderen Machine laufen kann. Im zweistufigen Modell werden die SQL-Anfragen an einen Anwendungsserver geschickt, der dann über JDBC mit dem DBMS verkehrt. Das hat den Vorteil, dass man mehr Kontrolle über den Zugang zur Datenbank hat und dass man benutzerfreundliche Schnittstellen anbieten kann, die vom Anwendungsserver in JDBC-Aufrufe übersetzt werden.

Wie geht JDBC mit SQL um? Man kann *jedes* SQL-Kommando absetzen. Damit läuft man aber Gefahr, dass der Code von einer Datenbank zur anderen nicht mehr ohne weiteres portierbar ist, da die verschiedenen Datenbanken vom SQL-Standard abweichen. Eine Möglichkeit ist, ODBC-ähnliche Escape-Sequenzen zu benutzen. JDBC bietet noch eine andere Möglichkeit, indem deskriptive Informationen über das DBMS verwendet werden, wozu eine `DatabaseMetaData`-Schnittstelle zu implementieren ist.

Die neueste Information über die Verfügbarkeit von Treibern erhält man über:

```
http://java.sun.com/products/jdbc
```

Die JDBC-Klassen befinden sich im Paket `java.sql` .

14.2 Verbindung `Connection`

Ein Objekt der Klasse `Connection` repräsentiert eine Verbindung mit der Datenbank. Eine Anwendung kann mehrere Verbindungen mit derselben oder verschiedenen Datenbanken haben. Normalerweise wird eine Verbindung mit der statischen Methode `getConnection ()` der Klasse `DriverManager` erzeugt, wobei die Datenbank durch eine URL repräsentiert wird. Der Treibermanager versucht, einen Treiber zu finden, der auf die gewünschte Datenbank passt. Der Treibermanager verwaltet eine Liste registrierter Treiber-Klassen (Klasse `Driver`). Wenn ein Treiber gefunden ist, dann wird mit der Methode `connect` die Verbindung zur Datenbank aufgebaut.

Beispiel:

```
String   url = "jdbc:mysql://rechner:3306/Spielbank";
Connection  verbindung = DriverManager.getConnection
                            (url, "Benutzername", "Passwort");
```

Mit einer JDBC-URL werden Datenbanken identifiziert, wobei der Protokollname `jdbc` verwendet wird. Der Treiberhersteller muss angeben, wie die Datenbank nach dem `jdbc:` identifiziert wird.

Die empfohlene Standard-Syntax einer JDBC-URL besteht aus drei durch Doppelpunkte getrennten Teilen:

`jdbc:<Subprotokoll>:<Subname>`

1. Der Protokollname `jdbc` ist zwingend.
2. Der `<Subprotokoll>`-Name ist der Name des Treibers oder des Datenbank-Verbindungs-Mechanismus. Berühmte Beispiele dafür sind `odbc` oder `mysql`. Es kann aber auch der Name eines Namensdienstes sein. JavaSoft registriert informell die JDBC-Subprotokollnamen. Wenn man einen Treiber geschrieben hat, dann kann man den Namen des Protokolls registrieren lassen, indem man Email an

```
jdbc@wombat.eng.sun.com
```

sendet.

3. Der `<Subname>` identifiziert irgendwie die Datenbank. Er kann z.B. die Form

   ```
   //hostname:port/kfg/db/Spielbank
   ```

   haben.

Das `odbc`-Subprotokoll hat die spezielle Syntax:

`jdbc:odbc:<Datenquellname>[;<Attributname>=<Attributwert>]*`

Damit können beliebig viele Attribute als Parameter übergeben werden. Z.B.:

```
jdbc:odbc:Spielbank;UID=kfg;PWD=Ratatui;CacheSize=20;
```

Weitere Methoden von `DriverManager` sind `getDriver`, `getDrivers` und `registerDriver`.

Ein Treiber kann auf zwei Arten geladen werden:

1. `Class.forName ("TreiberKlassenname");`
   Diese Anweisung lädt die Treiberklasse explizit. Wenn die Klasse so geschrieben ist (Verwendung von statischen Initialisatoren), dass beim Laden der Klasse ein Objekt der Klasse erzeugt wird und dafür die Methode

   ```
   DriverManager.registerDriver (objektDerTreiberKlasse)
   ```

   aufgerufen wird, dann steht dieser Treiber für die Erzeugung einer Verbindung zur Verfügung. Falls das Treiberobjekt nicht automatisch erzeugt wird, ist der Aufruf

   ```
   Class.forName ("TreiberKlassenname").newInstance ();
   ```

   sicherer.

2. Man kann auch den Treiber der `System`-Eigenschaft `jdbc.drivers` hinzufügen. Diese Eigenschaft ist eine durch Doppelpunkt getrennte Liste von Treiber-Klassennamen. Wenn die Klasse `DriverManager` geladen wird, dann wird versucht alle Treiberklassen zu laden, die in der Liste `jdbc.drivers` stehen.

Methode 1) wird empfohlen, da sie unabhängig von einer Umgebungsvariablen ist.

Aus Sicherheitsgründen weiß JDBC, welcher Klassenlader welche Treiberklasse zur Verfügung stellt. Wenn eine Verbindung hergestellt wird, dann wird nur der Treiber benutzt, der vom selben Klassenlader geladen wurde wie der betreffende (die Verbindung anfordernde) Code.

Die Reihenfolge der Treiber ist signifikant. Der `DriverManager` nimmt den *ersten* Treiber, der die Verbindung aufbauen kann. Die Treiber in `jdbc.drivers` werden zuerst probiert.

Beispiel:

```
Class.forName ("org.gjt.mm.mysql.Driver");
String   url = "jdbc:mysql://rechner:3306/Spielbank";
Connection  verbindung
   = DriverManager.getConnection (url, "Bozo", "hochgeheim");
```

14.3 SQL-Anweisungen

Nach dem Verbindungsaufbau wird das `Connection`-Objekt benutzt, um SQL-Anweisungen abzusetzen. JDBC macht bezüglich der Art der SQL-Statements keine Einschränkungen. Das erlaubt große Flexibilität. Dafür ist man aber selbst dafür verantwortlich, dass die zugrunde liegende Datenbank mit den Statements zurechtkommt. JDBC fordert, dass der Treiber mindestens das ANSI SQL2 Entry Level unterstützt, um JDBC COMPLIANT zu sein. Damit kann der Anwendungsprogrammierer sich mindestens auf diese Funktionalität verlassen.

Es gibt drei Klassen um SQL-Anweisungen abzusetzen.

- `Statement`
- `PreparedStatement`
- `CallableStatement`

14.3.1 Klasse `Statement`

Die Klasse `Statement` dient zur Ausführung einfacher SQL-Anweisungen. Ein `Statement`-Objekt wird mit der `Connection`-Methode `createStatement` erzeugt. Mit der Methode `executeQuery` werden dann SQL-Anfragen abgesetzt:

```
Statement   anweisung = verbindung.createStatement ();
ResultSet   ergebnis
   = anweisung.executeQuery ("SELECT a, b, c FROM tabelle");
while (ergebnis.next ())
   {
   int   x = ergebnis.getInt ("a");
   String   y = ergebnis.getString ("b");
   float z = ergebnis.getFloat ("c");
   // Tu was mit x, y, z ...
   }
```

Die Methode `executeQuery` gibt eine einzige Resultatmenge zurück und eignet sich daher für `SELECT`-Anweisungen.

Die Methode `executeUpdate` wird verwendet, um `INSERT`-, `UPDATE`- oder `DELETE`-Anweisungen und Anweisungen von SQL-DDL wie `CREATE TABLE` und `DROP TABLE` auszuführen. Zurückgegeben wird die Anzahl der modifizierten Zeilen (evtl. 0).

Die Methode `execute` bietet fortgeschrittene Möglichkeiten (mehrere Resultatmengen), die selten verwendet werden und daher kurz in einem eigenen Abschnitt behandelt werden.

Nicht benötigte `ResultSet`-, `Statement`- und `Connection`-Objekte werden vom GC automatisch deallokiert. Dennoch ist es gute Programmierpraxis gegebenenfalls ein explizites

```
ergebnis.close (); anweisung.close (); verbindung.close ();
```

aufzurufen, damit Resourcen des DBMS unverzüglich freigegeben werden, womit man potentiellen Speicherproblemen aus dem Weg geht.

Auf die Escape-Syntax, um Code möglichst Datenbank-unabhängig zu machen, wird hier nicht eingegangen.

Ein Objekt der Klasse `ResultSet` enthält alle Zeilen, die das Ergebnis einer `SELECT`-Anfrage sind. Mit

`get<Typ> ("Spaltenname")`-Methoden
oder
`get<Typ> (Spaltennummer)`-Methoden

kann auf die Daten der jeweils *aktuellen* Zeile zugegriffen werden. Mit

```
ergebnis.next ()
```

wird die nächste Zeile zur aktuellen Zeile. `next ()` gibt `true` oder `false` zurück, jenachdem ob die nächste Zeile existiert oder nicht. Der erste Aufruf von `next` macht die erste Zeile zur aktuellen Zeile. (Das ist praktisch für die Gestaltung von Schleifen, da man so die erste Zeile nicht gesondert behandeln muss.)

Benannte Kursoren und positionierte Manipulationen sind auch möglich.

Spaltennummern beginnen bei "1". Verwendung von Spaltennummern ist etwas effektiver und notwendig bei Namenskonflikten. Mit

```
findColumn (Name)
```

kann die Nummer einer Spalte gefunden werden.

Es gibt empfohlene und erlaubte SQL-Java-Datentyp-Wandlungen.

Es ist möglich, beliebig lange Datenelemente vom Typ `LONGVARBINARY` oder `LONGVARCHAR` als Stream zu lesen, da die Methode

```
InputStream ein1 = ergebnis.getBinaryStream (spaltennummer);
InputStream ein2 = ergebnis.getAsciiStream (spaltennummer);
Reader   ein3 = ergebnis.getUnicodeStream (spaltennummer);
```

einen Stream liefert.

Mit der Sequenz

```
ergebnis.get<Typ> (spaltennummer);
ergebnis.wasNull ();
```

kann auf ein NULL-Eintrag in der Datenbank geprüft werden. Dabei gibt `get<Typ>`

- `null` zurück, wenn Objekte zurückgegeben werden,
- `false` bei `getBoolean (spaltennummer)`
- und Null, wenn ein anderer Standardtyp zurückgegeben wird.

Die Methode `wasNull ()` gibt `boolean` zurück.

14.3.2 Methode `execute`

Es kann vorkommen, dass eine gespeicherte Datenbankroutine mehrer SQL-Statements absetzt und damit mehrere Resultatmengen oder Update-Counts erzeugt. Die Einzelheiten des Gebrauchs der verschiedenen beteiligten Methoden sind langwierig. Daher beschränken wir uns darauf, nur für den kompliziertesten Fall, nämlich dass man überhaupt nicht weiß, was die verschiedenen Resultate sind, im folgenden einen Beispiel-Code anzugeben.

```
import   java.sql.*;
public class   JDBCexecute
   {
   public static void   werteAus (Statement anweisung,
      String anfrageStringMitUnbekanntemResultat)
      throws SQLException
      {
      anweisung.execute (anfrageStringMitUnbekanntemResultat);
      while (true)
         {
         int   zeilen = anweisung.getUpdateCount ();
         if (zeilen > 0)
            // Hier liegt ein Update-Count vor.
            {
            System.out.println ("Geänderte Zeilen = " + zeilen);
            anweisung.getMoreResults ();
            continue;
            }
         if (zeilen == 0)
            // DDL-Kommando oder kein Update
            {
            System.out.println (
               "DDL-Kommando oder keine Zeilen wurden geändert.");
            anweisung.getMoreResults ();
            continue;
            }
         // zeilen < 0:
         // Wenn man soweit ist, dann gibt es entweder ein ResultSet
         // oder keine Resultate mehr.
         ResultSet   rs = anweisung.getResultSet ();
         if (rs != null)
            {
            // Benutze ResultSetMetaData um das rs auszuwerten
            // ...
            anweisung.getMoreResults ();
            continue;
            }
         break;   // Es gibt keine Resultate mehr.
         }
```

```
    }
}
```

14.3.3 Klasse `PreparedStatement`

Objekte der Klasse `PreparedStatement` repräsentieren eine kompilierte SQL-Anweisung mit ein oder mehreren `IN`-Parametern. Der Wert eines `IN`-Parameters wird nicht spezifiziert, sondern mit einem Fragezeichen "`?`" als Platzhalter offengelassen. Bevor das Statement ausgeführt wird, muss für jedes Fragezeichen die Methode

```
set<Typ> (parameternummer, parameterWert)
```

aufgerufen werden, z.B. `setInt`, `setString`. `parameternummer` beginnt bei 1. Gesetzte Parameter können für wiederholte Ausführungen der Anweisung verwendet werden.

Die drei Methoden `execute`, `executeQuery` und `executeUpdate` sind so modifiziert, dass sie keine Argumente haben. Die Formen mit Argumenten sollten für Objekte von `PreparedStatement` nie benutzt werden.

Ein Objekt von `PreparedStatement` wird folgendermaßen erzeugt:

```
PreparedStatement prepAnw = verbindung.prepareStatement (
   "UPDATE SP SET QTY = ? WHERE SNR = ?");
```

Weiterführende Hinweise: Mit `setNull` können `NULL`-Werte in der Datenbank gesetzt werden. Mit `setObject` kann im Prinzip jedes Java-Objekt akzeptiert werden und erlaubt generische Anwendungen. Auch gibt es Methoden, mit denen ein `IN`-Parameter auf einen Eingabe-Strom gesetzt werden kann.

14.3.4 Klasse `CallableStatement`

Objekte vom Typ `CallableStatement` erlauben den Aufruf von in einer Datenbank gespeicherten Prozeduren.

Die Syntax für einen Aufruf einer Prozedur ohne Rückgabewert lautet:

```
{call Prozedurname[ (?, ?, ...)]}
```

Die `[]`-Klammern beinhalten optionale Elemente der Syntax. Die Fragezeichen können `IN`- oder `OUT`- oder `INOUT`-Argumente sein. Mit Rückgabewert ist die Syntax:

```
{? = call Prozedurname[ (?, ?, ...)]}
```

Über die Klasse `DatabaseMetaData` kann man die Datenbank bezüglich ihrer Unterstützung von gespeicherten Prozeduren untersuchen (Methoden `supportsStoredProcedures ()` und `getProcedures ()`).

`CallableStatement` erbt von `Statement` alle Methoden, die generell mit SQL-Anweisungen zu tun haben. Von `PreparedStatement` werden Methoden geerbt, die mit `IN`-Argumenten zu tun haben. Neu hinzu kommen Methoden, die mit `OUT`- oder `INOUT`-Argumenten arbeiten.

Objekte werden erzeugt mit:

```
CallableStatement prozAnw = verbindung.prepareCall (
   "{call prozedurName (?, ?)}");
```

Ob die Fragezeichen `IN` oder `OUT` oder `INOUT` sind, hängt von der Definition der Prozedur ab. `OUT`- und `INOUT`-Parameter müssen registriert werden, bevor die Anweisung ausgeführt wird. Nach Ausführung der Anweisung sollten erst alle `ResultSet`s abgearbeitet werden, ehe mit `get<Typ>`-Methoden die `OUT`-Parameter geholt werden. Folgendes Code-Fragment soll das illustrieren, wobei der erste Paramter ein `INOUT`-Parameter, der zweite ein `OUT`-Parameter ist:

```
CallableStatement prozAnw = verbindung.prepareCall (
   "{call prozedurName (?, ?)}");
prozAnw.setInt (1, 25);
prozAnw.registerOutParameter (1, java.sql.Types.INTEGER);
prozAnw.registerOutParameter (2, java.sql.Types.DECIMAL, 3);
ResultSet   rs = prozAnw.executeQuery ();
// Verarbeite rs
int i = prozAnw.getInt (1);
java.math.BigDecimal d = prozAnw.getBigDecimal (2, 3);
```

14.3.5 SQL- und Java-Typen

Für eine professionelle Arbeit mit JDBC sollte man sich die Einzelheiten bezüglich der Entsprechung von SQL-Typen und Java-Typen in der JDBC-Referenz ansehen. Diese darzustellen führt hier zu weit. Es soll nur der Hinweis gegeben werden, dass da einiges zu beachten ist.

14.4 Beispiel `JDBC_Test`

```
import   java.lang.*;
import   java.io.*;
import   java.sql.*;
public class   JDBC_Test
   {
   public static void   main (String[] argument)
      {
      String dbURL = argument [0];
      try
         {
         Class.forName ("postgresql.Driver");
         }
         catch (ClassNotFoundException e)
            {
            System.err.println ("Fehler in JDBC_Test.main: " + e);
            }
```

```
    try
       {
       Connection  v = DriverManager.getConnection (dbURL, "kfg", "");
       try
          {
          Statement   s = v.createStatement ();
          try
             {
             s.executeUpdate ("DROP TABLE SUPPLIER");
             }
             catch (SQLException e)
                {
                System.err.println ("Fehler in JDBC_Test.main: ");
                System.err.println ("   DROP TABLE SUPPLIER: " + e);
                }
          s.executeUpdate ("CREATE TABLE SUPPLIER ("
             + "SNR CHAR (6) NOT NULL,"
             + "SNAME CHAR (12),"
             + "STATUS INTEGER,"
             + "CITY CHAR (12))");
          s.executeUpdate ("INSERT INTO SUPPLIER (SNR, SNAME, STATUS, CITY)"
             + "VALUES ('S1', 'Smith', 20, 'London')");
          s.executeUpdate ("INSERT INTO SUPPLIER (SNR, SNAME, STATUS, CITY)"
             + "VALUES ('S2', 'Jones', 10, 'Paris')");
          s.executeUpdate ("INSERT INTO SUPPLIER (SNR, SNAME, STATUS, CITY)"
             + "VALUES ('S3', 'Blake', 30, 'Paris')");
          s.executeUpdate ("INSERT INTO SUPPLIER (SNR, SNAME, STATUS, CITY)"
             + "VALUES ('S4', 'Clark', 20, 'London')");
          s.executeUpdate ("INSERT INTO SUPPLIER (SNR, SNAME, STATUS, CITY)"
             + "VALUES ('S5', 'Adams', 30, 'Athens')");
          ResultSet   r =
             s.executeQuery (
                "SELECT SNAME FROM SUPPLIER WHERE CITY = 'Paris'");
          System.out.println ("SUPPLIER.SNAME");
          System.out.println ("--------------");
          while (r.next ())
             {
             System.out.println (r.getString ("SNAME"));
             }
          System.out.println ("--------------");
          s.close ();
          }
          catch (SQLException e)
             {
             System.err.println ("Fehler in JDBC_Test.main: " + e);
             }
       v.close ();
       }
       catch (SQLException e)
          {
          System.err.println ("Fehler in JDBC_Test.main: ");
          System.err.println ("   DriverManager.getConnection: " + e);
          }
    }
}
```

14.5 Transaktionen

Eine Transaktion besteht aus einem oder mehreren SQL-Anweisungen, die nacheinander vollständig ausgeführt werden und entweder mit einem Commit oder Rollback abgeschlossen werden.

Eine neu erzeugte Verbindung ist im Auto-Commit Mode, was bedeutet, dass nach jedem

`executeUpdate` automatisch die Methode `commit` aufgerufen wird. Jede Transaktion besteht aus einer SQL-Anweisung.

Wenn der Auto-Commit Mode abgeschaltet ist (`setAutoCommit (false)`), dann läuft eine Transaktion solange, bis explizit `commit ()` oder `rollback ()` für die Verbindung aufgerufen wird.

Wenn JDBC eine Exception wirft, dann wird das Rollback nicht automatisch gegeben. D.h. die ProgramiererIn muss in solchen Fällen ein Rollback ins `catch` aufnehmen.

Mit z.B.

```
verbindung.setTransactionIsolation (TRANSACTION_READ_UNCOMMITTED)
```

können verschiedene Isolations-Niveaus eingestellt werden.

Bemerkung: Anstatt dieser Transaktionsmechanismen empfiehlt es sich direkt die Mechanismen des verwendeten Datenbanksystems zu verwenden. Hier gibt es sehr starke Unterschiede.

14.6 JDBC 2.0

Diese neue JDBC-Version unterstützt erweiterte Möglichkeit `ResultSet`-Objekte zu behandeln. Dazu kommen BLOB-Felder und andere kleinere Verbesserungen. JDBC 2.0 unterstützt scrollbare und aktualisierbare Ergebnis-Mengen. Wegen weiterführender Literatur verweisen wir auf das Buch von Flanagan et al. [11].

14.7 Beispiel `SQLBefehle`

Die Applikation in der Klasse `SQLBefehle` führt alle in einer Datei angegebenen, mit Semikolon abgeschlossenen SQL-Befehle aus und schreibt eventuelle Resultate nach Standard-Ausgabe.

Aufruf: `$ java SQLBefehle <` *Dateiname*
Abbruch nach dem ersten fehlgeschlagenen SQL-Statement.

Beim Aufruf sind noch folgende Optionen angebbar:

```
-u userid
-p password
-b dburl
-d driver
-f filename
```

Beispiel:

```
$ java SQLBefehle -u mck -p mckpw -b jdbs:mysql://localhost/mck -f sqldatei
$ java SQLBefehle -u mck -p mckpw -b jdbs:mysql://localhost/mck < sqldatei
```

Bei diesem Beispiel ist die Verwendung der Methode `getMetaData ()` zu beachten, die ein Objekt der Klasse `ResultSetMetaData` liefert, das man über die Eigenschaften des `ResultSet`-Objekts ausfragen kann, um etwa das `ResultSet` geeignet darzustellen.

```
import   java.lang.*;
import   java.io.*;
import   java.sql.*;
import   java.util.*;
public class   SQLBefehle
   {
   public static void   main (String[] argument)
      // Option: -u userid z.B. "-u kfg"
      // Option: -p password z.B. "-p hochgeheim"
      // Option: -b DatenbankURL z.B. "-b jdbc:postgresql://spielberg/it95"
      // Option: -d Datenbanktreiber z.B. "-d postgresql.Driver"
      // Option: -f SQL-File z.B. "-f selAlles"
      {
      String   userID = "mck";
      String   password = "mckpw";
      String   dbURL = "jdbc:mysql://localhost/mck";
      String   dbDriver = "org.gjt.mm.mysql.Driver";
      String   sqlFile = null;
      for (int i = 0; i < argument.length; i+=2)
         {
         if (argument[i].equals ("-u")) userID = argument[i + 1];
         else if (argument[i].equals ("-p")) password = argument[i + 1];
         else if (argument[i].equals ("-b")) dbURL = argument[i + 1];
         else if (argument[i].equals ("-d")) dbDriver = argument[i + 1];
         else if (argument[i].equals ("-f")) sqlFile = argument[i + 1];
         }
      try
         {
         BufferedReader f = null;
         if (sqlFile != null)
               {
               f = new BufferedReader (new FileReader (sqlFile));
               }
         else
            {
            f = new BufferedReader (new InputStreamReader (System.in));
            }
         Class.forName (dbDriver);
         String   sw = "";
         try
            {
            String   s = null;
            int   j;
            while ( (s = f.readLine ()) != null)
               {
               if ((j = s.indexOf ("--")) >= 0)
                  {
                  if (j == 0) s = "";
                  else s = s.substring (0, j);
                  }
               sw = sw + s;
               }
            }
            catch (IOException e)
               {
               System.err.println ("Fehler bei f.readLine (): " + e);
               }
         StringTokenizer   st
            = new StringTokenizer (sw, ";", false);
         Connection  v = DriverManager.getConnection (dbURL, userID,
            password);
         Statement   s = v.createStatement ();
         try
            {
            while (st.hasMoreElements ())
               {
               String   sql = ( (String)st.nextElement ()).trim ();
               if (sql.equals ("")) continue;
```

```
System.out.println ();
System.out.println (sql + ';');
if (sql.toLowerCase ().indexOf ("select") < 0
   && sql.toLowerCase ().indexOf ("show") < 0
   && sql.toLowerCase ().indexOf ("describe") < 0)
   {
   s.executeUpdate (sql);
   }
else
   {
   ResultSet r = s.executeQuery (sql);
   ResultSetMetaData rm = r.getMetaData ();
   int spalten = rm.getColumnCount ();
   int[] w = new int[spalten + 1];
   for (int i = 1; i <= spalten; i++)
      {
      /*
      w[i] = rm.getColumnDisplaySize (i);
      if (w[i] < rm.getColumnLabel (i).length ())
         w[i] = rm.getColumnLabel (i).length ();
      */
      w[i] = rm.getColumnLabel (i).length ();
      }
   String   rs;
   while (r.next ())
      {
      for (int i = 1; i <= spalten; i++)
         {
         rs = r.getString (i);
         if (rs == null) rs = "NULL";
         if (w[i] < rs.length ()) w[i] = rs.length ();
         }
      }
   for (int i = 1; i <= spalten; i++)
      {
      System.out.print ("--");
      for (int j = 0; j < w[i]; j++)
         System.out.print ('-');
      }
   System.out.println ("-");
   for (int i = 1; i <= spalten; i++)
      {
      System.out.print ("| ");
      System.out.print (rm.getColumnLabel (i));
      for (int j = 0;
         j < w[i] - rm.getColumnLabel (i).length (); j++)
         System.out.print (' ');
      }
   System.out.println ("|");
   for (int i = 1; i <= spalten; i++)
      {
      System.out.print ("|-");
      for (int j = 0; j < w[i]; j++)
         System.out.print ('-');
      }
   System.out.println ("|");
   r = s.executeQuery (sql);
   while (r.next ())
      {
      for (int i = 1; i <= spalten; i++)
         {
         System.out.print ("| ");
         rs = r.getString (i);
         if (rs == null) rs = "NULL";
         for (int j = 0;
            j < w[i] - rs.length (); j++)
            System.out.print (' ');
         System.out.print (rs);
```

```
                    }
                 System.out.println ("|");
                 }
              for (int i = 1; i <= spalten; i++)
                 {
                 System.out.print ("--");
                 for (int j = 0; j < w[i]; j++)
                    System.out.print ('-');
                 }
              System.out.println ("-");
              }
           System.out.println ();
           }
        }
        catch (SQLException e)
           {
           System.err.println ("Fehler in JDBC_Test.main: " + e);
           e.printStackTrace ();
           }
     s.close ();
     v.close ();
     }
     catch (ClassNotFoundException e)
        {
        System.err.println ("Fehler in JDBC_Test.main: " + e);
        }
     catch (FileNotFoundException e)
        {
        System.err.println ("Fehler in JDBC_Test.main: " + e);
        }
     catch (SQLException e)
        {
        System.err.println ("Fehler in JDBC_Test.main: ");
        System.err.println ("   DriverManager.getConnection: " + e);
        }
  }
}
```

Kapitel 15

Serialisierung

Objekt-Serialisierung unterstützt die Kodierung von Objekten in Byte-Ströme und umgekehrt, um Objekte in Streams zu schreiben oder von Streams zu lesen. Dabei wird der komplette Objektgraph berücksichtigt. Ferner ist es möglich eigene Kodier-Methoden zu schreiben. Anstatt ***serialize*** wird im Englischen oft auch der Begriff ***marshal*** verwendet, der in einigen Fehlermeldungen vorkommt.

Jede Klasse, deren Objekte serialisierbar sind, muß entweder die Schnittstelle `Serializable` oder die Schnittstelle `Externalizable` implementieren.

15.1 Default-Serialisierung

Die Schnittstelle `Serializable` ist eine reine Marker-Schnittstelle, die anzeigt, daß Objekte der diese Schnittstelle implementierenden Klasse serialisiert bzw deserialisiert werden dürfen. Sie zwingt nicht zur Implementation von irgendwelchen Methoden. Die Serialisierung wird von der Methode `writeObject` der Klasse `ObjectOutputStream`, die Deserialisierung von der Methode `readObject` der Klasse `ObjectInputStream` durchgeführt.

Klasse `java.io.ObjectOutputStream` :

```
public final void writeObject (Object ob) throws IOException;
```

Klasse `java.io.ObjectInputStream` :

```
public final Object  readObject ()
   throws OptionalDataException, ClassNotFoundException,
      IOException;
```

Wenn ein Objekt andere Objekte als Datenelemente enthält, dann wird für diese Objekte rekursiv `writeObject` bzw `readObject` aufgerufen, was dazu führt, daß ein ganzer Objekt-Graph serialisiert bzw deserialisiert wird. Das bedeutet aber, daß diese Objekte auch `Serializable` implementieren müssen. Gegenseitige Verweise der Objekte aufeinander führen nicht zu unendlicher Rekursion. Jedes Objekt wird nur einmal serialisiert. Superklassen werden auch serialisiert.

Felder sind auch serialisierbar.

`transient`-Klassenelemente werden nicht serialisiert.

Das folgende Beispiel zeigt die Syntax und veranschaulicht das Verhalten der Serialisierung. Eine `ProgrammiererIn` hat als Datenelemente eine `Sprache`, einen nicht zu serialisierenden String `laune` und eine andere `ProgrammiererIn` als KollegIn. Ferner wird der Name von einer Superklasse `Person` geerbt.

```
import java.io.*;

public class   ProgrammiererIn extends Person
   implements Serializable
   {
   public   ProgrammiererIn (String name, String sprache)
      {
      super (name);
      this.sprache = new Sprache (sprache);
      laune = new String ("unbestimmte");
      }

   public void zeige ()
      {
      System.out.println ("Ich heiße " + name + ", programmiere in "
         + sprache.name + " und habe " + laune + " Laune.");
      if (kollegIn != null)
         System.out.println ("   Ich arbeite mit "
            + kollegIn.name + " zusammen.");
      }

   public void setLaune (String laune) { this.laune = laune; }

   public void setKollegIn (ProgrammiererIn kollegIn)
      {
      this.kollegIn = kollegIn;
      }

   private Sprache   sprache;
   private transient String   laune;
   private ProgrammiererIn kollegIn;
   }

class Person
   implements Serializable
   {
   public   Person (String name)
      {
      this.name = name;
      }

   protected String  name;
```

```
   }

class Sprache
   implements Serializable
   {
   public   Sprache (String name)
      {
      this.name = name;
      }

   String   name;
   }
```

Das Anwendungsprogramm in der Klasse `Serialisierung` legt zunächst zwei ProgrammiererInnen an, die sich gegenseitig referenzieren. Sie werden als Objete serialisiert auf eine Datei geschrieben. Dann wird ein Feld von ProgrammiererInnen angelegt, die alle denselben Programmierer als Kollegen haben. Das ganze Feld wird serialisiert und auf die Datei geschrieben. Dabei ist zu bemerken (vergleiche Dateigröße), daß der gemeinsame Kollege nur *einmal* serialisiert wird.

```
import java.io.*;

public class   Serialisierung
   {
   public static void   main (String[] argument) throws IOException
      {
      FileOutputStream  f = new FileOutputStream (argument[0]);
      ObjectOutputStream   s = new ObjectOutputStream (f);
      ProgrammiererIn   p1 = new ProgrammiererIn ("Gustav", "Java");
      ProgrammiererIn   p2 = new ProgrammiererIn ("Hanna", "C++");
      p1.setLaune ("schlechte");
      p2.setLaune ("gute");
      p1.setKollegIn (p2);
      p2.setKollegIn (p1);
      p1.zeige ();
      p2.zeige ();
      s.writeObject (p1);
      s.writeObject (p2);

      ProgrammiererIn[] fp = new ProgrammiererIn[20];
      for (int i = 0; i < 20; i++)
         {
         fp[i] = new ProgrammiererIn ("Dummy" + i, "Cobol");
         fp[i].setKollegIn (p1);
         }
      s.writeObject (fp);

      s.close ();
```

```
      f.close ();
      }
   }
```

Schließlich wird in Klasse `Deserialisierung` die Datei wieder gelesen. Die transiente "Laune" erscheint als "null".

```
import java.io.*;

public class   Deserialisierung
   {
   public static void   main (String[] argument)
      throws IOException, ClassNotFoundException
      {
      FileInputStream   f = new FileInputStream (argument[0]);
      ObjectInputStream s = new ObjectInputStream (f);
      ProgrammiererIn   p;
      p = (ProgrammiererIn)s.readObject ();
      p.zeige ();
      p = (ProgrammiererIn)s.readObject ();
      p.zeige ();

      ProgrammiererIn[] fp = (ProgrammiererIn[])s.readObject ();
      for (int i = 0; i < 20; i++) fp[i].zeige ();

      s.close ();
      f.close ();
      }
   }
```

15.2 Serialisierung mit Vor- bzw Nachbehandlung

In unserem Beispiel wurde die Laune nicht serialisiert und erscheint dann nach der Deserialisierung als `null`. Das widerspricht dem Verhalten des Konstruktors, der in solchen Fällen die Laune "unbestimmte" vorsieht. Daher sollte man die Möglichkeit haben, insbesondere transiente Datenelemente nach dem Einlesen zu belegen oder sonstige Initialisierungen – etwa das Starten von Threads – durchzuführen. Auch beim Serialisieren mag es nötig sein, vorher oder nachher gewisse Arbeiten durchzuführen, z.B. das Suspendieren und dann wieder Aufnehmen eines Threads, um während des Serialisierens einen konsistenten Zustand zu haben, oder die Entfernung von nicht benötigtem Speicher.

Dazu ist es nötig, folgende `private`! deklarierte Methoden zu implementieren:

```
private void   writeObject (ObjectOutputStream aus)
   throws IOException
   {
   // Vorbehandlung des Objekts
   aus.defaultWriteObject ();
   // Nachbehandlung des Objekts
   }



private void   readObject (ObjectInputStream ein)
   throws IOException, ClassNotFoundException
   {
   ein.defaultReadObject ();
   // Nachbehandlung des Objekts
   }
```

Unser Beispiel nimmt dann folgende Form an, wobei wir als Vorbehandlung beim Schreiben den Namen in * einbetten und beim Lesen als Nachbehandlung die Laune auf "unbestimmte" setzen:

```
import java.io.*;

public class   ProgrammiererIn extends Person
   implements Serializable
   {
   public   ProgrammiererIn (String name, String sprache)
      {
      super (name);
      this.sprache = new Sprache (sprache);
      laune = new String ("unbestimmte");
      }

   public void zeige ()
      {
      System.out.println ("Ich heiße " + name + ", programmiere in "
         + sprache.name + " und habe " + laune + " Laune.");
      if (kollegIn != null)
         System.out.println ("   Ich arbeite mit "
            + kollegIn.name + " zusammen.");
      }

   public void setLaune (String laune) { this.laune = laune; }

   public void setKollegIn (ProgrammiererIn kollegIn)
      {
      this.kollegIn = kollegIn;
      }
```

```
   private Sprache   sprache;
   private transient String   laune;
   private ProgrammiererIn kollegIn;

   private void   writeObject (ObjectOutputStream aus)
      throws IOException
      {
      aus.defaultWriteObject ();
      }

   private void   readObject (ObjectInputStream ein)
      throws IOException, ClassNotFoundException
      {
      ein.defaultReadObject ();
      setLaune ("unbestimmte");
      }
   }

class Person
   implements Serializable
   {
   public   Person (String name)
      {
      this.name = name;
      }

   protected String  name;

   private void   writeObject (ObjectOutputStream aus)
      throws IOException
      {
      String   old = name;
      name = "*" + name + "*";
      aus.defaultWriteObject ();
      name = old;
      }

   }

class Sprache
   implements Serializable
   {
   public   Sprache (String name)
      {
      this.name = name;
      }

   String   name;
   }
```

An den Anwendungsprogrammen ändert sich überhaupt nichts.

Da `name` in der Superklasse `Person` definiert ist, müssen wir die Methode `writeObject` auch dort definieren und dort die Namensmanipulation durchführen. Die entsprechende Manipulation in der abgeleiteten Klasse `ProgrammiererIn` wäre für die Serialisierung wirkungslos, da das Subobjekt der Superklasse vor der abgeleiteten Klasse serialisiert wird.

15.3 Eigene Serialisierung

Durch Implementation der Schnittstelle `Externalizable` hat man die ganze Kontrolle über das Datenformat der Serialisierung. Der Mechanismus setzt allerdings voraus, daß bei externalisierbaren Klassen ein Default-Konstruktor implementiert ist. Diese Schnittstelle zwingt zur Implementation der Methoden:

```
public void writeExternal (ObjectOutput aus)
   throws IOException;

public void readExternal (ObjectInput ein)
   throws IOException, ClassNotFoundException;
```

Für die Serialisierung von geerbten Klassen muß man selbst sorgen. Entweder sollten diese auch die Schnittstelle `Externalizable` implementieren, oder der Programmierer sorgt dafür, daß die Datenelemnte der Basisklassen geschrieben bzw gelesen werden.

Der folgende Code zeigt unser Beispiel mit Implementation der Schnittstelle `Externalizable`, wobei die Anwendungsprogramme wieder unverändert bleiben.

```
import java.io.*;

public class   ProgrammiererIn extends Person
   implements Externalizable
   {
   public   ProgrammiererIn ()
      {
      }

   public   ProgrammiererIn (String name, String sprache)
      {
      super (name);
      this.sprache = new Sprache (sprache);
      laune = new String ("unbestimmte");
      }

   public void zeige ()
```

```
      {
      System.out.println ("Ich heiße " + name + ", programmiere in "
         + sprache.name + " und habe " + laune + " Laune.");
      if (kollegIn != null)
         System.out.println ("   Ich arbeite mit "
            + kollegIn.name + " zusammen.");
      }

   public void setLaune (String laune) { this.laune = laune; }

   public void setKollegIn (ProgrammiererIn kollegIn)
      {
      this.kollegIn = kollegIn;
      }

   private Sprache   sprache;
   private transient String   laune;
   private ProgrammiererIn kollegIn;

   public void writeExternal (ObjectOutput aus)
      throws IOException
      {
      String   old = name;
      name = "*" + name + "*";
      // Es geht nicht:
      // aus.writeObject ( (Person)this);
      // aber
      // aus.writeObject (name);
      // oder
      super.writeExternal (aus);

      aus.writeObject (sprache);
      aus.writeObject (kollegIn);
      name = old;
      }

   public void readExternal (ObjectInput ein)
      throws IOException, ClassNotFoundException
      {
      // Es geht nicht:
      // Person   p = (Person)ein.readObject ();
      // name = p.name;
      // aber
      // name = (String)ein.readObject ();
      // oder
      super.readExternal (ein);

      sprache = (Sprache)ein.readObject ();
      kollegIn = (ProgrammiererIn)ein.readObject ();
```

```
      setLaune ("unbestimmte");
      }
   }

class Person
   implements Serializable, Externalizable
   {
   public   Person ()
      {
      }

   public   Person (String name)
      {
      this.name = name;
      }

   protected String  name;

   public void writeExternal (ObjectOutput aus)
      throws IOException
      {
      aus.writeObject (name);
      }

   public void readExternal (ObjectInput ein)
      throws IOException, ClassNotFoundException
      {
      name = (String)ein.readObject ();
      }
   }

class Sprache
   implements Serializable
   {
   public   Sprache (String name)
      {
      this.name = name;
      }

   String   name;
   }
```

15.4 Klassenversionen

Bei der Serialisierung eines Objekts werden auch Informationen über die Klasse mitgegeben, damit beim Deserialisieren die richtige Klasse geladen werden kann. Objekte der Klasse `java.io.ObjectStreamClass` verwalten den Klassennamen und eine Versionsnummer. Eine Klasse kann ihre eigene Versionsnummer in einer `long`-Konstanten deklarieren:

```
static final long serialVersionUID = -7440944271964977635L;
```

Wenn die Klasse keine eigene Seriennummer definiert, berechnet `ObjectOutputStream` automatisch eine eindeutige Versionsnummer, indem der ***Secure-Hash-Algorithm*** auf den Namen und die Elemente der Klasse angewendet wird. Nur mit derselben Seriennummer der Klasse kann eine Serialisierung wieder deserialisiert werden.

Wenn der Automatismus stört, wenn etwa nur kleine Änderungen an der Klasse durchgeführt werden, muß die Klasse selbst die Seriennummer definieren. Der Automatismus benötigt ziemlich viel Zeit, da die Bestimmung einer eindeutigen Versionsnummer relativ lang dauert.

Das Programm `serialver` des JDK kann verwendet werden, um eine Seriennummer der Klasse zu berechnen. Der Aufruf

```
$ serialver ProgrammiererIn
```

liefert

```
ProgrammiererIn:    static final long serialVersionUID = -7440944271964977635L;
```

15.5 Applets

Das `<APPLET>`-Tag hat ein Attribut `OBJECT`, das anstelle des Attributs `CODE` verwendet werden kann. Das Attribut `OBJECT` verlangt eine serialisierte Objekt-Datei anstelle einer Klassendatei. Damit kann man Applets in einem vorinitialisierten Zustand ausliefern. Insbesondere wird nicht mehr der Code zur Initialisierung benötigt.

Das ist insbesondere für interaktive GUI-Builder interessant, wobei GUI-Komponenten interaktiv in einem Applet angelegt werden. Das Applet wird dann serialisiert und in einer Datei zur Verfügung gestellt. Allerdings sollte das Applet nur serialisierbare Komponenten enthalten.

Es ist Konvention, solch einer Datei die Erweiterung `.ser` zu geben.

Mit dem `appletviewer` kann nach `Stop` des Applets mit `Save` eine serialisierte Version des Applets angelegt werden. Als Beispiel nehmen wir das Applet `Schmier` aus dem Einleitungs-Kapitel. Dieses Applet muß nur noch die Schnittstelle `Serializable` implementieren, dann kann es mit dem `appletviewer` als `Schmier.ser` gespeichert werden, z.B. nachdem die Farbe "Grün" gewählt wurde. Wenn man jetzt das Applet-Tag im HTML-File auf

```
<APPLET object="Schmier.ser"width=500 height=300>
```

abändert und wieder mit dem `appletviewer` startet, wird es mit der Farbe "Grün" beginnen.

Kapitel 16

Remote Method Invocation (RMI)

Mit *Remote Method Invocation* (RMI, "Aufruf entfernter Methoden") ist es möglich, Java Anwendungen auf mehrere Rechner zu verteilen. Damit wird im Client/Server-Bereich eine aufwendige Socket-Programmierung überflüssig.

Das Prinzip ist schnell erklärt: Auf dem Server läuft ein Broker, bei dem sich beliebige Objekte unter Angabe ihrer öffentlich zugänglichen Methoden anmelden können. Zeigt nun ein anderes Java Programm Interesse an diesem Objekt, übermittelt der Broker die notwendigen Zugriffsinformationen. Mit diesen Informationen wird eine direkte Kommunikation der Objekte möglich.

Das RMI-Modell kann am besten mit einem Drei-Schichtenmodell beschrieben werden. Die Transportschicht etabliert und überwacht die Verbindungen zwischen der eigenen und fremden Maschine. Dabei werden zur Zeit Sockets oder HTTP-Packaging zur Kommunikation verwendet. Die Objekte werden mittels Objekt-Serialisierung in Datenströme verwandelt.

Die Remote-Referenz-Schicht regelt die Weiterleitung von Anfragen an Remote-Objekte.

Auf der Applikationsebene ist nur die Stub/Skeleton-Schicht sichtbar. Sie wird für den Zugriff benutzt. Ein Stub ist eine Art Stellvertreter beim Client für das entfernte Objekt. Ein Skeleton ist das Äquivalent beim Server. Stub und Skeleton werden durch den Compiler `rmic` erstellt.

Der Broker ist `rmiregistry` und nimmt standardmäßig Anfragen und Anmeldungen auf dem Port 1099 entgegen.

Garbage Collection ist normalerweise nur für eine lokale virtuelle Maschine notwendig. RMI Laufzeitsysteme verwalten aber eigene Referenzzähler, die durch den Versand von `referenced` und `unreferenced` Meldungen leben und gegebenenfalls ein Objekt löschen.

Die folgenden Code-Beispiele zeigen einen RMI-Server und einen RMI-Client als Applet. Ferner wird gezeigt, wie diese Programme zu übersetzen sind.

```
package  RMI.Beispiele;

public interface  RMIDienst extends java.rmi.Remote
   // Enthält alle entfernt aufrufbaren Methoden.
   // Jede Methode muß RemoteException werfen.
```

```
   {
   String   sagGutenTag (String vonMir) throws java.rmi.RemoteException;

   static final String  defaultPortNummer = "6567";
   }
```

```
package  RMI.Beispiele;

import   java.rmi.*;
import   java.rmi.server.UnicastRemoteObject;

public class   RMIDienstImpl extends UnicastRemoteObject
   implements RMIDienst
   {
   public static void   main ( String[] argument)
      {
      String   portNummer;
      if (argument.length == 0) portNummer = RMIDienst.defaultPortNummer;
      else portNummer = argument[0];

      //System.setSecurityManager (new RMISecurityManager ());
         // Erzeugung und Installations eines Sicherheitsmanagers

      try
         {
         RMIDienstImpl  dienst = new RMIDienstImpl ("Ronaldo");

         Naming.rebind ("//"
            + java.net.InetAddress.getLocalHost ().getHostName ()
            + ":" + portNummer +"/" + dienst.name, dienst);
            // RMIDienst-Objekt d wird registriert.
         System.err.println (dienst.name + " ist registriert.\n");
         }
         catch (Exception e)
            {
            System.err.println ("Fehler in RMIDienstImpl: "
               + e.getMessage ());
            e.printStackTrace ();
            }
      }

   public   RMIDienstImpl (String s) throws RemoteException
      {
      super ();
      name = s;
      }
```

```
   public String  sagGutenTag (String vonMir)
      {
      return "Dienst " + name + " sagt " + vonMir + " Guten Tag.";
      }

   public String  name; // Identifizierstring für das RMIDienst-Objekt
   }
```

```
<HTML>
<HEAD>
<TITLE>RMI Guten Tag</TITLE>
</HEAD>
<BODY>
<CENTER> <H1>RMI Guten Tag</H1> </CENTER>
Die Botschaft des RMIDienstes ist:
<P>

<APPLET  codebase="../.."
         code ="RMI.Beispiele.RMIKundenApplet.class"
         width=500 height=120>
</APPLET>

</BODY>
</HTML>
```

```
package  RMI.Beispiele;

import   java.rmi.*;

public class   RMIKundenApplet extends java.applet.Applet
   {

   public void init ()
      {
      String   dienstName = "Ronaldo";
      String   kundenName = "Rivaldo";
      String   portNummer = RMIDienst.defaultPortNummer;
      try
         {
         RMIDienst   dienst = (RMIDienst)Naming.lookup ("//"
```

```
            + getCodeBase ().getHost ()
            + ":" + portNummer + "/" + dienstName);
            // Hole Referenz auf entferntes Dienst-Objekt.

         botschaft = dienst.sagGutenTag (kundenName);
         }
         catch (Exception e)
            {
            System.err.println ("Fehler in RMIKundenApplet: "
               + e.getMessage ());
            e.printStackTrace ();
            }
      }

   public void paint (java.awt.Graphics g)
      {
      g.drawString (botschaft, 25, 50);
      }

   String   botschaft = "";
   }
```

Übersetzen der Server-Seite:

```
$ cd $HOME/lang/java/RMI/Beispiele
$ javac -d $HOME/public-html/codeBasis RMIDienst.java
               RMIDienstImpl.java RMIKundenApplet.java
```

Erzeugung der Stubs und Skeletons:

```
$ cd $HOME/public-html/codeBasis
$ rmic -d $HOME/public-html/codeBasis RMI.Beispiele.RMIDienstImpl
```

Starten des Brokers für den Port 6567 im Hintergrund:

```
$ rmiregistry 6567 &
```

Starten des Servers im Hintergrund:

```
$ cd $HOME/public-html/codeBasis
$ java -Djava.rmi.server.codebase=http://amadeus/~kfg/codeBasis/
                                    RMI.Beispiele.RMIDienstImpl &
```

Starten des Client-Applets:

```
$ cd $HOME/public-html/codeBasis/RMI/Beispiele
$ appletviewer RMIKundenApplet.html &
```

16.1 Übungen

Übung Fragenbeantworter:

Das Client/Server-Beispiel aus dem Netzwerk-Kapitel soll mit Hilfe von RMI umgeschrieben werden.

Kapitel 17

Native Methoden

Native Methoden sind Methoden, die in einer anderen Sprache als Java implementiert sind. Native Methoden werden in der Java-Klasse nur deklariert, d.h. anstatt der Implementation { ... } erscheint ein Semikolon ";".

Damit bietet sich auch die Möglichkeit, andere Systemaufrufe oder andere Programme von Java aus zu starten: Man schreibt eine Funktion in der nativen Sprache, etwa C, die einen Systemaufruf absetzt. Funktionsargument ist ein String, der den Systemaufruf enthält.

17.1 Native C oder C++ Methoden

17.1.1 Hello-World-Beispiel

Zur Implementation eines nativen Hello-World-Programms sind folgende Schritte durchzuführen:

1. Zunächst wird ein Anwendungsprogramm in der Klasse `HauptGutenTag` geschrieben, das ein Objekt der Klasse `NativGutenTag` benutzt, die ihrerseits eine native Methode definiert. Der Code der beiden Klassen lautet:

```
public class   HauptGutenTag
   {
   public static void   main (String[] argument)
      {
      new NativGutenTag ().gruessDieWelt ();
      }
   }
```

```
public class   NativGutenTag
```

```
{
public native void   gruessDieWelt ();

static
   {
   System.loadLibrary ("gutentag");
   }

}
```

Diese beiden Klassen werden wie üblich übersetzt:

```
$ javac HauptGutenTag.java
```

(und eventuell, normalerweise nicht)

```
$ javac NativGutenTag.java
```

2. Nun wird ein C-Header-File (auch C++-Header-File) erzeugt, indem das Programm `javah` mit dem Klassennamen (ohne Extension) der Klasse, die native Methoden enthält, aufgerufen wird:

   ```
   $ javah -jni NativGutenTag
   ```

   Der Headerfile heißt dann `NativGutenTag.h` und befindet sich im aktuellen Verzeichnis. Mit der Option `-d` kann man andere Verzeichnisse angeben.

   ```
   /* DO NOT EDIT THIS FILE - it is machine generated */
   #include <jni.h>
   /* Header for class NativGutenTag */

   #ifndef _Included_NativGutenTag
   #define _Included_NativGutenTag
   #ifdef __cplusplus
   extern "C" {
   #endif
   /*
    * Class:     NativGutenTag
    * Method:    gruessDieWelt
    * Signature: ()V
    */
   JNIEXPORT void JNICALL Java_NativGutenTag_gruessDieWelt
     (JNIEnv *, jobject);
   ```

```
#ifdef __cplusplus
}
#endif
#endif
```

3. Jetzt wird die native Methode in C oder C++ geschrieben. Sie muss genau die Signatur haben, die im Header-File angegeben ist. Der Filename ist gleichgültig, muss aber eine Erweiterung haben, mit der der C- oder C++-Compiler zurechtkommt, z.B. "`.c`" oder "`.cpp`".

   File `gruessDieWelt.c` :

```
#include "NativGutenTag.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_NativGutenTag_gruessDieWelt
   (JNIEnv * dies, jobject job)
   {
   printf ("Es grüßt die native C-Funktion mit Guten Tag!\n");
   return;
   }
```

4. Die "`.c`"-Files werden übersetzt und in eine Bibliothek mit dem unter Java verwendbaren Namen `gutentag` gebunden.

   - Unter HP-UX geht das folgendermaßen:

```
$ cc -Aa +e +z -c -I/opt/java/include
         -I/opt/java/include/hp-ux gruessDieWelt.c

$ ld -b -o libgutentag.sl gruessDieWelt.o
```

   - Unter Windows wird der C-Linker eingesetzt:

```
> cl -IC:\jdk1.2\include
   -IC:\jdk1.2\include\win32
   -IP:\devstudio\vc\include
   -IP:\devstudio\vc\lib -LD
   gruessDieWelt.c -Flgutentag.dll
```

     Die vorstehenden Zeilen bilden eine Kommando-Zeile. Die Pfade sind natürlich dem jeweiligen System anzupassen. In diesem Beispiel wurden Bibliotheken eingebunden, um auf die parallele Schnittstelle mit der Funktion `_outp (...)` zu schreiben.

     Oder:

```
> cl -Ix:\java\include
   -Ix:\java\include\win32
```

```
-MD -LD gruessDieWelt.c -Fegutentag.dll
```

In beiden Fällen entsteht dabei die Bibliothek `gutentag.dll`.

- Unter Linux geht das etwa folgendermaßen:

```
$ gcc -shared -I/usr/lib/java/include
   -I/usr/lib/java/include/linux
   gruessDieWelt.c -o libgutentag.so

$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

Es entsteht die Bibliothek `libgutentag.so`.

- Unter FreeBSD geht das etwa folgendermaßen:

```
$ gcc -shared -fpic -I/usr/local/jdk1.1.8/include
   -I/usr/local/jdk1.1.8/include/freebsd -o libgutentag.so gruessDieWelt.c

$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

Es entsteht die Bibliothek `libgutentag.so`.

Der Filename der Bibliothek `gutentag` hängt vom Betriebssystem ab. Unter HP-UX ist er `libgutentag.sl`, unter Sun-Solaris `libgutentag.so`, unter Windows `gutentag.dll`.

5. Eventuell ist eine Angabe nötig, wo sich die Bibliothek `gutentag` befindet.

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
$ export LD_LIBRARY_PATH
```

6. Mit

```
$ java HauptGutenTag
```

kann man das Programm schließlich laufen lassen.

17.1.2 Übergabe von Parametern

Mit JNI lassen sich natürlich auch die Übergabe von Argumenten und Rückgabewerte realisieren. Dabei sind aber manche Datentypen zu konvertieren.

Standarddatentypen wie `int` können beinahe direkt, – d.h. unter dem Namen `jint` – in den C-Code eingebunden werden. Bei komplexeren Datentypen wie `String`, Felder und Klassen benötigt man spezielle Umwandlungsfunktionen, die zwischen C und Java vermitteln.

Es ist auch möglich Java-Datenelemente zu verwenden. Hierauf gehen wir nicht ein.

Beispielhaft gehen wir hier noch auf die Verwendung von Strings und Feldern ein.

Strings

Bei Strings ist darauf zu achten, dass sie nur wie 8-Bit-Zeichen verwendet werden.

Wandeln von Java nach C:

```
char* cString = (*env)->GetStringUTFChars (
   JNIEnv* env, jstring javaString, 0);
```

Wandeln von Java nach C++:

```
char* cString = env->GetStringUTFChars (jstring javaString, 0);
```

Anlegen und Wandeln von C nach Java:

```
jstring  javaString = (*env)->NewStringUTF (
   JNIEnv* env, char* cString);
```

Anlegen und Wandeln von C++ nach Java:

```
jstring  javaString = env->NewStringUTF (char* cString);
```

Felder

Hier wird kurz gezeigt, wie Bytefelder zu wandeln sind:

Wandeln von Java nach C:

```
char* cByteFeld = (*env)->GetByteArrayElements (
   JNIEnv *env, jbyteArray jByteFeld, 0);
```

Wandeln von Java nach C++:

```
char* cByteFeld = env->GetByteArrayElements (jbyteArray jByteFeld, 0);
```

Anlegen in C:

```
jbyteArray jByteFeld = (*env)->NewByteArray (JNIEnv* env, int length);
```

Wandeln von C nach Java:

```
(*env)->SetByteArrayRegion (
   JNIEnv* env, jbyteArray jByteFeld, int start, int length,
   char* cByteFeld);
```

Anlegen in C++:

```
jbyteArray jByteFeld = env->NewByteArray (int length);
```

Wandeln von C++ nach Java:

```
env->SetByteArrayRegion (
   jbyteArray jByteFeld, int start, int length, char* cByteFeld);
```

In folgendem einfachen Beispiel wird ein Integer, String und ein Feld von Bytes als Argument übergeben. Zurückgeliefert wird ein String.

```
public class   HauptGutenTag
   {
   public static void   main (String[] argument)
      {
      int   j = 0;
      String   s = "Guten Tag und Grüß Gott";
      byte[]   f = new byte[] {(byte) 7, (byte) 84, (byte) 53};
      System.out.println ("Nativ bekommt die Argumente");
      int   i = f.length;
      System.out.println ("   int i als " + i);
      System.out.println ("   String s als \"" + s + "\"");
      System.out.print ("   und das Bytefeld f als ");
      for (int k = 0; k < f.length; k++)
         System.out.print (f[k] + " ");
      System.out.println ();
      System.out.println ();
      String t = new NativGutenTag ().gruessDieWelt (i, s, f);
      System.out.println ("Nativ gibt den String \"" + t + "\" zurück.");
      }
   }
```

```
public class   NativGutenTag
   {
   public native String gruessDieWelt (int i, String s, byte[] f);
      // i Länge des Feldes f

   static
      {
      System.loadLibrary ("gutentag");
      }

   }
```

```
#include "NativGutenTag.h"
```

```
#include <stdio.h>

JNIEXPORT jstring JNICALL Java_NativGutenTag_gruessDieWelt
   (JNIEnv * dies, jobject job, jint i, jstring javas, jbyteArray javaf)
   {
   int   j = 7;
   int   k = 0;
   int   m = 0;
   const char* cs = (*dies)->GetStringUTFChars (dies, javas, 0);
   char* cf = (*dies)->GetByteArrayElements (dies, javaf, 0);
   char ct[200];
   jstring  javat;
   printf ("Es grüßt die native C-Funktion mit\n");
   printf ("   i = %d\n", i);
   printf ("   j = %d\n", j);
   printf ("   s = %s\n", cs);
   printf ("   f = ");
   for (k = 0; k < i; k++)
      printf (" %d", cf[k]);
   printf ("\n");
   k = 0;
   ct[k++] = 7 + '0';
   ct[k++] = ' '; ct[k++] = 'm'; ct[k++] = 'a'; ct[k++] = 'l'; ct[k++] = ' ';
   while (*cs != '\0') ct[k++] = *cs++;
   for (m = 0; m < i; m++)
      {
      ct[k++] = ' ';
      ct[k++] = cf[m] / 10 + '0';
      ct[k++] = cf[m] - (cf[m] / 10) * 10 + '0';
      }
   ct[k++] = '\0';
   javat = (*dies)->NewStringUTF (dies, ct);
   return javat;
   }
```

Kapitel 18

Servlets

Mit Java-Servlets kann man die Funktionalität eines Web-Servers erweitern. Ein Servlet kann man sich als ein Applet vorstellen, daß anstatt auf der Client-Maschine auf der Server-Maschine läuft. Servlets arbeiten mit ***request/response***-Modell, wobei der Client eine Anfrage *(request)* an den Server sendet, der eine Antwort *(response)* zurückschickt.

Servlets werden typischerweise als ***Middleware*** eingesetzt, d.h. auf einer Stufe zwischen dem Client und z.B. einem Datenbank-Server. Damit bringen Servlets einem System sehr viel Flexibilität.

Servlets können für jede Client-Anfrage gestartet und wieder gestoppt werden. Oder sie können mit dem Web-Server gestartet werden und laufen dann solange, wie der Web-Server läuft. Auf die Servlet-Installations-Prozeduren der verschiedenen Web-Server kann hier nicht eingegangen werden. Für die hier gezeigten Beispiele verwenden wir das Java Servlet Development Kit (JSDK) der Firma Sun. Nach Entpacken des `tar`-Files kann man im `README` nachlesen, wie der Server gestartet und gestoppt wird. Eventuell muß die Datei `default.cfg` editiert werden, um Server-Namen und Internet-Adresse anzupassen.

Die Servlet-API befindet sich in den Paketen:

```
javax.servlet
javax.servlet.http
```

18.1 Beispiel `GutenTagServlet`

Für ein Hello-World-Servlet sind folgende Schritte durchzuführen:

1. Schreiben des Java-Codes:
   ```
   import   java.io.*;
   import   javax.servlet.*;
   import   javax.servlet.http.*;
   ```

```
public class   GutenTagServlet extends HttpServlet
   // Sendet eine einfache statische HTML-Seite zurück.
   {
   public void doGet
      (
      HttpServletRequest   anfrage,
         // Enthält Information, die der Browser schickt.

      HttpServletResponse  antwort
         // Enthält Information, die an den Browser geschickt wird.

      ) throws ServletException, IOException
      {
      antwort.setContentType ("text/html");
         // Legt den MIME-Typ fest.

      ServletOutputStream  aus = antwort.getOutputStream ();
         // Das ist der Stream zum Browser,
         // in den jetzt eine HTML-Seite geschrieben wird:

      aus.println ("<HTML>");
      aus.println ("<HEAD><TITLE>Guten Tag</TITLE></HEAD>");
      aus.println ("<BODY>");
      aus.println ("<H1>Guten Tag!</H1>");
      aus.println ("</BODY>");
      aus.println ("</HTML>");

      aus.close ();
      }
   }
```

Dieses Servlet generiert nur eine statische HTML-Seite.

2. Übersetzen des Servlets:
   `$ javac GutenTagServlet.java`

3. Installation im Web-Server, z.B im JSDK:

   (a) Lage der Servletklasse:

      i. Kopiere `GutenTagServlet.class` nach:

         `xxx/jsdk2.1/webpages/WEB-INF/servlets`

      Damit ist die Klasse `GutenTagServlet` im Klassenpfad des Web-Servers und kann so vom Server gefunden werden.

      ii. oder mach einen jar-File und stelle den in:

         `xxx/jsdk2.1/lib`

Dieser jar-File muss dann in `startserver` oder `startserver.bat` im `CLASSPATH` aufgenommen werden als `lib/xxx.jar` oder `lib\xxx.jar`.

(b) Editiere den File

`xxx/jsdk2.1/webpages/WEB-INF/servlets.properties`

und ergänze die Zeile:

`gttg.code=GutenTagServlet`
oder hier muß eventuell der Package-Name mit angegeben werden, wenn sich die Klasse in einer Bibliothek befindet:
`gttg.code=kj.tep.GutenTagServlet`

Damit wird ein vom Programmierer gewählter Servlet-Name `gttg` auf die Java-Klasse `GutenTagServlet` abgebildet. Das Servlet ist dann über die URL

`http://<host:port>/servlet/gttg`
z.B.
`http://bogart:8080/servlet/gttg`
oder
`http://localhost:7777/servlet/gttg`

aufrufbar.

(c) Durch Einträge im File

`xxx/jsdk2.1/webpages/WEB-INF/mappings.properties`

kann das Servlet auch auf andere URLs abgebildet werden. Z.B. wird durch die Ergänzung der Zeile

`/gutentag=gttg`

die Web-Server-URL

`http://<host:port>/gutentag`
z.B.
`http://bogart:8080/gutentag`

auf das Servlet `gttg` abgebildet.

4. Aufruf des Servlets mit einem Browser durch Angabe der URL, z.B:

`http://bogart:8080/gutentag`

18.2 Beispiel `ParameterServlet`

Dem Servlet `ParameterServlet` können Parameter übergeben werden. Die beiden Parameter heißen `Vorname` und `Nachname`. Dabei ist auf Groß- und Kleinschreibung zu achten.

Java-Code:

```
import   java.io.*;
import   javax.servlet.*;
import   javax.servlet.http.*;
```

```
public class   ParameterServlet extends HttpServlet
   // Sendet eine einfache statische HTML-Seite zurück,
   // die in Abhängigkeit von übergebenen Parametern verändert ist.
   {
   public void doGet
      (
      HttpServletRequest   anfrage,
      HttpServletResponse  antwort
      ) throws ServletException, IOException
      {
      antwort.setContentType ("text/html");
      ServletOutputStream  aus = antwort.getOutputStream ();

      String   vorname = anfrage.getParameter ("Vorname");
      String   nachname = anfrage.getParameter ("Nachname");
         // Die Parameter werden geholt.

      aus.println ("<HTML>");
      aus.println ("<HEAD><TITLE>Guten Tag mit Parametern</TITLE></HEAD>");
      aus.println ("<BODY>");
      aus.println ("<H1>Guten Tag " + vorname + " " + nachname + "!</H1>");
      aus.println ("</BODY>");
      aus.println ("</HTML>");

      aus.close ();
      }
   }
```

Eintrag im File `servlets.properties`:

```
pgttg.code=ParameterServlet
```

Eintrag im File `mappings.properties`:

```
/parameter=pgttg
```

Aufruf im Browser mit URL:

```
http://bogart:8080/parameter?Vorname=Karl%20Friedrich&Nachname=Gebhardt
```

Die Parameter erscheinen nach dem `?`. Das Blank im Vornamen muß als `%20` angegeben werden. Der zweite Parameter wird durch `&` abgetrennt.

18.3 Beispiel *Server Side Include* DatumServlet

Das Servlet DatumServlet erzeugt eine Ausgabe, die innerhalb einer Web-Seite benutzt werden kann *(Server Side Include,* SSI). Diese Servlet wird mit dem speziellen Tag SERVLET aktiviert, das nicht von allen Web-Servern unterstützt wird.

HTML-File:

```
<HTML>
<HEAD><TITLE>ServerSideInclude-Beispiel</TITLE></HEAD>
<BODY>
<H1>Guten Tag
<SERVLET code=DatumServlet>
<PARAM NAME="Vorname" VALUE="Karl Friedrich">
<PARAM NAME="Nachname" VALUE="Gebhardt">
</SERVLET>
Das hat funktioniert.
</H1>
</BODY>
</HTML>
```

Java-Code:

```
import   java.io.*;
import   java.util.*;
import   java.text.*;
import   javax.servlet.*;
import   javax.servlet.http.*;

public class   DatumServlet extends HttpServlet
   {
   public void service
      (
      HttpServletRequest   anfrage,
      HttpServletResponse  antwort
      ) throws ServletException, IOException
      {
      antwort.setContentType ("text/plain");
      ServletOutputStream  aus = antwort.getOutputStream ();

      String   vorname = anfrage.getParameter ("Vorname");
      String   nachname = anfrage.getParameter ("Nachname");

      String   heute = DateFormat.getDateInstance ().format (new Date ());

      aus.println (vorname + " " + nachname + "!");
      aus.println ("Heute ist der " + heute);

      aus.close ();
```

```
    }
  }
```

18.4 Beispiel `FormularServlet`

Mit diesem Servlet kann eine Tabelle, die und deren Struktur in Dateien abgelegt sind, editiert werden. Dabei sind typische Nebenläufigkeits-Probleme nicht berücksichtigt worden, da diese typischerweise von einem Datenbank-Management-System übernommen werden würden.

Dieses Servlet kann nach dem Übersetzen von

```
FormularServlet.java
SpaltenInformation.java
FormularBearbeitung.java
```

und Anlegen der Files

```
Struktur.dat
Daten.dat
```

in dem Verzeichnis, das in der Konstanten `FormularServlet.PFAD` steht, mit der URL:
`http://bogart:8080/servlet/`
`FormularServlet?Strukturdatei=Struktur.dat&Datendatei=Daten.dat`
gestartet werden.

Code von `FormularServlet`:

```
import   java.io.*;
import   java.util.*;
import   javax.servlet.*;
import   javax.servlet.http.*;

         /**
            Dieses Servlet wird ein Daten-Editier-Formular erzeugen.
            Es benötigt zwei Parameter:

               Strukturdatei: Der Name einer Datei, die die Struktur
                  der Information enthält
                  Format: <Kopf><TAB><Typ><TAB><Breite>

               Datendatei: Der Name einer Datei, die die Daten enthält,
                  die dargestellt werden sollen
```

```
                Format: <Wert Spalte 1><TAB><Wert Spalte 2><TAB>...

           Alle Daten werden als String gespeichert.
        **/

public class   FormularServlet extends HttpServlet
   {
   static final int  ZEICHENGRÖSSE = 10;
   static final String  PFAD
      = "/local/home/kfg/jsdk2.1/webpages/WEB-INF/servlets/";
      // Verzeichnis, wo sich die Strukturdatei und Datendatei befinden.

   static final String  SERVER = "localhost:8080/servlet/";

   public void doGet (HttpServletRequest frage, HttpServletResponse antwort)
      // Erzeugt HTML-Seite
      throws   ServletException, IOException
      {
      antwort.setContentType ("text/html");

      // Übernehmen der Parameter (Dateinamen):

      String   parameterStrukturDatei
         = PFAD + frage.getParameter ("Strukturdatei");
      if (parameterStrukturDatei == null)
         {
         fehler ("Parameter Strukturdatei fehlt!", null);
         return;
         }
      String   parameterDatenDatei
         = PFAD + frage.getParameter ("Datendatei");
      if (parameterDatenDatei == null)
         {
         fehler ("Parameter Datendatei fehlt!", null);
         return;
         }

      // Einlesen der Dateien:

      Vector   spalte = new Vector ();
      TabellenSpalte spaltenInfo;
      BufferedReader fs;
      String   eingabe = null;
      try
         {
         fs = new BufferedReader (
            new FileReader (parameterStrukturDatei));
         while ( (eingabe = fs.readLine () ) != null)
            {
```

```
            spaltenInfo = new TabellenSpalte (eingabe);
            spalte.addElement (spaltenInfo);
            }
         }
         catch (Exception e) { fehler ("Strukturfehler:", e); }

      try
         {
         StringTokenizer   st;
         fs = new BufferedReader (
            new FileReader (parameterDatenDatei));
         while ( (eingabe = fs.readLine () ) != null)
            {
            st = new StringTokenizer (eingabe, TabellenSpalte.TRENNER);
            for (int i = 0; i < spalte.size (); i++)
               {
               ( (TabellenSpalte) spalte.elementAt (i) )
                  .setFolgeWert (st.nextToken () );
               }
            }
         }
         catch (Exception e) { fehler ("Datenfehler:", e); }

      // Nun wird das HTML-Formular erzeugt:

      //ServletOutputStream   aus = antwort.getOutputStream ();
      PrintWriter aus = antwort.getWriter ();

      aus.println ("<HTML>");
      aus.println ("<HEAD><TITLE>Daten-Editier-Formular</TITLE></HEAD>");
      aus.println ("<BODY>");
      aus.println ("<FORM METHOD=POST ACTION="
         + "\"http://" + SERVER +"FormularBearbeitung\">");
         // Alles, was innerhalb des FORM-Tags steht, wird an das
         // Servlet FormularBearbeitung geschickt.

      aus.println ("<INPUT TYPE=HIDDEN NAME=Strukturdatei VALUE="
         + parameterStrukturDatei + ">");
      aus.println ("<INPUT TYPE=HIDDEN NAME=Datendatei VALUE="
         + parameterDatenDatei + ">");
      aus.println ("<TABLE WIDTH=100% BORDER=1>");

      // Spaltenköpfe:

      aus.println ("<TR>");
      for (int i = 0; i < spalte.size (); i++)
         {
         spaltenInfo = ( (TabellenSpalte) spalte.elementAt (i) );
         int b = spaltenInfo.getBreite ();
```

```
         String   k = spaltenInfo.getKopf ();
         if (b < k.length () ) b = k.length ();
         aus.println ("<TD WIDTH=" + (ZEICHENGRÖSSE * b) + ">" + k + "</TD>");
         }
      aus.println ("</TR>");

      // Zeilen:

      for (int z = 0;
         z < ( (TabellenSpalte) spalte.elementAt (0) ).getAnzahl (); z++)
         {
         aus.println ("<TR>");
         for (int i = 0; i < spalte.size (); i++)
            {
            spaltenInfo = ( (TabellenSpalte) spalte.elementAt (i) );
            String   k = spaltenInfo.getKopf ();
            aus.println ("<TD>");
            aus.println ("<INPUT TYPE=\"text\" NAME=\"" + k
               + "\" SIZE=\"" + (1 + spaltenInfo.getBreite () )
               + "\" VALUE=\"" + spaltenInfo.getWert (z)
               + "\">");
            aus.println ("</TD>");
            }
         aus.println ("</TR>");
         }
      aus.println ("</TABLE>");

      aus.println ("<BR>");
      aus.println ("<INPUT TYPE=\"submit\" NAME=\"CMD\" "
         + "VALUE=\"Änderungen Ausführen\">  "
         + "<INPUT TYPE=\"reset\">");

      aus.println ("</FORM>");

      aus.println ("</BODY>");
      aus.println ("</HTML>");
      }

   public String  getServletInfo ()
      {
      return "Ein Servlet, das ein Formular an den Browser schickt.";
      }

   static void fehler (String meldung, Object e)
      {
      System.err.println (meldung + " (" + e + ")");
      System.err.flush ();
      }
   }
```

Code von SpaltenInformation:

```
import   java.util.*;
         /**
            Representiert die Spalte einer Tabelle.
            Verwaltet Namen der Spalte, Typ der Spalte,
            Breite der Spalte für die Darstellung
            und alle Werte einer Spalte.
         **/
public class   TabellenSpalte
   {
   public static final String TRENNER = "\t";
      // Trennt die Einträge in der Strukturdatei und
      // in der Datendatei.

   private String kopf;
   private String typ;
   private int breite;
   private Vector wert = new Vector ();

   public   TabellenSpalte (String eingabe) throws Exception
      {
      StringTokenizer   st = new StringTokenizer (eingabe, TRENNER);
      kopf = st.nextToken ();
      typ = st.nextToken ();
      breite = Integer.parseInt (st.nextToken () );
      }

   public int  getBreite () { return breite; }
   public String  getKopf () { return kopf; }
   public String  getWert (int z) { return (String) wert.elementAt (z); }
   public void setFolgeWert (String w) { wert.addElement (w); }
   public void setWert (int z, String w) { wert.setElementAt (w, z); }
   public int  getAnzahl () { return wert.size (); }
   }
```

Code von FormularBearbeitung:

```
import   java.io.*;
import   java.util.*;
import   javax.servlet.*;
import   javax.servlet.http.*;

         /**
            Dieses Servlet empfängt Daten, die vom Browser
            mit HTTP POST gesendet werden. Es verwendet
```

```
            dieselben beiden Dateien, die FormularServlet
            verwendet.
         **/
public class   FormularBearbeitung extends HttpServlet
   implements  SingleThreadModel
      // Diese Schnittstelle hat keine Methdoden, sondern sorgt dafür,
      // daß die Service-Methode des Servlets unter gegenseitigem
      // Ausschluß durchgeführt wird.
   {
boolean  debug = true;
   public void doPost (HttpServletRequest frage, HttpServletResponse antwort)
      throws ServletException, IOException
      {
      antwort.setContentType ("text/html");
      ServletInputStream   ein = frage.getInputStream ();

      int   länge = frage.getContentLength ();
         // Man braucht die Länge des Inhalts.

      Hashtable   inhalt = HttpUtils.parsePostData (länge, ein);
         // Information des Clients kommt in eine Hash-Tabelle.

      String[] parameterStrukturDatei
         = (String[]) inhalt.get ("Strukturdatei");
      if (parameterStrukturDatei == null)
         {
         FormularServlet.fehler ("Parameter Strukturdatei fehlt!", null);
         return;
         }
      String[] parameterDatenDatei
         = (String[]) inhalt.get ("Datendatei");
      if (parameterDatenDatei == null)
         {
         FormularServlet.fehler ("Parameter Datendatei fehlt!", null);
         return;
         }

      // Öffnen der Dateien und lesen des Inhalts der Dateien:

      Vector   spalte = new Vector ();
      BufferedReader r = null;
      ServletContext context = null;
      try
         {
if (debug) FormularServlet.fehler ("Strukturdatei", parameterStrukturDatei[0]);
         context = getServletContext ();
         r = new BufferedReader (
            new InputStreamReader (
               //context.getResourceAsStream (
```

```
               // This does not work!!! NullPointer-Exceptions
               new FileInputStream (   // Workaround
                  parameterStrukturDatei[0]) ) );
         String   eingabe = null;
if (debug) FormularServlet.fehler ("Eingabe", eingabe);
if (debug) eingabe = r.readLine ();
if (debug) FormularServlet.fehler ("Eingabe", eingabe);
         while ( (eingabe = r.readLine () ) != null)
            {
if (debug) FormularServlet.fehler ("Eingabe", eingabe);
            TabellenSpalte spi = new TabellenSpalte (eingabe);
            if (spi != null) spalte.addElement (spi);
            }
         }
         catch (Exception e)
            {
            FormularServlet.fehler ("Fehler Strukturdatei! (r = " + r
               + ", context = " + context + ")", e);
            return;
            }
         finally
            {
            if (r != null) r.close ();
            }

      // Einlesen der Werte aus der Hash-Tabelle in TabellenSpalte-Objekte:

      for (int i = 0; i < spalte.size (); i++)
         {
         TabellenSpalte spi = (TabellenSpalte) spalte.elementAt (i);
         String[] wert = (String[]) inhalt.get (spi.getKopf () );
         for (int z = 0; z < wert.length; z++)
            {
            spi.setFolgeWert (wert[z]);
            }
         }

      // Jetzt wird das HTML-Formular gemacht,
      // und die Datendatei wird aktualisiert:

      PrintWriter dd = new PrintWriter (
         new FileWriter (parameterDatenDatei[0]) );
      PrintWriter aus = antwort.getWriter ();
      aus.println ("<HTML>");
      aus.println ("<HEAD><TITLE>Datei-Darstellungs-Formular</TITLE></HEAD>");
      aus.println ("<BODY>");
      aus.println ("<TABLE WIDTH=100% BORDER=1>");

      aus.println ("<TR>");
```

```
      for (int i = 0; i < spalte.size (); i++)
         {
         TabellenSpalte spi = (TabellenSpalte) spalte.elementAt (i);
         int   b = spi.getBreite ();
         String   k = spi.getKopf ();
         if (b < k.length () ) b = k.length ();
         aus.println ("<TH WIDTH=" + (FormularServlet.ZEICHENGRÖSSE * b) + ">");
         aus.println (k);
         aus.println ("</TH>");
         }
      aus.println ("</TR>");

      for (int z = 0;
         z < ( (TabellenSpalte) spalte.elementAt (0) ).getAnzahl (); z++)
         {
         aus.println ("<TR>");
         for (int i = 0; i < spalte.size (); i++)
            {
            TabellenSpalte spi
               = (TabellenSpalte) spalte.elementAt (i);
            aus.println ("<TD>");
            aus.println (spi.getWert (z) );
            dd.print (spi.getWert (z) );
            if (i < spalte.size () - 1) dd.print (TabellenSpalte.TRENNER);
            aus.println ("</TD>");
            }
         dd.println ();
         aus.println ("</TR>");
         }
      aus.println ("</TABLE>");
      aus.println ("</BODY>");
      aus.println ("</HTML>");
      aus.close ();
      dd.close ();
      }

   }
```

Datei `Struktur.dat`:

```
Z  String   1
Datum String   8
Bereich  String   2
Besteller   String   10
Preis String   10
Bezeichnung String   20
Firma String   20
```

Datei `Daten.dat`:

```
r  19.4.99  TI Elzmann  4300.00  Teamwork Sterling
b  -  TI Elzmann  19140.00 V32   -
r  23.12.98 TI Gebhardt 299.28   2 Hubs   inmac
r  28.1.99  TI Gebhardt 125.28   Beamer-Schlüssel  Schlüsselhilfe
r  4.10.99  TI Gebhardt 466.32   64MB Speicher, AGP-VGA-Karte  Arlt
b  -  TI Gebhardt 200.00   64MB Speicher  Arlt
r  22.11.99 TI Gebhardt 898.00   Robotics-Invention   Kurtz
r  3.12.99  TI Gebhardt 2538.75  M62, A201S  men
r  19.4.99  TI Herrler  99.00 Linux SuSE
r  23.4.99  TI Herrler  945.40   Laserdrucker   Bechtle
r  15.3.99  TI Messer   402.72   Umschalter  Dakota
r  30.3.99  TI Messer   115.00   MS Visual C++  Uni Karlsruhe
r  31.5.99  TI Messer   18.89 CD-Labels   Pearl
r  31.5.99  TI Richter  98.99 Lotus asknet
r  31.5.99  TI Richter  163.00   Lotus asknet
r  23.3.99  TI Schneider   324.66   Netzteil ua Conrad
r  25.3.99  TI Schneider   324.66   Werkzeug ua RS Components
r  29.9.99  TI Weghorn  378.62   Festplatte, CD Arlt
r  22.11.99 TI Richter  1385.05  2 Multimeter   Fluke
r  12.11.99 TI Richter  2015.00  Mathematica asknet
```

Das generelle DBMS-Problem, daß mehrere Benutzer ihre Aktualisierungen gegenseitig nicht überschreiben, wird hier trotz des verwendeten `SingleThreadModel`s nicht allgemein gelöst. Das ist kein leichtes Problem, da nicht leicht zu ermitteln ist, ob die gerade vorliegenden Daten auf die Platte zu schreiben sind.

Ein Möglichkeit ist, den Datum-Zeit-Stempel zu verwenden. Dazu müßte man ein zusätzliches `HIDDEN` Feld verwenden, das den Zeitpunkt enthält, zu dem die Daten aus der Datei gelesen wurden. Dann muß man vor dem Schreiben auf die Datei prüfen, ob das letzte Aktualisierungs-Datum der Datei noch davor liegt. Wenn das nicht der Fall war, weiß man, daß jemand anderes die Datei zwischendurch verändert hat. In dem Fall muß man die eigene Aktualisierung eventuell zurückweisen.

Kapitel 19

Reflexion

```
import   java.lang.reflect.*;

public class   Konstruktor
   {
   public static void   main (String[] arg) throws Throwable
      {
      Class klasse = Class.forName ("A");

      Class[]  einStringIntKonstruktorParameter
         = new Class[] {String.class, Integer.TYPE};
      Constructor konstruktor
         = klasse.getConstructor (einStringIntKonstruktorParameter);
      Object[] argumente
         = new Object[] {"Hello", new Integer (333)};
      A  a = (A) konstruktor.newInstance (argumente);
      System.out.println ("String-Int-Konstruktor: a : " + a);

      Class[]  einStringKonstruktorParameter
         = new Class[] {String.class};
      konstruktor
         = klasse.getConstructor (einStringKonstruktorParameter);
      argumente = new Object[] {"Guten Tag"};
      a = (A) konstruktor.newInstance (argumente);
      System.out.println ("String-Konstruktor: a : " + a);

      Class[]  einIntKonstruktorParameter
         = new Class[] {Integer.TYPE};
      konstruktor
         = klasse.getConstructor (einIntKonstruktorParameter);
      argumente = new Object[] {new Integer (12345)};
      a = (A) konstruktor.newInstance (argumente);
      System.out.println ("Int-Konstruktor: a : " + a);
```

```
      a = (A) klasse.newInstance ();
      System.out.println ("Default-Konstruktor: a : " + a);
      }
   }

class A
   {
   String   s;
   int   i;

   public   A (String s, int i)
      {
      this.s = s;
      this.i = i;
      }

   public   A (String s)
      {
      this (s, 113);
      }

   public   A (int i)
      {
      this ("unbekannt", i);
      }

   public   A ()
      {
      this (117);
      }

   public String  toString ()
      {
      return "s = " + s + "   i = " + i;
      }
   }
```

Kapitel 20

JavaBeans

Definition: *Ein **Bean** ist eine wiederverwendbare Softwarekomponente, die mit einem Generierungs-Tool manipuliert werden kann. JavaBeans ist ein Software-Komponenten-Modell für Java.*

Bemerkungen:

1. Für den **Anwender** eines Beans ist ein Bean eine Klasse wie jede andere auch.
2. Ein Bean ist eine Klasse, die von graphischen Designwerkzeugen oder Anwendungsgeneratoren (sogenannten **Beanbox-Tools**, z.B. **Beanbox** von Sun) verwendet werden kann, wenn sie **Bean-regelgerecht** erstellt wurde. Ein Bean kann visuell manipuliert werden.
3. Der **Entwickler** eines Beans muß die **Java-Beans-API** gut kennen und benutzen.
4. Ein Bean ist eine Klasse, die nach bestimmten Regeln entwickelt wurde.
5. Ein Bean muß nicht unbedingt eine GUI-Komponente sein.
6. Ein Bean kann eine einfache Klasse sein (z.B. eine AWT-Komponente) oder ein ganzes Software-Paket repräsentieren (z.B. eine einbettbare Tabellenkalkulation).

20.1 Bean-Regeln

Bei den Bean-Regeln geht es darum, durch welche Methoden die **Eigenschaften** ***(property)*** eines Beans zugänglich gemacht werden sollen. Eine Eigenschaft hat einen Namen, im folgenden immer den Namen `Eigen`, und eine Typ `T`. Der Name wird vorn immer groß geschrieben.

Die Implementation einer Eigenschaft spielt für das Bean keine Rolle. D.h. die Eigenschaft kann ein einfaches Datenelement sein, dessen Name durch die Bean-Regeln nicht festgelegt wird, oder sie kann irgendwie kompliziert berechnet werden oder aus einer Datenbank geholt werden.

Nicht alle `set-`, `get-` oder `is`-Methoden müssen für eine Eigenschaft implementiert werden. Damit sie aber überhaupt als eine Eigenschaft da ist, muß wenigstens eine dieser Methoden implementiert sein. Entsprechend unterscheidet man *read-only*, *write-only* und *read-write* Eigenschaften.

Bean

Klassenname:	**beliebig**
Superklasse:	**beliebig**
entweder:	**Konstruktor ohne Argumente**
oder:	**serialisierte Grundversion**
JAR-Datei-Manifesteintrag:	`Java-Bean: True`

Reguläre Eigenschaft vom Typ `T` mit Namen `Eigen`
(Implementation z.B. `private T eigen;`)

```
public T getEigen () {...}

public void setEigen (T wert) {...}
```

Boolesche Eigenschaft mit Namen `Eigen`
(Implementation z.B. `private boolean eigen;`)

```
public boolean getEigen () {...}
```
oder
```
public boolean isEigen () {...}

public void setEigen (boolean eigen) {...}
```

Indexierte Eigenschaft vom Typ `T` mit Namen `Eigen`
(indexed property)
(Implementation z.B. `private T[] eigen = new T[20];`
oder `private Vector eigen = new Vector ();`)

```
public T getEigen (int index) {...}

public void setEigen (int index, T eigen) {...}

public T[]  getEigen () {...}

public void setEigen (T[] eigen) {...}
```

Gebundene Eigenschaft vom Typ `T` mit Namen `Eigen`
Wenn sich eine gebundene Eigenschaft ***(bound property)*** ändert, können andere Objekte benachrichtigt werden.

get/set-Methoden wie bei den oben genannten Eigenschaften plus Registriermethoden für Event-Listener:
```
public void addPropertyChangeListener (PropertyChangeListener l) {..}

public void removePropertyChangeListener (PropertyChangeListener l) {..}
```

Optional kann ein Bean Methoden für die Registrierung von Listenern per Eigenschaft zur Verfügung stellen (siehe Klasse `PropertyChangeSupport` unten). Ferner kann ein Bean

optional Registriermethoden für jede gebundene Eigenschaft zur Verfügung stellen:
`public void addEigenListener (PropertyChangeListener l) {..}`

`public void removeEigenListener (PropertyChangeListener l) {..}`
Erst damit kann ein Entwicklungswerkzeug durch Inspektion der Klasse zwischen einer gebundenen und ungebundenen Eigenschaft unterscheiden.

Die set-Methoden werden so programmiert, daß sie ein Event feuern. Dazu wird in der set-Methode die Methode

```
public void propertyChange (PropertyChangeEvent e)
```

aller `PropertyChangeListener` aufgerufen. Die Benachrichtigungs-Granularität ist nicht die Eigenschaft. D.h. die aufgerufene Methode ist für alle Eigenschaften diesselbe.

Zur Implementation einer gebundenen Eigenschaft stellt das API die Klasse

```
PropertyChangeSupport
```

zur Verfügung mit den Methoden:

```
public void addPropertyChangeListener (PropertyChangeListener l)
public void addPropertyChangeListener (
                    String propertyName, PropertyChangeListener l)
public void removePropertyChangeListener (PropertyChangeListener l)
public void removePropertyChangeListener (
                    String propertyName, PropertyChangeListener l)
public boolean hasListeners (String propertyName)

public void firePropertyChange (PropertyChangeEvent e)
public void firePropertyChange (
               String propertyName, int oldValue, int newValue)
public void firePropertyChange (
               String propertyName, boolean oldValue, boolean newValue)
public void firePropertyChange (
               String propertyName, Object oldValue, Object newValue)
```

Eingeschränkte Eigenschaft vom Typ `T` mit Namen `Eigen`

Bei eingeschränkten Eigenschaften ***(constrained property)*** können benachrichtigte Objekte ein Veto einlegen. Diese Objekte legen ihr Veto ein, indem sie die

```
PropertyVetoException
```

werfen.

get-Methoden wie bei den oben genannten Eigenschaften, aber set-Methoden mit

```
throws PropertyVetoException
```

plus Registriermethoden für Event-Listener:

```
public void addVetoableChangeListener (VetoableChangeListener l) {..}

public void removeVetoableChangeListener (VetoableChangeListener l) {..}
```

Optional kann ein Bean Methoden für die Registrierung von Listenern per Eigenschaft zur Verfügung stellen (siehe Klasse `VetoableChangeSupport` unten). Ferner kann ein Bean optional Registriermethoden für jede gebundene Eigenschaft zur Verfügung stellen:
`public void addEigenListener (VetoableChangeListener l) {..}`

`public void removeEigenListener (VetoableChangeListener l) {..}`
Erst damit kann ein Entwicklungswerkzeug durch Inspektion der Klasse zwischen einer eingeschränkten und nicht eingeschränkten Eigenschaft unterscheiden.

Die set-Methoden werden wieder so programmiert, daß sie ein Event feuern, indem sie die Methode

```
public void vetoableChange (PropertyChangeEvent e)
```

aller `VetoableChangeListener` aufruft.

Zur Implementation einer eingeschränkten Eigenschaft stellt das API die Klasse

```
VetoableChangeSupport
```

zur Verfügung mit den Methoden:

```
public void addVetoableChangeListener (VetoableChangeListener l)
public void addVetoableChangeListener (
                    String propertyName, VetoableChangeListener l)
public void removeVetoableChangeListener (VetoableChangeListener l)
public void removeVetoableChangeListener (
                    String propertyName, VetoableChangeListener l)
public boolean hasListeners (String propertyName)

public void fireVetoableChange (PropertyChangeEvent e)
                                        throws PropertyVetoException
public void fireVetoableChange (
               String propertyName, int oldValue, int newValue)
                                        throws PropertyVetoException
public void fireVetoableChange (
               String propertyName, boolean oldValue, boolean newValue)
                                        throws PropertyVetoException
public void fireVetoableChange (
               String propertyName, Object oldValue, Object newValue)
                                        throws PropertyVetoException
```

Ereignisse, Events mit Namen `Ereignis`

Event-Klassenname:	`EreignisEvent`
Listener-Name:	`EreignisListener`
Listener-Methoden:	`public void beliebig (EreignisEvent e)`
Listener-Registrierung:	`public void addEreignisListener (EreignisEvent e)` `public void removeEreignisListener (EreignisEvent e)`

Unicast-Events
Nur ein Listener ist erlaubt. Wie oben, aber die add-Methode wirft die Exception

```
throws ToManyListenersException
```

Methoden
beliebig

20.2 Hilfsklassen

Die eigentliche Arbeit bei der Erstellung eines Beans ist die Bereitstellung der im folgenden beschriebenen Hilfsklassen. Allerdings sind diese Klassen sind alle optional. Insofern kann man sich die Arbeit eventuell sparen.

Im folgenden sei der Name des Beans `Bohne`.

1. Listener-Klassen müssen entsprechende Schnittstellen implementieren.
2. Klasse `BohneBeanInfo`
   Sie ist optional, aber deaktiviert die teilweise die automatische Inspektion der Beanklasse. Die `BohneBeanInfo`-Klasse stellt Informationen über die Klasse `Bohne` zur Verfügung. Dazu muß die Schnittstelle `BeanInfo` implementiert werden, bzw die Klasse `SimpleBeanInfo` erweitert werden.
3. Für jeden verwendeten Datentyp `Typ` ist ein Eigenschaftseditor zu schreiben oder zu besorgen. Das sind Klassen mit dem Namen:

   ```
   TypEditor
   ```

   Für die Standardtypen existieren solche Editoren bereits.
4. Es kann ein Customizer mit Namen

   ```
   BohneCustomizer
   ```

   erstellt werden. Er muß eine AWT- oder Swing-Komponente sein, z.B. `JPanel` und muß einen Konstruktor ohne Argumente haben.

   Ein Customizer ist dazu da, den Zustand des Beans insgesamt oder in einer besonderen Weise einzustellen. (Ein Eigenschaftseditor kann ja nur den Zustand einer einzelnen Eigenschaft ändern.)
5. Die Dokumentation des Beans kann in HTML-Format im File

   ```
   Bohne.html
   ```

   vorliegen.

Kapitel 21

Thread-Design

In diesem Kapitel werden Regeln für den Entwurf von nebenläufigen Programmen mit Tasks, insbesondere Java-Threads vorgestellt. Diese Regeln lassen sich ohne Weiteres auf andere nebenläufige Systeme verallgemeinern. Z.B. kann alles, was über die Verwendung von `synchronized` gesagt wird, auf alle Probleme des gegenseitigen Ausschlusses angewendet werden, und damit z.B. auf MUTEXe.

21.1 Thread Safety

Thread-Sicherheit ***(thread safety)*** bedeutet, dass die Datenelemente eines Objekts oder einer Klasse – dort sind es die Klassenvariablen – immer einen korrekten oder konsistenten Zustand aus der Sicht anderer Objekte oder Klassen haben, auch wenn von mehreren Threads gleichzeitig ***(shared)*** zugegriffen wird. (Lokale Variablen, Methodenparameter und Rückgabewerte sind davon nicht betroffen.)

Da Nebenläufigkeit ein wichtiges Merkmal von Java ist, muss man beim Design von Klassen grundsätzlich immer an Thread-Safety denken, da prinzipiell jedes Objekt oder jede Klasse in einer nebenläufigen Umgebung verwendet werden kann.

Thread-Safety kann folgendermaßen erreicht werden:

1. Kritische Bereiche erhalten ein `synchronized`.

   Alle Datenelemente, die inkonsistent werden können, werden `private` gemacht und erhalten eventuell `synchronized` Zugriffsmethoden. Ferner muss darauf geachtet werden, dass die Datenelemente nur über die Zugriffsmethoden verwendet werden. Für Konstanten gilt das nicht. ("inkonsistent werden können" bedeutet "durch mehr als einen Thread gleichzeitig veränderbar sein", ***mutable state variables***)

   `synchronized` ist notwendig, wenn mehrere Threads auf gemeinsame Daten (Objekte) zugreifen und mindestens ein Thread diese Daten (Objekte) verändert. Alle Methoden, die auf die Daten (Objekte) zugreifen, müssen als `synchronized` deklariert werden unabhängig davon, ob nur gelesen oder auch verändert wird.

Methoden-Aufrufe anderer Objekte sollten **nicht** in einer `synchronized` Umgebung erfolgen, weil das die Gefahr einer Verklemmung in sich birgt.

Der Zugriff auf einzelne Datenelemente vom Typ `boolean`, `char`, `byte`, `short`, `int` oder `float` ist unter Java Thread-sicher (nicht aber `long` und `double`).

2. Die Datenelemente werden unveränderbar (`final`) gemacht ***(immutable objects)***.

   Will man solche Objekte dennoch "ändern", dann muss man eventuell veränderte Kopien von ihnen anlegen.

3. Man benutzt einen Thread-sicheren Wrapper.

Warum kann man nicht einfach alles `synchronized` machen?

1. Synchronisierte Methoden-Aufrufe sind etwa 5 mal langsamer als nicht-synchronisierte Aufrufe.
2. Unnötige Synchronisationen (etwa bei Lesezugriffen) haben überflüssiges Blockieren von Threads zur Folge.
3. Immer besteht die Gefahr eines Deadlocks.
4. Auch wenn der einzelne Methodenaufruf Thread-sicher ist (`synchronized`), dann ist eine Kombination solcher Aufrufe trotzdem nicht Thread-sicher. Z.B. sind alle Methoden in der Klasse `Vector` `synchronized`, aber folgende Code-Folge ist nicht Thread-sicher:

   ```
   if (!vector.contains (element))
      {
      vector.add (element);
      }
   ```

   Um sie Thread-sicher zu machen, muss der ganze Block synchronisiert werden:

   ```
   synchronized (vector)
      {
      if (!vector.contains (element))
         {
         vector.add (element);
         }
      }
   ```

Die Verwendung von `synchronized` Blöcken hat den Nebeneffekt, dass Variablen, die eventuell im Cache waren, für andere Threads **sichtbar** werden. Wenn es allerdings nur darum geht, Variable für andere Threads sichtbar zu machen ***(sequential consistency)*** und nicht um Atomizität, dann kann – als mildere Form der Synchronisation – die Variable als `volatile` deklariert werden. Der Zugriff auf eine `volatile` Variable macht zu dem Zeitpunkt dann auch alle anderen Variablen für alle anderen Threads sichtbar.

Wenn Datenelemente nicht verändert werden ***(immutable)***, dann sind sie nach der vollständigen Objektkonstruktion Thread-sicher. Das kann auch noch für den Vorgang der Objektkonstruktion garantiert werden, wenn die Datenelemente `final` deklariert werden. Ferner kann auch kein Code erstellt werden, der diese Datenelemente verändert. Regel:

Alle Datenelemente sollten `final` deklariert werden, es sei denn, dass sie veränderlich sein sollen.

Bemerkung: Wenn Referenzen auf Objekte `final` deklariert werden, bedeutet das nicht, dass die referenzierten Objekte nicht verändert werden können, sondern nur, dass die `final` Referenz nicht ein anderes Objekt referenzieren kann.

21.2 Regeln zu `wait`, `notify` und `notifyAll`

21.2.1 Verwendung von `wait`, `notify` und `notifyAll`

Wenn durch einen Methodenaufruf der Zustand eines Objekts verändert wird **und** diese Änderung nur erfolgen darf, wenn eine gewisse **Wartebedingung** nicht mehr gegeben ist, dann hat diese Methode folgende Form:

```
synchronized void methode ()
   {
   while (Wartebedingung)
      {
      wait ();
      }

   // Mach Zustandsänderung.
   }
```

Das "`while`" ist notwendig, da nach Entlassung aus dem Wait-Set der Thread sich erst um das Lock bemühen muss, d.h. eventuell gewisse Zeit im Entry-Set verbringt, währenddessen ein anderer Thread die Wartebedingung wieder wahr machen könnte. Der zweite Grund sind spurious Wakeups (siehe unten).

Wenn eine Zustandsänderung dazu führt, dass eventuell wartende Threads weiterlaufen können, muss ein `notify` oder allgemeiner ein `notifyAll` aufgerufen werden:

```
synchronized void methode ()
   {
   // Mach Zustandsänderung.

   notifyAll ();
   }
```

Natürlich kann es sein, dass in einer Methode beide Fälle auftreten:

```
synchronized void methode ()
   {
   while (Wartebedingung)
      {
      wait ();
      }

   // Mach Zustandsänderung.

   notifyAll ();
   }
```

Nach **rein lesenden** Zugriffen muss grundsätzlich kein Thread geweckt werden (also **kein** `notify` oder `notifyAll`).

21.2.2 *Spurious Wakeup*

Es kann sein, dass ein Thread "aufwacht", ohne dass ein Timeout abgelaufen ist oder ein `notify`, `notifyAll` oder `interrupt` aufgerufen wurde. Dagegen muss man sich schützen. Das erreicht man i.A. dadurch, dass man die betroffenen `wait`s in eine `while`-Schleife bezüglich der Wartebedingung stellt (siehe Regel oben).

Spurious Wakeups kommen vor, weil z.B. das Multitasking des Host-Betriebssystems verwendet wird.

21.2.3 Timeout

Timeout "0" bedeutet bei `wait (timeout)` "kein Timeout", also unendlich langes Warten.

Das merkwürdige an der `wait (timeout)`-Methode ist, dass man nicht unterscheiden kann, ob der Timeout-Fall eingetreten ist, oder ob ein `notify` aufgerufen wurde. Man wird gezwungen ein Flag zu verwalten, das gesetzt wird, wenn ein `notify` aufgerufen wird, was nicht ganz trivial ist (wegen *spurious wakeup*).

21.2.4 Zusammenfassung `wait ()`, `notify ()`, `notifyAll ()`

Wir fassen die Verwendung von `wait ()`, `notify ()` und `notifyAll ()` in folgenden Regeln zusammen:

1. `wait`, `notify` und `notifyAll` können nur in einem Block aufgerufen werden, der für das aufrufende Objekt `synchronized` ist. Der Block kann natürlich auch eine `synchronized` Methode sein. (Die folgende Syntax bezieht sich auf diesen Fall.) Ferner verwenden wir **Sync** als Abkürzung für einen `synchronized` Block oder eine `synchronized` Methode.
2. `wait` gehört in eine While-Schleife:
   `while (Wartebedingung) wait ();`

3. Ein Sync muss ein `notify` oder `notifyAll` haben, wenn es mindestens ein `wait` gibt **und** wenn das Sync die `Wartebedingung` eines `wait`s möglicherweise `false` macht. (Ein Sync, das eine `Wartebedingung true` machen kann, braucht **kein** `notify` oder `notifyAll`.)

4. Ein `notify` genügt, wenn die Anwendung so ist, dass höchstens **ein** Thread die `wait`-Warteschlange verlassen soll und dass dieser Thread dann auch der "richtige" (der "gemeinte") ist.

5. In allen anderen Fällen muss ein `notifyAll` aufgerufen werden.

Wenn alle `wait`s in `while`-Schleifen verpackt sind, dann kann man immer ein `notifyAll` anstatt eines `notify` verwenden, ohne die Korrektheit des Programms zu beeinträchtigen. Auch überflüssige `notifyAll ()` schaden dann nicht. Das ist höchstens Performanz-schädlich.

21.3 `notify` oder `interrupt` ?

Mit `x.notify ()` wird **irgendein** Thread aus der `wait`-Warteschlange für `x` freigegeben.

Mit `t.interrupt ()` wird **genau** der Thread `t` aus einer `wait`-Warteschlange freigegeben.

`x.notify ()` kann nur in einer bezüglich `x synchronized` Umgebung aufgerufen werden. Abgesehen davon, dass man einen Monitor-Kontext braucht, ist `notify` performanter als `interrupt`, da keine Exception geworfen wird.

`notify` "verpufft" wirkungslos, wenn sich *kein* Thread in der `wait`-Warteschlange befindet.

`t.interrupt` weckt einen Thread `t` auch aus einem `sleep` oder `join`.

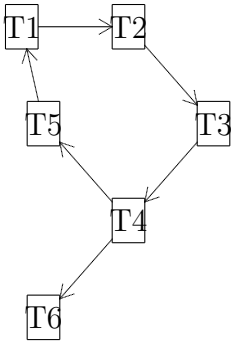
Bei `t.interrupt` wird eine `InterruptedException` geworfen.

`interrupt` setzt ein Interrupted-Flag, das dann bei Anlaufen eines `wait`, `sleep` oder `join` die `InterruptedException` auslöst, wobei dann das Flag zurückgesetzt wird. Ein `interrupt` "verpufft" also nicht. Mit `Thread.interrupted ()` kann das Flag vor dem Anlaufen eines `wait`, `sleep` oder `join` zurückgesetzt werden, um das Werfen einer Exception zu verhindern.

21.4 Verklemmungen

Vermeidung von **Verklemmungen**: Verklemmungen entstehen, wenn von verschiedenen Threads mehrere Betriebsmittel unter gegenseitigem Ausschluss angefordert werden. Solche Verklemmungen können detektiert werden, indem man einen **Vorranggraphen** erstellt, der aus Symbolen für Threads und Pfeilen besteht. Wenn ein Thread T_2 auf die Freigabe eines Betriebsmittels wartet, das ein anderer Thread T_1 benutzt, dann wird ein Pfeil $T_1 \longrightarrow T_2$ gezogen. Kommt es dabei zu einem Zyklus, dann liegt eine Verklemmung vor.

Achtung: Oft wird die Pfeilrichtung auch andersherum verwendet. Wie sie hier verwendet wird, erscheint uns am vernünftigsten und ist auch kompatibel mit anderen, ähnlichen Diagrammen, z.B. UML-Aktivitäts-Diagrammen.



Verklemmungen können folgendermaßen vermieden werden:

1. **Voranforderungsstrategie:** Ein Thread fordert alle Betriebsmittel, die er gerade benötigt, auf einen Schlag an.

   Oder: Solange ein Thread ein Betriebsmittel hat, darf er keine weiteren Betriebsmittel anfordern. Bezüglich des Vorranggraphen bedeutet das, dass ein Thread entweder nur eingehende oder nur ausgehende Pfeile hat. Damit kann es zu keinem Zyklus kommen.

   Diese Strategie vermindert – eventuell drastisch – die Nebenläufigkeit.

2. **Anforderung nach einer vorgegebenen Reihenfolge:** Mehrere Betriebsmittel dürfen nur in einer vorgegebenen Reihenfolge angefordert werden.

   Jedes Betriebsmittel könnte eine Nummer haben, nach der die Betriebsmittel – etwa aufsteigend – geordnet sind. Wenn ein Thread ein Betriebsmittel hat, dann kann er nur Betriebsmittel mit höherer Nummer anfordern. Das vermeidet Verklemmungen. Denn nehmen wir an, es gäbe im Vorranggraphen einen Zyklus. Wenn wir entlang des Zyklus gehen, dann muss die Nummer des jeweils angeforderten Betriebsmittels steigen. Damit ist aber kein Zyklus möglich.

   Das kann man i.A. nur über einen Betriebsmittel-Manager implementieren, der weiß, in welcher Reihenfolge die Betriebsmittel angefordert werden dürfen.

3. **Anforderung von Betriebsmitteln mit Bedarfsanalyse:** Jeder Thread gibt einem Betriebsmittel-Manager von vornherein bekannt, welche und wieviele Betriebsmittel er eventuell gleichzeitig benötigt. Der Manager teilt die Betriebsmittel nur dann zu, wenn dabei der **Zustand der Betriebsmittelbelegungen (Belegungszustand) sicher** bleibt. "Sicher" heißt, es kann zu keiner Verklemmung kommen. Oder genauer: Es gibt eine Folge von Belegungszuständen, so dass jeder Thread seine Restforderungen erfüllt bekommt, damit er zu Ende laufen kann und alle belegten Betriebsmittel schließlich freigibt.

   Wegen Einzelheiten des sogenannten **(Bankier-Algorithmus)** sei auf die Literatur [23] verwiesen.

Bei Datenbanksystemen nimmt man Verklemmungen von Transaktionen durchaus in Kauf, um möglichst hohe Nebenläufigkeit zu erreichen. Im Verklemmungsfall wird eben eine Transaktion zurückgesetzt (Rollback). Diese kann dann zu einem späteren Zeitpunkt wiederholt werden.

Reale Prozesse können aber oft nicht rückgängig gemacht werden. Man kann die Vermischung von zwei Flüssigkeiten nicht rückgängig machen. Zwei Züge, die bei der Reservierung von Gleisabschnitten in eine Deadlock-Situation geraten sind, könnte man eventuell wieder rückwärts fahren

lassen, aber das ist dann doch sehr peinlich. Daher muss man Verklemmungen bei Echtzeitsystemen unbedingt vermeiden.

21.5 Priorität

Faustregel:

Die Priorität ist umgekehrt proportional zur "Prozesszeit" zu vergeben.

D.h. ein Thread, der sehr schnell auf Input reagieren soll, sollte eine höhere Priorität haben, als ein langsam reagierender Thread.

Allgemein ist es günstig, **rechen-intensiven** Threads eine niedrigere Priorität zu geben als **I/O-intensiven** Threads.

Die Korrektheit eines Programms darf nicht von der Vergabe von Prioritäten abhängen.

Doug Lea[20] gibt folgende Empfehlungen:

Priorität	Thread-Typ
10	Krisen-Management
7 – 9	interaktiv, ereignisgesteuert
4 – 6	I/O-abhängig
2 – 3	Berechnung im Hintergrund
1	wenn wirklich nichts anderes ansteht

21.6 Komposition anstatt Vererbung/Realisierung

Das Erben der Klasse `Thread` hat zwei Nachteile. Man kann erstens nicht mehr von anderen Klassen erben, und zweitens – eventuell gravierender – stehen dem Anwender alle `Thread`-Methoden zur Verfügung. Auch wenn wir `Runnable` realisieren, besteht die Gefahr, dass der Anwender unsere Methode `run` überschreibt.

Daher mag es sinnvoll sein, den ganzen Thread-Mechanismus durch Komposition zu kapseln:

```
public class   MeinThread
   {
   private Thread task;

   MeinThread ()
      {
      task = new Thread (new MeinRunnable ());
      task.start ();
      }

   private class  MeinRunnable implements Runnable
      {
```

```
      public void run ()
         {
         // ...
         }
      }
   }
```

Oder eventuell noch kompakter:

```
public class   MeinThread
   {
   MeinThread ()
      {
      new Thread
         (
         new Runnable ()
            {
            public void run ()
               {
               // ...
               }
            }
         ).start ();
      }
   }
```

Bemerkung: Es ist **nicht** zu empfehlen, einen Thread in seinem Konstruktor zu starten, wie das hier gemacht wurde und obwohl das einen sehr eleganten Eindruck macht. (Siehe Diskussion unten.)

21.7 Vorzeitige Veröffentlichung von `this`

Das Starten eines Threads in seinem Konstruktor macht einen sehr eleganten Eindruck. Allerdings kann das zu ernsten Fehlern führen, weil das Objekt zu dem Zeitpunkt noch nicht vollständig konstruiert ist, insbesondere wenn der Thread weit oben in einer Vererbungshierarchie gestartet wird. Mit dem Thread-Start wurde der `this`-Zeiger vorzeitig veröffentlicht ***(escaped)***.

Das kann auch passieren, wenn innerhalb des Konstruktors das Objekt, etwa als Listener, irgendwo registriert wird.

Dieses Problem kann man eigentlich nur durch die Verwendung von protected Konstruktoren, publish-Methoden und öffentlichen Factory-Methoden lösen. Das sei an folgendem Beispiel für das Starten eines Threads gezeigt:

```
public class   AutoThreadStart
   {
   public static void   main (String[] arg)
      {
```

```
      BThread.newInstance ();
      }
   }
class AThread extends Thread
   {
   protected String  a;
   protected   AThread () { a = "Thread A"; }
   protected AThread publish ()
      {
      start ();   // oder registriere irgendwo
      return this;
      }
   public static AThread   newInstance ()
      {
      return new AThread ().publish ();
      }
   public void run ()
      {
      while (true)
         {
         System.out.println ("Jetzt läuft " + a);
         try { Thread.sleep (2000); } catch (InterruptedException e) {}
         }
      }
   }
class BThread extends AThread
   {
   protected   BThread () { a = "Thread B"; }
   public static AThread   newInstance ()
      {
      return new BThread ().publish ();
      }
   }
```

21.8 Zeitliche Auflösung

Einige Methoden haben als Argumente eine Zeitangabe (z.B. `sleep`, `wait`, `join`) oder geben eine Zeitangabe zurück (z.B. `System.currentTimeMillis`). Wie genau diese Zeitangabe eingehalten wird bzw. zurückgegeben wird, hängt leider vom zu Grunde liegenden Zeitsystem ab.

Die Zeitsysteme der verwendeten Betriebssysteme haben oft eine geringe Auflösung. Unter Windows liegt die Auflösung bei 10 bis 16 ms. Das bedeutet, dass `System.currentTimeMillis ()` Werte in Inkrementen von 10 bis 16 ms zurückliefert. Oder ein `Thread.sleep (5)` lässt den Thread 10 bis 16 ms schlafen.

Die Methode `System.nanoTime ()` benutzt i.A. ein genaueres Zeitsystem und liefert Nanosekunden zurück, die allerdings nur mit Werten aus anderen Aufrufen dieser Methode verglichen werden können. Daher kann diese Methode manchmal als Workaround verwendet werden.

21.9 Java Virtual Machine Profile Interface

Zeitanalysen werden oft mit Hilfe der Methoden

`System.currentTimeMillis ()`
oder besser
`System.nanoTime ()`

gemacht. Das reicht in vielen Fällen aus, ist aber eine sehr grobe Methode. Für genauere Analysen sei auf das JVMPI (Java Virtual Machine Profile Interface) und die entsprechende Literatur dazu hingewiesen.

21.10 Adhoc-Thread

Wenn innerhalb von sequentiellem Code eine Situation eintritt, in der plötzlich parallelisiert werden kann oder muss, dann kann an dieser Stelle im Code ein Thread als anonyme innere Klasse definiert, instanziiert und gestartet werden.

```
//... sequentiell
//... nun adhoc-Thread:
new Thread ()
   {
   public void run ()
      {
      //... paralleler Code
      }
   }.start ();
//... ab jetzt zwei Threads
```

Dazu folgendes Beispiel:

```
public class   CodeMitAdhocThread
   {
   public static void   main (String[] arg)
      {
      new CodeMitAdhocThread ().tuWas ();
      }
   public void tuWas ()
      {
      for (int i = 0; i < 3; i++)
         {
         System.out.println ("tuWas sequentiell (" + i + ").");
         }
      Thread   t = new Thread ()
         {
         public void run ()
            {
            for (int i = 0; i < 3; i++)
               {
               System.out.println ("                          "
                  + "paralleler adhoc-Code (" + i + ").");
               Thread.yield ();
               }
            System.out.println ("                         "
               + "paralleler adhoc-Code fertig.");
```

```
            }
         };
      t.start ();
      for (int i = 0; i < 7; i++)
         {
         if (t.isAlive ())
            System.out.println ("tuWas parallel (" + i + ").");
         else
            System.out.println ("tuWas sequentiell (" + i + ").");
         Thread.yield ();
         }
      }
   }
```

21.11 Thread-sichere API-Klassen

Die Klassen `java.util.Vector` und `java.util.Hashtable` sind Thread-sicher, d.h. gleichgültig, wieviele Threads gleichzeitig auf Objekte dieser Klassen zugreifen, ihr Zustand bleibt konsistent. Wenn der Anwender allerdings mehrere Methoden kombiniert, dann kann es zu Überraschungen kommen. Hier muss dann Client-seitiges Locking verwendet werden.

```
Vector<Object> v;
// ...
synchronized (v)  // Client-seitiges Locking
   {
   int   letzter = v.size () - 1;
   if (letzter >= 0)
      {
      v.remove (letzter);
      }
   }
// ...
```

Wenn man sowieso Client-seitig Sperren muss, dann empfiehlt sich aus Gründen der Effizienz die Verwendung der **nicht** Thread-sicheren Klassen `java.util.ArrayList` und `java.util.HashMap`.

Mit `java.util.Collections.synchronizedList (List<E> x)` kann man sich `synchronized` Varianten von allen möglichen Listen erzeugen.

Siehe auch Kapitel "Concurrent Utilities".

21.12 Übungen

21.12.1 StopAndGo

Schreiben Sie einen Thread, der von einem anderen Thread – etwa dem `main` angehalten, fortgesetzt und beendet werden kann.

(Man kann auch `Bewegung.java` nehmen und die mit "`// ???`" gekennzeichneten Methoden implementieren bzw. verändern. Möglicherweise müssen noch Datenelemente ergänzt werden. `StopAndGoGUI.java` ist dann eine Anwendung für die Klasse `Bewegung`.)

Kapitel 22

Concurrency Utilities

Die Programmierung paralleler Tasks ist schwierig und fehlerträchtig. Die Aufgabe kann wesentlich vereinfacht werden, wenn man eine Nebenläufigkeits-Bibliothek verwendet. Java bietet dafür seit J2SE 5.0 das Package `java.util.concurrent` an, mit dem wir uns in diesem Kapitel beschäftigen. Ferner werden noch einige fortgeschrittene Techniken der Parallel-Programmierung angesprochen.

22.1 Timer

Zunächst wollen wir auf eine Klasse hinweisen, die es schon in `java.util` gibt und mit der man Tasks zeitlich für einmalige oder wiederholte Ausführung einplanen kann:

```
public class   Timer
   {
   public   Timer ();
   public   Timer (boolean isDaemon);
   public   Timer (String name);
   public   Timer (String name, boolean isDaemon);
   public void cancel ();  // Beendet den Timer. Eingeplante Tasks werden ausgeplant.
   public int  purge ();   // Entfernt alle ausgeplanten TimerTasks.
   public void schedule (TimerTask task, Date time);
   public void schedule (TimerTask task, Date firstTime, long period);
   public void schedule (TimerTask task, long delay);
   public void schedule (TimerTask task, long delay, long period);
   public void scheduleAtFixedRate (TimerTask task, Date firstTime, long period);
   public void scheduleAtFixedRate (TimerTask task, long delay, long period);
   }

public abstract class   TimerTask implements Runnable
   {
   public   TimerTask ();
```

```
public boolean cancel ();  // Task wird ausgeplant.
                           // Wenn die Task gerade läuft, läuft sie zu Ende.
public abstract void run ();
public long scheduledExecutionTime ();
}
```

Für weitere Erläuterungen sei auf die Java-API-Dokumentation verwiesen.

22.2 Thread-Pools

Die Erzeugung und der Start eines Threads ist wesentlich aufwändiger als der Aufruf einer Prozedur oder Methode.

Anstatt für jede Task einen neuen Thread zu erzeugen und zu starten, kann man einen Thread aus einem Pool von Threads verwenden und ihm die Task zur Ausführung geben. Wenn die Task beendet wird, geht der Thread wieder in den Pool zurück.

Eine weitere wichtige Anwendungsmöglichkeit von Threadpools ist die Begrenzung der Anzahl der Threads, die gestartet werden können. Denn wenn die von Java oder dem zugrundeliegenden Betriebssystem gesetzte Grenze überschritten wird, kommt es zum Absturz. Typisches Beispiel ist ein multithreaded (Web-)Server, der unter hoher Last (zu viele Client-Threads) zusammenbricht.

Threadpools können dann am besten eingesetzt werden, wenn die Tasks **homogen** und **unabhängig** sind. Eine Mischung von langlaufenden und kurzen Tasks führt leicht zu einer Verstopfung des Threadpools, so dass die Ausführung kurzer Tasks unerwartet lange dauert. Bei voneinander abhängigen Tasks riskiert man ein Deadlock, indem z.B. eine Task unendlich lange im Threadpool verbleibt, weil sie auf das Ergenis einer anderen Task wartet, die aber nicht laufen kann, weil der Threadpool voll ist.

Zentral ist die Schnittstelle `Executor`:

```
public interface  Executor
   {
   void  execute (Runnable task);
   }
```

Die Klasse `ThreadPoolExecutor` ist eine Realisierung dieser Schnittstelle und bietet verschiedene Möglichkeiten für Pool-Operationen, auf die wir hier nicht eingehen.

Die einfachste Möglichkeit zu einem Threadpool zu kommen, besteht im Aufruf der `static` Methode `newFixedThreadPool (int anzThreads)` der Klasse `Executors`. Diese Methode gibt ein Objekt vom Typ `ExecutorService` zurück, der die Schnittstelle `Executor` mit der Methode `execute (Runnable)` realisiert. Die Einzelheiten werden im folgenden Beispiel gezeigt.

Zunächst eine Task, die Start- und Endezeit ausgibt und zwischendurch etwas wartet:

```
public class   Task
```

```
   implements  Runnable
   {
   private String name;
   private int wartezeit;
   public   Task (String name, int wartezeit)
      {
      this.name = name;
      this.wartezeit = wartezeit;
      }
   public void run ()
      {
      System.out.println ("Task " + name + " beginnt um "
         + System.currentTimeMillis () + ".");
      try
         {
         Thread.sleep (wartezeit);
         }
         catch (InterruptedException e ) {}
      System.out.println ("Task " + name + " ist fertig um "
         + System.currentTimeMillis () + ".");
      }
   }
```

Um diese Task auszuführen, benützen wir einen Thread-Pool. In dem folgenden Programm erzeugen wir einen Pool der Größe 3 und lassen 10 Tasks durchführen. Schließlich warten wir, bis alle Tasks zu Ende gekommen sind.

```
import   java.util.concurrent.*;
public class   ExecuteTask
   {
   public static void   main (String[] arg)
      {
      ExecutorService executor
         =  Executors.newFixedThreadPool (3);
         // Wir erzeugen einen Thread-Pool mit 3 Threads.
      int   sumWartezeit = 1000;
         // Wir summieren alle Wartezeiten, damit wir
         // wissen, wann wir den executor schließen können.
      java.util.Random  zufall = new java.util.Random ();
      for (int i = 0; i < 10; i++)
         {
         String   name = "" + i;
         int   wartezeit = zufall.nextInt (1000);
         sumWartezeit = sumWartezeit + wartezeit;
         Runnable task = new Task (name, wartezeit);
            // Tasks werden erzeugt
         System.out.println ("Task " + name +
            " mit Wartezeit " + wartezeit
            + " wird an den Threadpool übergeben.");
         executor.execute (task);
            // Tasks werden dem Executor übergeben.
         }
      try
         {
         Thread.sleep (sumWartezeit);
         executor.shutdown ();
         executor.awaitTermination (sumWartezeit,
            TimeUnit.MILLISECONDS);
         }
         catch (InterruptedException e) {}
      }
   }
```

Die Ausgabe dieses Programms ist:

```
Task 0 mit Wartezeit 406 wird hinzugefügt.
Task 1 mit Wartezeit 935 wird hinzugefügt.
Task 2 mit Wartezeit 247 wird hinzugefügt.
Task 3 mit Wartezeit 993 wird hinzugefügt.
Task 4 mit Wartezeit 402 wird hinzugefügt.
Task 5 mit Wartezeit 839 wird hinzugefügt.
Task 6 mit Wartezeit 14 wird hinzugefügt.
Task 7 mit Wartezeit 131 wird hinzugefügt.
Task 8 mit Wartezeit 613 wird hinzugefügt.
Task 9 mit Wartezeit 229 wird hinzugefügt.
Task 0 beginnt um 1103561602040.
Task 1 beginnt um 1103561602040.
Task 2 beginnt um 1103561602040.
Task 2 ist fertig um 1103561602280.
Task 3 beginnt um 1103561602280.
Task 0 ist fertig um 1103561602441.
Task 4 beginnt um 1103561602441.
Task 4 ist fertig um 1103561602841.
Task 5 beginnt um 1103561602841.
Task 1 ist fertig um 1103561602971.
Task 6 beginnt um 1103561602971.
Task 6 ist fertig um 1103561602981.
Task 7 beginnt um 1103561602981.
Task 7 ist fertig um 1103561603122.
Task 8 beginnt um 1103561603122.
Task 3 ist fertig um 1103561603272.
Task 9 beginnt um 1103561603272.
Task 9 ist fertig um 1103561603502.
Task 5 ist fertig um 1103561603682.
Task 8 ist fertig um 1103561603732.
```

Dabei ist zu bemerken, dass die ersten drei Tasks sehr schnell beginnen, während weitere Tasks warten müssen, bis wieder ein Thread zur Verfügung steht.

Da die JVM solange läuft, bis alle Tasks zu Ende gelaufen sind, ein Threadpool aber i.A. immer irgendwelche Tasks am laufen hat, bietet `ExecuteService` Methoden an, mit denen ein Threadpool ordentlich abgeschaltet werden kann. Nach einem `shutdown ()` wird der Threadpool nicht verwendete Threads abschalten und keine neuen Threads zum Laufen bringen. Allerdings muss er warten, bis laufende Threads zu Ende gekommen sind. Das kann eine gewisse Zeit dauern. Daher gibt es die Methode

```
awaitTermination (long timeout, TimeUnit timeunit),
```

um auf die Beendingung zu warten.

Das Threadpool-Framework bietet noch wesentlich mehr. Dazu sei auf die Dokumentation zu J2SE 5.0 verwiesen.

22.3 Semaphore

Das Package `java.util.concurrent` bietet eine Klasse `Semaphore` an, die einen **allgemeinen** oder ***counting*** Semaphor repräsentiert. Allgemeine Semaphore werden verwendet, um den Zugang zu einem Betriebsmittel zahlenmäßig zu begrenzen oder um eine Sende-Warteauf-Ereignis-Situation zu realisieren.

Die Klasse `Semaphore` hat zwei Konstruktoren:

```
Semaphore (int permits)
Semaphore (int permits, boolean fair)
```

Eine FIFO-Strategie für die Warteschlange wird nur garantiert, wenn `fair` auf `true` gesetzt wird. Um eine Aussperrung zu verhindern, sollte man i.A. `fair` auf `true` setzen.

Mit den Methoden

```
void  acquire ()
void  acquire (int permits)
```

kann ein Thread ein oder mehrere Permits beantragen. Wenn die Permits nicht sofort erhältlich sind, wartet der Thread, bis er die Permits erhält oder durch `interrupt` unterbrochen wird.

Bei den nächsten beiden Methoden kann der Thread nicht unterbrochen werden. Allerdings kann nach Erhalt der Permits festgestellt werden, ob ein Unterbrechungsversuch stattgefunden hat.

```
void  acquireUninterruptibly ()
void  acquireUninterruptibly (int permits)
```

Die folgenden Methoden blockieren nicht oder höchstens eine gewisse Zeit:

```
boolean  tryAcquire ()
boolean  tryAcquire (int permits)
boolean  tryAcquire (long timeout, TimeUnit unit)
boolean  tryAcquire (int permits, long timeout, TimeUnit unit)
```

Die Permits werden mit der Methode

```
void release ()
```

wieder freigegeben.

Die Permits entsprechen der Semaphor-Variablen eines allgemeinen oder counting Semaphors und können daher beliebige Werte, also auch Werte über dem Initialisierungswert annehmen. Insbesondere können die Semaphore auch mit 0 oder negativen Werten für die Permits initialisiert werden. In dem Fall sind dann ein oder mehrere `release`-Operationen vor dem ersten erfolgreichen `acquire` nötig.

Beispiel: Wir haben drei Postschalter, bei denen die Abfertigung ein bis zwei Sekunden dauert. Die Kunden warten höchstens zwei Sekunden, bis sie dran kommen, dann geben sie auf.

```
import   java.util.concurrent.*;
import   java.util.*;
public class   Postschalter
   {
   private static final int ANZ_SCHALTER = 3;
   private static final Semaphore   sema
      = new Semaphore (ANZ_SCHALTER, false);
      //= new Semaphore (ANZ_SCHALTER, true);   // Das funktioniert nicht!!
   private static final int ANZ_KUNDEN = 20;
   private static final int KUNDEN_INTERVALL = 300;   // Millisekunden
   private static final int MAX_WARTEZEIT = 2000;  // Millisekunden
```

```
private static final int MIN_ABFERTIGUNGSZEIT = 1000; // Millisekunden
private static final int MAX_ABFERTIGUNGSZEIT = 2000; // Millisekunden
private static final Random   zufall = new Random ();
static class   KundIn extends Thread
   {
   int   kdnr;
   public   KundIn (int kdnr)
      {
      this.kdnr = kdnr;
      }
   public void run ()
      {
      System.out.println ("Kunde " + kdnr + " wartet.");
      long  w = System.currentTimeMillis ();
      try
         {
         if (sema.tryAcquire (MAX_WARTEZEIT, TimeUnit.MILLISECONDS))
            {
            System.out.println ("   Kunde " + kdnr);
            int   z = MIN_ABFERTIGUNGSZEIT
               + zufall.nextInt (
                  MAX_ABFERTIGUNGSZEIT - MIN_ABFERTIGUNGSZEIT);
            System.out.println ("   Kunde " + kdnr);
            try { sleep (z); }
               catch (InterruptedException e) { e.printStackTrace (); }
            w = System.currentTimeMillis () - w;
            System.out.println ("Kunde " + kdnr + " wurde in " + z +
               " Millisekunden abgefertigt"
               + " nach insgesamt " + w + " Millisekunden.");
            }
         else
            {
            w = System.currentTimeMillis () - w;
            System.out.println ("Kunde " + kdnr
               + " hat aufgegeben nach " + w + " Millisekunden.");
            }
         }
         catch (InterruptedException e) { e.printStackTrace (); }
         finally
            {
            sema.release ();
            }
      }  // end run
   }
public static void   main (String[] arg)
   {
   ArrayList<KundIn> ka = new ArrayList<KundIn> ();
   for (int i = 0; i < ANZ_KUNDEN; i++)
      {
      KundIn   k = new KundIn (i);
      ka.add (k);
      k.start ();
      try { Thread.sleep (KUNDEN_INTERVALL); }
         catch (InterruptedException e) { e.printStackTrace (); }
      }
   while (true)
      {
      try
         {
         Thread.sleep (5000);
         }
         catch (InterruptedException e) { e.printStackTrace (); }
      System.out.print ("Die Kunden");
      for (KundIn k :ka)
         {
         if (k.isAlive ()) System.out.print ( " " + k.kdnr);
         }
      System.out.println (" sind immer noch aktiv!");
      }
```

```
    }
  }
```

22.4 Locks

Der `synchronized` Block ist die elementarste Methode, ein Lock für ein Codestück zu beantragen. `Lock`-Klassen im Paket

```
java.util.concurrent.locks
```

bieten darüber hinaus Timeouts, mehrwertige Bedingungsvariable, Read-Write-Locks und die Möglichkeit, das Warten auf ein Lock abzubrechen.

Das Benutzungmuster sieht folgendermaßen aus:

```
Lock  s = ...; // Siehe später.
s.lock ();     // Das Lock wird beantragt.
try
   {
   // Verwendung des geschützten Betriebsmittels
   }
   finally
      {
      s.unlock ();
      }
```

Da ein Lock auf jeden Fall wieder freigegeben werden muss, ist der `finally`-Block notwendig.

`s.lock ()` lässt den Thread warten, bis er das beantragte Lock bekommt. Der Thread kann dabei nicht unterbrochen werden. Dies aber erlaubt `s.lockInterruptibly`:

```
Lock  s = ...;
try
   {
   s.lockInterruptibly ();
   try
      {
      // Verwendung des geschützten Betriebsmittels
      }
      finally
         {
         s.unlock ();
         }
   }
   catch (InterruptedException e)
      {
```

```
      System.err.println ("Thread wurde beim Warten auf ein Lock unterbrochen.");
      }
```

Wenn für den Thread `interrupt ()` aufgerufen wird, dann wird das `s.lockInterruptibly ()` abgebrochen und eine `InterruptedException` geworfen. Das wird i.A. so programmiert, dass bei einem Interrupt der geschützte Code nicht durchlaufen wird. Da der Thread kein Lock bekommen hat, darf das Lock auch nicht freigegeben werden. Daher wird der geschachtelte `try-try-finally-catch`-Konstrukt benötigt.

Mit `tryLock (...) :boolean` kann man Timeouts setzen.

```
Lock  s = ...; // Siehe später.
if (s.tryLock (200, TimeUnit.MILLISECONDS))
   {
   try
      {
      // Verwendung des geschützten Betriebsmittels
      }
      finally
         {
         s.unlock ();
         }
   }
else
   {
   // Tu was anderes ...
   }
```

`tryLock ()` ohne Argumente kommt sofort mit `false` zurück, wenn das Lock nicht erhalten wird, mit Argumenten erst nach der angegebenen Zeit.

Das Interface `Lock` sieht folgendermaßen aus:

```
public interface  Lock
   {
   void  lock ();               // Beantragt das Lock. Blockiert.
                                // Erhöht den hold count.

   void  lockInterruptibly ();  // Beantragt das Lock. Kann durch interrupt ()
                                // abgebrochen werden. Blockiert.
                                // Erhöht den hold count, wenn das Lock gegeben
                                // wird.

   Condition   newCondition ();  // Siehe Bemerkung unten.

   boolean  tryLock ();         // true: Wenn das Lock frei ist, sonst false.
                                // Blockiert nicht.
```

```
boolean  tryLock (long time, TimeUnit unit);
                              // true: Wenn das Lock innerhalb time frei ist,
                              // sonst false.
                              // Blockiert höchstens time.

void  unlock ();              // Erniedrigt hold count. Gibt das Lock bei
                              // hold count == 0 frei.
}
```

Ein Lock bezieht sich immer auf einen Thread, d.h. ein Thread besitzt das Lock oder eben nicht. Wenn ein Thread ein Lock hat, dann bekommt er dasselbe Lock sofort wieder, wenn er es noch einmal beantragt. Dabei wird der *hold count* um eins erhöht. Bei `unlock ()` wird der *hold count* um eins erniedrigt. Wenn Null erreicht wird, wird das Lock freigegeben.

Das Interface `Lock` wird durch folgende Klassen realisiert:

```
ReentrantLock
ReentrantReadWriteLock.ReadLock
ReentrantReadWriteLock.WriteLock
```

Üblicherweise wird für den gegenseitigen Ausschluss die Klasse `ReentrantLock` verwendet. Objekte der anderen beiden Klassen sind über die Klasse `ReentranReadWriteLock` zugänglich und werden zur Lösung des Reader-Writer-Problems verwendet (ähnlich einer Bolt-Variablen):

```
ReadWriteLock  rwl = new ReentrantReadWriteLock ();
Lock  readLock = rwl.readLock ();
Lock  writeLock = rwl.writeLock ();
```

Die Leser müssen dann das `readLock` wie ein `ReentrantLock` zum Schützen des Lesebereichs verwenden. Die Schreiber verwenden das `writeLock` zum Schützen des Schreibbereichs. Das `writeLock` ist ein **exklusives** Lock, während das `readLock` ein **shared** Lock ist, deren Verhalten wir mit folgender Kompatibilitäts-Matrix darstellen:

	`readLock`	`writeLock`
`readLock`	ja	nein
`writeLock`	nein	nein

D.h.: Wenn ein Thread ein `readLock` hat, dann kann ein zweiter (oder n-ter) Thread auch ein `readLock` bekommen, aber kein `writeLock`.

Offenbar sind die Klassen so programmiert, dass die Schreiber Vorrang haben. D.h., wenn ein Schreiber schreiben möchte, dann wird kein weiterer Leser zugelassen.

Die Methode `newCondition ()` wird von `ReadLock` und `WriteLock` *nicht* unterstützt.

Bemerkungen:

1. `newCondition` gibt ein Bedingungsobjekt vom Typ `Condition` zurück, das an das Lock gebunden ist und das die Methode `await ()` anbietet. Der Mechanismus ist mit dem `wait` vergleichbar: `await ()` kann nur aufgerufen werden, wenn man das zugehörige Lock hat. Wenn `await ()` aufgerufen wird, wird der Thread in eine Warteschlange bezüglich des verwendeten Bedingungsobjekts befördert. Dabei wird das Lock freigegeben. Wenn für das Bedingungsobjekt `signal ()` oder `signalAll ()` aufgerufen wird, dann wird ein bzw. werden alle Threads aus der Warteschlange entfernt. Diese Threads müssen sich dann wieder um das Lock bemühen.

   Ein Lock kann mehrere unterschiedliche `Condition`-Objekte liefern. Damit kann man unterschiedliche Wait-Sets verwalten.

   Für das `await` gibt es Varianten mit Timeout oder eine Variante, die nicht unterbrechbar ist.

22.5 Barrieren

Eine Barriere ***(barrier)*** ist ein Synchronisationspunkt ***(barrier point)***, an dem mehrere Threads aufeinander warten, bevor sie mit der Ausführung ihres Codes weitermachen. Die Klasse `CyclicBarrier` bietet Methoden an, solch einen Treffpunkt zu definieren. Die Barriere ist zyklisch, weil sie wiederverwendet werden kann, nachdem sich die Threads getroffen haben.

Mit den beiden Konstruktoren

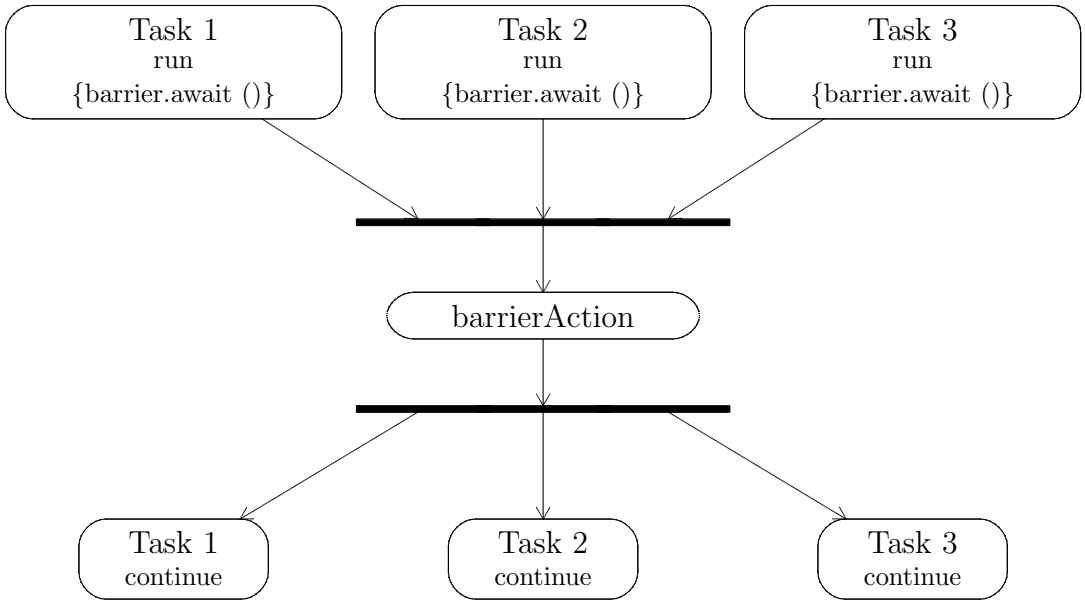
```
CyclicBarrier (int parties)
CyclicBarrier (int parties, Runnable barrierAction)
```

kann man die Anzahl von Teilnehmern am Treffpunkt definieren. Außerdem ist es möglich, eine Aktion als `Runnable` anzugeben, die nach dem Eintreffen der Teilnehmer gestartet wird und zu Ende läuft, bevor die Teilnehmer wieder freigegeben werden.

Die Teilnehmer müssen am Synchronisationspunkt für die Barriere

```
void  await ()
void  await (long timeout, TimeUnit unit)
```

aufrufen. Dabei wird die Exception `BrokenBarrierException` geworfen, wenn ein Teilnehmer den Treffpunkt vorzeitig (etwa durch einen Interrupt) verlässt.



Übung: Darstellung der Barriere in der Ablaufdiagramm-Notation.

Beispiel: Wir haben drei Threads, die je eine Roboterachse steuern. Bei jedem Positionspunkt sollen die Threads aufeinander warten, da der Mainthread einen neuen Positionspunkt bestimmen muss.

```
import   java.util.concurrent.*;
import   java.util.*;
public class   RobAchsen
   {
   private static final int   ANZ_ACHSEN = 3;
   private static final int   FAHRZEIT = 1000;
   private static final Random   zufall = new Random ();
   //private static CyclicBarrier   barrier;
   //private static MyCyclicBarrier barrier;
   private static MyCyclicBarrier2  barrier;
   private static ArrayList<Achse>  achse = new ArrayList<Achse> ();
   private static class Achse extends Thread
      {
      int   achsNr;
      int   posNr;
      String   s = "";
      public   Achse (int achsNr)
         {
         this.achsNr = achsNr;
         for (int i = 0; i < achsNr; i++) s = s + "\t";
         }
      public void run ()
         {
         while (true)
            {
            System.out.println (s + "Achse " + achsNr
               + " fährt nach Pos " + posNr + ".");
            vertueEtwasZeit ();
            System.out.println (s + "Achse " + achsNr
               + " hat Pos " + posNr + " erreicht und wartet an Barriere.");
```

```
            try
               {
               barrier.await ();
               }
               catch (InterruptedException e) { e.printStackTrace (); }
               catch (BrokenBarrierException e ) { e.printStackTrace (); }
            vertueEtwasZeit ();
            System.out.println (s + "Achse " + achsNr
               + " nimmt neue Pos " + posNr + " auf.");
            vertueEtwasZeit ();
            }
         }
      }
   private static class Position
      implements  Runnable
      {
      int   pos = 0;
      public void run ()
         {
         pos = pos + 1;
         System.out.println ("Barrier-Action-" + Thread.currentThread ()
            + " Position setzt neue Positionsnummer " + pos + ".");
         for (Achse a : achse)
            {
            a.posNr = pos;
            }
         }
      }
   public static void   main (String[] arg)
      {
      for (int i = 0; i < ANZ_ACHSEN; i++)
         {
         Achse a = new Achse (i + 1);
         achse.add (a);
         }
      Position position = new Position ();
      //barrier = new CyclicBarrier (ANZ_ACHSEN, position);
      //barrier = new MyCyclicBarrier (ANZ_ACHSEN, position);
      barrier = new MyCyclicBarrier2 (ANZ_ACHSEN, position);
      position.run ();  // Nur um die erste Position zu setzen.
      for (Achse a :achse) a.start ();
      }
   public static void   vertueEtwasZeit ()
      {
      try
         {
         Thread.sleep (zufall.nextInt (FAHRZEIT));
         }
         catch (InterruptedException e) { e.printStackTrace (); }
      }
   }
```

22.6 Latches

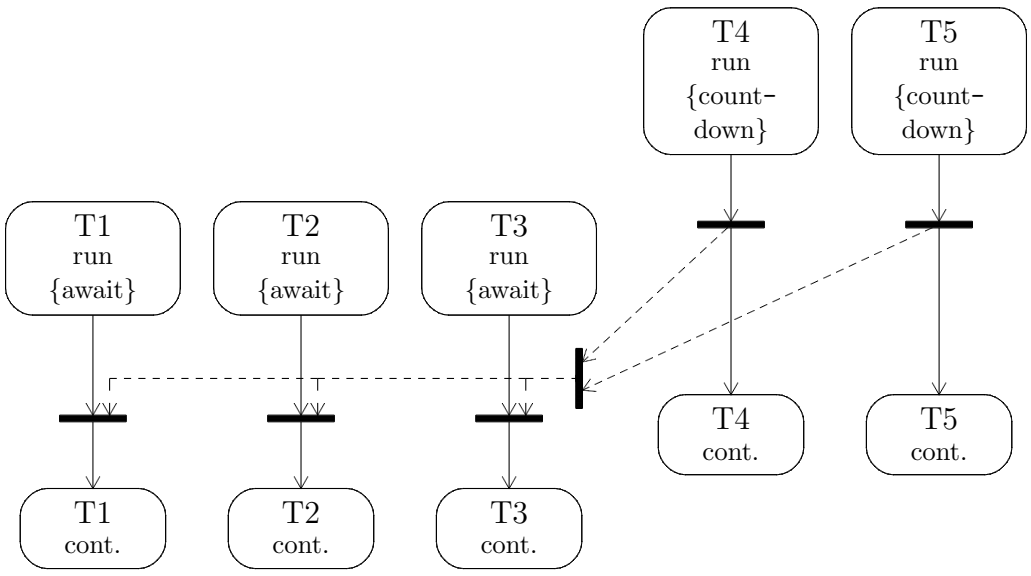
Die Klasse `CountDownLatch` hat eine ähnliche Funktion wie `CyclicBarrier` – nämlich die Synchronisation mehrerer Tasks – mit dem Unterschied, dass die wartenden Tasks auf ein Signal warten, dass durch andere Tasks gegeben wird, indem diese einen Zähler herunterzählen. Wenn der Zähler Null erreicht, dann wird eben dieses Signal generiert, dass die wartenden Tasks freigibt.

Die Klasse `CountDownLatch` kann mit einer ganzen Zahl ***(count)*** (anfänglicher Zählerstand) initialisiert werden. Konstruktor und Operationen:

```
CountDownLatch (int count)

void  await ()
void  await (long timeout, TimeUnit unit)

void  countDown ()
int   getCount ()
```

Die Anwendung ist typisch für solche Situationen, in denen eine oder mehrere Tasks warten (`latch.await ()`) müssen, bis eine oder mehrere Operationen – typischerweise in anderen Tasks – erledigt (`latch.countDown ()`) sind.

Wenn Null einmal errreicht ist, bleiben weitere `await`-Aufrufe wirkungslos. Einen `CountDownLatch` kann man nur einmal verwenden *(one-shot-device)*.



Beispiel: Wir haben fünf Arbeiter, die anfangen zu arbeiten, wenn die Werkzeuge bereitliegen, wenn der Chef seinen Kaffee getrunken hat, und wenn es 8:15 Uhr geschlagen hat. Einen zweiten Latch verwenden wir für das Ende des Arbeitstages: Der Chef kann erst nach Hause gehen, wenn jeder der fünf Arbeiter seine Arbeit erledigt hat.

```
import   java.util.*;
import   java.util.concurrent.*;
public class   ArbeiterChef
   {
   public static CountDownLatch  startSignal;
   public static CountDownLatch  endeSignal;
   //public static MyCountDownLatch startSignal;
```

```
   //public static MyCountDownLatch endeSignal;
   public static void   main (String[] arg)
      {
      int   anzArbeiter = 5;
      Arbeiter[]  arbeiter = new Arbeiter[anzArbeiter];
      startSignal = new CountDownLatch (3);
      endeSignal = new CountDownLatch (anzArbeiter);
      //startSignal = new MyCountDownLatch (3);
      //endeSignal = new MyCountDownLatch (anzArbeiter);
      for (int i = 0; i < anzArbeiter; i++)
         {
         arbeiter[i] = new Arbeiter (i + 1);
         }
      Chef  chef = new Chef ();
      Werkzeuge   werkzeuge = new Werkzeuge ();
      Uhr   uhr = new Uhr ();
      }  // end main
   static Random  random = new Random ();
   static void tue (int maxzeit, String was)
      {
      System.out.println (was);
      try
         {
         Thread.sleep (random.nextInt (maxzeit));
         }
         catch (InterruptedException e) { e.printStackTrace (); }
      }
   }  // end class ArbeiterChef
class Arbeiter extends Thread
   {
   int   nummer;
   public   Arbeiter (int nummer)
      {
      this.nummer = nummer;
      start ();
      }
   public void run ()
      {
      ArbeiterChef.tue (100, "Arbeiter " + nummer
         + " wartet auf Beginn der Arbeit.");
      try
         {
         ArbeiterChef.startSignal.await ();
         }
         catch (InterruptedException e) { e.printStackTrace (); }
      ArbeiterChef.tue (1000, "Arbeiter " + nummer + " tut seine Arbeit.");
      ArbeiterChef.tue (1, "Arbeiter " + nummer + " ist fertig.");
      ArbeiterChef.endeSignal.countDown ();
      }
   }  // end class Arbeiter
class Chef extends Thread
   {
   public   Chef ()
      {
      start ();
      }
   public void run ()
      {
      ArbeiterChef.tue (1000, "Chef trinkt seinen Kaffee.");
      ArbeiterChef.tue (1, "Chef ist fertig mit Kaffeetrinken.");
      ArbeiterChef.startSignal.countDown ();
      ArbeiterChef.tue (1, "Chef wartet verantwortungsvoll "
         + "bis die Arbeiter fertig sind.");
      try
         {
         ArbeiterChef.endeSignal.await ();
         }
```

```
         catch (InterruptedException e) { e.printStackTrace (); }
      ArbeiterChef.tue (1, "Chef geht nun auch nach Hause.");
      }
   }  // end class Chef
class Werkzeuge extends Thread
   {
   public   Werkzeuge ()
      {
      start ();
      }
   public void run ()
      {
      ArbeiterChef.tue (1000, "Werkzeuge werden bereitgelegt.");
      ArbeiterChef.tue (1, "Werkzeuge sind bereit.");
      ArbeiterChef.startSignal.countDown ();
      }
   }  // end class Werkzeuge
class Uhr extends Thread
   {
   public   Uhr ()
      {
      start ();
      }
   public void run ()
      {
      ArbeiterChef.tue (1000, "Uhr wartet bis es 8:15 Uhr ist.");
      ArbeiterChef.tue (1, "Es ist 8:15 Uhr!.");
      ArbeiterChef.startSignal.countDown ();
      }
   }  // end class Uhr
```

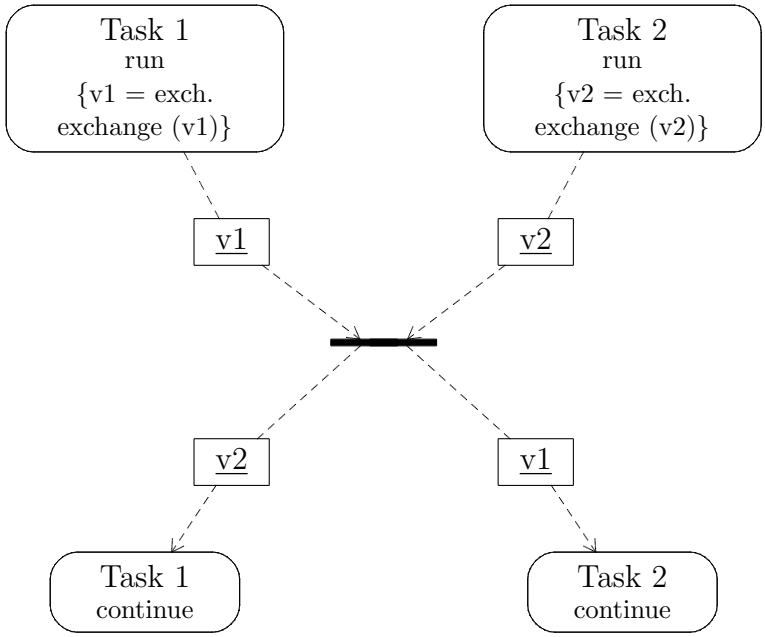
22.7 Austauscher

Mit der Klasse `Exchanger<V>` können Datenstrukturen `V` zwischen zwei Tasks ausgetauscht werden. Jede Task ruft einfach nur die Methode `exchange` auf mit dem auszutauschenden Objekt als Argument. Zurück bekommt sie das getauschte Objekt.

Die beiden Tasks **warten aufeinander** am `exchange`.

```
Exchanger<V> exch = new Exchanger<V> ();
```

Task 1	Task 2
`// ...`	`// ...`
`// ...`	`V v2 = ...`
`V v1 = ...`	`//  ...`
`v1 = exch.exchange (v1);`	`v2 = exch.exchange (v2);`
`// v1 ist nun das v2`	`// v2 ist nun das v1`
`//  von rechts.`	`//  von links.`



Beispiel: Ein Drucker leert eine Tonerkassette, die dann mit einer vollen Kassette ausgetauscht wird, die die Wartung bereitstellt. Die Wartung füllt die leere Kassette wieder.

```
import   java.util.*;
import   java.util.concurrent.*;
public class   Toner
   {
   public static void   main (String[] arg)
      {
      Exchanger<Tonerkassette>   exch = new Exchanger<Tonerkassette> ();
      Drucker  drucker = new Drucker (exch);
      Wartung  wartung = new Wartung (exch);
      }  // end main
   static Random  random = new Random ();
   static void tue (int maxzeit, String was)
      {
      System.out.println (was);
      try
         {
         Thread.sleep (random.nextInt (maxzeit));
         }
         catch (InterruptedException e) { e.printStackTrace (); }
      }
   }  // end class Toner
class Drucker extends Thread
   {
   Exchanger<Tonerkassette>   exch;
   public   Drucker (Exchanger<Tonerkassette> exch)
      {
      this.exch = exch;
      start ();
      }
   public void run ()
```

```
      {
      Tonerkassette  t = new Tonerkassette ("Nr. 1");
      while (true)
         {
         Toner.tue (1, "Drucker will Tonerkassette "
            + t.name + " tauschen.");
         try
            {
            t = exch.exchange (t);
            }
            catch (InterruptedException e) { e.printStackTrace (); }
         Toner.tue (3000, "Drucker hat neue Tonerkassette "
            + t.name + " und druckt.");
         }
      }
   }  // end class Drucker
class Wartung extends Thread
   {
   Exchanger<Tonerkassette>   exch;
   public   Wartung (Exchanger<Tonerkassette> exch)
      {
      this.exch = exch;
      start ();
      }
   public void run ()
      {
      Tonerkassette  t = new Tonerkassette ("Nr. 2");
      while (true)
         {
         t.füllen ();
         Toner.tue (1, "Wartung bietet Tonerkassette "
            + t.name + " zum Tausch an.");
         try
            {
            t = exch.exchange (t);
            }
            catch (InterruptedException e) { e.printStackTrace (); }
         }
      }
   }  // end class Wartung

class Tonerkassette
   {
   String   name;
   public   Tonerkassette (String name)
      {
      this.name = name;
      }
   public void füllen ()
      {
      Toner.tue (3000, "Tonerkassette " + name + " wird gefüllt.");
      }
   }  // end class Tonerkassette
```

Übung: Exchanger

Entwickeln Sie eine Implementierung der Klasse `Exchanger`, indem Sie die Methode `exchange` implementieren. (Falls Sie mit Generics wenig Erfahrung haben, lassen Sie sich nicht stören: `V` ist einfach der Typ der Objekte, die ausgetauscht werden können. Sie können den Typ `V` in der Klasse `Exchanger` einfach verwenden.)

Für die Übung heißt die Klasse `AufgabeAustauscher` und realisiert die Schnittstelle `Austauscher`, die die Methode `exchange` deklariert.

```
public class   AufgabeAustauscher<V> implements Austauscher {
   public         V  exchange (V v) throws InterruptedException {
      // ... Ihr Code
      return v;
   }
}  // end class
```

Der Code befindet sich im Verzeichnis `code`. Das Testprogramm wird mit der Klasse `Anwendung` gestartet und läuft eventuell mit dem Exchanger der Java-API (`JavaExchangerAustauscher`). Dort müssen Sie dann eventuell Ihren Austauscher

```
= new AufgabeAustauscher<Bierkasten> ();
```

einkommentieren.

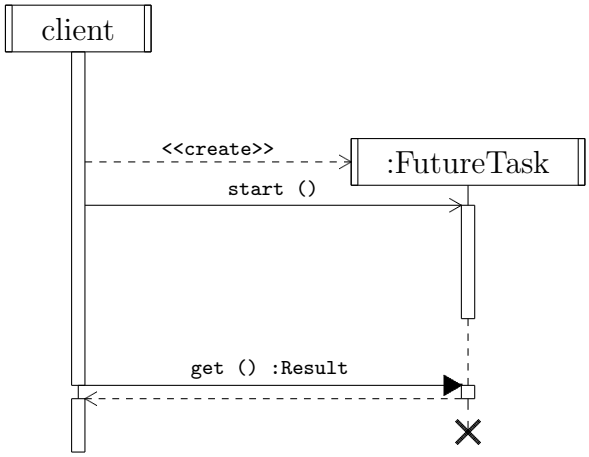
22.8 *Future*

Wenn das Ergebnis einer lang dauernden Aktivität, z.B. einer längeren Berechnung, erst zu einem späteren Zeitpunkt benötigt wird, dann kann es sinnvoll sein, diese Aktivität möglichst frühzeitig anzustoßen und das Ergebnis später, wenn es schließlich benötigt wird, abzuholen. Die Klasse `FutureTask` erlaubt eine einfache Lösung dieses Problems.

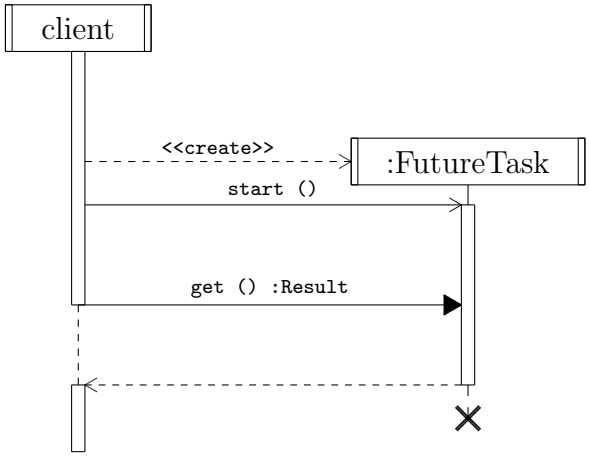
Der Kontruktor von `FutureTask<Resultat>` bekommt als Argument ein `Callable`-Objekt mit, dessen `call`-Methode so implementiert wird, dass die lang dauernde Aktivität aufgerufen wird und dass deren Resultat zurückgegeben wird. `Resultat` ist eine Klasse des Anwenders, deren Objekte das Ergebnis der Aktivität repräsentieren.

Die Objekte von `FutureTask` sind `Runnable`s und können als Thread gestartet werden. Der Mechanismus ist so, dass dabei `call ()` aufgerufen wird, also die lang dauernde Aktivität gestartet wird.

Das Ergebnis der Aktivität wird schließlich mit `get ()` des `FutureTask`-Objekts geholt.



Dieses `get ()` blockiert, falls die Aktivität noch nicht zu Ende gelaufen ist.



Folgendes kommentierte Beispiel dürfte die Einzelheiten des Mechanismus hinreichend erläutern:

```
import   java.util.concurrent.*;
public class   Vorausberechnung
   {
   public static class  Resultat
      // Objekte dieser Klasse repräsentieren das Resultat
      // einer langandauernden Berechnung.
      {
      double[] ergebnis = new double[10];
      }
   public Resultat   berechne ()
      // Dies ist die lange dauernde Aktivität.
      {
      Resultat r = new Resultat ();
      double   x;
```

```
   for (int i = 0; i < 10; i++)
      {
      r.ergebnis[i] = Math.random ();
      System.out.println ("                    Berechnung: " + i);
      try { Thread.sleep (100); }
         catch (InterruptedException e)   { e.printStackTrace (); }
      }
   return r;
   }

// Nun geben wir diese Aktivität (Berechnung)
// als ein Callable einem FuturTask-Objekt future:
public FutureTask<Resultat> future
   = new FutureTask<Resultat>
      (
      new Callable<Resultat> ()
         {
         public Resultat   call ()
            {
            return berechne ();
            }
         }
      );
public static void   main (String[] arg)
   {
   // Jetzt wird das Ganze angewendet:
   Vorausberechnung  vb = new Vorausberechnung ();
   System.out.println ("Berechnung wird als Future gestartet.");
   new Thread (vb.future).start ();
      // Damit wird die Aktivität (Berechnung) gestartet.
   // Wir brauchen das Ergebnis noch nicht.
   // Wir haben noch anderes zu tun:
   long  d = 0;
   System.out.println ("Main tut was anderes.");
   try { Thread.sleep (d = (long) (Math.random () * 1500)); }
      catch (InterruptedException e)   { e.printStackTrace (); }
   System.out.println ("Main braucht dafür " + d + " ms.");
   // Aber jetzt brauchen wir das Ergebnis:
   System.out.println ("Main holt jetzt das Resultat,");
   System.out.println ("   muss aber eventuell auf restliche");
   System.out.println ("   Berechnungen warten.");
   try
      {
      Resultat r = vb.future.get ();
         // Hier wird das Resultat geholt und wir müssen
         // an dieser Stelle eventuell warten, bis das Resultat
         // fertig ist.
      System.out.print ("Resultat: ");
      for (int i = 0; i < 10; i++)
         {
         System.out.print (r.ergebnis[i] + " ");
         }
      System.out.println ();
      }
      catch (ExecutionException e) { e.printStackTrace (); }
      catch (InterruptedException e) { e.printStackTrace (); }
   }
}
```

Bemerkung: Unter C# gibt es das sogenannte Task-based Asnychronous Pattern (TAP), das Ähnliches leistet.

22.9 *Message Queues*

Für Botschaften-Kanäle können verschiedene Klassen verwendet werden, die die Schnittstellen `BlockingQueue` und/oder `Queue` implementieren.

Die Schnittstelle `Queue<E>` hat folgende Methoden:

```
boolean  add (E e);
   // Fügt das Element e in die Queue ein.
   // Wirft eine IllegalStateException, wenn das nicht sofort
   // möglich ist.

E  element ();
   // Gibt den Kopf der Queue zurück ohne ihn zu entfernen.
   // Wirft eine NoSuchElementException, wenn die Queue leer ist.

boolean  offer (E e);
   // Fügt e am Ende der Queue ein, wenn das möglich ist.

E  peek ();
   // Gibt den Kopf der Queue zurück ohne ihn zu entfernen.
   // Gibt null zurück, wenn die Queue leer ist.

E  poll ();
   // Gibt den Kopf der Queue zurück und entfernt ihn.
   // Gibt null zurück, wenn die Queue leer ist.

E  remove ();
   // Gibt den Kopf der Queue zurück und entfernt ihn.
   // Wirft eine NoSuchElementException, wenn die Queue leer ist.
```

Die Meisten Methoden werfen noch weitere Exceptions. Wegen Einzelheiten sei auf die API-Dokumentation verwiesen.

Die Schnittstelle `BlockingQueue<E>` erbt alle Methoden von `Queue<E>` und hat u.a. folgende zusätzliche Methoden:

```
boolean  offer (E e, long  time, TimeUnit unit);
   // Fügt e am Ende der Queue ein, wenn das möglich ist.
   // Blockiert höchstens time.

E  poll (long  time, TimeUnit unit);
   // Gibt den Kopf der Queue zurück und entfernt ihn.
   // Blockiert höchstens time.
   // Gibt null zurück, wenn die Queue leer ist.

void  put (E e);
   // Fügt e am Ende der Queue ein, wenn das möglich ist.
```

```
   // Blockiert, bis es möglich ist.

E  take ();
   // Gibt den Kopf der Queue zurück und entfernt ihn, wenn es möglich ist.
   // Blockiert, bis es möglich ist.
```

Wie bei der Schnittstelle `Queue<E>` werfen die meisten Methoden auch Exceptions. Wegen Einzelheiten sei auf die API-Dokumentation verwiesen.

Diese Schnittstelle wird von folgenden Klassen realisiert:

```
ArrayBlockingQueue
DelayQueue
LinkedBlockingQueue
PriorityBlockingQueue
SynchronousQueue
```

Am häufigsten dürfte `ArrayBlockingQueue<E>` eingesetzt werden. Sie hat eine begrenzte Kapazität und ihre Elemente sind FIFO-geordnet. Ihre wichtigsten Konstruktoren und Methoden sind:

```
public ArrayBlockingQueue<E> (int capacity)
public ArrayBlockingQueue<E> (int capacity, boolean fair)
void  clear ()
```

(`fair = true` gibt wartenden Erzeugern und Verbrauchern in FIFO-Reihenfolge Zutritt zur Queue.)

`ArrayBlockingQueue:` Begrenzte Kapazität.

`DelayQueue:` Objekte müssen eine gewisse Zeit in der Queue verbringen, ehe sie daraus entfernt werden können.

`LinkedBlockingQueue:` "Unendliche" Kapazität.

`PriorityBlockingQueue:` Die Objekte können verglichen werden. Die kleinsten oder größten Objekte werden zuerst aus der Queue entfernt.

`SynchronousQueue:` Enthält keine Objekte. D.h. ein Erzeuger muss mit dem Ablegen eines Objekts so lange warten, bis ein Verbraucher das Objekt aus der Queue entnehmen will. Und umgekehrt. D.h. eine `SynchronousQueue` wird nur dazu verwendet, um zwei Threads aufeinander warten zu lassen (Rendezvous).

22.9.1 *Deques*

`Deque` and `BlockingDeque` sind ***double ended queues,*** die die Entnahme und Ablage von Objekten an beiden Enden erlauben.

22.10 Atomare Variable

Auch Operationen von Standardtypen sind im Prinzip nicht Thread-sicher. Da das Anlegen einer Thread-sicheren Umgebung oft aufwendig ist, gibt es im Paket `java.util.concurrent.atomic` für die Standardtypen spezielle Klassen wie z.B.

```
AtomicInteger
AtomicLong
AtomicReference<V>
...
```

mit denen atomare Operationen wie z.B.

```
int   addAndGet (int delta)
int   getAndAdd (int delta)
int   decrementAndGet ()
int   getAndDecrement ()
int   incrementAndGet ()
int   getAndIncrement ()
int   getAndSet (int newValue)
boolean  compareAndSet (int expect, int update)
```

durchgeführt werden können.

22.11 Collections

22.11.1 Thread-sichere Collections

`Vector` und `Hashtable` sind Thread-sicher, nicht aber `ArrayList`, `HashMap` und andere Collections. Man kann allerdings Thread-sichere Collections erhalten mit:

```
List<E>  liste = Collections.synchronizedList (new ArrayList<E> ());
Map<K, E>   map = Collections.synchronizedMap (new HashMap<K, E> ());
```

Die Methoden von `liste` und `map` sind dann alle Thread-sicher.

22.11.2 Schwach Thread-sichere Collections

Die Thread-sicheren Collections haben den Nachteil, dass sie eine Collection oft sehr lange sperren und damit die Parallelität stark einschränken. Aus diesem Grund wurden sogenannte ***Concurrent Collections*** entwickelt, die mehr Parallelität erlauben, dafür aber nur **schwach konsistent** ***(weakly consistent)*** sind. Als Sperr-Strategie wird das sogenannte ***lock striping*** verwendet. Dabei werden nur Teile einer Collection gesperrt. In der Klasse `ConcurrentHashMap` werden nur die gerade benutzten Hash-Eimer gesperrt.

Bei `ConcurrentHashMap` sind die wichtigen Methoden

```
get
put
containsKey
remove
putIfAbsent
replace
```

Thread-sicher, nicht aber die weniger wichtigen Methoden wie:

```
size
isEmpty
```

Auch die Iteratoren dieser Klasse sind nur schwach konsistent.

`CopyOnWriteArrayList` ist ein concurrent Ersatz für ein synchronized `ArrayList`. `CopyOnWriteArraySet` ist ein concurrent Ersatz für ein synchronized `ArraySet`. Die Collections werden in diesem Fall als unveränderliche Objekte veröffentlicht und im Fall einer Veränderung kopiert. Da das Kopieren bei großen Collections aufwendig ist, eignen sich diese Klassen insbesondere, wenn anstatt von Modifikation Iteration weitaus häufiger ist.

22.12 GUI-Frameworks

Die meisten GUI-Frameworks sind ***single-threaded***. D.h. es gibt einen ***event dipatch thread*** **(EDT)**, der alle GUI-Ereignisse **sequentiell** abarbeitet. Multithreaded Frameworks hatten immer Probleme mit gegenseitigem Ausschluss. Insbesondere bestand die Gefahr eines Deadlocks.

Die sequentielle Behandlung von GUI-Ereignissen bedeutet, dass die individuelle Behandlung sehr schnell abgeschlossen sein muss, um die Antwortgeschwindigkeit *(responsiveness)* einer Anwendung zu gewährleisten. Daher muss eine lang andauernde Ereignisbehandlung in einen eigenen Thread ausgelagert werden. Die Probleme des Multithreadings werden also auf den Anwender geschoben.

22.13 Übungen

22.13.1 Concurrency Utilities

Als eine sehr gute Übung empfiehlt sich eine eigene Implementierung einiger Klassen der Pakete `java.util.concurrent` und `java.util.concurrent.locks`.

22.13.2 Pipes, Queues

Schreiben Sie zwei Threads, die über eine Pipe oder Queue miteinander kommunizieren. Als Grundlage könnte man das Herr-und-Knecht-Beispiel nehmen.

22.13.3 Pipes and Queues

Viele Herren schreiben Befehle in die gleiche Queue. Viele Knechte entnehmen diese Befehle dieser Queue und bearbeiten sie. Ein Befehl wird von einem Knecht bearbeitet. Nach der Bearbeitung sendet der Knecht eine Antwort über eine Pipe an den Herren, der den Befehl gegeben hat. Das bedeutet, dass ein Befehls-Objekt Informationen über den Herren, etwa die Pipe oder den Input-Pipe-Kopf enthalten muss. Der Herr liest die Antwort des Knechtes. Herren und Knechte haben Namen.

Literaturverzeichnis

[1] Joshua Bloch, "Effective Java Programming Language Guide", Addison-Wesley

[2] Mary Campione und Kathy Walrath, "The Java Tutorial Object-Oriented Programming for the Internet", Addison-Wesley 1996

[3] Patrick Chan, Rosanna Lee und Doug Kramer, "The Java Class Libraries, Second Edition, Volume 1" Addison-Wesley 1997

[4] Patrick Chan und Rosanna Lee, "The Java Class Libraries, Second Edition, Volume 2" Addison-Wesley 1997

[5] Robert Eckstein, Marc Loy und Dave Wood, "Java Swing" O'Reilly & Associates 1998

[6] Klaus Echtle und Michael Goedicke, "Lehrbuch der Programmierung mit Java", dpunkt-Verlag 2000

[7] David Flanagan, "Java in a Nutshell", 1. Auflage, O'Reilly & Associates 1996

[8] David Flanagan, "Java in a Nutshell", 2. Auflage, O'Reilly & Associates 1997

[9] David Flanagan, "Java in a Nutshell", 3. Auflage, O'Reilly & Associates 1999

[10] David Flanagan, "Java Foundation Classes in a Nutshell", O'Reilly & Associates 2000

[11] David Flanagan, Jim Farley, William Crawford und Kris Magnusson, "Java Enterprise in a Nutshell", O'Reilly & Associates 1999

[12] David M. Geary, "Graphic Java. Die JFC beherrschen (AWT)", Prentice Hall 1999

[13] David M. Geary, "Graphic Java. Die JFC beherrschen (Swing)", Prentice Hall 1999

[14] Brian Goetz, "Java Concurrency in Practice", Addison-Wesley Longman

[15] James Gosling, Bill Joy und Guy Steele, "The Java Language Specification"

[16] Mark Grand und Jonathan Knudsen, "Java Fundamental Classes Reference", O'Reilly & Associates 1997

[17] Peter Haggar, "Practical Java. Programming Language Guide", Addison-Wesley 2000

[18] Cay S. Horstmann und Gary Cornell, "Core Java. Band 1 – Grundlagen", Prentice Hall 1999

[19] Cay S. Horstmann und Gary Cornell, "Core Java. Band 2 – Expertenwissen", Prentice Hall 2000

[20] Doug Lea, "Concurrent Programming in Java: Design Principles and Patterns", Addison-Wesley

[21] Laura Lemay und Roger Cadenhead, "Java 2", Markt und Technik 1999

[22] Scott Oaks und Henry Wong, "Java Threads", OŽReilly

[23] Rainer Oechsle, "Parallele und verteilte Anwendungen in Java", Hanser 2007

[24] Lothar Piepmeyer, Vorlesungsskriptum "Java Threads", `lothar.piepmeyer@hs-furtwangen.de`

[25] Aaron Walsh und John Fronckowiak, "Die Java Bibel", MITP 1998