Java — Eine pragmatische Einführung

Holger Arndt

Sommersemester 2003

basierend auf alten Versionen von Dr. Benedikt Großer (2000) und Prof. Dr. Andreas Frommer (2001, 2002)

Aktueller Stand: Mittwoch, 9. Juli 2003, 16:39 Uhr

Inhaltsverzeichnis

1 Allgemeine Hinweise

1.1 Zur Geschichte von Java

1992	Sun entwickelt „*7“ (Palm-ähnlich) mit Green-OS und Interpreter Oak
Apr. 1993	NCSA Mosaic (erster grafischer Web-Browser)
März 1995	Hotjava (Browser von Sun, der Java-Applets ausführen kann)
23. Mai 1995	offizieller Geburtstag von Java
Dez. 1995	Netscape 2.0 (mit eingebautem Java-Interpreter)
23. Jan. 1996	JDK 1.0
18. Feb. 1997	JDK 1.1 (JDBC, Beans, . . .)
Anfang 1998	Browserunterstützung für Java 1.1
8. Dez. 1998	JDK 1.2, seither Java 2 Platform genannt (Swing, Java2D, . . .)
8. Mai 2000	JDK 1.3 (Performanceverbesserungen, . . .)
Feb. 2002	JDK 1.4

1.2 Software, Dokumentation, Literatur

- Downloads der Java 2 Standard Edition (J2SE):
  http://java.sun.com/j2se/downloads.html

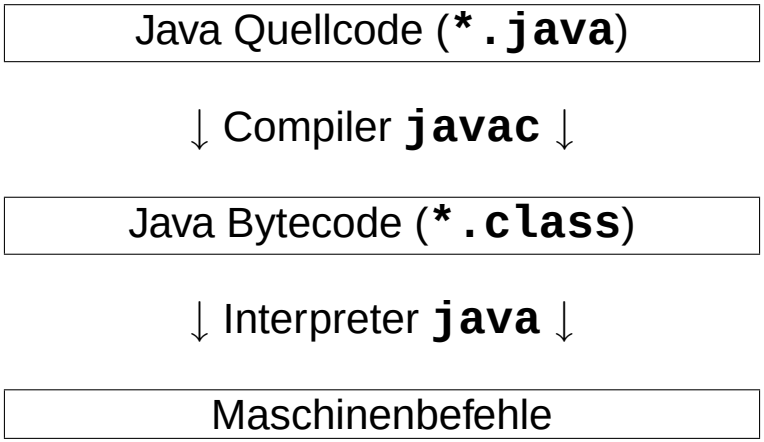
- API-Referenz zur Version 1.4.1:
  http://java.sun.com/j2se/1.4.1/docs/api/index.html
- The Java Tutorial von SUN:
  http://java.sun.com/docs/books/tutorial/
- Guido Krüger, Handbuch der Java-Programmierung, 3. Auflage:
  http://www.javabuch.de/
- Hubert Partl: Java Einführung, mit sehr gutem Teil über Datenbanken:
  http://www.boku.ac.at/javaeinf/
- Judith Bishop, Java Gently, Third Edition, Addison Wesley, 2001:
  http://javagently.cs.up.ac.za/jg3e/
- Axel Rogat, Objektorientiertes Programmieren mit C++ und Java, Hochschulrechenzentrum Universität Wuppertal, 2001; ältere Version unter:
  http://www.math.uni-wuppertal.de/~axel/skripte/oop/oop.html
- lokale Dokumentation auf den Rechnern `wmit`*XY*:
  file:/home/wmit00/javakurs/Doku/index.html

2 Erste Schritte

Inhalte / Ziele. *Ziel dieses Abschnittes ist das Erlernen der grundlegenden Begriffe und Arbeitsschritte der Programmiersprache Java. Dabei werden die Werkzeuge des Java-Development-Kits* **`javac`**, **`java`** *und* **`appletviewer`** *vorgestellt.*

2.1 Die virtuelle Maschine

Java-Programme werden nicht direkt in Maschinencode übersetzt, sondern in einen *plattformunabhängigen* Bytecode. Dieser wird zur Laufzeit von einer *Java Virtual Machine* interpretiert oder durch einen *Just-in-Time*-Compiler weiter übersetzt.



Die bei den folgenden Beispielen angegebenen Befehle zum Compilieren und Ausführen von Java-Programmen beziehen sich auf die Installation auf den IT-Rechnern.

2.2 „Hello World“ als Applikation

- Quelltext: **`Hello1.java`**
- Compilieren mit JDK 1.1.8: **`javac Hello1.java`**
- dabei entsteht: **`Hello1.class`**
- Ausführen: **`java Hello1`**
- Compilieren als Java-2-Programm mit Java-2-SDK 1.4.1:
  `/usr/java/j2sdk1.4.1_02/bin/javac Hello1.java`
- Ausführen des Java-2-Programms: **`/usr/java/j2sdk1.4.1_02/bin/java Hello1`**

HelloWorld/Hello1.java

```
1  /**
2     Hello-World-Applikation
3     @author Holger Arndt
4     @version 13.02.2003
5  */
6  public class Hello1
7  {
8    public static void main(String[] args)
9    {
10     System.out.println("Hello World!");
11   }
12 }
```

- Basis jeder Java-Applikation
- Klassen bilden die grundlegenden Bausteine von Java. Eine Klassendefinition erfolgt mit dem Schlüsselwort **`class`**.
- Das Schlüsselwort **`public`** bezieht sich auf Zugriffsrechte.
- Der Interpreter **`javac`** erwartet, dass die Java-Quelltextdatei mindestens eine **`class`**-Definition gleichen Namens beinhaltet. (Also: Eine Java-Quelltextdatei **`Xyz.java`** enthält mindestens die Klasse **`class Xyz`**.) Es ist erlaubt, aber nicht konventionell, mehrere Klassen in einer Java-Quelltextdatei zu definieren.

- Eine Java-Applikation muss eine **`class`**-Definition enthalten, die die Methode **`main()`** implementiert.
- Der Interpreter **`java`** benutzt die Methode **`main()`** als Startpunkt für den Programmablauf. Er arbeitet alle Anweisungen der **`main()`**-Methode ab.
- Die Applikation endet nach Ausführung des letzten Anweisung der **`main()`**-Methode.

Quelltextanalyse

1. Definition einer Klasse:

```
6  public class Hello1
7  {
12 }
```

   - Mit dem Schlüsselwort **`class`** beginnt die Definition einer Klasse, in diesem Falle **`Hello`**.
   - Ein Paar geschweifter Klammern begrenzt den Klassenkörper, in dem wir dann Klassenattribute und -methoden implementieren können.

2. Die Methode **`main()`**:

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

   - Jede Java-Applikation muss eine **`main()`**-Methode enthalten, die wie oben angegeben definiert wird. Der Interpreter **`java`** sucht für den Programmeinstieg genau nach dieser Signatur.

- Dem Methodennamen **`main()`** sind drei Schlüsselworte vorangestellt:
  - **`public`** bezieht sich auf *Zugriffsrechte*.
  - **`static`** zeigt an, dass **`main()`** eine *Klassenmethode* ist.
  - **`void`** bezieht sich auf den *Datentyp* / *Rückgabewert*.
- Hinter dem Methodennamen **`main()`** schließt sich in runden Klammern noch eine Argumentliste an: **`String argv[]`**. Mit diesem Mechanismus kann das Betriebssystem beim Starten der Applikation Kommandozeilenargumente übergeben.

3. Ausgabe von Text mit `System.out.println()`:

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

2.3 Ein grafisches „Hello World“

HelloWorld/HelloAWT.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  /**
5     Grafische Hello-World-Applikation
6     @author Holger Arndt
7     @version 25.02.2003
8  */
9
10 public class HelloAWT extends Frame
11 {
12   HelloAWT()
13   {
14     add(new Label("Hello World!", Label.CENTER));
15     addWindowListener(new WindowAdapter()
16       {
17         public void windowClosing(WindowEvent e)
18         {
19           System.exit(0);
20         }
21       });
22     setTitle("Halloooo");
23     setSize(200, 100);
24     setVisible(true);
25   }
26
27   public static void main(String[] args)
28   {
29     new HelloAWT();
30   }
31 }
```

2.4 Ein grafisches „Hello World“ mit Swing

HelloWorld/HelloSwing.java

```
1  import javax.swing.*;
2  import java.awt.event.*;
3
4  /**
5     Grafische Hello-World-Applikation mit Swing
6     @author Holger Arndt
7     @version 25.02.2003
8  */
9
10 public class HelloSwing extends JFrame
11 {
12   HelloSwing()
13   {
14     getContentPane().add(new JLabel("<HTML><BIG>Hello World!</BIG></HTML>",
15                                     SwingConstants.CENTER));
16     addWindowListener(new WindowAdapter()
17       {
18         public void windowClosing(WindowEvent e)
19         {
20           System.exit(0);
21         }
22       });
23     setTitle("Halloooo");
24     setSize(200, 100);
25     setVisible(true);
26   }
27
28   public static void main(String[] args)
29   {
30     new HelloSwing();
31   }
32 }
```

Swing-Programme werden später in 13 behandelt. Bereits hier ist aber zu erkennen, dass sich am grundlegenden Prinzip grafischer Applikationen im Gegensatz zu AWT nicht viel ändert. Zum Compilieren ist mindestens die JDK-Version 1.2 notwendig.

2.5 „Hello World“ als Applet

HelloWorld/HelloApplet.java

```
1  import java.awt.*;
2  import java.applet.Applet;
3
4  /**
5     Hello-World-Applet
6     @author Holger Arndt
7     @version 21.02.2003
8  */
9  public class HelloApplet extends Applet
10 {
11   public void paint(Graphics gc)
12   {
13     gc.drawString("Hello World", 50, 50);
14   }
15 }
```

Der Java-Bytecode wird in eine HTML-Datei namens **HelloApplet.html** eingebettet:

HelloWorld/HelloApplet.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>
<TITLE>Hello3</TITLE>
<META HTTP-EQUIV="content-type" CONTENT="text/html;charset=ISO-8859-1">
</HEAD>

<BODY>
<APPLET CODE="HelloApplet.class" HEIGHT=100 WIDTH=200></APPLET>
</BODY>
</HTML>
```

Da viele Browser Java 2 noch nicht unterstützen, sollten Applets immer für virtuelle Maschinen der Version 1.1 übersetzt werden. Bei neueren Java-Versionen erreicht man dies durch die Option **`-target`**:

`/usr/java/j2sdk1.4.1_02/bin/javac -target 1.1 HelloApplet.java`

Ausführung mit dem Appletviewer oder in einem Browser:

`appletviewer HelloApplet.html` oder **`netscape HelloApplet.html`**

Wir erfahren später mehr über Applets (siehe 7).

2.6 HTML

Das HTML-Grundgerüst aus dem letzten Abschnitt wird für die meisten Fälle ausreichen.

Referenzen zu HTML

- Stefan Münz: SELFHTML, http://selfhtml.teamone.de/
- Hubert Partl: http://www.boku.ac.at/htmleinf/
- Axel Rogat: http://www.math.uni-wuppertal.de/~axel/skripte/oop/oopE.html
- W3C: http://www.w3.org/MarkUp/ (enthält u. A. die offizielle Spezifikation)

2.7 Zusammenfassende Fragen

- Warum bezeichnet man Java als plattformunabhängige Sprache?
- Welche beiden unterschiedlichen Arten von ausführbaren Programmen gibt es?
- Wie implementiert und startet man eine „Hello World!“-Applikation?
- Wie implementiert und startet man ein „Hello World!“-Applet?
- Welche virtuellen Maschinen kann man jeweils verwenden?
- Wie kann man in einer Applikation Text auf den Bildschirm drucken lassen?
- Wie erstellt man HTML-Dokumente?

3 Grundlagen kurzgefasst

Inhalte / Ziele. *Ziel dieses Abschnittes ist die Beschreibung der elementaren Bestandteile eines Java-Programmes. Im Anschluss an diesen Abschnitt sollten wir einfache Javaprogramme schreiben und kompliziertere zumindest lesen können. Ferner lernen wir das Werkzeug* **`javadoc`** *des Java-Development-Kits kennen.*

3.1 Ein- und Ausgabe

Java ermöglicht den Zugriff auf die Standardein- und ausgabe über **`System.in`** und **`System.out`**.

Grundlagen/EinAusgabe/SystemOut.java

```
6  public class SystemOut
7  {
8    public static void main (String argv[])
9    {
10     System.out.println("6 * 7 = " + (6 * 7));
11   }
12 }
```

Das Einlesen von der Standardeingabe ist etwas mühsam. Wir werden uns später ausführlich damit beschaftigen (siehe 9.2). Das folgende Programm liefert nicht das erwünschte Ergebnis!

Grundlagen/EinAusgabe/SystemIn.java

```
14          int val = System.in.read();
15          byte b = (byte) val;
16          char c = (char) val;
17          System.out.println("Eingabe als int:  " + val);
18          System.out.println("Eingabe als byte: " + b);
19          System.out.println("Eingabe als char: " + c);
```

3.2 Lexikalische Struktur

- kein Präprozessor
- Eingabezeichen aus dem Unicode-Zeichensatz.

  Das folgende Beispielprogramm muss mit der Option **`-encoding utf-8`** übersetzt werden.

HelloWorld/HelloUnicodeSwing.java

```
19      String Czech_česky = "Dobrý den";
21      String Esperanto = "Saluton (Ĥoŝanĝo ĉiuĵaŭde)";
37      String German_DeutschSüd = "Grüß Gott";
39      String Greek_ Ελληνικά  = "Γειά σας";
53      String Russian_Русский = "Здравствуйте!";
63      String Vietnamese_Tiếng_Việt = "Chào bạn";
```

- Namenskonventionen
  - Klassen: **`Hello`**, **`Button`**, **`Color`**, **`Applet`**
  - Methoden: **`main`**, **`setLabel`**, **`toString`**, **`getCodeBase`**
  - Attribute: **`argv`**, **`height`**, **`startAngle`**
  - „Konstanten“: **`PI`**, **`MIN_VALUE`**, **`MAX_PRIORITY`**
- Kommentare
  - einzeilige Kommentare: `//xxx`
  - mehrzeilige Kommentare: `/*xxx */`
  - Dokumentationskommentare: `/**xxx */`

  Die ein- und mehrzeiligen Kommentare sind sinnvoll für die *Programmierer* von Klassen. Die Dokumentationskommentare sind für *Benutzer* von Klassen gedacht und arbeiten mit **`javadoc`** (siehe 4.2.1) zusammen.

3.3 Einfache Datentypen und Variablen

Die acht einfachen *Datentypen* in Java zeichnen sich durch feste Länge und Standardwerte aus.

Typname	Länge in Byte	Wertebereich	Standardwert
`boolean`	1	`true`, `false`	`false`
`char`	2	alle Unicode-Zeichen	`'\000'` bzw. `'\u0000'`
`byte`	1	$-2^{-7}, \ldots, 2^7 - 1$	`0`
`short`	2	$-2^{-15}, \ldots, 2^{15} - 1$	`0`
`int`	4	$-2^{-31}, \ldots, 2^{31} - 1$	`0`
`long`	8	$-2^{-63}, \ldots, 2^{63} - 1$	`0`
`float`	4	$\pm 1,4024 \cdot 10^{-45}, \ldots, \pm 3,4028 \cdot 10^{38}$	`0.0f`
`double`	8	$\pm 4,0407 \cdot 10^{-324}, \ldots, \pm 1,7977 \cdot 10^{308}$	`0.0`

Java kennt keine expliziten Zeiger, Recordtypen oder Bitfelder.

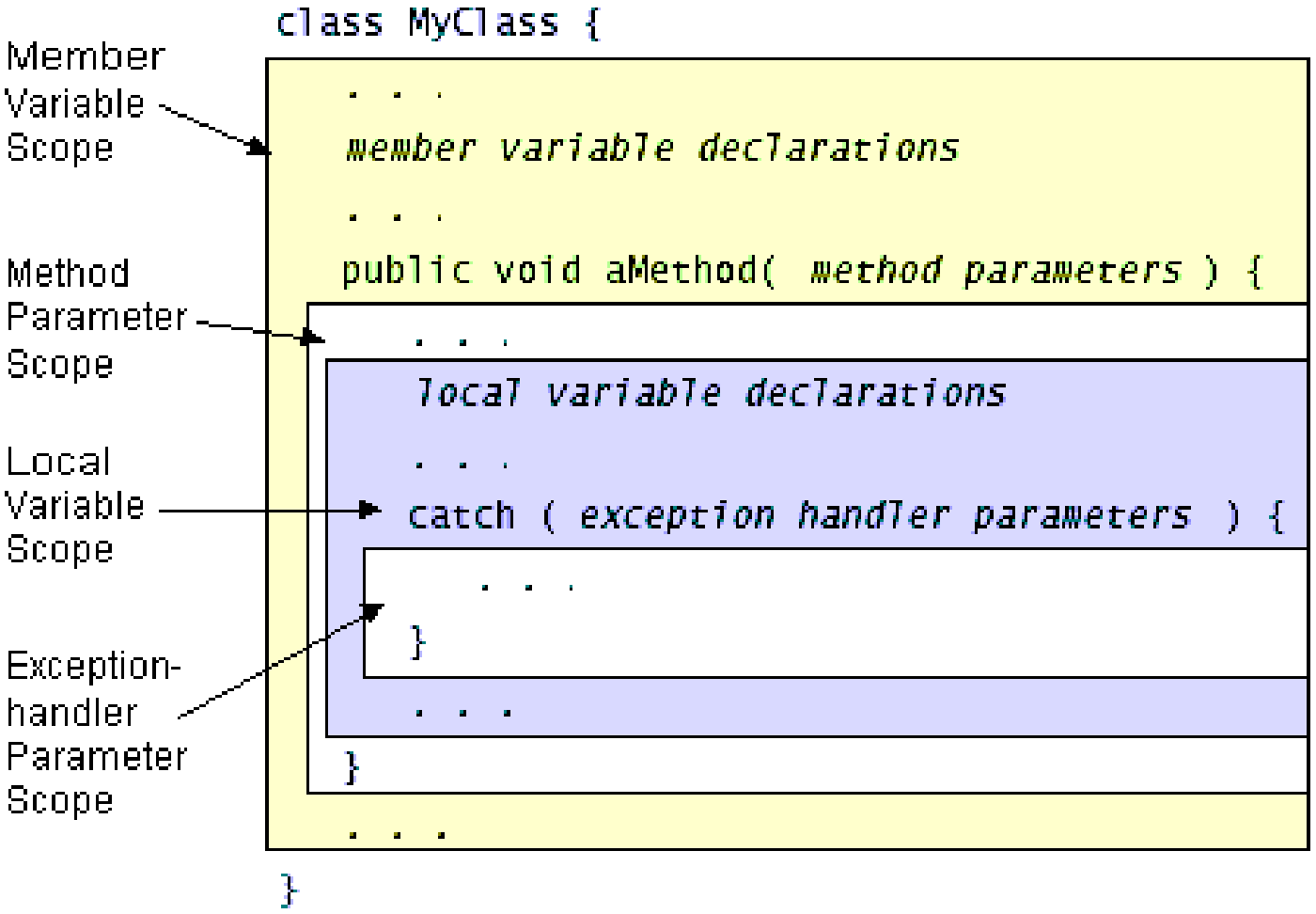
Eine *Variable* ist gekennzeichnet durch

Datentyp: Neben einfachen Datentypen lernen wir später noch Referenzdatentypen kennen.

Name: siehe Namenskonventionen

Scope: (deutsch Gültigkeitsbereich, Lebensraum). Man unterscheidet in

- Member-Variablen
- Lokale Variablen
- Variablen aus Parameterlisten
- Exception-Handler Variablen



Wert: Der Compiler **`javac`** lässt nicht zu, dass eine nicht mit einem Wert initialisierte Variable einer Anderen zugewiesen wird. Man kann das Anlegen und Initialisieren zusammenfassen. Wird eine Member-Variable deklariert, so erhält sie den Standardwert des Datentyps.

Grundlagen/Datentypen/TypBsp.java

```
10      int a;
11      a = 3;
12      int b = 5;
13      int c = a * b;
```

Mit dem Schlüsselwort **final** kann man unterbinden, dass eine Variable nach der ersten Belegung nochmals geändert wird.

```
14      final int d = 4;
15      //d = 5; // Fehler
```

3.4 Referenzdatentypen

Neben den einfachen Datentypen gibt es in Java noch die Referenzdatentypen. Darunter fallen alle *Objekte*, aber auch *Arrays* und *Strings*.

- Diese Typen werden „per Referenz“ verarbeitet, d. h. die Adresse des Objekts (Arrays oder Strings) wird in einer Variablen abgelegt.
- Referenzdatentypen werden zur Laufzeit erzeugt.
- Variablen vom Referenzdatentyp haben den Standardwert **null**.
- Objekte können ihrerseits wieder über Methoden und Variablen verfügen.

3.4.1 Arrays

Die Deklaration einer Array-Variablen erfolgt, indem man an einen einfachen Datentyp ein Paar eckiger Klammern anhängt:

```
19      double[] daten;
```

Bis jetzt ist noch nicht spezifiziert, wieviele Einträge das Array haben soll. Mit dem **new**-Operator können wir das festlegen und entsprechenden Speicherplatz anfordern:

```
20      daten = new double[10];
```

Man kann Deklaration und die so genannte *Instantiierung* wieder zusammenfassen:

```
22      int[] zensuren = new int[6];
```

Referenzierung von Array-Elementen erfolgt mit Hilfe der eckigen Klammern. Dabei wird das erste Element mit **0** angesprochen.

```
23      zensuren[0] = 2;
24      zensuren[1] = 4;
```

Man kann Deklaration, Instantiierung und Initialisierung auch zusammenfassen, ohne den **new**-Operator zu benutzen:

```
26      int[] zensuren2 = { 2, 4, 5, 3, 2, 1 };
```

Als Referenzdatentypen verfügen Arrays über das Attribut **length**, das die Länge des Arrays zum Inhalt

hat.

```
27      System.out.println(zensuren2[zensuren2.length - 1]);
28      // druckt 1
```

Mehrdimensionale Arrays sind möglich:

```
30      char[][] schachbrett = new char[8][8];
```

Die Klammern **[]** können auch hinter der Variable angegeben werden.

```
33      double daten2[] = new double[12];
```

Mit der Methode **arraycopy** aus **System** lassen sich (Teile eines) Arrays kopieren.

```
34      System.arraycopy(daten, 2, daten2, 5, 4);
35      // daten2[5..8] = daten[2..5]
```

3.4.2 Strings

Strings (Zeichenketten) werden in Java ebenfalls als Referenzdatentypen verwendet.

Deklaration und Initialisierung separat:

```
37    String dateiname1;
38    dateiname1 = "Hello1.java";
```

Deklaration und Initialisierung zusammengefasst:

```
40    String dateiname2 = "Hello1.java";
```

Der Referenzdatentyp **`String`** verfügt über den einzigen überladenen Operator (**+**) in Java:

```
42    String rakete = "Apollo";
43    rakete = rakete + 3 * 4;
44    System.out.println(rakete); // druckt Apollo12
```

Als Referenzdatentypen verfügen Strings über Methoden wie **`length()`** und **`charAt()`**.

```
45    char x = rakete.charAt(1);
46    System.out.println("Zeichen 1 = " + x);
47    System.out.println("Zeichenanzahl = " + rakete.length());
48    // druckt Zeichenanzahl = 8
```

Prinzipiell ändern Strings ihren Wert *nie*. In Java sind dennoch Zuweisungen auf Strings erlaubt. Intern legt der Compiler dann jeweils einen neuen String an.

Siehe API-Referenz, um mehr über Strings zu erfahren.

3.4.3 Identität und Äquivalenz

Bei Referenzdatentypen muss man zwischen *Identität* und *Äquivalenz* unterscheiden:

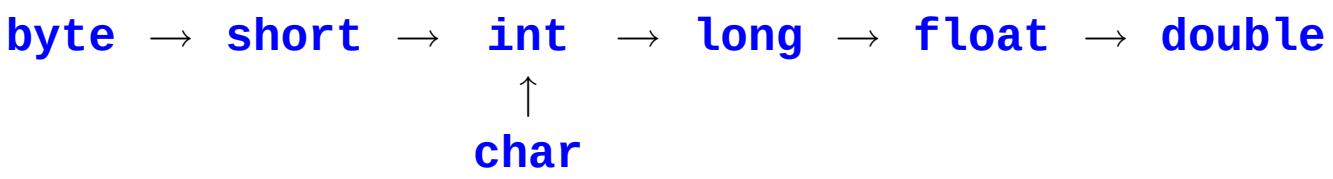
- Identität prüft, ob zwei Referenzen auf die gleiche Adresse verweisen. Der zugehörige Operator lautet **==**.
- Äquivalenz prüft, ob zwei Adressen gleichen Inhalt haben. Die zugehörige Methode lautet **`equals()`**.

Grundlagen/IdentitaetAequivalenz/IAcheck.java

```
28      String wort1 = "Color.java";
29      String wort2 = "Color";
30      boolean icheck, aecheck;
33      wort2 = wort2 + ".java";
36      icheck = (wort1 == wort2);          // false
39      aecheck = (wort1.equals(wort2)); // true
42      wort1 = wort2;
44      icheck = (wort1 == wort2);          // true
47      wort1 = new String(wort2);
48      icheck = (wort1 == wort2);          // false
```

3.5 Typkonvertierungen

Bei einer Wertübertragung von einem einfachen Datentyp in einen Anderen sind in Java Regeln zur Typkonvertierung zu beachten. Man unterscheidet *erweiternde* und *einschränkende* Konvertierungen. Grob gesagt geht es darum, ob bei einer Konvertierung der Wertebereich vergrößert oder verringert wird. Folgende Tabelle zeigt, welche einfachen Datentypen ineinander konvertiert werden können:



Konvertierungen in Pfeilrichtung sind erweiternd und Konvertierungen entgegen der Pfeilrichtung sind einschränkend. Insbesondere sind Konvertierungen mit dem Datentyp **`boolean`** nicht erlaubt. Einfache Umwandlungen lassen sich hier mit Hilfe des ternären Operators **`? :`** erzielen.

Erweiternde Konvertierungen werden in Java automatisch vorgenommen:

Grundlagen/Datentypen/CastBsp.java

```
10      //float einzelgewicht = 1.3;      // Fehler: double -> float
11      float einzelgewicht = 1.3f;
12      byte  anzahl = 5;
13      float gesamtgewicht = anzahl * einzelgewicht;
14      //                                  // anzahl: byte -> float
15
16      double PI = 3.1415927;
17      double radius = 2.5;
18      double umfang = 2 * PI * radius; // 2 -> 2.0: int -> double
```

Einschränkende Konvertierungen müssen in Java explizit mit Hilfe des Type-Cast-Operators durchgeführt werden:

```
20      double x = 2.0 * 0.5;
21      int y;
22      //y = x;                          // Fehler: double -> int
23      y = (int)x;                       // okay (expliziter Cast)
```

3.6 Operatoren und Ausdrücke

Operatoren führen bestimmte Operationen auf einer, zwei oder drei Variablen aus (*unäre*, *binäre*, *ternäre* Operatoren).

- Unäre Operatoren können in *Präfix*- oder *Postfix*-Notation benutzt werden:

  Präfix-Operator **`operator op`** z. B. **`++i`**
  Postfix-Operator **`op operator`** z. B. **`i++`**

- Binäre Operatoren benutzen *Infix*-Notation:

  Infix-Operator **`op1 operator op2`** z. B. **`x + y`**

- In Java gibt es auch einen ternären Operator:

  `expr ? op1 : op2` z. B. **`(i > 0) ? 1 : -1`**

Man unterteilt die Operatoren in Java in

- Arithmetische Operatoren (**`+`**, **`-`**, **`*`**, **`/`**, **`%`**; **`++`**, **`--`**)
- Vergleichsoperatoren (**`>`**, **`>=`**, **`<`**, **`<=`**, **`==`**, **`!=`**)
- Logische Operatoren (**`&&`**, **`||`**, **`!`**, **`&`**, **`|`**, **`^`**)
- Bitweise Operatoren (**`<<`**, **`>>`**, **`>>>`**, **`&`**, **`|`**, **`^`**, **`~`**)
- Zuweisungsoperatoren (**`+=`**, **`-=`**, **`*=`**, **`/=`**, **`%=`**, **`&=`**, **`|=`**, **`^=`**, **`<<=`**, **`>>=`**, **`>>>=`**)
- sonstige (**`? :`**, **`[ ]`**, **`.`**, **`( )`**, **`new`**, **`instanceof`**)

Für eine Liste dieser Operatoren sei beispielsweise auf
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/operators.html verwiesen.

Ein *Ausdruck* ist eine Folge von Variablen, Operatoren und Methodenaufrufen, die zur Laufzeit ausgewertet wird und deren Ergebnis einer einfachen Variablen zugewiesen wird.

```
x = (i-- < (a + b) / 2) ? Math.cos(3 * (a - b)) : x + b * b / 10;
```

Die Reihenfolge der Auswertung eines Ausdrucks ist durch gewisse Operator-Vorrangregeln bestimmt. Durch Klammerung kann sie beeinflusst werden. Siehe auch
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/expressions.html.

Grundlagen/Operatoren/OpBsp.java

```
10      int i = 4;
11      int j, k;
12
13      // unäre Operatoren:
14      // präfix ++i
15      // postfix i++
20      j = ++i;
23
24      // binäre Operatoren :
25      i = 37;
26      j = 3;
27      k = i % 3 + 2 * j;
```

```
29
30      // Der tertiäre Operator:
31      int signum;
32      double x;
33      x = 3.0;
34      signum = (x > 0.0) ? 1 : -1;
39
40      // Vergleichsoperatoren
41      i = 3;
42      k = 6;
44      if (i > k)
45        System.out.println("Es␣gilt:␣i␣>␣k");
48
49      // Zuweisungsoperatoren
50      i = k;
52      i = i + 2;
54      i += 3;
56      i /= 2;
58
59      // Logische Operatoren mit und ohne Kurzschlussauswertung
62      x = ((k > i) & (j++ > 5)) ? 1 : 0;       // j wird immer erhöht
64      x = ((k != 0) && (1 / k > 0.2)) ? k : 0; // keine Div. durch 0
```

3.7 Kontrollstrukturen

Diese Strukturen steuern den Ablauf eines Programmes. In Java gibt es diese Kontrollstrukturen:

- **`if`** — **`else`**
- **`switch`** — **`case`** — **`break`**
- Schleifen: **`for`**, **`while`**, **`do`** — **`while`**
- Ausnahmebehandlung: **`try`** — **`catch`** — **`finally`**, **`throw`**
- Sonstiges: **`break`**, **`continue`**, **`return`**

Beispiel: `if` — `else`

Grundlagen/Kontrollstrukturen/IfElseBsp.java

```
12      if (x >= 0)
13        {
14          y = Math.sqrt(x);
15        }
28      if (x == 0)
29        {
30          signum = 0;
31        }
32      else if (x > 0)
33        {
34          signum = 1;
35        }
36      else
37        {
38          signum = -1;
39        }
```

Beispiel: `switch` — `case` — `break`

Grundlagen/Kontrollstrukturen/SwitchBsp.java

```
11      switch (farbe)
12        {
13          case 0:
14            System.out.println("rot");
15            break;
16          case 1:
17            System.out.println("blau");
18            break;
23          default:
24            System.out.println("Kenne␣diese␣Farbe␣nicht");
25        }
```

Beispiel: `for`-Schleife

Grundlagen/Kontrollstrukturen/ForBsp.java

```
11      int summe = 0;
12      int n = 5;
13      for (int i = 1; i <= n; i++)
14        {
16          summe += i * i;
17        }
21      for (int i = 1, j = 9999; i < j; i *= 2, j -= 444)
```

Beispiel: `while`-Schleife

Grundlagen/Kontrollstrukturen/WhileBsp.java

```
11      while (n != 1)
12        {
13          n = n % 2 == 1 ? 3 * n + 1 : n / 2;
14          System.out.println(" n = " + n);
15        }
```

Beispiel: `do`-`while`-Schleife

Grundlagen/Kontrollstrukturen/DoWhileBsp.java

```
11      do
12        {
13          System.out.println(n);
14          n--;
15        }
16      while (n >= 0);
```

Ausnahmebehandlungen betrachten wir später (siehe 6).

Mit **`break`** kann man Schleifen abbrechen, **`continue`** beendet den aktuellen Schleifendurchlauf. Beide sollten möglichst vermieden werden („Unser Motto ohne Got(t)o“).

3.8 Kommandozeilenparameter

Auf Kommandozeilenparameter kann über den **`String`**-Array-Parameter der Methode **`main`** (üblicherweise **`argv`**) zugriffen werden.

Grundlagen/Kommandozeilenparameter/ArgvBsp.java

```
8   public static void main (String argv[])
9   {
10    for (int i = 0; i < argv.length; i++)
11      {
12        System.out.println(argv[i]);
13      }
27  } // Ende main()
```

3.9 Zusammenfassende Fragen

- Was sind einfache Datentypen?
- Welche einfache Datentypen gibt es in Java?
- Was ist der Lebensraum einer Variablen?
- Wie deklariert und initialisiert man Variablen?
- Wie kann man Variablen mit unveränderbarem Wert erzeugen?

- Was ist der Unterschied zwischen einfachen und Referenzdatentypen?
- Wie deklariert und initialisiert man Arrays?
- Wie kann man auf Arrays zugreifen?
- Wie deklariert und initialisiert man Strings?
- Wie kann man auf Strings zugreifen?
- Was ist der Unterschied zwischen Identität und Äquivalenz?
- Was bedeutet erweiternde und einschränkende Typkonvertierung?
- Was für Operatoren sind in Java zulässig?
- Welche Operator-Vorrangregeln sind in Java gültig?
- Welche Kontrollstrukturen gibt es?
- Welche Kommentarzeichen gibt es?

4 Objektorientierung in Java, Teil 1

Inhalte / Ziele. *Die zentralen und für das praktische Arbeiten mit Java wichtigen Begriffe des objektorientierten Softwareentwurfs werden vorgestellt. Das Zusammenspiel zwischen der Deklaration von Klassen einerseits und dem Lebenszyklus von Objekten andererseits steht dabei im Mittelpunkt.*

4.1 Grundbegriffe

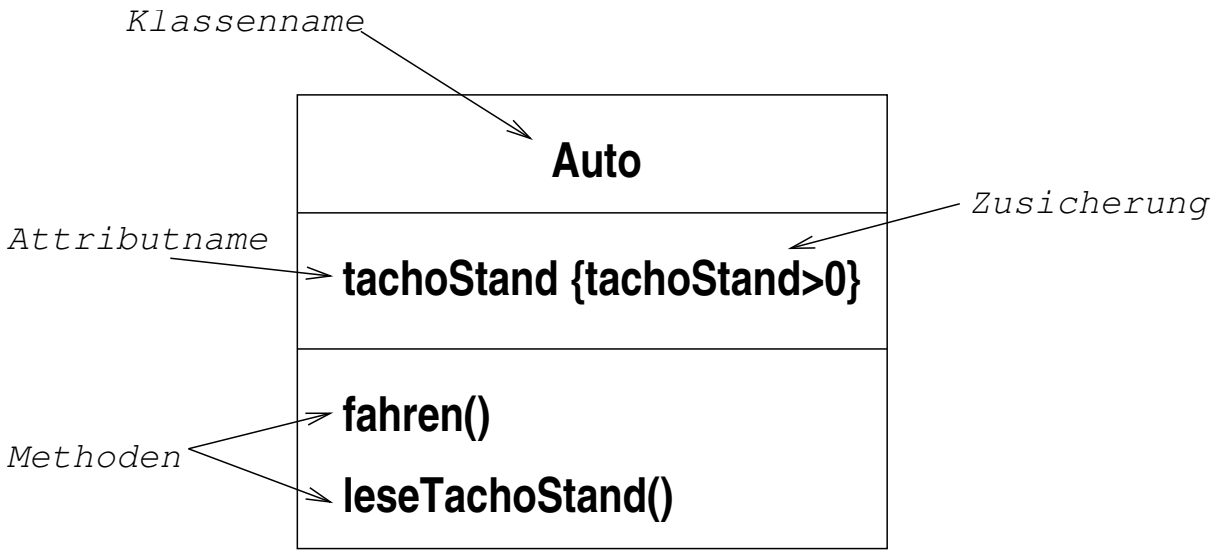
Beim objektorientierten Softwareentwurf versucht man, die in der realen Welt vorkommenden Gegenstände als Objekte zu interpretieren. Bei der Abbildung der realen Welt nach Software beschränkt man sich dabei auf das Wesentliche:

Entwickler: „Was ist Euch wichtig?“
Anwender: „Autos“
Entwickler: „Was ist denn ein Auto, welche Merkmale sind für Euch relevant?“
Anwender: „Ein Auto hat einen Tachostand und man kann damit fahren.“

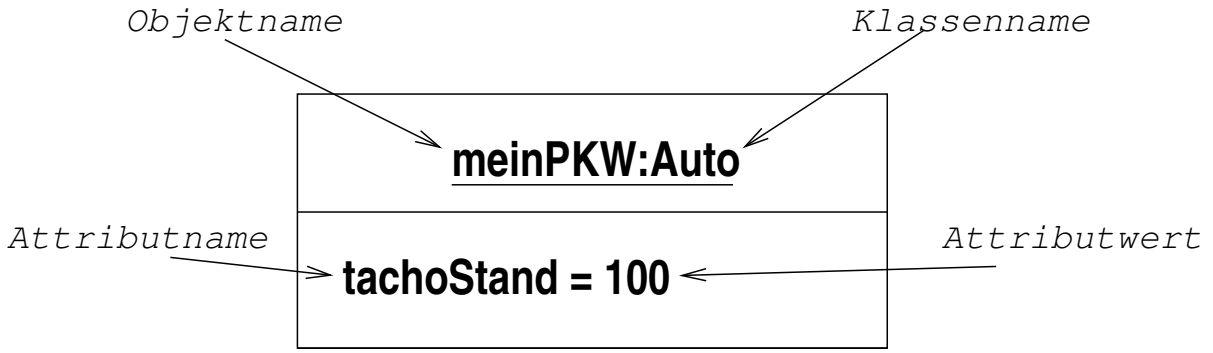
Der Entwickler entwirft nun eine Vorlage, in der alle geforderten Merkmale berücksichtigt sind. Nach dieser Vorlage können dann beliebig viele konkrete Exemplare neu erzeugt werden.

In einer objektorientierten Programmiersprache heißen solche Vorlagen *Klassen*. Die konkreten Exemplare nennen wir *Objekte*. Das Erzeugen von Objekten nach der Vorlage einer Klasse heißt *Instantiieren*. Die Merkmale können wir unterteilen in *Attribute*, *Methoden* und *Zusicherungen*.

Eine Klasse kann man wie folgt darstellen:



Objekte — also die Exemplare der Klassenvorlagen — stellt man ähnlich dar. Zur Unterscheidung wird der Objektname unterstrichen.



Erstes wichtiges Konzept:

Im objektorientierten Softwareentwurf werden in den Klassen bzw. Objekten *sowohl* Attribute *als auch* Methoden *und* Zusicherungen zusammengefasst.

4.2 Klassen und Objekte in Java

Softwareentwicklung unter Java erfolgt also auf zwei Ebenen:

Zum Einen entwerfen und implementieren wir Klassen. Ein Java-Quellcode für unsere Klasse **`Auto`** könnte so aussehen:

Objektorientierung/Autos/01/Auto.java

```
6  public class Auto
7  {
8    // Objektattribute
9    double tachoStand;
10
11   // Objektmethoden
12   public void fahren(double distanz)
13   {
14     tachoStand = tachoStand + distanz;
15   }
16
17   public double leseTachoStand()
18   {
19     return tachoStand;
20   }
21
22   public void ausgeben()
23   {
24     System.out.println("Tachostand = " + tachoStand + " km");
25   }
38 } // ende: public class Auto
```

Auf der anderen Ebene können wir von einer Klasse beliebig viele Objekte instantiieren und auf ihre Merkmale zugreifen:

```
28   public static void main(String argv[])
29   {
30     Auto meinPKW = new Auto();
31     meinPKW.fahren(100);
32     meinPKW.ausgeben();
33     meinPKW.fahren(20);
34     System.out.println(meinPKW.leseTachoStand());
37   }
```

4.2.1 Deklaration von Klassen

Klassen implementieren wir in Java-Quelltextdateien **`*.java`**. In einer Datei **`Xyz.java`** muss mindestens eine Klasse mit dem Namen **`Xyz`** deklariert werden. Üblicherweise verwendet man für jede Klassendeklaration eine eigene Datei.

Die Klassendeklaration wird durch das Schlüsselwort **`class`** angezeigt. Für Eigenschaften, die sich auf Zugriffsrechte oder Vererbung beziehen, können zusätzliche Modifizierer wie **`public`** oder **`final`** verwendet werden.

Nach der Festlegung des Namens erfolgt in geschweiften Klammern die Implementierung des Klassenkörpers.

Im Beispiel **`Auto`** sind bereits einige *Objektattribute* und *Objektmethoden* implementiert.

Um eine Klasse lokal auszutesten, kann man die Methode **`main()`** benutzen.

Java bietet dem Entwickler die Möglichkeit, parallel zum Implementieren Dokumentationskommentare zu verfassen, die mit dem SDK-Werkzeug **`javadoc`** ausgewertet werden können.

Objektorientierung/Autos/02-javadoc/Auto.java

```
1  /**
2   * Klasse Auto (2).
3   * eine Basis-Implementierung einer Klasse zur Simulation
4   * eines Autos.
5   * <p>
6   * Mit Hilfe von Dokumentationskommentaren kann der Entwickler
7   * einer Klasse dem Anwender den Leistungsumfang der Klasse und
8   * die Benutzung ihrer Schnittstellen darstellen.
9   *
10  * @author Benedikt Gro&szlig;er, Holger Arndt
11  * @version 04.04.2003
12  * @since JDK 1.0
13  */
14 public class Auto
```

```
20    /** simuliert die Fortbewegung um eine gewisse Distanz.
21     *  @param distanz die zu fahrende Distanz.
22     */
23    public void fahren(double distanz)

28    /** erlaubt Zugriff auf den aktuellen Tachostand des Autos.
29     *  @return den aktuellen Tachostand des Autos.
30     */
31    public double leseTachoStand()

36    /** gibt alle Informationen des Autos aus.
37     */
38    public void ausgeben()
```

Quelltextanalyse

- Mit dem Kommando **`javadoc Auto.java`** wird ein Dokumentationssystem aus HTML-Seiten generiert: `Quelltexte/Objektorientierung/Autos/02-javadoc/javadoc11/tree.html`, `Quelltexte/Objektorientierung/Autos/02-javadoc/javadoc14/index.html`
- Innerhalb der Dokumentationskommentare im Java-Quellcode dürfen die üblichen HTML-Tags verwendet werden.

- Zur Strukturierung gibt es Markierungen, die mit dem Zeichen @ eingeleitet werden:

Markierung	Dokumentation	Verwendung bei
@author	Autoreneintrag	Klasse, Interface
@version	Versionseintrag (Datum)	Klasse, Interface
@see	Querverweis	Klasse, Interface, Attribut, Methode
@param	Parameter der Methode	Methode
@return	Rückgabewert der Methode	Methode
@throws (alt: **@exception**)	Ausnahmen	Methode
@deprecated	zeigt eine veraltete Methode an	Methode

- Details zur Benutzung von **javadoc** findet man beispielsweise unter
  http://java.sun.com/j2se/javadoc/index.html

4.2.2 Instantiieren von Objekten

Klassen dienen als Vorlage für viele einzelne Exemplare (Objekte). Objekte sind Referenzdatentypen.

Objektorientierung/Autos/03-Instantiierung/Auto.java

```
30      Auto bmw; // erzeugt eine Referenz auf ein Objekt vom Typ Auto
```

Instantiierung von Objekten erfolgt mit dem **new**-Operator.

```
32      bmw = new Auto(); // instantiiert das Objekt

34      Auto zweiCV = new Auto(); // beide Schritte kombiniert
```

Zugriff auf Attribute und Methoden erfolgt mit Hilfe der Punkt-Notation. Objektattribute lassen sich via **Objekt.Attribut** und Objektmethoden via **Objekt.Methode()** ansprechen.

```
36      double tsbmw = bmw.tachoStand;
37      zweiCV.tachoStand = 44444;
38
39      tsbmw = bmw.leseTachoStand();
40      zweiCV.fahren(500);
```

Zweites wichtiges Konzept:

Gutes objektorientiertes Design zeichnet sich durch *Datenkapselung* aus. Der Zugriff auf Attribute sollte ausschließlich über Methoden erfolgen, um z. B. die zugesicherten Merkmale eines Objektes zu garantieren. Mit Hilfe von *Zugriffsrechten* (siehe 5.3) kann man dies erreichen.

4.3 Objekte sind Referenzdatentypen

Objekte werden „per Referenz“ verarbeitet, d. h. die Adresse des Objekts wird in einer Variablen abgelegt. Standardwert ist **`null`**. Erst zur Laufzeit durch Anwenden des **`new`**-Operators wird Speicherplatz für eine Instanz angefordert. Wieder muss man zwischen Identität und Äquivalenz unterscheiden.

Objektorientierung/Autos/04-Referenzen/Auto.java

```
31      Auto meinPKW = new Auto();
32      Auto bmw = new Auto();
33      Auto zweiCV;
38      System.out.println(bmw == meinPKW); // false
39      zweiCV = bmw;
40      System.out.println(bmw == zweiCV);  // true
41      zweiCV = null;
42      System.out.println(zweiCV == null); // true
```

Identität zweier Variablen vom Referenzdatentyp ist gegeben, wenn sie auf das gleiche Objekt verweisen:

Äquivalenz prüft, ob zwei verschiedene Objekte den gleichen Inhalt haben. Wir werden später (siehe 5.9) sehen, wie man dazu die Methode **`equals()`** verwenden kann.

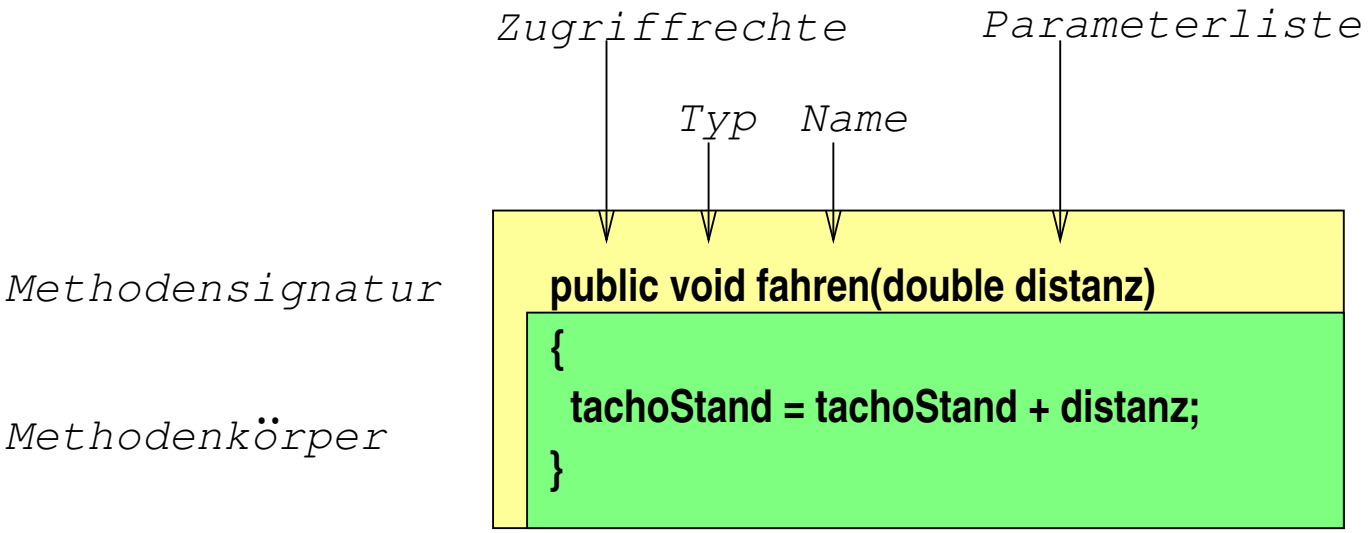
4.4 Über Methoden

Die Deklaration einer Methode enthält mindestens:

- *Typ*
- *Name*
- *Parameterliste* (evtl. leer)
- *Methodenkörper*

In der *Methodensignatur* sind optional zu verwenden:

- *Modifizierer* für Zugriffsrechte etc. (siehe auch 5.3)
- *Klauseln* zur Ausnahmeerzeugung etc. (siehe auch 6.4)



4.4.1 Typ einer Methode

Als Rückgabetyp einer Methode sind alle einfachen Datentypen und Referenzdatentypen erlaubt. Die letzte Anweisung des Methodenkörpers ist dann zwingend die **`return`**-Anweisung.

```
return ausdruck;
```

`ausdruck` muss zuweisungskompatibel zum Rückgabetyp sein.

Erlaubt ist also auch die Rückgabe von Arrays:

```
static int[] xxx() { return new int[]{ 4, 3 }; }
```

Der Ausdruck **`xxx()[1]`** würde dann 3 liefern.

Es gibt ferner den Rückgabetyp **`void`**, der keine **`return`**-Anweisung erwartet.

4.4.2 Parameter einer Methode

Parameter werden *call-by-value* übergeben. Für Parameter mit Referenzdatentypen haben wir dann die Semantik einer *call-by-reference*-Übergabe: Es wird eine Kopie des Verweises übergeben; referenziert wird jedoch das Originalobjekt.

Objektorientierung/Autos/05-Parameter/Auto.java

```
27  public void abschleppen(Auto auto, double distanz)
28  {
29    this.fahren(distanz);
30    auto.fahren(distanz); // Test call-by-reference
31    distanz = 0;          // Test call-by-value
32    auto = null;          // Test call-by-value
33  }
```

```
45    bmw.abschleppen(meinPKW, d);
```

Bei der Instantiierung eines Objekts wird automatisch eine Referenz namens **`this`** erzeugt. Sie zeigt auf das Objekt selbst. Man kann **`this`** in der Punktnotation verwenden, um deutlich zu machen, dass es sich um ein Merkmal des Objekts selbst handelt. Intern ergänzt der Compiler **`javac`** alle Memberattribute und -methoden durch Voranstellen von **`this`**.

In dem Beispiel soll durch die Anweisung **`this.fahren(distanz);`** die Methode auf das Objekt selbst angewendet werden. Bei **`auto.fahren(distanz);`** ist dagegen das übergebene Objekt mit der Referenz **`auto`** gemeint.

4.4.3 Überladen von Methoden

Der Compiler **javac** unterscheidet Methoden anhand der vollständigen Methodensignatur. Man kann also innerhalb einer Klasse mehrere Methoden gleichen Namens definieren. Insbesondere kann man Methoden *überladen*, d. h. gleichnamige Methoden mit unterschiedlichen Parameterlisten deklarieren.

Objektorientierung/Autos/06-Ueberladen/Auto.java

```
22   public void ausgeben()
23   {
24     System.out.println("Tachostand = " + tachoStand + " km");
25   }
27   public void ausgeben(String name)
28   {
29     System.out.println("Tachostand von " + name + " = " + tachoStand + " km");
30   }
```

Welche Methode gewählt wird, wird anhand der Signatur entschieden.

```
37     bmw.ausgeben();
38     bmw.ausgeben("BMW Z3");
```

4.5 Konstruktoren, Destruktor und Garbage Collector

Anwender: „Ich habe einen Gebrauchtwagenhandel. Ich möchte direkt den Tachostand festlegen. Was geschieht beim Instantiieren von Objekten?“
Entwickler: „*Konstruktoren* arbeiten mit dem new-Operator zusammen“
Anwender: „Was geschieht mit Objekten, die ich nicht mehr brauche?“
Entwickler: „Der *Destruktor* arbeitet mit dem *Garbage Collector* zusammen.“

Wir betrachten wieder die zwei Ebenen bei der Entwicklung mit Java:

1. Implementierung der Klasse
2. Instantiierung von Objekten nach Vorlage der Klasse

4.5.1 Konstruktoren

Wie kann man das Instantiieren von Objekten beeinflussen?

- Ein Konstruktor dient zur Generierung von Objekten.
- Er wird innerhalb des Klassenkörpers implementiert.
- Er heißt genauso wie die Klasse.
- Er wird bei der Instantiierung eines Objektes durch den Operator **new** aufgerufen.
- Mehrere Konstruktoren für Objekte einer Klasse sind zulässig. Java unterscheidet die Konstruktoren anhand der Signatur, also anhand der Datentypen aus der Parameterliste.
- Es besteht die Möglichkeit, Konstruktoren zu verketten.
- Wird vom Entwickler kein Konstruktor implementiert, so verwendet Java einen Standardkonstruktor.

Wir erweitern unsere Klasse **Auto** um zwei Konstruktoren:

Objektorientierung/Autos/07-Konstruktoren/Auto.java

```
11  //Konstruktoren
12  /** erzeugt ein Auto mit Standard-Initialisierungen.
13   */
14  public Auto()
15  {
16    this.tachoStand = 100;
17  }
18
19  /** erzeugt ein Auto mit spezifizierten Initialisierungen.
20   *  @param tachoStand der spezifizierte Tachostand. Sollte >= 0 sein.
21   */
22  public Auto(double tachoStand)
23  {
24    if (tachoStand >= 0)
25      this.tachoStand = tachoStand;
26    else
27      this.tachoStand = 100;
28  }
```

4.5.2 Destruktor und Garbage Collector

Was geschieht mit Objekten, die nicht mehr gebraucht werden?

- Zum Entfernen nicht mehr benötigter Objekte stellt Java den Mechanismus der *Garbage Collection* bereit.
- Der *Garbage Collector* sucht in gewissen Abständen nach Objekten, die nicht mehr referenziert werden.
- Werden solche Objekte gefunden, so werden sie aus dem Speicher entfernt.
- Vor dem endgültigen Entfernen wird noch die Methode **`finalize()`** *(Destruktor)* abgearbeitet.
- Wir können also beeinflussen, was das Objekt noch tun soll, bevor es zerstört wird, indem wir diese Methode implementieren. Beispielsweise kann man noch geöffnete Dateien schließen usw.
- Wir können aber kaum Einfluss darauf nehmen, *wann* das geschieht.
- Es kann nur einen Destruktor geben. Er hat die Signatur

```
protected void finalize()
```

- Es gibt einen Standarddestruktor.

- Um ein nicht mehr benötigtes Objekt für den Garbage Collector freizugeben, müssen wir dafür sorgen, dass keine Referenz mehr darauf verweist. Wir können dies beispielsweise so erreichen:

```
bmw = null;
```

- Der Aufruf

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

  ist eine Empfehlung an die Virtual Machine, den Garbage Collector jetzt aufzurufen.

4.6 Klassenattribute und -methoden

Anwender: „Ich möchte wissen, wie viele Auto-Objekte meine Anwendung verwaltet.“
„Gibt es in Java globale Funktionen und Variablen?“
Entwickler: „Nein. Aber wir können mit `static`-Merkmalen arbeiten.“

- Wir haben bislang *Objekt*attribute und -methoden kennen gelernt. Solche Merkmale sind an das Objekt gebunden.
- Ein *Klassenattribut* ist an die Klasse und nicht an eine bestimmte Instanz der Klasse gebunden.
- Es gibt nur eine Kopie eines Klassenattributes, unabhängig davon, wie viele Instanzen erzeugt werden.

- Klassenattribute werden in der Klassendeklaration durch das Schlüsselwort **`static`** gekennzeichnet.

Objektorientierung/Autos/08-static/Auto.java

```
8   // Klassenttribute
9   static int anzahl = 0;
```

- Zugriff erfolgt mit Hilfe der Punkt-Notation **`Klassenname.Attributname`**:

```
Auto.anzahl;
Color.black;
```

- Klassenmethoden werden in der Klassendeklaration ebenfalls durch das Schlüsselwort **`static`** gekennzeichnet.

```
36  // Klassenmethoden
37  public static int anzahlAutos()
38  {
39    return anzahl;
40  }

65  public static void main(String argv[])
```

- Zugriff erfolgt mit Hilfe der Punkt-Notation **`Klassenname.Methodenname()`**:

```
69    System.out.println(Auto.anzahlAutos()); // druckt 1

      Math.sin(3.0);
```

- Klassenmethoden können nur auf Klassenvariablen verweisen und nur andere Klassenmethoden aus der aktuellen Klasse aufrufen.

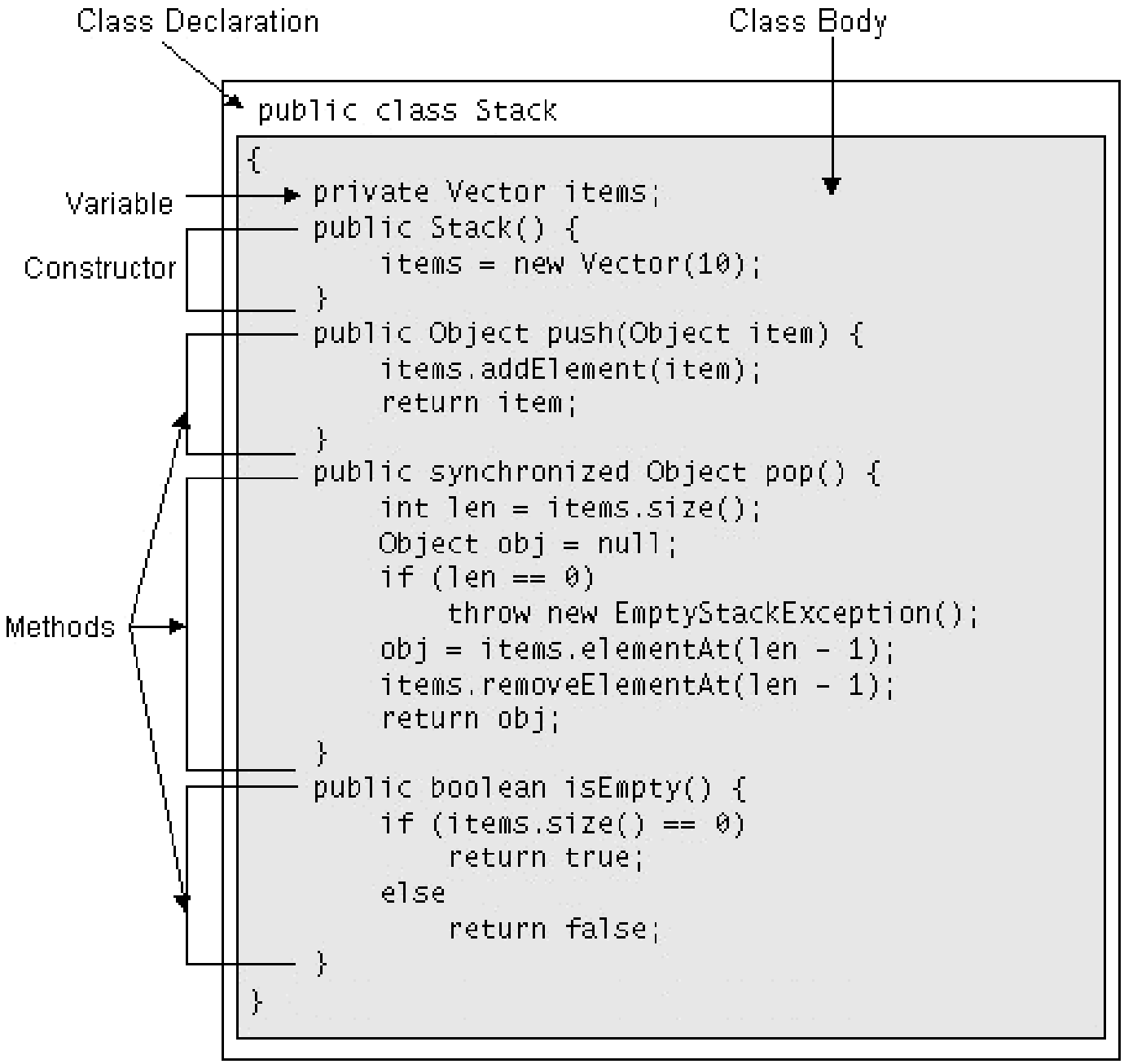
```
14  //Konstruktoren
15  public Auto()
16  {
17    this.tachoStand = 100;
18    anzahl = anzahl + 1;
19  }

30  // Destruktor
31  protected void finalize()
32  {
33    anzahl = anzahl - 1;
34  }
```

4.7 Gemischtes

Anwender: „Kann ich nochmal eine Klassendeklaration sehen?“

Entwickler: „Ja. Hier ein schönes Beispiel von SUN.“


Class Declaration
Class Body
Variable
Constructor
Methods
public class Stack
{
private Vector items;
public Stack() {
items = new Vector(10);
}
public Object push(Object item) {
items.addElement(item);
return item;
}
public synchronized Object pop() {
int len = items.size();
Object obj = null;
if (len == 0)
throw new EmptyStackException();
obj = items.elementAt(len - 1);
items.removeElementAt(len - 1);
return obj;
}
public boolean isEmpty() {
if (items.size() == 0)
return true;
else
return false;
}
}

Anwender: „Kann ich jetzt den Gültigkeitsbereich von Variablen besser verstehen?“
Entwickler: „Hoffentlich. Siehe 3.3.“

Anwender: „Wie kann ich Konstanten definieren?“
Entwickler: „Mit Hilfe des Schlüsselwortes `final`.“

Deklariert man ein Klassenattribut oder Objektattribut mit dem Modifizierer **`final`**, so kann das Attribut nach der erstmaligen Initialisierung nicht mehr geändert werden.

```
static final double ERDBESCHLEUNIGUNG = 9.81;
final double DICHTE_DES_OBJEKTS = 7.9;
```

Anwender: „Wie kann ich abfragen, von welcher Klasse ein Objekt erzeugt wurde?“
Entwickler: „Mit Hilfe der Methode **`getClass()`** oder mit dem Operator **`instanceof`**.“

Objektorientierung/Autos/09-instanceof/Auto.java

```
32    System.out.println(bmw.getClass());
33    if (bmw instanceof Auto)
34      System.out.println("ist ein Auto");
35    if ("FIAT" instanceof String)
36      System.out.println("ist ein String");
```

Anwender: „Wie komme ich an mathematische Standardfunktionen?“
Entwickler: „Verwende die Klasse **`java.lang.Math`**.“

```
double x;
x = Math.cos(4.0) * Math.exp(1.1) - Math.PI;
```

Anwender: „Java ist plattformunabhängig. Wie komme ich trotzdem an Informationen über das Betriebssystem?“

Entwickler: „Verwende die Klasse **`java.lang.System`**.“

4.8 Zusammenfassende Fragen

- Was sind Klassen?
- Was sind Objekte?
- Was sind Merkmale?
- Was ist der Unterschied zwischen Identität und Äquivalenz?
- Wie deklariert man Konstruktoren?
- Wie erzeugt man Objekte?
- Wie deklariert man Objektattribute / Objektmethoden?
- Wie greift man auf Objektattribute / Objektmethoden zu?
- Wie deklariert man Klassenattribute / Klassenmethoden?

- Wie greift man auf Klassenattribute / Klassenmethoden zu?
- Wie arbeitet der Garbage Collector?
- Was heißt Datenkapselung?
- Was sind Dokumentationskommentare?
- Schlüsselworte: `class`, `new`, `static`, `final`

5 Objektorientierung in Java, Teil 2

Inhalte / Ziele. *Wir lernen weitere Konzepte des objektorientierten Softwareentwurfs kennen. Wie konzentrieren uns dabei auf die Frage, wie in Java* Vererbung, Interfaces, Zugriffsrechte, Packages *usw. realisiert werden.*

5.1 Neues Konzept: Vererbung

Anwender: „Ich möchte mich spezialisieren. Ich habe Geländewagen, Sportwagen, Limousinen usw. im Angebot.“

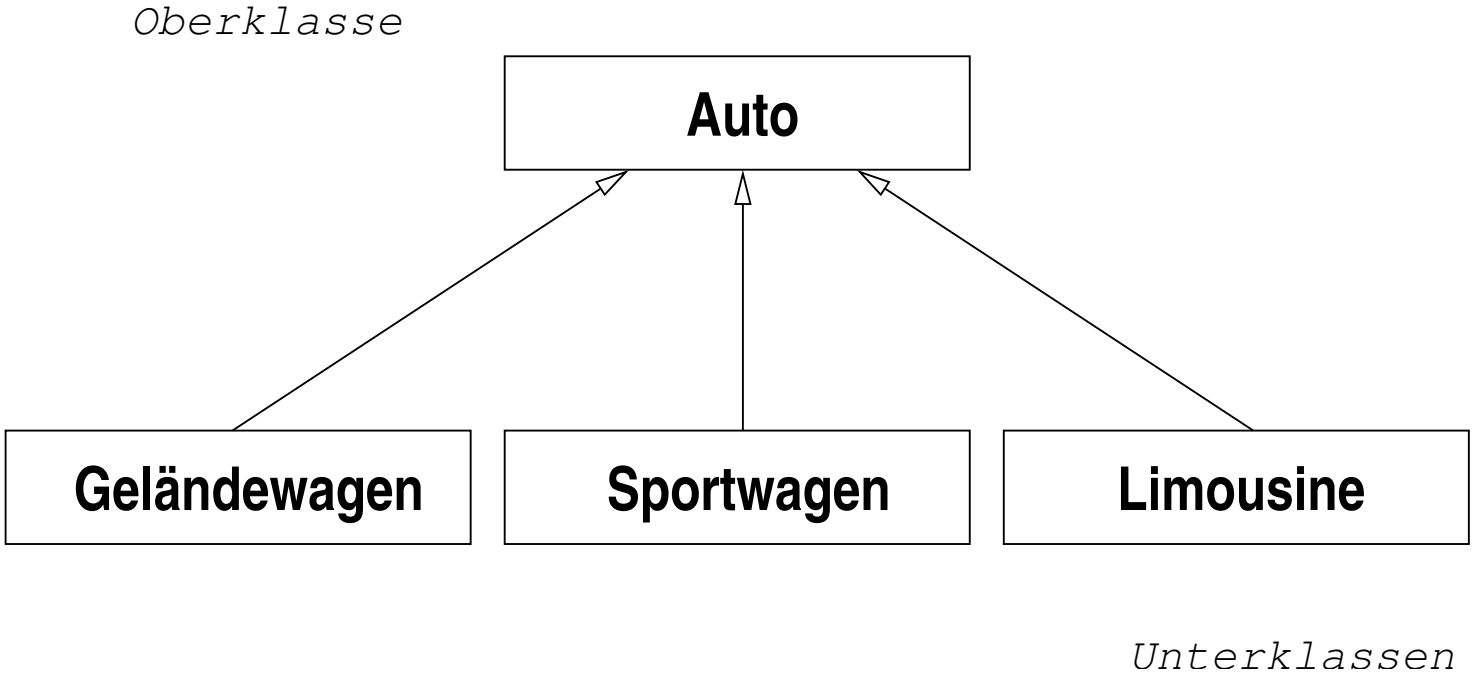
Entwickler: „Dafür gibt es in Java das Konzept der *Vererbung*.“

Wir haben gelernt, dass objektorientierte Sprachen wie Java Attribute und Methoden in Klassen vereinigen und dass Datenkapselung für die Einhaltung der Zusicherungen sorgt.

Drittes wichtiges Konzept:

Objektorientierte Sprachen bieten die Möglichkeit der Vererbung. Wir haben die Möglichkeit, Merkmale vorhandener Klassen auf neue Klassen zu übertragen („zu vererben“).

Bei der Abbildung der realen Welt nach Software lassen sich oft Hierarchien ausnutzen:



- Das Bild gibt eine *Klassenhierarchie* wieder. Die *Subklassen* **`Geländewagen`**, **`Sportwagen`** und **`Limousine`** *erben* von der Superklasse **`Auto`**. (Subklasse = Unterklasse, Superklasse = Oberklasse)

- Eine Subklasse *erbt* Attribute und Methoden von ihrer Superklasse (z. B. **`tachoStand`**, **`fahren(double distanz)`**. Sie kann diese verwenden, als wären sie in der Subklasse selbst deklariert.

Spezialisierung: In den Subklassen können die Methoden und Attribute der Superklasse entweder *wiederverwendet* oder *überlagert* werden. Zusätzlich dürfen die Subklassen um neue Methoden und Attribute *ergänzt* werden.

Weitervererbung: Subklassen können selber *weitervererbt* werden. Soll eine Klasse nicht mehr weiter vererbt werden können, kennzeichnet man dies durch das Schlüsselwort **`final`**. Genauso kann man das Überlagern von Attributen und Methoden durch eine Subklasse verbieten.

Mehrfachvererbung: Java unterstützt keine *Mehrfachvererbung*, also das Erben von mehreren Superklassen. Als Alternative gibt es die so genannten *Interfaces*.

Abstrakte Klassen: Es besteht die Möglichkeit, Klassen so zu definieren, dass erst die Subklassen die Definitionen mit Leben füllen müssen. Man kann diese Klassen erkennen an dem Schlüsselwort **`abstract`**. Solche Klassen können nicht instantiiert werden.

5.2 Vererbung in Java

Anwender: „Okay, wie erbt man jetzt in Java?“

Entwickler: „Mit dem Schlüsselwort **`extends`**.“

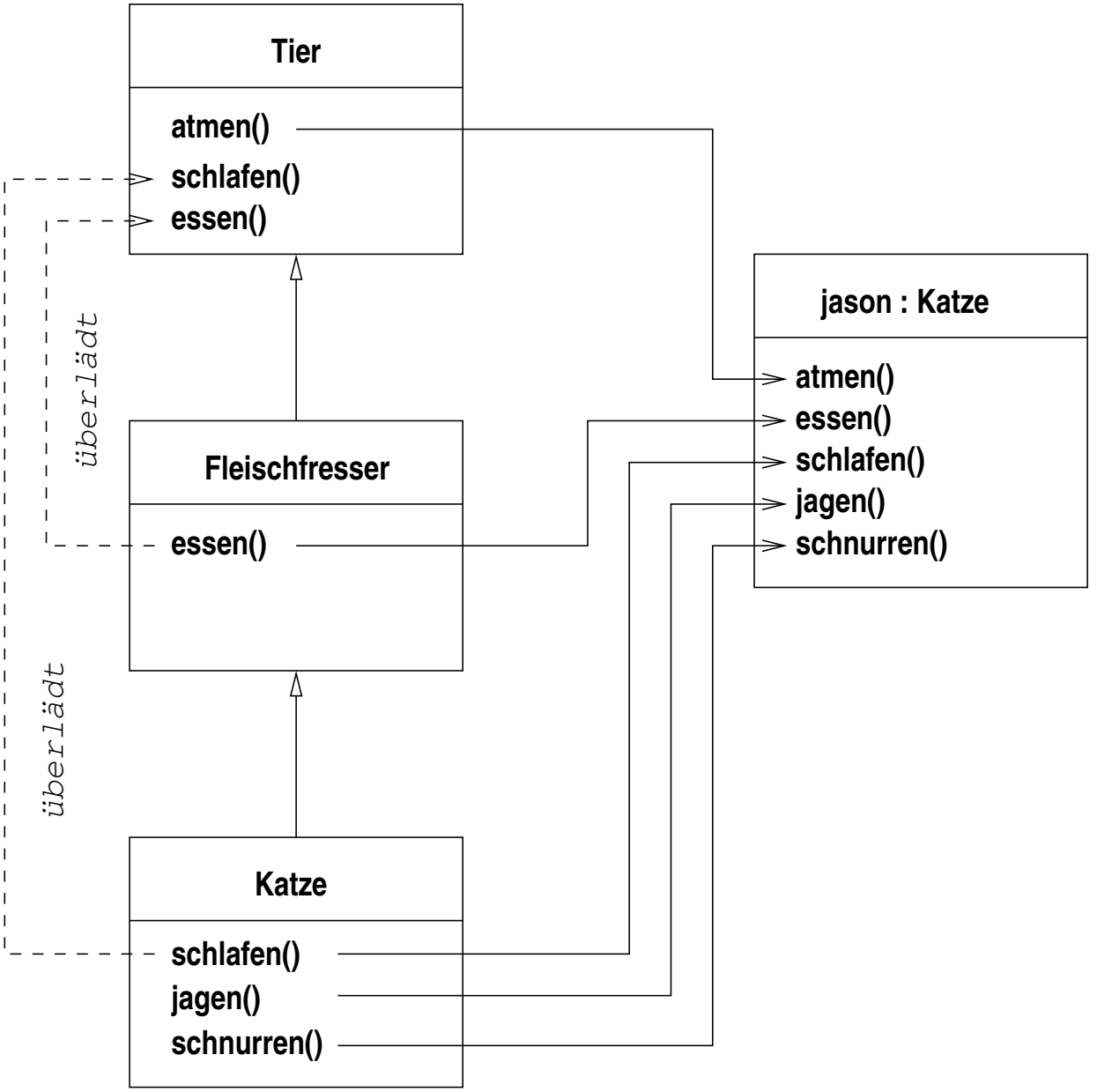
Objektorientierung/Autos/10-Vererbung/Limousine.java

```
6  public class Limousine extends Auto
7  {
9    int sicherheitsKategorie;
12   public Limousine()
13   {
14     this.sicherheitsKategorie = 3;
15   }
29   public void setzeSicherheitsKategorie(int sicherheitsKategorie)
30   {
31     if ((sicherheitsKategorie > 0) && (sicherheitsKategorie < 4))
32       this.sicherheitsKategorie = sicherheitsKategorie;
33   }
34
35   public int leseSicherheitsKategorie()
36   { return this.sicherheitsKategorie; }
39   public void ausgeben()
40   {
43   }
```

Quelltextanalyse

- Die Klasse **`Limousine`** erweitert die Klasse **`Auto`**.
- Ergänzung: Objektattribut **`sicherheitsKategorie`**
- Ergänzung: Objektmethoden **`setzeSicherheitsKategorie()`** (mit Zusicherung) und **`leseSicherheitsKategorie()`**
- Überlagerung: Die Methode **`ausgeben()`** der Superklasse **`Auto`** wird neu definiert und erhält eine andere Implementierung.
- Anderes Beispiel für **`extends`** siehe 2.5:

```
public class HelloApplet extends Applet
```


Tier
atmen()
schlafen()
essen()
Fleischfresser
essen()
Katze
schlafen()
jagen()
schnurren()
jason : Katze
atmen()
essen()
schlafen()
jagen()
schnurren()
überlädt
überlädt

5.2.1 Typkonvertierungen

Anwender: „Objekte sind Referenzdatentypen.“
„Kann ich eine Referenz vom Typ `Auto` auf eine Instanz vom Typ `Limousine` verweisen lassen?“
„Und umgekehrt?“
„Hat eine Typkonvertierung irgendwelche Auswirkungen?“

Entwickler: „Hm. Dazu muss ich etwas weiter ausholen.“

- Wir unterscheiden prinzipiell zwischen *erweiternden* Typkonvertierungen und *einschränkenden* Typkonvertierungen.
- Die Umwandlung eines Referenzdatentyps in eine seiner Superklassen nennen wir erweiternde Konvertierung.

Objektorientierung/Autos/10-Vererbung/Limousine.java

```
48      Limousine jaguar = new Limousine();
52      Auto meinPKW = new Auto();
56      meinPKW = jaguar;
```

- Erweiternde Konvertierungen werden in Java automatisch vorgenommen. Dies kennen wir schon bei einfachen Datentypen:

```
int m;
byte j = 120;
m = j;
```

- Folgende Anweisungen sind also erlaubt:

```
59    Auto zweiCV = new Auto();
60    zweiCV.abschleppen(jaguar, 200);
```

- Die Umwandlung eines Referenzdatentyps in eine seiner Subklassen nennen wir einschränkende Konvertierung. Sie muss mit dem Type-Cast-Operator vorgenommen werden und erfordert sorgfältige Behandlung:

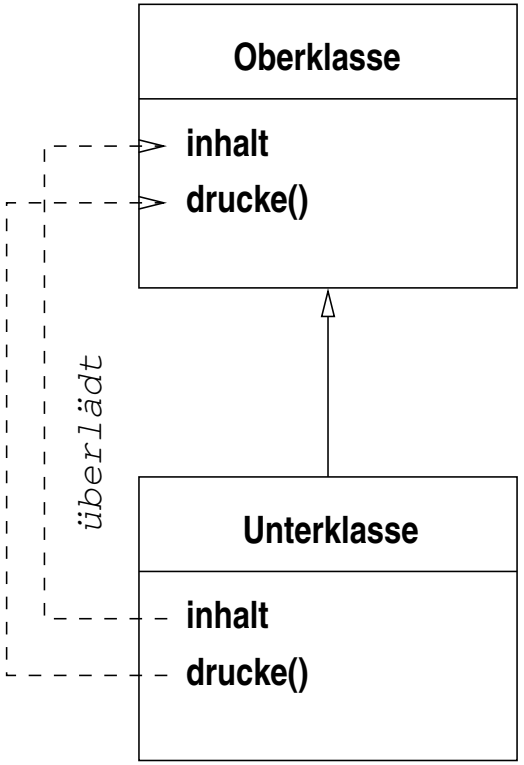
```
64    Limousine daimler = new Limousine(5000, 2);
68        daimler = (Limousine)zweiCV;  // Fehler
77        daimler = (Limousine)meinPKW; // OK, meinPKW ist Limousine
```

```
j = (byte)m;
```

5.2.2 Überlagerung und Binden

Anwender: „. . . Hat eine Typkonvertierung irgendwelche Auswirkungen?“
Entwickler: „Was für Auswirkungen genau?“
Anwender: „Zum Beispiel folgendes Problem: Welche Methoden ruft ein konvertiertes Objekt auf? Die der Subklasse oder die der Superklasse?“

Wir haben gesehen, dass eine Variable der Klasse **`Auto`** auf Objekte verschiedener Subklassen verweisen kann (z.B. **`Limousine`**, **`Sportwagen`**). Jedes Objekt einer Subklasse ist gleichzeitig ein Objekt der Superklasse. (Eine **`Limousine`** ist ein **`Auto`**. Ein **`byte`** ist ein **`int`**.)
Beim Übersetzen mit **`javac`** kann der Compiler dann noch nicht wissen, welche der überlagerten Methoden aufgerufen werden soll.
Wir nehmen jetzt an, dass in einer Subklasse **`Unterklasse`** Merkmale (= Attribute und Methoden) der Superklasse **`Oberklasse`** überladen werden.



Objektorientierung/BindTest/1/BindTest.java

```
28 class Oberklasse
29 {
30   String inhalt = "Attribut Oberklasse";
31
32   public void drucke() { System.out.println("Methode  Oberklasse" + "\n"); }
33 }
34
35 class Unterklasse extends Oberklasse
36 {
37   // Ueberlagerung eines Attributs
38   String inhalt = "Attribut Unterklasse";
39
40   // Ueberlagerung einer Methode
41   public void drucke() { System.out.println("Methode  Unterklasse" + "\n"); }
42 }
```

- Welche Merkmale sind gemeint, wenn ein Objekt **o** der Superklasse ein Objekt **u** der Subklasse zugewiesen bekommt?
- Welche Merkmale sind gemeint, wenn ein Objekt **u** der Subklasse als ein Objekt der Superklasse verwendet wird?
- Diese Fragen beantwortet das *Binden*:
  Dynamisches Binden liefert weiterhin das Merkmal der Subklasse.
  Statisches Binden liefert dagegen das Merkmal der Superklasse.
- Java verwendet dynamisches Binden für Methoden und statisches Binden für Attribute.

```
11      Unterklasse u = new Unterklasse();
12      System.out.println(u.inhalt); // druckt Attribut Unterklasse
13      u.drucke();                   // druckt Methode Unterklasse
14
15      Oberklasse o = new Oberklasse();
16      System.out.println(o.inhalt); // druckt Attribut Oberklasse
17      o.drucke();                   // druckt Methode Oberklasse
18
19      o = u;
20      System.out.println(o.inhalt); // druckt Attribut Oberklasse
21      o.drucke();                   // druckt Methode Unterklasse
22
23      System.out.println(((Oberklasse)u).inhalt); // druckt Attribut Oberklasse
24      ((Oberklasse)u).drucke();     // druckt Methode Unterklasse
```

5.2.3 Die Referenz super

- Die Referenz **super** können wir benutzen, um auf Merkmale einer Superklasse zu verweisen, die überlagert wurden.
- **super** veranlasst, dass die Suche nach der Methode, dem Attribut oder dem Konstruktor nicht in der aktuellen Klasse, sondern erst in der unmittelbaren Superklasse beginnt.

Objektorientierung/Autos/10-Vererbung/Limousine.java

```
39    public void ausgeben()
40    {
41      super.ausgeben();
42      System.out.println("Kategorie = " + sicherheitsKategorie);
43    }
```

Objektorientierung/BindTest/2-super/BindTest.java

```
37    public void zeigeInhalte()
38    {
39      System.out.println("Eigener Inhalt:       " + inhalt);
40      System.out.println("überlagerter Inhalt: " + super.inhalt);
41    }
44    public void druckeAlles()
45    {
46      drucke();
47      super.drucke();
48    }
```

```
11      Unterklasse u = new Unterklasse();
12      u.zeigeInhalte();
13      // Eigener Inhalt:      Attribut Unterklasse
14      // überlagerter Inhalt: Attribut Oberklasse
15      u.druckeAlles();
16      // Methode  Unterklasse
17      // Methode  Oberklasse
```

5.2.4 Verkettung von Konstruktoren

Anwender: „Wie kann ich `this` und `super` bei Konstruktoren einsetzen?“
Entwickler: „Dafür gibt es in Java Regeln:“

- Ist die erste Anweisung eines Konstruktors eine normale Anweisung — also kein Aufruf von **`this()`** oder **`super()`** — so ergänzt Java automatisch einen impliziten Aufruf von **`super()`** zum Aufruf des Standardkonstruktors der Superklasse. Nach Rückkehr von diesem Aufruf initialisiert Java die Instanzattribute der aktuellen Klasse und fährt mit der Ausführung der Anweisungen des aktuellen Konstruktors fort.
- Ist die erste Anweisung eines Konstruktors der Aufruf eines Superklassenkonstruktors über **`super()`**, ruft Java den gewünschten Superklassenkonstruktor auf. Nach Rückkehr von diesem Aufruf initialisiert Java die Instanzattribute der aktuellen Klasse und fährt mit der Ausführung der Anweisungen des aktuellen Konstruktors fort.
- Ist die erste Anweisung eines Konstruktors der Aufruf eines überladenen Konstruktors über **`this()`**, ruft Java den gewünschten Konstruktor auf und führt danach einfach die Anweisungen im aktuellen Konstruktor aus. Der Aufruf des Superklassenkonstruktors vollzieht sich explizit oder implizit innerhalb des überladenen Konstruktors, so dass die Initialisierung der Instanzvariablen bereits dort stattgefunden hat.

Instantiierung von „oben nach unten“.

Objektorientierung/Autos/10-Vererbung/Limousine.java

```
17  public Limousine(double tachoStand, int sicherheitsKategorie)
18  {
19    super(tachoStand);
20    if ((sicherheitsKategorie > 0) && (sicherheitsKategorie < 4))
21      this.sicherheitsKategorie = sicherheitsKategorie;
22    else
23      this.sicherheitsKategorie = 3;
24  }
```

5.3 Datenkapselung

Designmerkmale einer objektorientierten Sprache wie Java:

- Klassen vereinigen Attribute, Methoden und Zusicherungen.
- Vererbung erlaubt das Übertragen von Merkmalen vorhandener Klassen auf neue Klassen.
- Datenkapselung durch Zugriffsrechte.

Klassen, ihre Attribute und Methoden können durch Zugriffsbezeichner modifiziert werden. Durch diese Bezeichner kann man die Zugriffsrechte und -möglichkeiten genauer definieren.

In Java unterscheidet man vier Zugriffsrechte:

Zugriffsrecht	Schlüsselwort	Beispiel
private	`private`	`private int i;`
protected	`protected`	`protected int i;`
package		`int i;`
public	`public`	`public int i;`

Man benutzt Zugriffsrechte, um Datenkapselung zu erzielen. Um z. B. zu vermeiden, dass Objektattribute unkontrolliert von außerhalb geändert werden können, kann man das Schlüsselwort `private` verwenden.

Objektorientierung/Geld/1/Sparstrumpf.java

```
10   // Auf k kann von aussen nicht zugegriffen werden.
11   private float k;

27   // Eine Methode, die von aussen aufgerufen werden kann.
28   public float auszahlen(float r)

41   // Eine Methode, die nicht von aussen aufgerufen werden kann.
42   private void umrechnenYen()
```

Objektorientierung/Geld/1/Test.java

```
13     bar = s.auszahlen(1000);

15     // System.out.println("Kapital = " + s.k); // Fehler b. Compilieren
16     // s.umrechnenYen();                       // Fehler b. Compilieren
```

5.4 Packages

- Ein Package ist eine Sammlung verwandter Klassen, die Zugriffsrechte und Namensgebung verwaltet.
- Es gibt im JDK bereits eine Reihe nützlicher Packages, z. B. **`java.applet`**, **`java.awt`**, **`java.io`**, **`java.net`** usw.
- Das Package **`java.lang`** wird automatisch eingebunden. Es enthält neben den elementaren Sprachkonstrukten auch die Klassen **`Math`** (4.7), **`System`** (4.7), **`Thread`** (11), **`Object`** (5.9), **`Class`** (5.9) sowie diverse Wrapperklassen (5.10) für einfache Datentypen.
- Man kann natürlich auch eigene Packages erstellen.
  - Man kann Packages einbinden durch die Anweisung **`import`**.

Objektorientierung/Packages/Test.java

```
1 import autos.*;
```

  - Man kann Packages erzeugen durch die Anweisung **`package`**.

Objektorientierung/Packages/autos/Auto.java

```
1 package autos;
```

Objektorientierung/Packages/autos/Limousine.java

```
1 package autos;
```

- Die Quelltextedateien sollten in einem gleichnamigen Unterverzeichnis liegen.
- Man kann Packages auch ohne **`import`** verwenden, wenn man den Klassennamen vollständig qualifiziert.

Objektorientierung/Packages/Test2.java

```
10      autos.Limousine jaguar = new autos.Limousine();
```

- Trifft der Compiler **`javac`** auf eine **`import`**-Anweisung, so durchsucht er den **`CLASSPATH`**, um die Verzeichnisse mit den Packages zu finden.
- Gibt man beim Aufruf keinen speziellen Pfad an, so wird zuerst das aktuelle Verzeichnis und dann das Standardverzeichnis der mitgelieferten JDK-Packages durchsucht.
- Benötigt der Compiler Klassen, die noch nicht compiliert sind, so werden diese *automatisch* mit übersetzt.

5.5 Zugriffsrechte bei Vererbung und Packages

Siehe **`Objektorientierung/Zugriff/Test.java`**, **`Objektorientierung/Zugriff/paket/A.java`**, **`Objektorientierung/Zugriff/paket/B.java`**, **`Objektorientierung/Zugriff/C.java`**, **`Objektorientierung/Zugriff/paket/ASub1.java`**, **`Objektorientierung/Zugriff/ASub2.java`**.

Die Ergebnistabelle fasst die Zugriffsrechte zusammen:

	Zugriff von			
auf A/M mit Zugriffsrecht	Klasse	Unterklasse (aber nicht Package)	Package	fremde Klasse
private	*			
protected	*	*	*	
package	*		*	
public	*	*	*	*

- Zugriffsrechte wirken auf Attribute und Methoden gleich.
- Das Attribut **`public`** ist zusätzlich auch bei der Klassendeklaration von Bedeutung.
- Nur Klassen, die als **`public`** deklariert wurden, sind außerhalb ihres Packages sichtbar.
- Pro Java-Quelltextdatei darf nur eine **`public`**-Klasse angelegt werden.

5.6 Der Modifizierer `final`

- Modifiziert man ein *Attribut* mit dem Schlüsselwort **`final`**, so darf sein Wert nicht mehr geändert werden. Das Attribut kann man dann als Konstante verwenden.
- Modifiziert man eine *Methode* mit dem Schlüsselwort **`final`**, so darf sie nicht mehr überlagert werden. Das erspart dem Compiler das dynamische Binden und kann zur Geschwindigkeitsoptimierung benutzt werden.
- Modifiziert man eine *Klasse* mit dem Schlüsselwort **`final`**, so darf sie keine Subklassen mehr bilden.

5.7 Abstrakte Klassen

Anwender: „Ich möchte ein Wertpapierdepot verwalten. Alle Wertpapiere sollen über eine Methode `berechneWert()` verfügen. Der Wert eines Sparbuchs berechnet sich aber ganz anders als der von Aktien.“

Entwickler: „Wir sollten eine abstrakte Klasse `Wertpapier` mit einer abstrakten Methode `berechneWert()` deklarieren. Jede Klasse, die davon erben will, *muss* dann eine konkrete Implementierung vornehmen.“

Objektorientierung/Geld/2/Wertpapier.java

```
6  public abstract class Wertpapier
7  {
8    public abstract float berechneWert();
9  }
```

Quelltextanalyse

- Das Schlüsselwort **`abstract`** kennzeichnet eine Klasse oder Methode als abstrakt.
- Bei einer abstrakten Methode ersetzt ein Semikolon den Klassenkörper.
- Eine abstrakte Klasse kann neben abstrakten Methoden durchaus normale (nicht abstrakte) Attribute und Methoden enthalten.
- Von einer abstrakten Klasse können keine Instanzen gebildet werden.

- Um Objekte zu erzeugen, müssen wir zunächst eine Subklasse bilden:

Objektorientierung/Geld/2/Sparbuch.java

```
6  public class Sparbuch extends Wertpapier
25   public float berechneWert()
26   {
27     float wert = k;
28     for (int i = 1; i <= l; i++)
29       wert = wert * (1 + p);
30     return wert;
31   }
```

Objektorientierung/Geld/2/Aktie.java

```
17   public float berechneWert()
18   {
19     return anzahl * kurswert;
20   }
```

Quelltextanalyse

- Wir *müssen* dann die abstrakte Methode **`public float berechneWert()`** implementieren.
- Für eine abgeleitete Klasse **`Aktie`** kann eine ganz andere Implementierung der Methode vorgenommen werden.
- Von den Subklassen dürfen wir Objekte instantiieren.

Objektorientierung/Geld/2/Depot.java

```
10      Sparbuch sb = new Sparbuch(5112.92f, 0.0125f, 5);
11      System.out.println(sb.berechneWert());
```

5.8 Interfaces

Anwender: „Die Steuererklärung steht an. Ich möchte für meine Autos, Sparbücher und Hunde eine einheitliche Methode `berechneSteuer()` einführen. Leider gehören meine Klassen zu *verschiedenen* Vererbungshierarchien. Was tun?“

Entwickler: „Verwende *Interfaces*.“

- Eine grundsätzlich Designentscheidung der Java-Erfinder war, keine *Mehrfachvererbung* zu unterstützen.

  Das heißt, dass jede Klasse nur von einer einzigen Oberklasse erben kann.

- Nehmen wir einmal an, dass wir die folgenden Vererbungshierarchien aufgebaut haben:
  - **Object** $\longrightarrow$ **Tier** $\longrightarrow$ **FleischFresser** $\longrightarrow$ **Hund**
  - **Object** $\longrightarrow$ **Auto** $\longrightarrow$ **Limousine**
  - **Object** $\longrightarrow$ **Wertpapier** $\longrightarrow$ **Sparbuch**
  - Siehe **Objektorientierung/Interfaces**.

- Nun möchten wir unsere Steuererklärung in Java programmieren. Zu versteuern sind alle Instanzen der Klassen **`Hund`**, **`Auto`** und **`Sparbuch`**.
- Es wäre nützlich, wenn für jede dieser Klassen eine Methode **`berechneSteuer()`** zur Verfügung stünde.
- Mit der Hilfe von *Interfaces* bietet Java eine Konstruktion an, um solche Anforderungen zu lösen:
- Wir erstellen zunächst ein Interface **`Steuer`**.

Objektorientierung/Interfaces/Steuer.java

```
6  public interface Steuer
7  {
8    public double berechneSteuer();
9  }
```

- Dort listen wir die Methoden auf, die implementiert werden müssen, wenn wir das Interface benutzen wollen.
- Alle Methoden müssen **`public`** sein.
- Interfaces dürfen keine nicht abstrakten Methoden enthalten.
- Eine Klasse, die das Interface benutzen will, zeigt dies durch das Schlüsselwort **`implements`** an. Wir müssen dann alle Methoden des Interfaces implementieren.

Objektorientierung/Interfaces/Hund.java

```
6  public class Hund extends Fleischfresser implements Steuer
7  {
18   public double berechneSteuer()
19   {
20     double steuerBetrag = 120;
21     return steuerBetrag;
22   }
23 }
```

- Siehe **Objektorientierung/Interfaces/Sparbuch.java**.
- Siehe **Objektorientierung/Interfaces/Auto.java**.
- Siehe **Objektorientierung/Interfaces/Limousine.java**.

- Nun verfügen alle Objekte dieser Klassen über eine Methode **berechneSteuer()**, die man so verwenden kann:

Objektorientierung/Interfaces/Test.java

```
10    Hund bello = new Hund();
11    Sparbuch sb = new Sparbuch(1200, 0.015f, 1);
15    double steuern;
16    steuern = bello.berechneSteuer();
17    System.out.println(steuern);
```

- Eine Klasse kann mehrere Interfaces implementieren.

- Wann benutzen wir Interfaces?
  - Wenn wir ereignisgesteuerte Benutzeroberflächen programmieren wollen (z.B. **`ActionListener`**, siehe 7.6.3).
  - Wenn wir *Threads* programmieren wollen (**`Runnable`**, siehe 11.2).
- Wenn eine Anwendung sehr viele Konstanten definiert, kann man diese in einem Interface zusammenfassen. Jede Klasse, die diese Konstanten verwenden will, kann das Interface dann benutzen.

Objektorientierung/Interfaces/Defs.java

```
6  public interface Defs
7  {
14   static final int BUFFERLENGTH = 8;
15   static final int DEFAULTTAG = 0;
16   static final int DEFAULTERR = -1;
17 }
```

Diese Konstanten haben immer den Status **`static final`**, egal ob dies explizit angegeben wird oder nicht.

5.9 `Object` und `Class`

- Die Klasse **`Object`** aus dem Package **`java.lang`** ist die Mutter aller Klassen.
- Eine Klassendeklaration ohne die Klausel **`extends`** erweitert der Compiler **`javac`** automatisch zu **`extends Object`**.
- Die Klasse **`Object`** verfügt über einige nützliche Methoden, die wir bei der Defintion eigener Klassen überlagern können (siehe auch API-Referenz).
  - **`toString()`**: Gibt eine Darstellung des Objekts als String zurück.
  - **`equals()`**: Testet auf Äquivalenz zweier Objekte (siehe 4.3)
  - **`clone()`**: Erzeugt ein neues Objekt derselben Klasse. Klassen, die eine **`clone`**-Methode implementieren, sollten dies durch **`implements Cloneable`** anzeigen.
  - **`finalize()`**: Arbeitet mit dem Garbage Collector zusammen (siehe 4.5.2) .
  - **`getClass()`**: Gibt eine Referenz auf die Klasse zurück, von der das Objekt abgeleitet ist.

Objektorientierung/Autos/11-equals-clone-toString/Auto.java

```
6  public class Auto implements Cloneable
7  {
50  // ueberlagert die Methode equals() von Object
51    public boolean equals(Object obj)
52    {
53      if (obj != null && obj instanceof Auto)
54        return this.tachoStand == ((Auto)obj).tachoStand;
55      return false;
56    }
58    // ueberlagert die Methode clone() von Object
59    public Object clone()
60    {
61      return new Auto(this.tachoStand);
62    }
64    // ueberlagert die Methode toString() von Object
65    public String toString()
66    {
67      return "Instanz von Auto, tachoStand = " + tachoStand;
68    }
```

```
71    public static void main(String argv[])
72    {
73      Auto meinPKW = new Auto();
74      Auto bmw = new Auto();
75      Auto zweiCV;
76
77      // Identitaet und Aequivalenz mit == und equals()
78      meinPKW.fahren(100);
79      bmw.fahren(100);
80      System.out.println(bmw == meinPKW);       // false
81      System.out.println(bmw.equals(meinPKW)); // true
82      bmw.fahren(100);
83      System.out.println(bmw.equals(meinPKW)); // false
84      System.out.println(bmw.equals("Hallo")); // false
85      zweiCV = (Auto)bmw.clone();               // clone erfordert Cast!
86      System.out.println(bmw==zweiCV);          // false
87      System.out.println(bmw.equals(zweiCV));  // true
88      zweiCV = null;
89      System.out.println(bmw.equals(zweiCV));  // false

92      System.out.println(bmw);              // Instanz von Auto, tachoStand = 300.0
93      System.out.println(new Object());       // gibt z.B. java.lang.Object@df6ccd
```

- Klassen im Java-Quellcode werden zur Laufzeit durch Instanzen der Klasse **`Class`** aus dem Package **`java.lang`** verwaltet.
- Zu jeder Klasse, die wir verwenden, gibt es ein **`Class`**-Objekt.
- Dieses ist für die Erstellung von Instanzen dieser Klasse verantwortlich.
- Die Klasse **`Class`** verfügt ebenfalls über einige nützliche Methoden (siehe API-Referenz).

Objektorientierung/Autos/12-Ahnen/Ahnen.java

```
10  static void zeigeAhnenGalerie(Object obj)
11  {
12    System.out.print("Der␣Stammbaum␣von␣");
13    Class tmp = obj.getClass();
14    while (tmp != null)
15      {
16        System.out.println("␣␣" + tmp.getName());
17        tmp = tmp.getSuperclass();
18      }
19  }
```

5.10 Spezielle Klassen

Anwender: „Wie sind einfache Datentypen wie `int` oder `double` in die Klassenhierarchie einzuordnen?“

Entwickler: „Mit Hilfe von *Wrapperklassen*.“

- Eine Designentscheidung von Java ist es, einfache Datentypen *nicht* als Objekte zu betrachten.
- Es gibt jedoch Möglichkeiten, Variablen von einfachen Datentypen in eine objektorientierte Hülle einzuwickeln:

Einfacher Datentyp	Wrapperklasse
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`
`void`	`Void`

- Mit Hilfe dieser Klassen aus dem Package **`java.lang`** lassen sich einfache Datentypen in Referenzdatentypen umwandeln.
- Insbesondere ist die Konvertierung von und nach **`String`** durch entsprechende Methoden der Klassen erleichtert.

Objektorientierung/Wrapperklassen/WrapBsp.java

```
10      double dx = 10;
11      Double dxw = new Double(dx);
12      String sx = "3.1415927";
13      Double sxw = new Double(sx);
14      double pi = sxw.doubleValue();
```

5.11 Zusammenfassende Fragen

- Die folgende Tabelle enthält alle Schlüsselworte der Sprache Java. Welche kennen wir schon und welche Bedeutung haben sie? (Im unteren Abschnitt sind neuere Schlüsselwörter, die bisher noch keine Bedeutung haben.)

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

- Was sind die drei wichtigsten Konzepte des objektorientierten Softwareentwurfs?
- Was bedeutet Weitervererbung?

- Was bedeutet Mehrfachvererbung?
- Was ist der Unterschied zwischen Ergänzung und Überlagerung?
- Was ist der Unterschied zwischen erweiternder und einschränkender Konvertierung?
- Was bedeutet Binden?
- Was ist der Unterschied zwischen statischem und dynamischem Binden?
- Was sind Zugriffsrechte und wozu werden sie eingesetzt?
- Wie sind in Java die Zugriffsrechte geregelt?
- Was sind Packages? Wieviele Packages gehören zur aktuellen Java-Version?
- Was sind abstrakte Klassen?
- Was sind Interfaces?
- Welche Bedeutung haben die Klassen **`Object`** und **`Class`**?
- Was sind Wrapper-Klassen?

6 Ausnahmen

Inhalte / Ziele. *Das Konzept der Ausnahmebehandlung in Java stellt ein elegantes und mächtiges Instrument zum Umgang mit Laufzeitfehlern zur Verfügung. Wir werden in den weiteren Abschnitten diverse Packages des Java-API vorstellen. Da eine Vielzahl von Klassen die Behandlung von Ausnahmen erfordert, ist es wichtig, sich mit dem Konzept vertraut zu machen.*

6.1 Was sind Ausnahmen?

- Eine *Ausnahme (exception)* ist ein Ereignis, das zur Laufzeit eines Programmes auftritt und den normalen Ablauf der Anweisungen unterbricht.
- Ausnahmen können beispielsweise auftreten bei
  - Rechenoperationen (**`ArithmeticException`**)
  - Zugriffen auf nicht vorhandene Feldelemente (**`ArrayIndexOutOfBoundsException`**)
  - Lese- und Schreiboperationen (**`IOException`**)
- Tritt in einer Java-Methode ein Fehler auf, so kann diese Methode ein Objekt der Klasse **`Exception`** (oder einer Subklasse) erzeugen und an das Laufzeitsystem (Interpreter) weitergeben. In Java wird dies als „werfen einer Ausnahme“ (*„throw an exception“*) bezeichnet.

- Wirft eine Methode eine Ausnahme, so tritt das Laufzeitsystem in Aktion, um diese Ausnahme zu behandeln. Es wird nach einem *exception handler* gesucht. Ein *exception handler* „fängt eine Ausnahme“ *(„catch an exception“)*.
- Die Sprache Java unterscheidet zwischen *allgemeinen Ausnahmen* (z. B. fehlende Dateien, nicht verfügbare Rechner) und *Laufzeitausnahmen* (das sind alle Nachfahren von **`RuntimeException`**, z. B. Division durch Null bei ganzzahligen Datentypen, Zugriff auf nicht vorhandene Feldelemente).
- Allgemeine Ausnahmen *müssen*, Laufzeitausnahmen *können* abgefangen werden.
- Wenn wir keinen *exception handler* implementieren, bricht das Programm mit einer entsprechenden Fehlermeldung ab.

6.2 Behandlung einer Laufzeitausnahme

Exceptions/ExcBsp1.java

```
10      int[] a = new int[2];
11      a[0] = 1;
12      a[1] = 2;
13
14      System.out.println(a[2]);
15
16      System.out.println("Mich␣sieht␣man␣nur,␣wenn␣keine␣Ausnahmen␣auftraten.");
```

Quelltextanalyse

- Das obige Programm provoziert eine Laufzeitausnahme:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
        at ExcBsp1.main(ExcBsp1.java:14)
```

Tritt in einem Java-Programm eine Ausnahme auf, so wird ein Objekt der Klasse **`Exception`** (oder einer erbenden Klasse) erzeugt und an das Laufzeitsystem (Interpreter) weitergeben. Wir bezeichnen dies als „werfen einer Ausnahme“ („throw an exception“).

In unserem Fall können wir die Ausnahme vom Laufzeitsystem behandeln lassen oder sie selbst abfangen („catch an exception“).

Exceptions/ExcBsp2.java

```
14      try
15        {
16          System.out.println(a[2]);
17          System.out.println("Mich␣sieht␣man␣nur,␣wenn␣keine␣Ausnahmen␣auftraten.");
18        }
19      catch (ArrayIndexOutOfBoundsException aioobe)
20        {
21          System.out.println("Habe␣gefangen:␣" + aioobe);
22          System.out.println("Die␣Message␣lautet:␣" + aioobe.getMessage());
23          aioobe.printStackTrace();
24         }
25
26      System.out.println("Sieht␣man␣mich?");
```

Quelltextanalyse

- Wir verwenden hier die **`try`**-**`catch`**-Kontrollstruktur.
- Die Anweisungen, die eine Ausnahme auslösen können, befinden sich innerhalb eines **`try`**-Blockes.
- Tritt zur Laufzeit eine Ausnahme auf, so springt der Interpreter an das Ende dieses **`try`**-Blockes.
- Im Anschluss daran gibt es einen oder mehrere **`catch`**-Blöcke. Jeder dieser Blöcke behandelt eine bestimmte Ausnahme.
- In unserem Fall erhalten wir eine Instanz der Klasse **`ArrayIndexOutOfBoundsException`**. Diese enthält diverse Zusatzinformationen.

- Sind mehrere **`catch`**-Blöcke vorhanden, wird in den ersten Block gesprungen, der Exceptions der geworfenen Klasse *oder einer Oberklasse* fängt.
- Deswegen sollten immer zuerst spezielle und danach allgemeinere Exceptions gefangen werden.

Exceptions/ExcBsp3.java

```
25    finally
26      {
27        System.out.println("Mich␣sieht␣man.␣Immer.");
28      }
```

Quelltextanalyse

- An die **`catch`**-Blöcke kann man noch einen **`finally`**-Block anschließen.
- Dort kann man Anweisungen eintragen, die immer (egal, welche Ausnahme auftrat, auch wenn gar keine auftrat) ausgeführt werden sollen. Die Anweisungen werden sogar dann ausgeführt, wenn im **`try`**- oder **`catch`**-Block eine **`return`**-Anweisung steht!

Allgemeine Syntax der **try-catch-finally**-Struktur:

```
try {
  Anweisungen
  }
catch (AException a) {
  Anweisungen zum Behandeln von AException a
  }
catch (BException b) {
  Anweisungen zum Behandeln von BException b
  }
finally {
  von Ausnahmen unabhängige Anweisungen
  }
```

6.3 Mehr über den `try`-Block

- Tritt innerhalb eines **`try`**-Blockes eine Ausnahme auf, so werden die nachfolgenden Anweisungen übergangen.
- Dies hat eventuell Auswirkungen auf das Initialisieren von Variablen.
- Welche Möglichkeiten gibt es, dieses Problem zu umgehen?

Exceptions/ExcBsp4.java

```
13      int b, c;
15      try
16        {
17          b = 5;
20        }
32      //c = b;   // FEHLER: variable b might not have been initialized
```

6.4 Eine Ausnahme werfen

Konstruktoren und Methoden können Ausnahmen werfen.

Sie zeigen dies durch eine Klausel wie **`throws Exception`** an.

Diese Klausel wird auch vom Dokumentationsgenerator **`javadoc`** registriert, um dem Benutzer einer Klasse anzuzeigen, dass er ggf. eine **`try-catch`**-Umgebung aufbauen muss.

Wir demonstrieren dies am Beispiel **Sparbuch**. Es wird jedes mal, wenn die Zusicherung $k \geq 0$ in Gefahr ist, eine Ausnahme geworfen.

Exceptions/Geld/3/Sparbuch.java

```
18  public Sparbuch(float k, float p, int l) throws Exception
19  {
20    if (k < 0 || p < 0 || l < 0)
21      throw new Exception("Fehler beim Erzeugen des Sparbuchs.");

28  public float abheben(float s) throws Exception
29  {
30    if (k < s)
31      throw new Exception("Fehler beim Abheben.");

39    try
40      {
41        Sparbuch sb = new Sparbuch(100, 0.01f, 3);
42        sb.abheben(200);
43      }
44    catch (Exception e)
45      {
46        System.out.println(e);
47      }
```

Quelltextanalyse

- Sowohl der Konstruktor als auch die Methode **abheben()** zeigen durch die Klausel **throws Exception** an, dass sie eine allgemeine Ausnahme werfen könnten.

- Für den Anwender einer Klasse heißt das, dass er das Instantiieren und das Aufrufen der Methode in eine **`try`**-**`catch`**-Umgebung einbetten *muss*.
- Mit dem Schlüsselwort **`throw`** können wir eine Ausnahme werfen. In diesem Beispiel wird eine neue **`Exception`** erzeugt, die wir mit einem bestimmten String versehen.
- In der **`catch`**-Umgebung wird dieser String dann ausgedruckt.

6.5 Eine spezielle Ausnahme erzeugen

Man kann in der Fehlerbehandlung noch viel stärker differenzieren, indem man eigene Klassen von Ausnahmen erzeugt.

Exceptions/Geld/4/SparbuchException.java

```
6  public class SparbuchException extends Exception
7  {
8    public static final int UNBEKANNT = 0;
9    public static final int KONSTRUKTOR = 1;
10   public static final int ABHEBEN = 2;
11
12   private int fehlerNummer;
13
14   public SparbuchException()
15   {
16     super();
17     this.fehlerNummer = UNBEKANNT;
18   }
19
20   public SparbuchException(int fn)
21   {
22     super();
23     switch (fn)
24       {
25         case KONSTRUKTOR: this.fehlerNummer = fn; break;
26         case ABHEBEN: this.fehlerNummer = fn; break;
27         default: this.fehlerNummer = UNBEKANNT;
28       }
29   }
30
31   public int leseFehlerNummer() { return fehlerNummer; }
32
33   public String toString()
34   {
35     switch (fehlerNummer)
36       {
37         case KONSTRUKTOR: return "Sparbuch: Fehler beim Erzeugen des Sparbuchs";
38         case ABHEBEN: return "Sparbuch: Fehler beim Abheben";
39         default: return "Sparbuch: unbekannter Fehler";
40       }
41   }
42 }
```

Quelltextanalyse

- Mit der Klausel **`extends Exception`** zeigt die Klasse **`SparbuchException`** an, dass sie als spezielle Ausnahme zu verwenden ist.
- Wir erweitern hier die allgemeine Ausnahme um einen speziellen Fehlercode mit dem Attribut **`fehlerNummer`** und um die Methode **`leseFehlerNummer()`**.

Exceptions/Geld/4/Sparbuch.java

```
28  public float abheben(float s) throws SparbuchException
29  {
30    if (k < s)
31      throw new SparbuchException(SparbuchException.ABHEBEN);
```

Quelltextanalyse

- Der Konstruktor von **`Sparbuch`** und die Methode **`abheben()`** werfen nun die spezialisierte Ausnahme **`SparbuchException`**.
- Beim Erzeugen der Ausnahme mit **`new`** wird nun eine entsprechende Fehlernummer übergeben.
- In der **`catch`**-Anweisung kann diese dann durch Aufruf der Methode **`leseFehlerNummer()`** ausgewertet werden.

6.6 Zusammenfassende Fragen

- Was sind Ausnahmen und was geht uns das an?
- Was ist der Unterschied zwischen allgemeinen Ausnahmen und Laufzeitausnahmen?
- Was bedeutet Werfen und Fangen einer Ausnahme?
- Wie fängt man Ausnahmen?
- Welche Möglichkeiten hat man, mit einer gefangenen Ausnahme umzugehen?
- Wie realisiert man das Einbetten in die `try`-`catch`-`finally`-Kontrollstruktur?
- Was ist dabei besonders zu beachten?
- Wie kann man herausfinden, wann ein Einbetten notwendig ist?
- Wie wirft man Ausnahmen?
- Wie erzeugt man eigendefinierte Ausnahmen?
- Schlüsselworte `try`, `catch`, `finally`, `throw` und `throws`.

7 Applets

Inhalte / Ziele. *Erstellung von Applets und graphischen Benutzerschnittstellen. Ein Grund für die schnelle Verbreitung von Java ist die Möglichkeit, Applets zu programmieren und damit die Interaktionsmöglichkeiten von Internetseiten stark aufzuwerten. In diesem Abschnitt wird ein Applet erstellt, mit dem man ein einfaches Kniffelspiel simulieren kann. Dabei werden die wichtigsten Konzepte der Appletprogrammierung dargestellt. Einhergehend mit der Entwicklung des Kniffelspiels erfolgt ein Crash-Kurs zu objektorientierten graphischen Benutzerschnittstellen.*

7.1 Ein Beispiel: Kniffel

Das fertige Applet (mit Erläuterungen zu den Regeln) steht unter
`Applets/Kniffel/Fertig/Kniffel.html`.

7.2 Erste Arbeitsschritte mit Applets

An dieser Stelle sei nochmal daran erinnert, dass jedes Applet in eine HTML-Datei eingebettet werden muss, die entweder mit dem Appletviewer oder mit einem Browser angezeigt werden kann.

7.3 Erben von `java.applet.Applet`

```
java.lang.Object
    ↘ java.awt.Component
        ↘ java.awt.Container
            ↘ java.awt.Panel
                ↘ java.applet.Applet
```

- Applets erben von der Klasse **`java.awt.Component`**. Diese abstrakte Klasse stellt Attribute, die z. B. Position, Erscheinungsform und Zustand beschreiben, sowie Methoden zum Zeichnen und zur Benutzerinteraktion bereit. AWT ist die Abkürzung für *Abstract Windowing Toolkit*.

- Die abstrakte Klasse **`java.awt.Container`** stellt die verschiedene Gerüste bereit, um Komponenten geeignet zu positionieren (Layoutmanager).

- Die Klasse **`java.awt.Panel`** wird benutzt, um die Komponenten und ihre Benutzerinteraktion zu implementieren. Ein Panel ist vergleichbar mit einer Staffelei, auf der man sowohl eigene Zeichnungen anfertigen als auch vorgefertigte Elemente wie Buttons einfügen kann. Außerdem können wir hier auf Benutzereingaben reagieren. In der Vererbungshierarchie ist **`Panel`** die erste Klasse, von der wir Objekte instantiieren können.

- Die Klasse **`java.applet.Applet`** stellt Methoden bereit, um mit einem Browser oder einem Appletviewer zusammenarbeiten zu können. Von besonderer Bedeutung sind dabei die Methoden **`init()`**, **`start()`**, **`stop()`** und **`destroy()`**.
- Ein Applet ist also ein rechteckiges Bildschirmelement, das eine Größe und Position hat. Es kann Ereignisse empfangen und ist in der Lage, graphische Ausgaben vorzunehmen. Außerdem kann es mit einem Browser zusammenspielen.

Das Zusammenspielen zwischen Applets und einem Browser folgt einem weiteren grundlegenden Konzept von objektorientierten Programmiersprachen:

Inversion of control:

„Don’t call us, we’ll call you!“

7.4 Leben mit Applets

Neben der Methode **`paint()`** verfügt ein Applet über vier weitere Methoden, die mit der Java-Laufzeitumgebung zusammenarbeiten. Wir können diese Methoden überlagern.

- Die Methode **`init()`** wird nur einmal direkt nach der Instantiierung des Applets aufgerufen.

- Die Methode **`start()`** wird aufgerufen, wenn das Applet sichtbar wird.
- Wird ein Applet unsichtbar, wird die Methode **`stop()`** aufgerufen.
- Die Methode **`destroy()`** kommt zum Zuge, bevor das Applet endgültig entfernt wird.

Problem:

Verschiedene Browser-Implementierungen haben in Detailfragen unterschiedliche Ansichten darüber, wann ein Aufruf von **`init()`**, **`start()`**, **`stop()`** oder **`destroy()`** erfolgen soll. Die meisten Browser halten sich aber an folgende Grundregeln:

- Wenn ein Applet geladen wird, passiert folgendes:
  1. Es wird ein Objekt der entsprechenden Subklasse von **`Applet`** instantiiert.
  2. Das Applet initialisiert sich selbst (d. h. die Methode **`init()`** wird abgearbeitet).
  3. Das Applet startet (Methode **`start()`**).
- Wenn der Benutzer die HTML-Seite mit dem Applet verlässt, wird die Methode **`stop()`** aufgerufen.
- Wird die HTML-Seite neu geladen, wird erneut die Methode **`start()`** aufgerufen.
- Die Methode **`destroy()`** tritt in Aktion, wenn der Browser oder Appletviewer beendet wird. In der Regel wird diese Methode nicht überlagert.

7.5 Die Meilensteine im Leben eines Applets

- Auf der Seite **Applets/Simple/Simple.html** befindet sich ein Applet.

Applets/Simple/Simple.java

```
10 public class Simple extends Applet
11 {
12   int count = 0;
13
14   public void init()
15   {
16     setBackground(Color.green);
17     System.out.println(count++ + ": init()    wird aufgerufen ...");
18   }
19
20   public void start()
21   { System.out.println(count++ + ": start()   wird aufgerufen ..."); }
```

- Dieses Applet tut eigentlich gar nichts. Es wartet darauf, dass die Java-Laufzeitumgebung (**appletviewer**, Browser) eine der Methoden **init()**, **start()**, **stop()**, **destroy()** oder **paint()** aufruft.

- Wie können wir solche Aufrufe veranlassen und beobachten?

7.6 Nochmal: Kniffel

Auf den folgenden Seiten werden wir das Kniffel-Applet schrittweise aufbauen.

- Die Klasse **`java.applet.Applet`** erweitern
- Ein Layout für das Applet definieren
- Interaktion mit Knöpfen (in vier Variationen)
- Würfeln, Teil 1 + 2
- Würfel zeichnen, Teil 1 – 3
- Würfel interaktiv machen

7.6.1 Die Klasse `java.applet.Applet` erweitern

Applets/Kniffel/1-AppletErweitern/Kniffel.java A

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

9  public class Kniffel extends Applet
10 {
11   // ueberlagere die Methode init()
12   public void init()
13   { setBackground(Color.white); }
14 } // Ende public class Kniffel extends Applet
```

Quelltextanalyse

- Mit der Klausel **`extends Applet`** wird eine Subklasse von **`java.applet.Applet`** erzeugt.
- Mit den **`import`**-Anweisungen binden wir die Packages **`java.applet`** und **`java.awt`** ein.
- Die Methode **`init()`** der Klasse **`Applet`** wird überlagert. Momentan wird nur die Hintergrundfarbe auf **`Color.white`** gesetzt.

7.6.2 Ein Layout für das Applet definieren

Applets/Kniffel/2-Layout/Kniffel.java

```
12  public void init()
13  {
16    Panel unten = new Panel();
17    Button k1 = new Button("Neues Spiel");
18    k1.setBackground(Color.lightGray);
19    unten.add(k1);
20    Button k2 = new Button("Würfeln");
21    k2.setBackground(Color.lightGray);
22    unten.add(k2);
23    setLayout(new BorderLayout());
24    add(unten, BorderLayout.SOUTH);
25  }
```

Quelltextanalyse

- Es empfiehlt sich, die darzustellenden Elemente des Applets zu untergliedern. Wir benutzen dazu *Panels*, die mit Hilfe von *Layoutmanagern* verwaltet werden.
- Ein Panel können wir uns vorstellen als eine Staffelei. Auf dieser können wir malen oder vordefinierte Objekte darstellen lassen. Insbesondere können Panels auf Benutzereingaben reagieren.
- Wir erzeugen ein Panel **`unten`**, auf dem wir später zwei Knöpfe „Würfeln“ und „Neues Spiel“ verwalten werden.

- Das Package **`java.awt`** stellt eine Klasse **`Button`** zur Verfügung. Mit

```
17      Button k1 = new Button("Neues␣Spiel");
```

  instantiieren wir ein Objekt dieser Klasse und nennen es **`k1`**.

- Mit der Methode **`setBackground()`** der Klasse **`Button`** lässt sich die Hintergrundfarbe ändern.

- Durch die Anweisung **`unten.add(k1);`** wird das Button-Objekt dem Panel hinzugefügt.

- Die Klasse **`java.applet.Applet`** erbt von **`java.awt.Container`** die Möglichkeit, *Layout-Manager* einzusetzen. Mit **`setLayout(new BorderLayout());`** wird ein Layout gewählt, dass das Applet in fünf Gebiete (Norden, Osten, Süden, Westen und Zentrum) aufteilt.

- Man kann sowohl einfache, bereits vorhandene Komponenten (z.B. Buttons), als auch komplexere Komponenten in ein solches Layout einfügen.

- Mit **`add(unten, BorderLayout.SOUTH);`** fügen wir das fertige Panel dem Applet hinzu.

7.6.3 Interaktion mit Knöpfen

Ab der Version 1.1 von Java sind zwei Aufgaben zu erledigen, um auf Benutzereingaben bei Buttons (oder anderen graphischen Benutzerschnittstellen) zu reagieren.

1. Wir müssen einen **`ActionListener`** implementieren.
2. Wir müssen diesen ActionListener für den Knopf registrieren.

Im Package **`java.awt.event`** gibt es ein Interface (siehe 5.8) **`ActionListener`**. Dieses Interface verlangt, dass wir die Methode

```
public void actionPerformed(ActionEvent ae) {}
```

implementieren.

Hierzu werden im Folgenden vier verschiedene Möglichkeiten vorgestellt.

7.6.3.1 Das Applet implementiert einen ActionListener

Zuerst soll das Applet **`Kniffel`** selbst den ActionListener implementieren.

Wir erweitern also die Klassendeklaration um die Klausel **`implements ActionListener`** und überlagern die Methode **`actionPerformed()`**.

Applets/Kniffel/3-Interaktion/a-ActionListener/Kniffel.java

```
3  import java.awt.event.*;
11 public class Kniffel extends Applet implements ActionListener
32   public void actionPerformed(ActionEvent ae)
33   {
34     String cmd = ae.getActionCommand();
35     if (cmd.equals("Neues Spiel"))
36       System.out.println("Neues Spiel gedrückt.");
37     else if (cmd.equals("Würfeln"))
38       System.out.println("Würfeln gedrückt.");
39   }
```

Quelltextanalyse

- Mit **`import java.awt.event.*;`** binden wir das Package zur Ereignisbehandlung ein.
- Die Methode **`actionPerformed()`** bekommt eine Instanz **`ae`** der Klasse **`ActionEvent`** übergeben.
- Mit dessen Methode **`getActionCommand()`** können wir das Label des Knopfes ermitteln.
- Aus den zur Laufzeit übergebenen Informationen können wir also abfragen, welche Aktion vom Benutzer ausgeführt wurde.
- Die Anweisungen **`System.out.println()`** dienen nur zur Kontrolle.

```
21      k1.addActionListener(this);
25      k2.addActionListener(this);
```

Im zweiten Schritt registrieren wir unseren **`ActionListener`** bei den Knöpfen **`k1`** und **`k2`**. Da in unserem Falle das Applet selbst den **`ActionListener`** implementiert, übergeben wir die Referenz **`this`**.

7.6.3.2 Ein ActionListener als lokale Klasse

Soll unser Applet den ActionListener nicht selbst implementieren, so können wir eine eigene Klasse **`KniffelActionListener`** verwenden, die als *lokale Klasse* in die Klasse **`Kniffel`** eingebettet wird.

Applets/Kniffel/3-Interaktion/b-LokaleKlasse/Kniffel.java

```
11 public class Kniffel extends Applet
12 {
14   public void init()
15   {
19     KniffelActionListener kal = new KniffelActionListener();
22     k1.addActionListener(kal);
30   }
33   class KniffelActionListener implements ActionListener
34   {
35     public void actionPerformed(ActionEvent ae)
36     {
37       String cmd = ae.getActionCommand();
38       if (cmd.equals("Neues Spiel"))
39         System.out.println("Neues Spiel gedrückt.");
40       else if (cmd.equals("Würfeln"))
41         System.out.println("Würfeln gedrückt.");
42     }
43   } // Ende class KniffelActionListener implements ActionListener
44 } // Ende public class Kniffel extends Applet
```

Der Methode **addActionListener** muss nun statt der Referenz **this** ein Objekt der Klasse **KniffelActionListener** übergeben bekommen.

Alternativ könnten wir auch für jeden Knopf eine eigene Listener-Klasse implementieren; dann braucht man in **actionPerformed** keine Fallunterscheidung.

7.6.3.3 Ein ActionListener als anonyme Klasse

Die Klasse **KniffelActionListener** wird nur an einer einzigen Stelle benötigt und instantiiert. Wir können stattdessen an dieser Stelle eine *anonyme Klasse* definieren, die von **ActionListener** erbt.

Applets/Kniffel/3-Interaktion/c-AnonymeKlasse/Kniffel.java

```
19      ActionListener al = new ActionListener()
20        {  // Beginn der anonymen Subklasse von ActionListener
21         public void actionPerformed(ActionEvent ae)
22         {
28         }
29        }; // Ende der anonymen Subklasse von ActionListener
32      k1.addActionListener(al);
```

- Eine anonyme Unterklasse der Klasse **Kl** wird erzeugt, indem bei der Instantiierung **new Kl(...)** das sonst folgende Semikolon durch einen Klassenrumpf **{ ... }** ersetzt wird.
- Auf die selbe Art können wir mit anonymen Klassen Interfaces implementieren.
- Die Vererbung bzw. Implementierung geschieht hier ohne die Schlüsselworte **extends** bzw. **implements**.

7.6.3.4 Einzelne anonyme Klassen für jeden Knopf

Anonyme Klassen können sogar direkt bei der Registrierung des ActionListeners für einen Knopf erzeugt werden: in der Methode **`addActionListener`**.

Applets/Kniffel/3-Interaktion/d-ZweiAnonymeKlassen/Kniffel.java

```
21      k1.addActionListener(new ActionListener()
22        {   // Beginn der anonymen Subklasse von ActionListener
23          public void actionPerformed(ActionEvent ae)
24          { System.out.println("Neues Spiel gedrückt."); }
25        }); // Ende der anonymen Subklasse von ActionListener
```

7.6.4 Würfeln, Teil 1

Wir legen nun eine Klasse **`Wuerfelbrett`** an.

`Wuerfelbrett` soll von **`java.awt.Panel`** erben und auf Mausklicks reagieren. Hierzu gibt es wieder mehrere Möglichkeiten.

7.6.4.1 Implementierung eines MouseListeners

Applets/Kniffel/4-Wuerfeln1/a-MouseListener/Wuerfelbrett.java

```
10 class Wuerfelbrett extends Panel implements MouseListener
11 {
12   // Konstruktor
13   Wuerfelbrett() { addMouseListener(this); }
14
15   // Die Methoden fuer das Interface MouseListener
16   public void mousePressed(MouseEvent me)
17   {
18     int mx = me.getX();
19     int my = me.getY();
20     System.out.println("mousePressed: " + mx + " " + my);
21   } // mousePressed
22
23   // Dummies
24   public void mouseClicked(MouseEvent me) {}
25   public void mouseReleased(MouseEvent me) {}
26   public void mouseEntered(MouseEvent me) {}
27   public void mouseExited(MouseEvent me) {}
28 } // classWuerfelbrett extends Panel implements MouseListener
```

Quelltextanalyse

- Das **`Wuerfelbrett`** soll das Interface **`MouseListener`** implementieren.
- Das Interface **`MouseListener`** verlangt das Implementieren von folgenden Methoden: **`mousePressed()`**, **`mouseClicked()`**, **`mouseReleased()`**, **`mouseEntered()`**, **`mouseExited()`**

- Uns interessiert lediglich **`mousePressed()`**. Die übrigen Methoden versehen wir daher nur mit einem leeren Methodenkörper.
- Die Methode **`mousePressed()`** erhält ein Objekt **`me`** der Klasse **`MouseEvent`**.
- Diesem Objekt können wir mit der Methode **`getX()`** die aktuelle x-Koordinate entlocken.

Applets/Kniffel/4-Wuerfeln1/a-MouseListener/Kniffel.java

```
10 public class Kniffel extends Applet
11 {
12   Wuerfelbrett mitte;
15   public void init()
16   {
41     mitte = new Wuerfelbrett();
42     add(mitte, BorderLayout.CENTER);
```

In der Methode **`init()`** des Applets **`Kniffel`** fügen wir jetzt eine Instanz von **`Wuerfelbrett`** ein. Wir legen dabei die Referenz **`mitte`** als Memberattribut des Applets an.

7.6.4.2 Verwendung eines MouseAdapters

Applets/Kniffel/4-Wuerfeln1/b-MouseAdapter/Wuerfelbrett.java

```
13  Wuerfelbrett()
14  {
15    addMouseListener(new MouseAdapter()
16      {
17        public void mousePressed(MouseEvent me)
18        {
19          int mx = me.getX();
20          int my = me.getY();
21          System.out.println("mousePressed: " + mx + " " + my);
22        }
23      });
24  }
```

- Zu allen Listener-Interfaces, die auf mehr als ein Ereignis reagieren, existieren entsprechende Adapter-Klassen, die alle Listener-Methoden mit leerem Rumpf implementieren.
- Da keine Mehrfachvererbung möglich ist, kann unser **`Wuerfelbrett`** allerdings nicht von solch einem Adapter erben.
- Wir verwenden daher wieder eine anonyme Klasse, die diesmal von **`MouseAdapter`** abgeleitet wird, und überschreiben nur die Methode **`mousePressed`**.

7.6.5 Würfeln, Teil 2

Nun kümmern wir uns um die Programmlogik, indem wir die Klasse **`Wuerfelbrett`** erweitern.

Applets/Kniffel/5-Wuerfeln2/Wuerfelbrett.java

```
11    int anzahlWuerfe = 0;
12    int[] wuerfel = { 0, 0, 0, 0, 0 };
13    boolean[] fest = { false, false, false, false, false };
30    public void neuesSpiel()
31    {
32      for (int i = 0; i < wuerfel.length; i++)
33        {
34          wuerfel[i] = 0;
35          fest[i] = false;
36          System.out.println(i + ": " + wuerfel[i]);
37        }
38      anzahlWuerfe = 0;
39    }
41    public void wuerfeln()
42    {
43      double zufallszahl;
44      if (anzahlWuerfe < 3)
45        {
46          for (int i = 0; i < wuerfel.length; i++)
47            {
48              if (!fest[i])
49                {
50                  zufallszahl = Math.ceil(6.0 * Math.random());
51                  wuerfel[i] = (int)Math.round(zufallszahl);
52                } //if
53              System.out.println(i + ": " + wuerfel[i]);
54            } // for
55        } // if
56      anzahlWuerfe++;
57    }
```

Quelltextanalyse

- Wir erzeugen und initialisieren zwei Arrays der Länge 5 durch `int[] wuerfel = { 0, 0, 0, 0, 0 };` und `boolean[] fest = ...`
- Die Methoden **`neuesSpiel()`** bzw. **`wuerfeln()`** belegen das Array **`wuerfel`** mit Nullen bzw. Zufallszahlen zwischen 1 und 6.
- Die **`System.out.println()`**-Anweisungen dienen zur Kontrolle.

Dann sorgen wir dafür, dass das Applet **`Kniffel`** mit dem Panel **`Wuerfelbrett`** Botschaften austauschen kann.

Applets/Kniffel/5-Wuerfeln2/Kniffel.java A

```
24          public void actionPerformed(ActionEvent ae)
25          { mitte.neuesSpiel(); }
32          public void actionPerformed(ActionEvent ae)
33          { mitte.wuerfeln(); }
```

Quelltextanalyse

- Wir wollen die Methoden **`neuesSpiel()`** und **`wuerfeln()`** aufrufen, indem wir die entsprechenden Knöpfe im Applet **`Kniffel`** anklicken.
- Dazu verwenden wir die Referenz **`mitte`** und ersetzen die Druckanweisungen durch die Methodenaufrufe in Punktnotation.

7.6.6 Würfel zeichnen, Teil 1

Applets/Kniffel/6-WuerfelZeichnen1/Wuerfelbrett.java

```
30  public void paint(Graphics g)
31  {
32    g.setColor(Color.black);
33    g.drawRect(10, 10, 40, 40);
34    System.out.println("paint() wird aufgerufen.");
35  }
```

Quelltextanalyse

- Die Klasse **`Panel`** bietet die Methode **`paint()`** als Schnittstelle zum Zeichnen an.
- Sie dient dazu, das Panel neu zu zeichnen, wenn verdeckte Teile wieder freigelegt werden.
- Wie die Methoden **`init()`**, **`start()`**, **`stop()`** oder **`destroy()`** eines Applets wird sie *implizit* aufgerufen, je nachdem, wie der Browser sich verhält.
- Neben **`paint()`** gibt es noch die Methode **`update()`**. Diese verhält sich ähnlich, nur lässt sich hier durch inkrementelles Zeichnen Geschwindigkeit gewinnen (siehe 11.7.3).
- Beide Methoden haben ein Objekt der Klasse **`java.awt.Graphics`** als Parameter. Dies ist der Graphik-Kontext der jeweiligen Zeichenoperation. Dort ist z. B. die Information über Clipping-Fenster etc. enthalten.

- Die Klasse **`Graphics`** verfügt über viele Methoden zum Zeichnen einfacher geometrischer Figuren.
- Ein Studium der API-Referenz lohnt sich.

7.6.7 Würfel zeichnen, Teil 2

Applets/Kniffel/7-WuerfelZeichnen2/Wuerfelbrett.java

```
30    public void paint(Graphics g)
31    {
32      // Uebermale alles, was vorher war,
33      // mit einem grossen weissen Rechteck.
34      g.setColor(Color.white);
35      g.fillRect(10, 10, 241, 41);
36      // Male nun die Wuerfel
37      zeichneWuerfel(g);
38    }
41    public void neuesSpiel()
42    {
50      repaint();
51    }
53    public void wuerfeln()
54    {
69      repaint();
70    }
72    public void zeichneWuerfel(Graphics g)
73    {
74      g.setColor(Color.black);
75      for (int i = 0; i < 5; i++)
76        g.drawRect(10 + 50 * i, 10, 40, 40);
77    }
```

Quelltextanalyse

- In **`paint()`** wird nun der Bereich der Würfel mit der Hintergrundfarbe überschrieben und dann die Methode **`zeichneWuerfel()`** aufgerufen.
- In **`neuesSpiel()`** und **`wuerfeln()`** wird **`repaint()`** aufgerufen, um das ganze Panel neu zu malen.
- In **`neuesSpiel()`** und **`wuerfeln()`** werden die Druckanweisungen entfernt.

7.6.8 Würfel zeichnen, Teil 3

Applets/Kniffel/8-WuerfelZeichnen3/Wuerfelbrett.java

```
70  public void maleAugen(Graphics g, int x, int y, int Augen, boolean f)
71  {
72    if (Augen != 0)
73      {
74        if (f)
75          g.setColor(Color.lightGray);
76        else
77          g.setColor(Color.blue);
78        // links oben
79        if (Augen != 1) g.fillOval(x + 3, y + 3, 8, 8);
   ...
96  public void zeichneWuerfel(Graphics g)
97  {
   ...
99    for (int i = 0; i < 5; i++)
   ...
102       maleAugen(g, 10 + 50 * i, 10, wuerfel[i], fest[i]);
```

Quelltextanalyse

- Wir führen eine Methode **`maleAugen()`** ein, die von **`zeichneWuerfel()`** aufgerufen wird.
- Die Parameterliste gibt die Informationen wieder, was und wie gezeichnet werden soll.

7.6.9 Würfel interaktiv machen

Applets/Kniffel/9-WuerfelInteraktiv/Wuerfelbrett.java

```
20          public void mousePressed(MouseEvent me)
21          {
22            int mx = me.getX();
23            int my = me.getY();
24
25            if (my >= 10 && my <= 50)
26              for (int i = 0; i < wuerfel.length; i++)
27                if (mx >= 10 + i * 50 && mx <= 50 + i * 50)
28                  if (wuerfel[i] != 0)
29                    {
30                      fest[i] = !fest[i];
31                      repaint(10 + 50 * i, 10, 40, 40);
32                    }
33          }
```

Quelltextanalyse

- Wir vervollständigen jetzt die Methode **`mousePressed()`** von **`Wuerfelbrett`**.
- Man benutzt die Koordinaten **`mx`** und **`my`** nun, um die entsprechenden Würfel zu bearbeiten.

- Um ein Flackern aller Würfel zu vermeiden, wir nur der Bereich des angeklickten Würfel neu gemalt.

7.7 Zusammenfassende Fragen

- Wie implementiert und startet man das „Hello World!“-Applet?
- Worin unterscheiden sich Applets und Applikationen?
- Durch die Klausel `extends Applet` erbt man diverse Merkmale von Superklassen. Welche?
- Was sind die Meilensteine im Leben eines Applets?
- Welche Packages stellen Klassen zur Programmierung von graphischen Benutzerschnittstellen *(Graphical User Interface, GUI)* bereit?
- Welche Layoutmanager gibt es?
- Was ist ein Panel?
- Wie stellt man einen Knopf dar?
- Wie kann man auf einen Knopfdruck reagieren?
- Wie kann man in einem Panel auf Mausereignisse reagieren?

- Was ist der Unterschied zwischen einem **`MouseListener`** und einem **`MouseAdapter`**?
- Wie können verschiedene Komponenten eines Applets untereinander Botschaften austauschen?
- Was ist ein Graphik-Kontext?
- Wie zeichnet man einfache geometrische Figuren?
- Was bedeutet das Prinzip *Inversion of control*?

8 AWT

Inhalte / Ziele. *Erstellung eigenständiger graphischer Applikationen mit Hilfe des* Abstract Windowing Toolkit (AWT)*. Anhand des Kniffelspiels werden einige Elemente des AWT vorgestellt. Ein weiteres Beispielprogramm demonstriert die Reaktionen auf verschiedene Mausereignisse. Auf eine umfassendere Darstellung des AWT wird zugunsten von Swing (13) verzichtet.*

8.1 Umwandlung des Applets in eine AWT-Applikation

Die Klasse **`Kniffel`** wird nun nicht mehr von **`Applet`** abgeleitet, sondern erbt von **`Frame`**.

AWT/Kniffel/1-Frame/Kniffel.java

```
9  public class Kniffel extends Frame
10 {
```

↘ **`java.awt.Container`**
↘ **`java.awt.Window`**
↘ **`java.awt.Frame`**

- **`java.awt.Container`** ist der letzte gemeinsame Vorgänger von **`Applet`**s und **`Frame`**s.
- **`java.awt.Window`** ist einfach ein rechteckiger Graphikbereich — ohne Rand oder Menü. Jedes **`Window`** muss entweder in ein weiteres **`Window`** oder in einen **`Frame`** eingebettet werden.
- Ein **`java.awt.Frame`** entspricht eher dem, was man sich unter einem Fenster vorstellt. Es enthält einen Rand sowie eine Titelzeile.

```
13    // aus der init()-Methode des Applets wird nun der Konstruktor
14    public Kniffel()
15    {
16      super("Kniffel");     // setzt den Fenstertitel
17      setSize(270, 160);    // setzt die Größe (war beim Applet im HTML-Code)
18      setResizable(false); // die Fenstergröße darf nicht geändert werden

50      setVisible(true);     // den Frame anzeigen
51    }
```

- Die Methode **`init()`** des Applets wird nun zum Konstruktor des Frames.
- Die Größe, die vorher in der HTML-Datei festgelegt war, muss nun im Programm gesetzt werden.
- Außerdem sorgen wir dafür, dass der Anwender die Fenstergröße nicht ändern kann (**`setResizable(false);`**).
- Neue Frames sind nach der Konstruktion zunächst unsichtbar. Daher benötigen wir am Ende noch ein **`setVisible(true);`**.

```
55    public static void main(String[] argv)
56    {
57      // erzeuge eine neue Instanz des Kniffel-Frames
58      new Kniffel();
59    }
60  } // Ende public class Kniffel extends Frame
```

- Die **`main()`**-Methode macht nichts anderes, als ein neues Kniffel-Objekt, also ein Fenster, zu erzeugen.

Nach den bisherigen Änderungen ist es noch nicht möglich, das Fenster mit den üblichen Mitteln (Klick auf den Schließen-Knopf in der Titelzeile, **`Alt-F4`**) zu schließen. Es bleibt nur **`Ctrl-C`** oder das Kommando **`kill`**.

```
46      // sorge dafür, dass das Fenster geschlossen werden kann
47      addWindowListener(new WindowAdapter()
48        { public void windowClosing(WindowEvent e) { beenden();} });

53    void beenden() { System.exit(0); }
```

- Durch hinzufügen eines **`WindowAdapter`**s können wir auf **`WindowEvent`**s reagieren.
- **`System.exit(0);`** beendet die virtuelle Maschine und gibt den Parameter (hier 0 für normales Ende) an das Betriebssystem weiter.

An der Klasse **`Wuerfelbrett`** sind keinerlei Änderungen notwendig.

8.2 Ergänzung eines Menüs

Wir ergänzen ein Menü **Spiel** mit den Menüpunkten **Neu** und **Ende**.

AWT/Kniffel/2-Menu/Kniffel.java

```
47      MenuBar mBar = new MenuBar();
48      Menu spielMenu = new Menu("Spiel");
49      MenuItem spielNeu = new MenuItem("Neu", new MenuShortcut(KeyEvent.VK_N));
50      spielNeu.addActionListener(new ActionListener()
51        { public void actionPerformed(ActionEvent e) { mitte.neuesSpiel();} });
52      spielMenu.add(spielNeu);
57      mBar.add(spielMenu);
58      setMenuBar(mBar);
```

- Für jeden Menüpunkt wird ein **MenuItem**-Objekt benötigt. Diese werden zu einem **Menu** zusammengefasst, und mehrere **Menu**s zu einem **MenuBar**. Letzterer schließlich wird dem **Frame** hinzugefügt.

- Durch einen zusätzlichen **MenuShortcut**-Parameter können den Menüpunkten Tastaturkürzel zugeordnet werden. Mit **Ctrl-N** kann hier alternativ zu dem Button ein neues Spiel gestartet werden.

8.3 Reaktion auf verschiedene Maus-Ereignisse

AWT/MausEreignisse/MausEreignisse.java

```
9  public class MausEreignisse extends Frame
10   implements MouseListener, MouseMotionListener
11 {
```

Für die Reaktion auf **MouseEvent**s gibt es in Java zwei verschiedene Interfaces: **MouseListener** für die Reaktion auf gedrückte Maustasten sowie das Verlassen und Betreten des Graphikbereichs kennen wir schon; **MouseMotionListener** sind für die Bewegung der Maus zuständig.

```
36   public void mouseClicked(MouseEvent e)
37   {
38     Graphics g = getGraphics();
39     g.setColor(chooseColor(e));
40     g.drawString("C", e.getX(), e.getY());
41     System.out.println("mouseClicked: " + e.getX() + " " + e.getY());
42   }
```

- Ausnahmsweise umgehen wir die Methode **paint** und malen direkt. Der benötigte Graphik-Kontext wird mit der Methode **java.awt.Component.getGraphics()** beschafft.
- Der Nachteil hiervon ist, dass der Fensterinhalt nicht wieder hergestellt wird, wenn das Fenster verdeckt war.

```
24  // die Abfrage der verschiedenen Mousetasten funktiert erst seit Java 1.4
25  Color chooseColor(MouseEvent e)
26  {
27    switch(e.getButton())
28      {
29        case MouseEvent.BUTTON1: return Color.red;
30        case MouseEvent.BUTTON2: return Color.green;
31        case MouseEvent.BUTTON3: return Color.blue;
32      }
33    return Color.magenta;
34  }
```

- Die Methode **`MouseEvent.getButton()`** testet, welche Maustaste gedrückt wurde. Die Konstanten **`BUTTON1`**, **`BUTTON2`** und **`BUTTON3`** stehen für die linke, mittlere bzw. rechte Maustaste.
- Diese Methode wurde erst in Java 1.4 eingeführt. In älteren Versionen und insbesondere bei Applets lassen sich die Tasten daher nicht unterscheiden.
- Die Methoden des **`MouseMotionListener`**s haben als Parameter zwar auch ein **`MouseEvent`**, dies enthält aber keine Information über die Maustasten.

Beobachtung

- Ein Ziehen der Maus ruft die drei Methoden **`mousePressed`**, **`mouseDragged`** (mehrfach) und **`mouseReleased`** auf.

- Ein Mausklick ruft nacheinander **mousePressed**, **mouseReleased** und **mouseClicked** auf.

8.4 Weitere Beispiele

Weitere Beispiele zu verschiedene AWT-Elementen finden sich in den folgenden Beispiel-Applets von Sun.

AWT/SunBeispiele/ButtonDemo.java A
AWT/SunBeispiele/CheckBoxDemo.java A
AWT/SunBeispiele/ChoiceDemo.java A
AWT/SunBeispiele/ListDemo.java A
AWT/SunBeispiele/TextDemo.java A

9 Das Package java.io

Inhalte / Ziele. *Das Package* **java.io** *enthält eine Sammlung von Klassen, die die Ein- und Ausgabe sowie Dateiverwaltung unterstützen. Die Ein- / Ausgabeoperationen in Java basieren auf* Streams. *Bei* Applets *im Internet sind die Dateifunktionen erheblich eingeschränkt. Wir werden eine kleine Java-Applikation zur Textverarbeitung erstellen. Diese erzeugt aus aus einer Java-Quelltextdatei eine HTML-Datei, in dem der Java-Quelltext zu Dokumentationszwecken optisch aufbereitet wird.*

9.1 Streams

Programme müssen häufig Informationen von einer externen Quelle lesen oder in ein externes Ziel schreiben.

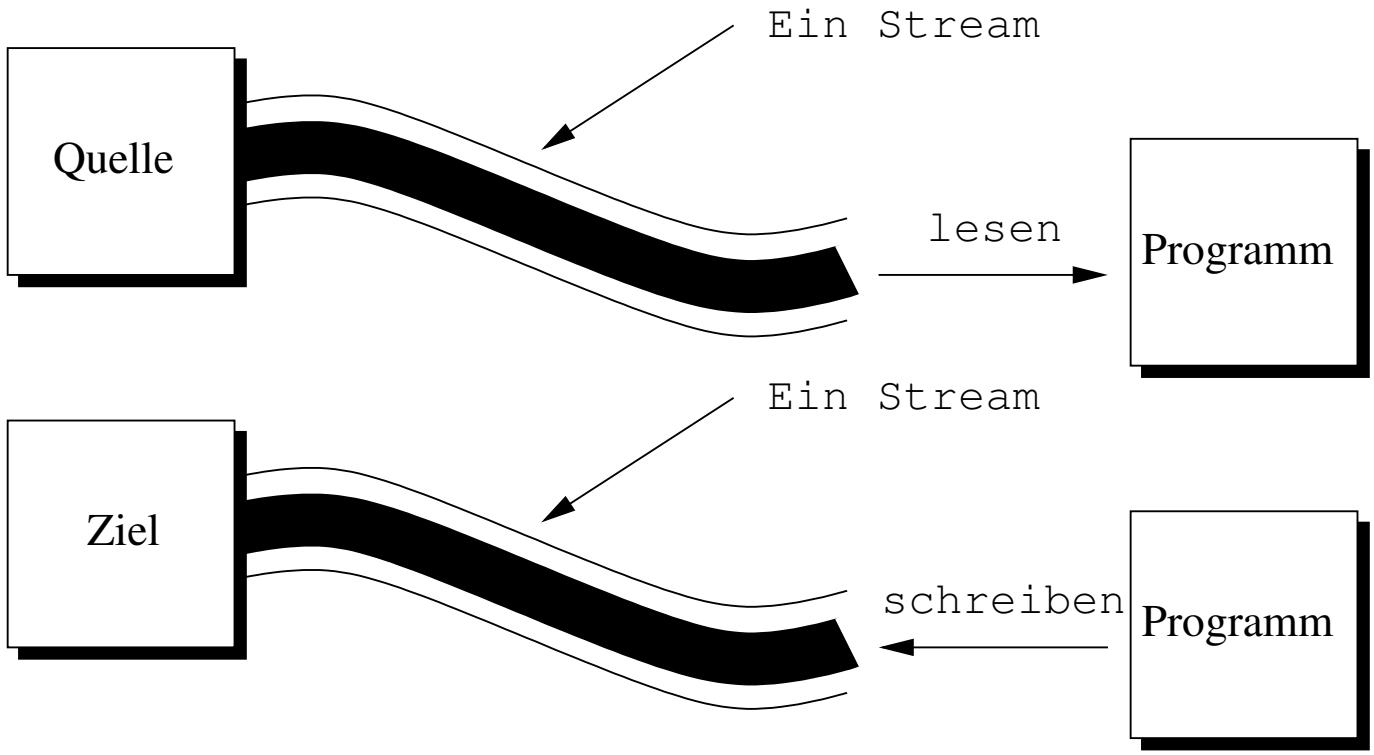
Die Informationen können von beliebiger Art sein: Objekte, Zeichenketten, Zahlen, Bilder oder Töne.

Quelle und Ziel der Informationen kann eine Datei, eine Konsole oder eine Netzwerkadresse sein.

Um lesend / schreibend auf externe Informationen zuzugreifen, werden in Javaprogrammen so genannte *Streams* verwendet.

Ein Stream stellt einen universellen Datenfluss oder Kommunikationskanal dar.

Es gibt Streams zum lesenden *(Reader)* und zum schreibenden *(Writer)* Zugriff.



Durch Vererbung werden dann diverse Streams für konkrete Anwendungen erzeugt und im Package **java.io** als Klassen bereitgestellt.

In Java erfolgt jedes Arbeiten mit Streams nach folgendem Pseudocode:

```
Eingabestream erzeugen
solange noch Information
    lese Information
Eingabestream schliessen
```

```
Ausgabestream erzeugen
solange noch Information
    schreibe Information
Ausgabestream schliessen
```

9.2 Ein- / Ausgabe am Terminal

Java ermöglicht den Zugriff auf die Standardein- und ausgabe über **System.in** und **System.out**.

Grundlagen/EinAusgabe/SystemOut.java

```
10      System.out.println("6 * 7 = " + (6 * 7));
```

Grundlagen/EinAusgabe/SystemIn.java

```
12      try
13        {
14          int val = System.in.read();
20        }
21      catch (IOException ioe)
22        {
23          System.out.println("Habe gefangen: " + ioe);
24        }
```

Quelltextanalyse

- **System.in** repräsentiert den Standard-Eingabestream.
- Dieser wird verwendet, um Tastatureingaben zu lesen.
- Die Methode **read()** liest das nächste Byte aus dem Stream.
- Der Wert wird als **int**-Variable (!) zurückgegeben und liegt im Bereich **0** bis **255**.
- Falls kein Byte mehr gelesen werden kann (z. B. Dateiende), ist der Rückgabewert **-1**.

- Die Methode wirft ferner eine **IOException**. Dies macht die Einbettung in eine **try-catch**-Umgebung nötig.

JavaIO/Terminal/Terminal.java

```
12      try
13        {
22          InputStreamReader isr = new InputStreamReader(System.in);
23          BufferedReader br = new BufferedReader(isr);
24          String zeile = br.readLine();
25          System.out.println(zeile);
26        }
27      catch (IOException ioe) { System.out.println("Habe gefangen: " + ioe); }
```

Quelltextanalyse

- Streams können ineinander verschachtelt werden.
- Ein einfacher **InputStream** wie **System.in** liefert einen Strom aus Bytes.
- Ein **InputStreamReader** wandelt einen Bytestrom in einen Characterstrom um.
- Auf diesen Characterstrom kann ein **BufferedReader** angesetzt werden.
- Ein **BufferedReader** liest Text von einem Characterstrom. Hierbei werden die **char**s gepuffert eingelesen.

- Er erkennt z. B. Zeilenendezeichen und bietet eine Methode **readLine()** an, mit der man eine vollständige Zeile einlesen kann.

Hinweis:

Für die Umwandlung von **String** nach einfachen numerischen Datentypen kennen wir die Wrapper-Klassen (siehe 5.10).

9.3 Die Klasse File

Bevor eine Datei geschrieben wird, sollte man überprüfen, ob die Zugriffsrechte ausreichen und ob evtl. bereits eine Datei mit dem vorgesehen Namen existiert.

Die Klasse **File** bietet daneben noch Methoden, die es erlauben, Dateien umzubenennen, Verzeichnisse zu erzeugen oder zu durchsuchen.

JavaIO/Files/FileBsp1.java

```
13      String fileSep = System.getProperty("file.separator");
14      System.out.println("fileSep = " + fileSep);
18      System.out.println(datei.exists());
19      System.out.println(datei.getName());
20      System.out.println(datei.getPath());
25      System.out.println(datei.canRead());
26      System.out.println(datei.length());
28      System.out.println(new Date(datei.lastModified()));
```

JavaIO/Files/FileBsp2.java

```
12      String fileSep = System.getProperty("file.separator");
13      File verzeichnis = new File(".." + fileSep + ".." + fileSep
14                                  + "Objektorientierung" + fileSep
15                                  + "Interfaces");
20      if (verzeichnis.exists())
21        if (!verzeichnis.isFile())
22          {
23            System.out.println();
24            System.out.println("Listing aller Dateien");
25            String[] dateien = verzeichnis.list();
26            for (int i = 0; i < dateien.length; i++)
27              System.out.println(dateien[i]);
28          }
```

9.4 Datei lesen

In dieser Anwendung soll eine Datei eingelesen und auf dem Bildschirm ausgegeben werden.

JavaIO/Files/Lesen/1/LeseDatei.java

```
12      String nameEingabedatei;
13      File eingabedatei;
14      FileReader fr;

16      try
17        {
18          nameEingabedatei = argv[0];
19          System.out.println("Inhalt von " + nameEingabedatei);
20          eingabedatei = new File(nameEingabedatei);
21          fr = new FileReader(eingabedatei);
22        }
23      catch (ArrayIndexOutOfBoundsException aioobe)
24        { System.out.println("Aufruf mit: java LeseDatei eingabedatei"); }
25      catch (FileNotFoundException fnfe)
26        { System.out.println("Fehler: " + fnfe.getMessage()); }
```

Quelltextanalyse

- Der Name der Datei soll als Kommandozeilenargument beim Aufruf übergeben werden und wird mit **`nameEingabedatei = argv[0];`** ausgewertet.
- Vergisst der Anwender das Kommandozeilenargument, fangen wir dies durch die Ausnahme **`ArrayIndexOutOfBoundsException`** ab.
- Die Anweisung **`eingabedatei = new File(nameEingabedatei);`** erzeugt uns eine Instanz von **`File`**, die die Informationen über die Datei kapselt.
- Das Anlegen einer **`File`**-Instanz wirft *keine* Ausnahme.

- Die Klasse **FileReader** erzeugt einen Stream von einer Datei.
- Sollte die Datei nicht vorhanden oder nicht lesbar sein, wird die Ausnahme **FileNotFoundException** geworfen.

JavaIO/Files/Lesen/2/LeseDatei.java

```
13      String zeile;
16      BufferedReader br;

24          br = new BufferedReader(fr);
25          zeile = br.readLine();
26          while (zeile != null)
27            {
28              System.out.println(zeile);
29              zeile = br.readLine();
30            }
31          br.close();

37      catch (IOException ioe)
38        { System.out.println("Fehler: " + ioe.getMessage()); }
```

Quelltextanalyse

- Die Klasse **FileReader** erzeugt einen Stream von einer Datei.
- Um auf diesen Stream komfortabler zugreifen zu können, verpacken wir ihn wieder in einen Stream, der Daten gepuffert lesen kann.
- Wir erzeugen dafür eine Instanz von **BufferedReader**.

- Mit der Anweisung **`zeile = br.readLine()`** kann nun eine ganze Zeile in einen String eingelesen werden.
- Die Methode **`readLine()`** wirft die Ausnahme **`IOException`**.
- Sie liefert **`null`** zurück, sobald das Ende des Streams (hier der Datei) erreicht ist.
- Danach kann der **`BufferedReader`** mit der Methode **`close()`** wieder geschlossen werden.

9.5 Datei schreiben

Diese Anwendung erzeugt eine einfache HTML-Datei.

JavaIO/Files/Schreiben/1/SchreibeDatei.java

```
12      String nameAusgabedatei;
14      File ausgabedatei;
15      FileWriter fw;
16      BufferedWriter bw;
18      try
19        {
20          nameAusgabedatei = argv[0];
21          ausgabedatei = new File(nameAusgabedatei + ".html");
22          fw = new FileWriter(ausgabedatei);
23          bw = new BufferedWriter(fw);
24          bw.write("Hallo");
25          bw.close();
26        }
```

Quelltextanalyse

- Die Klasse **FileWriter** ist der symmetrische Gegenpart von **FileReader**.
- Auch zu **BufferedReader** gibt es eine symmetrische Ergänzung: **BufferedWriter**

Es werden nun Kopf und Fuß einer HTML-Datei geschrieben.

JavaIO/Files/Schreiben/2/SchreibeDatei.java

```
51      String newline;
56      newline = System.getProperty("line.separator");
63          schreibeKopf(bw, newline, nameAusgabedatei);
65          schreibeFuss(bw, newline);
10    static void schreibeKopf(BufferedWriter bw, String newline, String code)
11      throws IOException
12    {
17      bw.write("<HTML>" + newline);
35    } // static void schreibeKopf
```

Quelltextanalyse

- Zwei Klassenmethoden **schreibeKopf()** und **schreibeFuss()** erzeugen den Inhalt des HTML-Dokuments.
- Im String **newline** speichern und benutzen wir das systemabhängige Zeichen für den Zeilenumbruch.

9.6 Exkurs: Zeichenkettenverarbeitung

Die Klassen **java.lang.StringBuffer** und **java.util.StringTokenizer** stellen nützliche Werkzeuge zur Zeichenkettenverarbeitung zur Verfügung.

- Nach seiner ersten Initialisierung ist der Wert eines Strings nicht mehr veränderbar. Auch bei der Benutzung des überladenen Operators **+** werden implizit immer wieder neue String-Objekte erzeugt:

Zeichenketten/StringBuffer/StrBufBsp.java

```
10      String wort = "Java";
11      wort = wort + "quellcode";
12      wort = wort + "datei";
```

- Dies kann sich negativ auf die Effizienz auswirken.
- Die Klasse **java.lang.StringBuffer** ist eine veränderliche Variante der Klasse **String**. Sie kooperiert eng mit **String**, verfügt aber zusätzlich über diverse Methoden zur Manipulation der Zeichenkette.

```
14      StringBuffer zksb = new StringBuffer("Javacode");
15      zksb.append("datei");
16      zksb.insert(4,"quell");
17      zksb.reverse();
18
19      // Umwandlung in einen String mit der Methode toString();
20      String zks = zksb.toString();
21      System.out.println(zks); // druckt ietadedoclleuqavaJ
```

- Die Klasse **StringTokenizer** aus dem Package **java.util** zerteilt einen String in einzelne Tokens, die durch ein bestimmtes Zeichen voneinander getrennt sind.

- Die Methoden **nextToken()** und **hasMoreTokens()** dienen zum Auslesen und zur Überprüfung der Verfügbarkeit von einzelnen Tokens. Die Methode **countTokens()** gibt die Anzahl der verfügbaren Tokens zurück.

Zeichenketten/StringTokenizer/StrTokBsp1.java

```
14      StringTokenizer st1 = new StringTokenizer("Alle Jahre wieder");
15      while (st1.hasMoreTokens())
16        {
17          aktuellesToken = st1.nextToken();
18          System.out.println(aktuellesToken);
19        }
20
21      StringTokenizer st2 = new StringTokenizer("http://www.wdr.de",":/");
22      if (st2.countTokens()==2)
23        {
24          System.out.println("Protokoll = " + st2.nextToken());
25          System.out.println("Host      = " + st2.nextToken());
26        }
```

Zeichenketten/StringTokenizer/StrTokBsp2.java

```
12      StringBuffer sb = new StringBuffer();
13      for (int i = 0; i < argv.length; i++)
14        sb.append(argv[i] + "␣");
15      System.out.println(sb);
16
17      String s = sb.toString();
18      sb.setLength(0);
19
20      StringTokenizer st = new StringTokenizer(s, "␣");
21      String aktuellesToken;
22      while (st.hasMoreTokens())
23        {
24          aktuellesToken = st.nextToken();
25          System.out.println(aktuellesToken);
26        } // while
```

Quelltextanalyse

- Diese Anwendung fasst zunächst alle Kommandozeilenargumente aus dem String-Array **`argv[]`** zusammen.
- Im zweiten Schritt wird ein **`StringTokenizer`** verwendet, um den **`StringBuffer`** wieder in einzelne Strings zu verwandeln.
- Beim Instantiieren des StringTokenizer kann optional ein zweites Argument als Separatorzeichen verwendet werden. **`st = new StringTokenizer(s, "e");`** separiert dann nicht zwischen Leerzeichen, sondern beim Zeichen **`"e"`**.

- Nach dem Übersetzen kann das Programm getestet werden mit z. B.

```
java StrTokBsp2 Esel essen Nesseln gern.
```

9.7 Eine Datei einlesen, verarbeiten und schreiben

Die Java-Applikation **J2H** soll aus einer Java-Quelltextdatei ein HTML-Dokument erzeugen. Dazu fassen wir zunächst die Ergebnisse von **LeseDatei.java** und **SchreibeDatei.java** zusammen.

JavaIO/j2h/1/J2H.java

```
9 public class J2H
```

Ausführung und Wirkung

- Wir können dieses Programm übersetzen mit **javac J2H.java** und beispielsweise testen mit **java J2H Auto**.
- Aus dem Java-Quellcode **Auto.java** ist dann eine Datei **Auto.html** entstanden, die mit einem Browser betrachtet werden kann. Vorher müssen wir aber noch einige Erweiterungen ergänzen.

Version 2

- Wir wollen nun jede eingelesene Zeile Wort für Wort auswerten.
- Dazu verwenden wir die Klassenmethode **`auswerten()`**.
- Wir ersetzen also in der Methode **`main()`** die Anweisung **`bw.write(zeile + newline)`** durch **`auswerten(...)`**.
- Innerhalb von **`auswerten()`** besteht die erste Aufgabe darin, führende Leerzeichen auszudrucken.
- Danach erzeugen wir einen **`StringBuffer`** und zerlegen ihn mit dem **`StringTokenizer`** in einzelne Tokens. Dazu müssen wir das Package **`java.util`** einbinden.

JavaIO/j2h/2/J2H.java

```
49  static void auswerten(BufferedWriter bw, String zeile, String newline)
50    throws IOException
51    {
52      int i = 0;
53      while (i < zeile.length()
54             && (zeile.charAt(i) == ' ' || zeile.charAt(i) == '\t' ))
55        bw.write(zeile.charAt(i++));
56
57      StringBuffer sb = new StringBuffer(100);
58      for (i = 0; i < zeile.length(); i++)
59        sb.append(zeile.charAt(i));
60
61      String aktuellesToken;
62      StringTokenizer st = new StringTokenizer(sb.toString());
63      while (st.hasMoreTokens())
64        {
65          aktuellesToken = st.nextToken();
66          bw.write(aktuellesToken + " ");
67        }
68      bw.write(newline);
69    } // static void auswerten
```

Version 3

- Wir verfeinern jetzt die Klassenmethode **auswerten()**.
- Wir fügen dazu eine statische **boolean**-Variable **auskommentiert** hinzu.
- Ferner brauchen wir eine lokale **boolean**-Variable **zeileAuskommentiert**.
- Wir prüfen, ob das aktuelle Token entweder *//* oder */** bzw. **/* entspricht. In solchen Fällen setzen wir die Schriftfarbe auf **RED** (bzw. zurück) und aktualisieren die entsprechenden **boolean**-Variablen.
- Die Zeichen **<** und **>**, die im Javaquellcode auftreten können, werden durch die entsprechenden HTML-Tags ersetzt.

JavaIO/j2h/3/J2H.java

```
12  static String COMMENT_BEG = "<I><FONT COLOR=RED>";
13  static String COMMENT_END = "</FONT></I>";
14  static boolean auskommentiert = false;

89        if (aktuellesToken.equals("/*"))
90          {
91            bw.write(COMMENT_BEG);
92            auskommentiert = true;
93          }

71        if (c == '<')
72          sb.append("&lt;");
73        else if (c == '>')
74          sb.append("&gt;");
75        else
76          sb.append(c);
```

Version 4

- Mit Hilfe der Methode **isKey()** lassen sich Java-Schlüsselworte blau färben.

JavaIO/j2h/4/J2H.java

```
55  static boolean isKey(String wort)
56  {
57    boolean erg = false;
59    String[] keyWords = { "abstract", "boolean", "break", "byte", "case",
68                          "void", "volatile", "while" };
69    for (int i = 0; i < keyWords.length; i++)
70      if (wort.equals(keyWords[i]))
71        erg = true;
72    return erg;
73  }

117       istSchluesselwort = isKey(aktuellesToken);
118       if (istSchluesselwort && !zeileAuskommentiert && !auskommentiert)
119         bw.write(KEYWORD_BEG);
```

Version 5

- Die letzten Änderungen erzeugen im HTML-Dokument eine Übersicht über die Paare geschweifter Klammern.

JavaIO/j2h/5/J2H.java

9.8 Zusammenfassende Fragen

- Was sind Streams?
- Wie realisiert man Eingaben über die Tastatur?
- Welche Methoden zur Dateiverwaltung bietet die Klasse **`File`**?
- Warum verschachtelt man Streams (Wrapping)?
- Wie verschachtelt man Streams?
- Wie funktioniert das Einlesen einer Textdatei?
- Welche Ausnahmen sind abzufangen?
- Wie funktioniert das Schreiben in eine Datei?
- Welche Ausnahmen sind abzufangen?
- Was leisten die Klassen **`StringBuffer`** und **`StringTokenizer`**?

10 Das Package java.text

Inhalte / Ziele. *Wir können nun Texte und Zahlen in Terminals und in Dateien schreiben und hieraus lesen. Es fehlt bisher eine Möglichkeit, Zahlen formatiert auszugeben, z. B. immer mit zwei Nachkommastellen. Zu diesem Zweck gibt es das Package* **java.text***. Es ermöglicht außerdem das Einlesen spezieller Formate sowie lokalisierte Ausgaben.*

Das folgende Beispielprogramm enthält die wichtigsten Anwendungen von **java.text**. Weitere Details können in der API-Referenz nachgelesen werden.

JavaText/FormatBsp.java

```
2  import java.text.*;
```

- Das Package **java.text** muss importiert werden.

```
15     String s;
16     double d = 0;
18     try
19       {
20         System.out.print("Eingabe: d = ");
21         s = (new BufferedReader(new InputStreamReader(System.in))).readLine();
22         d = Double.parseDouble(s);
23       }
24     catch (IOException ioe)
25       { System.out.println("Fehler: " + ioe.getMessage()); }
26     catch (NumberFormatException nfe)
27       { System.out.println("Fehler: " + nfe.getMessage()); }
29     System.out.println("normale Ausgabe:                    " + d);
```

- Hier wird die Standardform für die Ausgabe verwendet.

```
31 NumberFormat nf;
32 nf = NumberFormat.getNumberInstance();
33 nf.setMinimumFractionDigits(3);
34 System.out.println("NumberFormat␣(MinFrDig=3):␣␣␣␣␣␣␣␣␣␣" + nf.format(d));
```

- **NumberFormat** ist eine abstrakte Basisklasse für alle Zahlenformate.
- Die Methode **getNumberInstance()** erzeugt eine Instanz einer abgeleiteten Klasse, in der Regel **DecimalFormat**.
- Die Methode **format()** gibt es in zwei überladenen Varianten, für **double** und **long**. Sie liefert einen String mit der gewünschten Formatierung der Zahl zurück.
- Es werden im Beispiel mindestens drei Nachkommastellen ausgegeben.

```
36 System.out.println("NumberFormat.getCurrencyInstance():␣"
37                    + NumberFormat.getCurrencyInstance().format(d));
```

- Die Methode **getCurrencyInstance()** erzeugt ein Format zur Ausgabe von Währungsbeträgen.
- Hier kann es noch zu Problemen kommen, da bisher nicht alle Zeichensätze das €-Zeichen enthalten.

```
39    nf = NumberFormat.getCurrencyInstance(new Locale("en", "GB"));
40    System.out.println("␣␣␣␣␣␣␣...(new␣Locale(\"en\",␣\"GB\")):␣"
41                       + nf.format(d));
```

- Den Methoden **`getNumberInstance()`** und **`getCurrencyInstance()`** kann als zweiter Parameter ein **`Locale`**-Objekt für regional angepasste Ausgaben übergeben werden. **`Locale`**-Objekte werden konstruiert aus einem Sprachkürzel (*`"en"`*) und einem Länderkürzel (*`"GB"`*).
- Wird kein **`Locale`**-Objekt übergeben, wird die Umgebungsvariable **`$LANG`** ausgewertet. Sie kann in der **`bash`** z. B. mit **`export LANG=fr_FR`** geändert werden.

```
43    System.out.println("DecimalFormat(\"000.0000\"):␣␣␣␣␣␣␣␣␣"
44                       + new DecimalFormat("000.0000").format(d));
```

- Es werden hier *immer* mindestens drei Vor- und vier Nachkommastellen ausgegeben.
- Das Zeichen *`'0'`* steht für eine beliebige Ziffer.
- Das Zeichen *`'.'`* ist das Dezimaltrennzeichen.

```
46    System.out.println("DecimalFormat(\"##0.####\"):␣␣␣␣␣␣␣␣␣"
47                       + new DecimalFormat("##0.####").format(d));
```

- Das Zeichen *`'#'`* unterdrückt führende Nullen und Nullen als letzte Nachkommastellen.

```
49    System.out.println("DecimalFormat(\"#,##0.0###\"):␣␣␣␣␣␣␣␣"
50                       + new DecimalFormat("#,##0.0###").format(d));
```

- Das Zeichen `','` ist das Tausendertrennzeichen, mit dem lange Zahlen in Dreiergruppen eingeteilt werden.

```
52    System.out.println("DecimalFormat(\"#.##E00\"):␣␣␣␣␣␣␣␣␣␣␣␣"
53                       + new DecimalFormat("#.##E00").format(d));
```

- Das Zeichen `'E'` aktiviert die Exponentialdarstellung.

```
55    Date date = new Date();
56    SimpleDateFormat sdf = new SimpleDateFormat("EE,␣dd.MM.yyyy,␣hh:mm:ss");
57    System.out.println("SimpleDateFormat:␣" + sdf.format(date));
```

- Mit **`SimpleDateFormat`** lassen sich Daten und Uhrzeiten formatiert ausgeben.
- **`new Date()`** initialisiert ein **`Date`**-Objekt mit dem aktuellen Datum und der aktuellen Zeit.
- Nähere Informationen zum Umgang mit Daten und Uhrzeiten findet sich in der API-Dokumentation zu **`java.util.Date`** und **`java.util.Calendar`**.

```
59      try
60        {
61          System.out.print("Eingabe (in der Form 1.234,56): d = ");
62          s = (new BufferedReader(new InputStreamReader(System.in))).readLine();
63          DecimalFormat df = new DecimalFormat("#,##0.0");
64          d = df.parse(s).doubleValue();
65          System.out.println("eingegeben: " + d);
66        }
67      catch (IOException ioe)
68        { System.out.println("Fehler: " + ioe.getMessage()); }
69      catch (ParseException pe)
70        { System.out.println("Fehler: " + pe.getMessage()); }
```

- Die Formatklassen kann man nicht nur zur Ausgabe sondern auch zur Eingabe in bestimmten Formaten verwenden.
- Die Methode **`parse()`** liefert ein **`Number`**-Objekt (Vorfahr von **`Double`**) zurück, dessen Wert man mit **`doubleValue()`** auslesen kann.
- Das obige Beispiel akzeptiert **`double`**-Ausdrücke in der Form **`1.234,78`**.
- Wird eine ungültige Zeichenfolge eingegeben, wirft **`parse()`** eine **`ParseException`**.

11 Threads

Inhalte / Ziele. *Java ist eine multithread-fähige Programmiersprache. Sie erlaubt also, dass der Interpreter mehrere so genannte Threads gleichzeitig ausführen lassen kann. Dafür stehen die Klasse* **`java.lang.Thread`** *und das Interface* **`java.lang.Runnable`** *zur Verfügung. Wichtige Unterthemen sind Implementierung, Prioritäten, Synchronisation, Zusammenspiel mit Applets, Animationen.*

11.1 Ein Beispiel: Threads

Das Applet **`Threads/ThreadSort/threadsort.html`** demonstriert die Mächtigkeit von Threads.

Ein *Thread* ist ein einzelner sequentieller Kontrollfluss innerhalb eines Programmes.

Multi-Threading bedeutet, mehrere Aufgaben (Tasks) nebeneinander (parallel) laufen zu lassen.

11.2 Einen Thread implementieren

Einen Thread erzeugen und starten wir in drei Schritten.

1. Wir erzeugen eine Klasse, die das *Interface* **`Runnable`** implementiert (Interfaces siehe 5.8).
2. Wir übergeben die Referenz auf ein Objekt dieser Klasse an den Konstruktor der Klasse **`Thread`**.
3. Die neue Instanz von Thread kann mit **`start()`** zur Ausführung gebracht werden.

Interfaces sind abstrakte Klassen (siehe 5.7). Solche Klassen schreiben lediglich vor, welche Attribute und Methoden enthalten sein sollen. Die konkrete Implementierung wird vom Programmierer vorgenommen, wenn er eine Subklasse bildet.

Das Interface **`Runnable`** sieht so aus:

```
public
interface Runnable {
    public abstract void run();
}
```

Wir müssen also die Methode **`run()`** implementieren.

Threads/JaNein/Runnable/RandomRun.java

```
6  class RandomRun implements Runnable
7  {
8    private String meinName;
11   public RandomRun(String str) { meinName = str; }
14   public String getName() { return meinName; }
17    public void run()
18    {
19      for (int i = 0; i < 10; i++)
20        {
21          System.out.println(i + "␣mal␣" + getName());
22          try
23            {
24              Thread.sleep((int)(Math.random() * 1000));
25            }
26          catch (InterruptedException e) {}
27        }
28      System.out.println("Die␣Antwort␣lautet:␣" + getName());
29    }
30 }
```

Quelltextanalyse

- Durch die Klausel **`implements Runnable`** zeigt die Klasse an, dass sie das Interface **`Runnable`** implementiert.
- Wir müssen daher die Methode **`run()`** implementieren.
- Die Methode **`run()`** ist das Herzstück eines **`Runnable`**s.
- Hier wird konkret eine Zählschleife implementiert. Später soll ein Thread, der die Methode ausführt, den Zähler und seinen Namen ausgeben und danach eine bestimmte Zeit schlafen.
- Die **`Thread`**-Methode **`sleep()`** wirft eine Ausnahme, die gefangen werden sollte.
- Wir überlagern den Konstruktor, indem wir den Namen des Threads angeben.
- Mit der Methode **`getName()`** lassen wir uns den Namen zurückgeben.

Vom Hauptprogramm der Applikation **`ThreadTest`** aus werden mehrere Threads erzeugt und gestartet.

Threads/JaNein/Runnable/ThreadTest.java

```
10      RandomRun rr1 = new RandomRun("Ja");
11      Thread t1 = new Thread(rr1);
12      RandomRun rr2 = new RandomRun("Nein");
13      Thread t2 = new Thread(rr2);
17      t1.start();
18      t2.start();
```

Quelltextanalyse

- Es werden zwei Objekte **`rr1`** und **`rr2`** der Klasse **`RandomRun`** erzeugt.
- Mit **`Thread t1 = new Thread(rr1);`** wird eine Instanz von Thread erzeugt.
- Ein Thread erwartet, dass ihm eine Methode **`run()`** bekannt gegeben wird, die er ausführen soll.
- Wir erledigen dies, indem wir seinem Konstruktor eine Referenz auf ein **`Runnable`** (**`rr1`**) übergeben.
- Mit **`t1.start()`** wird der Thread gestartet.
- Es wird nun die Methode **`run()`** des **`Runnable`**s **`rr1`** ausgeführt.
- Man kann „beliebig“ viele Threads starten. Zum Beispiel könnte man einen Thread namens ***`"Vielleicht"`*** hinzufügen.

11.2.1 Von `Thread` erben

Manchmal werden Threads auch wie im folgenden Beispiel erzeugt:

Threads/JaNein/ExtendsThread/RandomRun.java

```
6  class RandomRun extends Thread
7  {
9    public RandomRun(String str) { super(str); }
12   public void run()
13   {
24   }
25 }
```

Threads/JaNein/ExtendsThread/ThreadTest.java

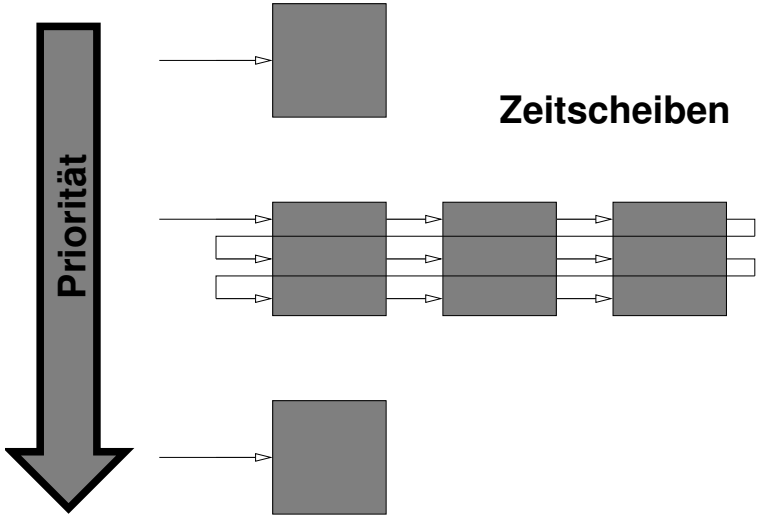
```
10     RandomRun rr1 = new RandomRun("Ja");
15     rr1.start();
```

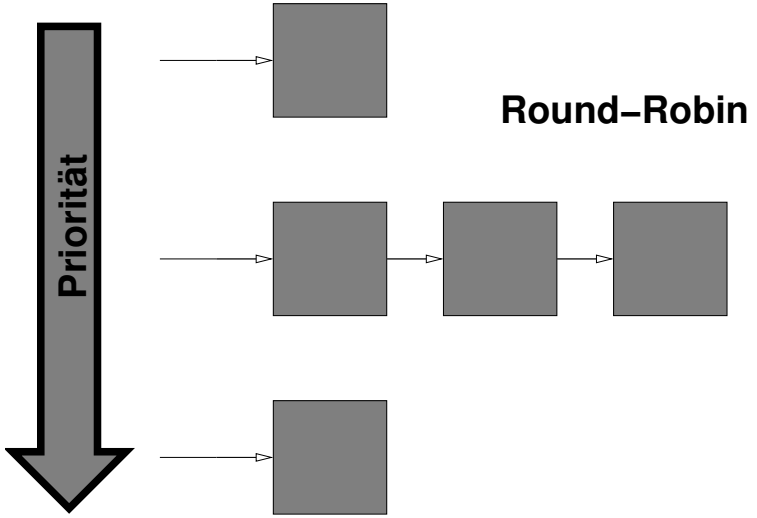
Quelltextanalyse

- Die Klasse **`Thread`** selbst implementiert auch das Interface **`Runnable`**.
- Wir können daher auch direkt von **`Thread`** eine neue Threadklasse ableiten.
- Auch wenn diese zweite Variante etwas kürzer ist, wird empfohlen ein **`Runnable`** zu implementieren, sofern nur **`run()`** implementiert werden soll, aber keine weiteren **`Thread`**-Methoden überlagert werden sollen.

11.3 Prioritäten

- Auf einer einzigen CPU können mehrere Threads nicht wirklich parallel laufen.
- Die Ausführung mehrerer Threads auf einem einzelnen Prozessor nennt man *Scheduling*.





- In Java kann man Threads mit einer Priorität versehen.
- Die Realisierung des Scheduling und damit die Auswirkung von Prioritäten ist allerdings abhängig von der jeweiligen Implementierung der virtuellen Maschine.

Threads/Prioritaeten/PriorRun.java

```
18  public void run()
19  {
20    while (tick < 10000000)
21      {
22        tick++;
23        if ((tick % 1000000) == 0)
24          System.out.println("Thread " +  meinName + ", tick = " + tick);
25      }
26  }
```

Quelltextanalyse

- Wir programmieren zuerst eine Klasse **`PriorRun`**, die **`Runnable`** implementiert.
- Innerhalb der zu überlagernden Methode **`run()`** wird die Variable **`tick`** hochgezählt und ab und zu ausgegeben.

Threads/Prioritaeten/Rennen.java

```
15    t1.setPriority(Thread.NORM_PRIORITY);
16    t2.setPriority(Thread.NORM_PRIORITY + 3);
```

Quelltextanalyse

- Wir erzeugen zwei Instanzen der Klasse **`PriorRun`** und übergeben zur Unterscheidung verschiedene Strings.
- Wir erzeugen zwei Threads **`t1`** und **`t2`**, die mit einer Priorität belegt werden.
- Zuletzt wird über **`start()`** die jeweilige Methode **`run()`** angestoßen.

11.4 Zusammenarbeit zwischen Threads

- Sollen Threads zusammenarbeiten, so sind die Zugriffe auf gemeinsame Daten zu regeln.
- Die grundlegenden Konzepte dafür sollen am „Hersteller-Verbraucher“-Beispiel vorgestellt werden.
- Wir erzeugen Klassen **`Hersteller`**, **`Verbraucher`**, **`Korb`** und **`HVKTest`**.

Threads/HVK/1/HVKTest.java

```
6  public class HVKTest
7  {
8    public static void main(String[] args)
9    {
10     Korb k = new Korb();
11
12     Thread th = new Thread(new Hersteller(k));
13     Thread tv = new Thread(new Verbraucher(k));
14
15     th.start();
16     tv.start();
17   }
18 }
```

Quelltextanalyse

- Die Applikation **`HVKTest`** erzeugt je ein Objekt der Klassen **`Hersteller`**, **`Verbraucher`** und **`Korb`**.
- Die Klassen **`Hersteller`** und **`Verbraucher`** implementieren **`Runnable`**s, die mit zwei Threads **`th`** und **`tv`** gestartet werden.

11.4.1 Der Hersteller-Thread

Threads/HVK/1/Hersteller.java

```
6  public class Hersteller implements Runnable
7  {
8    private Korb derKorb;
9
10   public Hersteller(Korb derKorb) { this.derKorb = derKorb; }
11
12   public void run()
13   {
14     for (int i = 1; i <= 10; i++)
15       {
16         derKorb.lege(i);
17         System.out.println("Hersteller legte " + i + " in den Korb.");
18         try { Thread.sleep((int)(Math.random() * 100)); }
19         catch (InterruptedException e) {}
20       }
21   }
22 }
```

Quelltextanalyse

- Die Klasse **`Hersteller`** implementiert ein **`Runnable`**.
- Im Konstruktor wird eine Referenz auf das **`Korb`**-Objekt der Applikation **`HVKTest`** angelegt.
- Die Methode **`run()`** wird überlagert. In einer Schleife versucht der Thread, Zugriff auf den Korb zu erhalten, um den aktuellen Wert des Schleifenzählers dort abzulegen (**`derKorb.lege(i)`**). Nach einer Meldung verzögert der Hersteller-Thread seine Arbeit, indem er sich eine gewisse Zeit schlafen legt.

11.4.2 Der Verbraucher-Thread

Threads/HVK/1/Verbraucher.java

```
6  public class Verbraucher implements Runnable
7  {
8    private Korb derKorb;
9
10   public Verbraucher(Korb derKorb) { this.derKorb = derKorb; }
11
12   public void run()
13   {
14     int wert;
15     for (int i = 1; i <= 10; i++)
16       {
17         wert = derKorb.hole();
18         System.out.println("Verbraucher␣holte␣" + wert + "␣aus␣dem␣Korb.");
19       }
20   }
21 }
```

Quelltextanalyse

- Die Klasse **`Verbraucher`** implementiert ein **`Runnable`**.
- Es wird jeweils aus dem Korb ein Wert ausgelesen.

11.4.3 Der Korb

- Die Programmteile, auf die mehrere unabhängige Threads zugreifen, nennt man *kritischen Sektionen*.
- Wir beginnen mit einem ersten Entwurf für die Klasse **`Korb`**.
- In dieser Form tut die Applikation allerdings noch nicht das, was wir erwarten.

Threads/HVK/1/Korb.java

```
6  public class Korb
7  {
8    private int inhalt;
9
10   public int hole() { return inhalt; }
11
12   public void lege(int wert) { inhalt = wert;  }
13 }
```

Es ist notwendig, dass wir den Zugriff auf das **`Korb`**-Objekt *synchronisieren*.

Threads/HVK/2-synchronized/Korb.java

```
10  public synchronized int hole()
11  {
12    // Der Verbraucher-Thread sperrt alle
13    // synchronized-Methoden.
14    return inhalt;
15    // Der Verbraucher-Thread gibt die
16    // synchronized-Methoden wieder frei.
17  }

19  public synchronized void lege(int wert)
```

- Durch das Schlüsselwort **`synchronized`** wird eine Klasse oder eine Instanz davor geschützt, dass sie gleichzeitig von mehreren Threads manipuliert wird.
- Innerhalb einer Klassendefinition können mehrere Methoden als **`synchronized`** markiert werden.
- Sobald ein Thread eine dieser Methoden aufruft, sind alle als **`synchronized`** markierten Methoden für andere Threads gesperrt. (Unterscheide Synchronisation von Klassen- und Objektmerkmalen.)
- Man sagt auch, dass der Thread die **`Sperre`** auf das Objekt besitzt.
- Verlässt der Thread die Methode, wird die Sperre frei und ein anderer Thread kann diese erwerben.

Synchronisation alleine reicht nicht aus. Wir können jetzt zwar garantieren, dass immer nur ein Thread Zugriff auf das Objekt hat. Aber es kann passieren, das ein Thread mehrmals hintereinander die Sperre erwirbt. Zwei Threads müssen in der Lage sein, sich gegenseitig zu informieren, wenn sie mit ihrer Arbeit wirklich fertig sind.

11.4.4 Die Methoden `wait()` und `notify()`

Mit Hilfe einer **`boolean`**-Variable **`verfuegbar`** können wir einen neuen Entwurf wagen:

(Technisches Detail: Manche Compiler optimieren leere **`while`**-Schleifen weg. Daher erzeugen wir eine Dummy-Anweisung **`i++`**.)

Threads/HVK/3-deadlock/Korb.java

```
11   public synchronized int hole()
12   {
13     int i = 0;
14     System.out.println("betrete hole()");
15     while (!verfuegbar)
16       {
17         // warte darauf, dass ein anderer Thread (Hersteller)
18         // verfuegbar auf true setzt.
19         i++;
20       }
21     verfuegbar = false;
22     System.out.println("verlasse hole()");
23     return inhalt;
24   }
25
26   public synchronized void lege(int wert)
27   {
28     int i = 0;
29     System.out.println("betrete lege()");
30     while (verfuegbar)
31       {
32         // warte darauf, dass ein anderer Thread (Verbraucher)
33         // verfuegbar auf false setzt.
34         i++;
35       }
36     inhalt = wert;
37     verfuegbar = true;
38     System.out.println("verlasse lege()");
39   }
```

Quelltextanalyse

- Die obige Version produziert einen klassischen *Deadlock* (Todessperre). Wir wissen nicht, wann welcher Thread auf den Korb zugreifen will.
- Wenn z. B. die Variable **`verfuegbar`** den Wert **`false`** hat und ein Thread die Methode **`hole()`** aufruft, bleibt das Programm stehen.
- Wünschenswert wäre es also, wenn der aktive Thread seine Sperre aufheben könnte, ohne die Methode zu verlassen.

Neben dem Konzept der Sperren auf synchronisierte Blöcke tritt nun das Konzept der *Warteliste*. Für diesen Mechanismus werden die Methoden **`wait()`** und **`notify()`** bzw. **`notifyAll()`** eingesetzt.

- Diese Methoden erbt man von der Klasse **`java.lang.Object`**.
- **`wait()`** und **`notify()`** dürfen nur innerhalb eines synchronisierten Blockes verwendet werden.
- Mit **`wait()`** gibt ein Thread innerhalb eines synchronisierten Blockes die von ihm gehaltene Sperre frei und begibt sich in eine Warteliste. Ein Aufruf von **`wait()`** bewirkt also, dass die **`synchronized`**-Methoden für andere Threads zugänglich werden.

- Der gerade aktive Thread hat über **`notify()`** bzw. **`notifyAll()`** die Möglichkeit, andere wartende Threads aus der Warteliste zu informieren, dass sich der Zustand eines Objektes geändert hat. Die Threads aus der Warteschlange konkurrieren dann wieder um die Sperre, sobald sie freigegeben wird.

- Das folgende Beispiel zeigt, wie **`wait()`** und **`notify()`** zusammenwirken.

Threads/HVK/4-wait-notify/Korb.java

```
11  public synchronized int hole()
12  {
13    int i = 0;
14    while (!verfuegbar)
15      {
16        try { wait(); }
17        catch (InterruptedException e) {}
18      }
19    verfuegbar = false;
24    notify();
25    return inhalt;
26  }
28  public synchronized void lege(int wert)
29  {
30    int i = 0;
31    while (verfuegbar)
32      {
33        try { wait(); }
34        catch (InterruptedException e) {}
35      }
36    inhalt = wert;
37    verfuegbar = true;
38    notify();
39  }
```

11.5 Threads und Applets

Kombiniert man die Möglichkeiten von Threads und Applets, so kann man sehr leicht Animationen erzeugen.

Frage: Wie kombinieren wir Threads und Applets?

Wiederholung: Einen Thread erzeugen und starten wir in drei Schritten.

1. Wir erzeugen eine Klasse, die das Interface **`Runnable`** implementiert. — *Diese Klasse ist das Applet selbst. Wir überlagern die Methode* **`run()`**.
2. Wir übergeben die Referenz auf ein Objekt dieser Klasse an den Konstruktor der Klasse **`Thread`**. — *In der Methode* **`start()`** *von Applet wird ein* **`Thread`***-Objekt erzeugt. Mit* **`this`** *übergeben wir eine Referenz auf das Applet selbst.*
3. Die neue Instanz von Thread kann mit **`start()`** zur Ausführung gebracht werden. — *In der Methode* **`start()`** *von Applet wird die Methode* **`start()`** *von* **`Thread`** *aufgerufen.*

Threads/UhrApplet/Uhr.java A

```
11 public class Uhr extends Applet implements Runnable
12 {
13   Thread uhrThread = null;
14
15   // ueberlagert Methode start() von Applet
16   public void start()
17   {
24         uhrThread = new Thread(this, "Uhr");
25         uhrThread.start();
27   }
30   public void run()
31   {
39   }
```

Quelltextanalyse

- Mit den Klauseln

  ```
  public class Uhr extends Applet implements Runnable
  ```

  erbt man einerseits von **`java.awt.Applet`**. Gleichzeitig erhält man die Möglichkeit, die Methode **`run()`** überlagern zu können (und damit später Threads mit Anweisungen zu versorgen.)

- Innerhalb der Methode **`start()`** des Applets kann ein Thread instantiiert werden. Es gibt einen Konstruktor von **`Thread`**, der neben einer Referenz auf ein **`Runnable`**-Objekt auch einen **`String`** zur Bezeichnung akzeptiert.

- Die Anweisung **`uhrThread.start();`** bewirkt den Aufruf der Methode **`run()`**, die wir programmiert haben.

11.6 Der Rest der Uhr

Das Uhr-Applet zeigt die wichtigsten Meilensteine im Leben eines Threads:

1. Einen Thread erzeugen
2. Einen Thread starten
3. Einen Thread anhalten (pausieren lassen)
4. Einen Thread stoppen (beenden)

```
15  // ueberlagert Methode start() von Applet
16  public void start()
17  {
18    if (uhrThread == null)
19      {
20        // Erzeuge eine Instanz von uhrThread.
21        // Wir wollen, dass die Methode run()
22        // des Applets selbst ausgeführt wird.
23        // Daher uebergeben wir als Runnable this.
24        uhrThread = new Thread(this, "Uhr");
25        uhrThread.start();
26      }
27  }
```

```
29  // ueberlagert Methode run() von Runnable
30  public void run()
31  {
32    Thread meinThread = Thread.currentThread();
33    while (uhrThread == meinThread)
34      {
35        repaint();
36        try { Thread.sleep(1000); }
37        catch (InterruptedException e){}
38      } // while
39  }
40
41  // ueberlagert Methode paint() von Applet
42  public void paint(Graphics g)
43  {
44    Calendar now = Calendar.getInstance();
45    now.setTimeZone(TimeZone.getTimeZone("ECT"));
46    now.setTime(now.getTime());     // workaround
47    DecimalFormat df = new DecimalFormat("00");
48    g.drawString(df.format(now.get(Calendar.HOUR_OF_DAY)) + ":" +
49                 df.format(now.get(Calendar.MINUTE)) + ":" +
50                 df.format(now.get(Calendar.SECOND)), 10, 20);
51  }
52
53  // ueberlagert Methode stop() von Applet
54  public void stop() { uhrThread = null; }
```

Quelltextanalyse

- Mit `Thread uhrThread = null;` legen wir als Memberattribut eine Referenz auf `Thread` an.
- Innerhalb der Methode `start()` wird, falls noch nicht geschehen, durch `uhrThread = new Thread(this, "Uhr");` der zugehörige Konstruktor für den Thread aufgerufen.
- Mit der Anweisung `uhrThread.start();` wird der Thread zur Ausführung gebracht mit der eigens implementierten Methode `run()`.
- Innerhalb der Methode `run()` geschieht Folgendes:
  - Es wird lokal eine Thread-Referenz `meinThread` erzeugt, die auf den aktuell laufenden Thread (`uhrThread`) zeigt.
  - Solange die beiden Referenzen identisch sind, wird zuerst durch `repaint()` eine neue Ausgabe erzeugt.
  - Danach legt sich der Thread für eine gewisse Anzahl von Millisekunden schlafen. Man lässt den Thread pausieren.
  - Dieses Applet könnte also eine Basis für Animationen sein.
- Gestoppt wird die Ausführung von `run()`, sobald in der Methode `stop()` von Applet die Referenz von `uhrThread` auf `null` gesetzt wird.

11.6.1 Dasselbe mit der Methode `interrupt()`

Das Starten und Anhalten des Uhr-Threads lässt sich auch eleganter programmieren — mit Hilfe der Methoden **`interrupt()`** und **`isInterrupted()`**. Leider reagieren manche ältere Browser (z. B. Netscape 4) aber nicht auf solche Unterbrechungen eines Threads.

Threads/UhrApplet/interrupt/Uhr.java A

```
14   Thread uhrThread;
17   public void start()
18   {
23     uhrThread = new Thread(this, "Uhr");
24     uhrThread.start();
25   }
28   public void run()
29   {
30     Thread meinThread = Thread.currentThread();
31     while (!meinThread.isInterrupted())
32       {
33         repaint();
34         try { Thread.sleep(1000); }
35         catch (InterruptedException e){ meinThread.interrupt(); }
36       } // while
37   }
52   public void stop() { uhrThread.interrupt(); }
```

Quelltextanalyse

- Der Aufruf **`uhrThread.interrupt()`** signalisiert dem **`uhrThread`**, dass er unterbrochen werden soll.
- Der Thread muss mit der Methode **`isInterrupted()`** selbst nachsehen, ob eine solche Unterbrechungsanforderung vorliegt.
- Trifft die Unterbrechung während der **`sleep`**-Phase ein, so wird *diese* sofort unterbrochen und das Abbruchsflag zurückgesetzt. Der Thread muss sich im **`catch`**-Block anschließend noch einmal selbst beenden.

11.7 Wer kennt Duke?

Das ist er:

Threads/Duke/1/Duke.java

```
9  public class Duke extends Applet
10 {
11   int hoehe, breite;
12   Image dukeBild;

14   public void init()
15   {
16     dukeBild = getImage(getCodeBase(), "hungryduke.gif");
17   }
```

Quelltextanalyse

- Wir überlagern die Methode **`init()`** und veranlassen das Applet, mit dem Aufruf der Methode **`getImage()`** mit dem Laden einer Datei eines bestimmten Bildformats (hier gif) zu beginnen.

```
19   public void paint(Graphics g)
20   {
21     breite = dukeBild.getWidth(this);
22     hoehe = dukeBild.getHeight(this);
23     g.drawImage(dukeBild, 10, 10, breite, hoehe, this);
24     System.out.println("gezeichnet mit paint");
25   }
26
27   public void update(Graphics g)
28   {
29     breite = dukeBild.getWidth(this);
30     hoehe = dukeBild.getHeight(this);
31     g.drawImage(dukeBild, 10, 10, breite, hoehe, this);
32     System.out.println("gezeichnet mit update");
33   }
```

Quelltextanalyse

- Die Methode **`paint()`** dient dazu, das Applet neu zu zeichnen, wenn verdeckte Teile wieder freigelegt werden.
- Die Methode **`update()`** können wir überlagern, wenn wir durch inkrementelles Zeichnen Geschwindigkeit gewinnen wollen. **`update()`** wird aufgerufen, wenn ein asynchrones Ereignis eintritt. Ein solches ist beispielsweise der Abschluss des Ladevorganges einer Bilddatei.
- Zum Unterschied zwischen **`paint()`** und **`update()`** später mehr (siehe 11.7.3).

11.7.1 Duke bewegt sich auf Knopfdruck

Threads/Duke/2/Duke.java A

```
10 public class Duke extends Applet implements MouseListener
11 {
12   int hoehe, breite;
13   int nr = 0;
14   Image[] dukeBild = new Image[3];
15
16   public void init()
17   {
18     addMouseListener(this);
19     dukeBild[0] = getImage(getCodeBase(), "hungryduke.gif");
20     dukeBild[1] = getImage(getCodeBase(), "rightspoonduke.gif");

46   public void mousePressed( MouseEvent me )
47   {
48     nr = (nr + 1) % 3;
49     repaint();
50   }
```

Quelltextanalyse

- Es werden nun drei Bilddateien geladen und entsprechend in Arrays verwaltet.
- Durch die Klausel **`implements MouseListener`** verpflichtet sich das Applet, die entsprechenden Methoden des Interfaces zu implementieren und kann im Gegenzug auf Mausereignisse reagieren.
- Mit **`addMouseListener(this);`** wird der MouseListener registriert.

- In der Methode **`mousePressed()`** wird das Memberattribut **`nr`** geändert und **`repaint()`** aufgerufen. Dies bewirkt die Ausführung der unter **`update()`** implementierten Anweisungen.
- Ein explizites **`repaint()`** bewirkt den Aufruf von **`update()`**, während das durch den Browser veranlasste Neuzeichnen die Methode **`paint()`** aufruft.

11.7.2 Duke winkt

Threads/Duke/3/Duke.java A

```
15  Thread dukeThread = null;
```

Quelltextanalyse

- Wir erzeugen einen Thread durch Implementierung des Interfaces **`Runnable`**.
- Die Referenz unseres Threads namens **`dukeThread`** soll vorerst auf **`null`** zeigen.
- Wir überlagern die Methoden **`start()`**, **`run()`** und **`stop()`** analog zum Uhr-Applet.

11.7.3 `paint()` und `update()`

Threads/Duke/4/Duke.java A

```
24  public void paint(Graphics g)
25  {
26    for (int i = 0; i < 100; i++)
27      for (int j = 0; j < 100; j++)
28        {
29          Color prisma = new Color(i ,j, 0);
30          g.setColor(prisma);
31          g.drawLine(i, j, i, j);
32        }
```

Quelltextanalyse

- Wir blähen die Methode **`paint()`** künstlich auf, indem wir einen aufwändigen Hintergrund malen.
- Jeder explizite Aufruf von **`repaint()`** bewirkt die Durchführung von **`update()`**.
- Wir haben die Methode **`update()`** überlagert und sparen uns dabei das aufwändige Malen des Hintergrundes.
- Auf diese Weise können wir selbst bestimmen, welche Teile unseres Grafikbereichs tatsächlich neu gezeichnet werden müssen.
- Wenn wir **`update()`** nicht überlagern, tritt die Standard-Implementierung in Kraft. Diese tut im Wesentlichen zwei Dinge. Erstens wird der gesamte Grafikbereich mit der Hintergrundfarbe übermalt. Danach wird die Ausführung der Methode **`paint()`** veranlasst.

- Wir können dieses Phänomen beobachten, indem wir „unsere“ überlagerte Methode **`update()`** auskommentieren.

11.8 Zusammenfassende Fragen

- Was ist ein Thread?
- Was bedeutet Multi-Threading?
- Wie können wir einen Thread erzeugen und starten?
- Wie arbeiten dabei das Interface **`Runnable`** und die Klasse **`Thread`** zusammen?
- Wie kann das Scheduling bei der Implementierung einer virtuellen Maschine implementiert sein?
- Was sind kritische Sektionen?
- Was bewirkt das Schlüsselwort **`synchronized`**?
- Was sind Deadlocks?
- Was sind Sperren?
- Was sind Wartelisten?

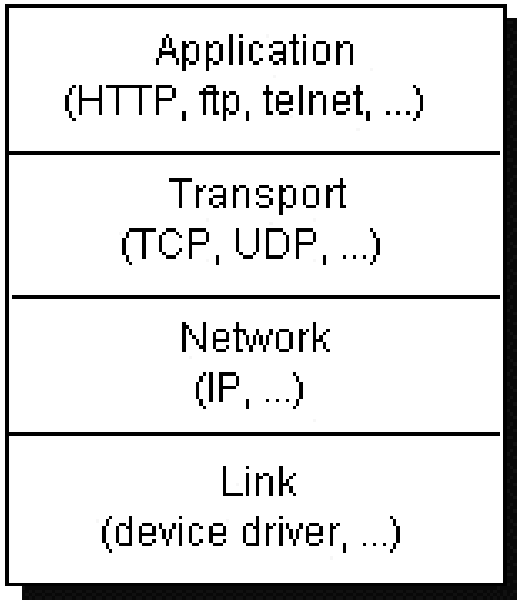
- Was bewirken die Methoden **`wait()`**, **`notify()`** und **`notifyAll()`**?
- Wie kann ein Applet Threads verwalten?
- Wie lassen sich Bilddateien in ein Applet laden und darstellen?
- Was ist der Unterschied zwischen **`paint()`** und **`update`**?

12 Das Package `java.net`

Inhalte / Ziele. *Wir lernen in diesem Abschnitt die Möglichkeiten zur Netzwerkprogrammierung mit Java kennen. Die Sprache unterstützt unter Verwendung des Packages* **`java.net`** *Kommunikation sowohl unter der dem* Transmission Control Protocol (TCP) *als auch unter dem* User Datagram Protocol (UDP). *Auf diese Weise ist eine einfache Handhabung von Internetdiensten gegeben und die Entwicklung von Client-Server-Anwendungen erleichtert.*

12.1 Grundlagen

Kommunikation in Computernetzwerken wird geregelt durch Protokolle im OSI-Schichtenmodell.



Im so genannten *Transport Layer* gibt es zwei Protokolle, die von Java durch Bereitstellen entsprechender Klassen gut unterstützt werden: *TCP* und *UDP*.

TCP

TCP *(Transmission Control Protocol)* ist ein *verbindungsorientiertes* Protokoll, das einen gesicherten Datenfluss zwischen zwei Computern herstellt. Anwendungen wie *http*, *ftp* oder *telnet* basieren auf diesem Protokoll.

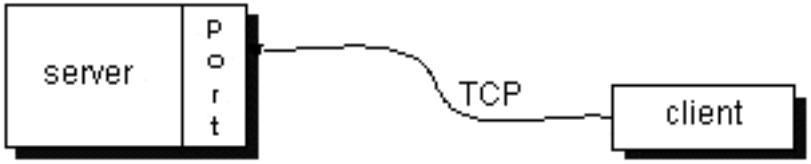
UDP

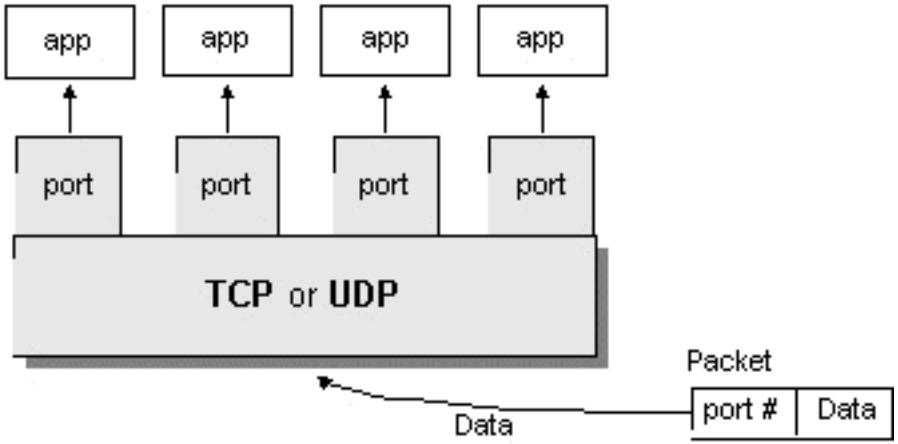
UDP *(User Datagram Protocol)* ist dagegen ein *verbindungsloses* Protokoll. Es sendet unabhängige Datenpakete, so genannte *Datagramme* von einem Computer zu einem Anderen.

Hinweis: Viele Firewalls und Router erlauben keine UDP-Pakete.

Ports

Da ein Computer im Netzwerk simultan mehrere Dienste anbietet oder in Anspruch nimmt, muss es eine Regelung geben, welcher Dienst über welchen Kommunikationskanal laufen darf. *Ports* stellen für TCP und UDP eine Möglichkeit dar, ein- und ausgehende Daten den jeweiligen Anwendungen zuzuordnen.





Es gibt Portnummern zwischen 0 und 65535, wobei die Nummern 0 bis 1023 von Standarddiensten wie *http* belegt sind.

Klassen aus `java.net`

JDK-Klassen für die Kommunikation unter TCP sind **`URL`**, **`URLConnection`**, **`Socket`** und **`ServerSocket`**.

JDK-Klassen für die Kommunikation unter UDP sind **`DatagramPacket`**, **`DatagramSocket`** und **`MulticastSocket`**.

Diese Klassen sind *systemunabhängig*. Um einfache Netzwerkanwendungen zu schreiben, müssen wir lediglich die Mächtigkeit dieser Klassen nutzen und brauchen uns nicht mit den Implementierungsdetails auseinanderzusetzen.

12.2 Die Klasse URL

- Wir betrachten zuerst Klassen, die mit TCP arbeiten.
- Ein *Uniform Resource Locator (URL)* wird in Java durch Instanzen der Klasse **`java.net.URL`** verwaltet.
- Es gibt diverse Konstruktoren für **`URL`**, siehe API-Referenz.
- Die Konstruktoren werfen die Ausnahme **`MalformedURLException`**.
- Eine einmal erzeugte Instanz kann nicht mehr geändert werden.

- Praktisch sind die Methoden zum Herausfiltern des Protokolls usw.

JavaNet/URL/URLInfo.java

```
14      try
15        {
16          URL url = new URL(argv[0]);
17          System.out.println("Protokoll: " + url.getProtocol());
18          System.out.println("Port:      " + url.getPort());
19          System.out.println("Host:      " + url.getHost());
20          System.out.println("Datei:     " + url.getFile());
21          System.out.println("Referenz:  " + url.getRef());
22        }
23      catch (MalformedURLException mue)
24        { System.out.println(mue.getMessage()); }
```

12.3 Verbindung aufbauen mit openStream()

JavaNet/URL/URLLader/1/URLLader.java

```
19          InputStream urlinstr = url.openStream();
21          InputStreamReader isr = new InputStreamReader(urlinstr);
23          BufferedReader br = new BufferedReader(isr);
25          String zeile = br.readLine();
```

Quelltextanalyse

- Das Instantiieren eines **URL**-Objekts alleine erzeugt noch keine Netzwerkverbindung.

- Erst die Anweisung **`InputStream urlinstr = url.openStream();`** baut eine Verbindung zur der spezifizierten URL auf.
- Die Methode **`openStream()`** gibt einen einfachen **`InputStream`** zurück.
- Auf diesen **`InputStream`** wird ein **`InputStreamReader`** angesetzt.
- Auf diesen **`InputStreamReader`** wird ein **`BufferedReader`** angesetzt, um komfortabel auf einzelne Zeilen zugreifen zu können.
- Über die Methode **`readLine()`** der Klasse **`BufferedReader`** wird eine Zeile des HTML-Dokuments in einen String eingelesen. Dieser wird auf der Konsole ausgegeben.
- Dies wird solange wiederholt, bis das Dateiende erreicht ist.
- Danach wird der Eingabestrom geschlossen.
- Neben einer **`MalformedURLException`** ist noch eine **`IOException`** zu fangen.

12.4 Verbindung aufbauen mit openConnection()

JavaNet/URL/URLLader/2/URLLader.java

```
20          URLConnection urlconn = url.openConnection();
22          InputStream urlconninstr = urlconn.getInputStream();
```

Quelltextanalyse

- Statt mit **openStream()** lediglich einen einfachen **InputStream** von der URL anzufordern, können wir mit der Methode **openConnection()** eine Instanz von **URLConnection** erzeugen.
- Die Klasse **URLConnection** hat einen sehr großen Funktionsumfang. Siehe API-Referenz.
- Unter anderem gibt es die Objektmethode **getInputStream()**, die wieder einen **InputStream** zurückliefert.

12.5 Ausgabe in eine lokale Datei schreiben

JavaNet/URL/URLLader/3/URLLader.java

```
30          File ausgabeDatei = new File("vomWeb.html");
32          FileWriter fw = new FileWriter(ausgabeDatei);
34          BufferedWriter bw = new BufferedWriter(fw);
40              bw.write(zeile);
41              bw.newLine();
```

Quelltextanalyse

- Im Abschnitt 9.5 haben wir gesehen, wie wir eine Datei zeilenweise beschreiben können.
- Wir erzeugen zuerst eine Instanz der Klasse **File**, beschreiben sie mit einer Instanz von **FileWriter**, die wir noch in eine Instanz von **BufferedWriter** verpacken.

- Die Ausgabe auf die Konsole wird ersetzt durch die Anweisungen **`bw.write(zeile);`** und **`bw.newLine()`**.
- Zuletzt wird der Ausgabestrom geschlossen.

12.6 Schreibender Zugriff auf eine URL

Mit Hilfe der Klasse **`URL`** und **`URLConnection`** kann man eine Java-Applikation veranlassen, ein Perl-Skript auf einem entfernten Server aufzurufen.

12.6.1 Ein Perl-Skript

- Wir werden gleich die Java-Applikation **`ScriptStarter`** betrachten. Sie kann eine Verbindung zu einem Perl-Skript namens **`backwards.pl`** öffnen und es zur Ausführung bringen.
- Das Skript **`backwards.pl`** erwartet eine Zeichenkette als Eingabe. Es sendet als Antwort die Zeichenkette in umgekehrter Reihenfolge. (Konkret wird aus der Eingabe **`string=Mauer`** die Ausgabe **`reuaM`** erzeugt.)
- Das CGI-Skript sieht wie folgt aus:

JavaNet/URL/perl/backwards.pl

```
#!/usr/bin/perl
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
@pairs = split(/&/, $buffer);
foreach $pair (@pairs)
{
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+/ /;
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
    # Stop people from using subshells to execute commands
    $value =~ s/~!/ ~!/g;
    $FORM{$name} = $value;
}

print "Content-type: text/plain\n\n";
print "$FORM{'string'} reversed is: ";
$foo=reverse($FORM{'string'});
print "$foo\n";
exit 0;
```

12.6.2 Das Perl-Skript ausführen lassen

Welche Schritte muss eine Java-Applikation durchführen, um mit CGI-Skripten arbeiten zu können?

1. **URL** erzeugen.
2. **URLConnection** erzeugen, d. h. eine Verbindung aufbauen.
3. Die Verbindung beschreibbar machen.
4. Einen Ausgabestrom auf die Verbindung setzen. Der Ausgabestrom wird verbunden mit dem Standard-Eingabestrom des CGI-Skripts auf dem Server.
5. Den Ausgabestrom beschreiben und damit das CGI-Skript mit Parametern zur Ausführung bringen.
6. Den Ausgabestrom schließen.
7. Eventuell einen Eingabestrom erzeugen, um die Antworten des CGI-Skripts zu empfangen.

Das Perl-Skript gibt es auch bei Sun (unter dem Namen **backwards**) und kann so aufgerufen werden:

```
java ScriptStarter http://java.sun.com/cgi-bin/backwards Mauer
```

JavaNet/URL/perl/ScriptStarter.java

```
*/
public class ScriptStarter
{
  public static void main(String[] argv)
  {
    try
      {
        String text = argv[1];
        // Konvertiert den String in das x-www-form-urlencoded Format.
        // zweiter Paramter für encode() ab Java 1.4
        String textcodiert = URLEncoder.encode(text, "ISO-8859-1");

        // Eine URL erzeugen. argv[0] verweist hoffentlich auf ein Perl-Script.
        URL url = new URL(argv[0]);
        // Eine Verbindung zu dieser URL aufbauen und eine Instanz
        // von URLConnection erhalten.
        URLConnection urlconn = url.openConnection();
        // Setzt ein internes Flag, so dass in die Verbindung geschrieben
        // werden kann.
        urlconn.setDoOutput(true);
        // Einen Ausgabestrom auf die Verbindung setzen.
        OutputStream urlconnoutstr = urlconn.getOutputStream();
        // Auf diesen Stream einen OutputStreamWriter ansetzen.
        OutputStreamWriter osr = new OutputStreamWriter(urlconnoutstr);
        // Den OutputStreamWriter in einem BufferedWriter verpacken.
        BufferedWriter bw = new BufferedWriter(osr);
        // Eine Zeile schreiben.
        bw.write("string=" + textcodiert);
        // BufferedWriter schliessen
```

Quelltextanalyse

- Mit `String textcodiert = URLEncoder.encode(text, "ISO-8859-1");` wird der String **`text`** in ein versendbares Format konvertiert. (Leerzeichen und nicht-alphanumerische Zeichen müssen codiert werden.)
- Dann erzeugen wir eine **`URL`**, die auf das Skript verweist.
- Anschließend wird eine Verbindung aufgebaut.
- Per Default wird auf eine Verbindung lesend zugegriffen. Durch die Anweisung `urlconn.setDoOutput(true);` schalten wir um auf schreibenden Zugriff.
- Die nächsten drei Anweisungen erzeugen einen Ausgabestrom, mit dem unsere Applikation komfortabel einen String versenden kann.
- Nach dem Schreiben wird der Stream wieder geschlossen.
- Falls wir Antwort von der **`URL`** erwarten, können wir auf der Verbindung einen Eingabestrom aufsetzen.

12.7 Internetseiten via Applets laden

Auf den nächsten Seiten soll folgendes Applet entwickelt werden:

12.7.1 Ein URLButton

- Das Applet besteht aus einer Anzahl von Knöpfen.
- Jedem Knopf ist eine Internet-Adresse (URL) zugeordnet.
- Zu diesem Zweck wird eine Klasse **URLButton** definiert, die von **java.awt.Button** erbt.

JavaNet/URL/Applets/URLButton.java

```
9  class URLButton extends Button
10 {
11   private URL url;
12
13   public URLButton(String aufdruck, URL url)
14   {
15     super(aufdruck);
16     this.url = url;
17   }
18
19   public URL getURL() { return url; }
20 }
```

12.7.2 Ein Applet mit einem URLButton

JavaNet/URL/Applets/ZumWDR.java

```
11 public class ZumWDR extends Applet
12 {
13   private URLButton knopf;
14
15   public void init()
16   {
17     setLayout(new GridLayout(1, 1));
18     try
19       {
20         URL url = new URL("http://www.wdr.de");
21         knopf = new URLButton("zum WDR", url);
29         add(knopf);
30       }
31     catch (MalformedURLException mue)
32       { System.out.println("Fehlerhafte URL: " + mue.getMessage()); }
```

Quelltextanalyse

In der Methode **`init()`** werden folgende Schritte ausgeführt:

- Definition eines Layouts: **`GridLayout`**
- Dem Objekt **`url`** der Klasse **`URL`** wird die Internet-Adresse des WDR zugeordnet. Dieser Vorgang kann eine Ausnahme werfen, die gefangen werden muss.
- Ein neues Objekt **`knopf`** der Klasse **`URLButton`** wird erzeugt.

- Mit **`add(knopf);`** wird der neue **`URLButton`** dem Applet zugefügt.

Um den Knopf interaktiv zu machen, müssen wir wieder einen **`ActionListener`** implementieren und registrieren:

```
22          knopf.addActionListener(new ActionListener()
23            {
24              public void actionPerformed(ActionEvent ae)
25              {
26                getAppletContext().showDocument(knopf.getURL());
27              }
28            });
```

Quelltextanalyse

- Wir erzeugen wieder direkt bei der Registrierung des Listeners eine anonyme **`ActionListener`**-Klasse.
- Die Anweisung

  `getAppletContext().showDocument(knopf.getURL());`

  löst sich wie folgt auf:

  - **`knopf`** ist ein Objekt der Klasse **`URLButton`**, deren Methode **`getURL()`** die entsprechende URL zurück gibt.
  - Diese URL dient als Eingabe für **`getAppletContext().showDocument()`**.
  - Mit dieser Anweisung wird zunächst bestimmt, in welcher Umgebung das Applet sich befindet.

- Die Methode **showDocument()** fordert die virtuelle Maschine auf, das sich unter der übergebenen URL befindliche Dokument anzuzeigen.
- Wird das Applet nicht in einem Browser sondern z. B. im Appletviewer ausgeführt, dann wird der **showDocument()**-Aufruf ignoriert.

12.7.3 Ein Applet mit mehreren URLButtons

Wir können aus einem HTML-Dokument Parameter an Applets übergeben. Für unser Beispiel sieht das HTML-Dokument so aus.

JavaNet/URL/Applets/URLWahl.html

```
9  <APPLET CODE="URLWahl.class" WIDTH=150 HEIGHT=100>
10   <PARAM NAME="Anzahl" VALUE="3">
11
12   <PARAM NAME="Link0" VALUE="http://www.wdr.de">
13   <PARAM NAME="Name0" VALUE="WDR">
14
15   <PARAM NAME="Link1" VALUE="http://www.uni-wuppertal.de">
16   <PARAM NAME="Name1" VALUE="Uni␣Wuppertal">
17
18   <PARAM NAME="Link2" VALUE="http://www.google.de">
19   <PARAM NAME="Name2" VALUE="Google">
20 </APPLET>
```

Innerhalb eines **<PARAM>**-Tags sind die Attribute **NAME** und **VALUE** anzugeben. Ein Applet kann die Parameter dann mit der Methode **getParameter()** auslesen.

JavaNet/URL/Applets/URLWahl.java A

```
13   private URLButton[] knoepfe;
14   private int anzahlKnoepfe;
15
16   public void init()
17   {
18     anzahlKnoepfe = Integer.parseInt(getParameter("Anzahl"));
19     setLayout(new GridLayout(anzahlKnoepfe, 1));
20     knoepfe = new URLButton[anzahlKnoepfe];
```

Quelltextanalyse

- Das Applet liest den Wert das Parameters **`Anzahl`** und wandelt ihn in den einfachen Datentyp **`int`** um: **`anzahlKnoepfe`**
- Es wird ein **`GridLayout`** mit der entsprechenden Anzahl Zeilen angelegt.
- Ferner wird ein Array aus **`URLButton`**s der selben Anzahl angelegt.

Wir ergänzen jetzt die **class URLWahl**:

```
23          for (int i = 0; i < anzahlKnoepfe; i++)
24            {
25              URL url = new URL(getParameter("Link" + i));
26              knoepfe[i] = new URLButton(getParameter("Name" + i), url);
27              knoepfe[i].addActionListener(new ActionListener()
28                {
29                  public void actionPerformed(ActionEvent ae)
30                  {
31                    getAppletContext().showDocument(((URLButton)ae.getSource()).getURL());
32                  }
33                });
34              add(knoepfe[i]);
35            }
```

Quelltextanalyse

- Innerhalb des **try**-Blocks läuft jetzt eine Schleife.
- Es wird jeweils ein neues Objekt von der Klasse **URL** erzeugt, wobei der String als Parameter ausgelesen wird.
- Auch der auf den Knöpfen ausgedruckte Name wird als Parameter ausgelesen.
- Innerhalb der Methode **actionPerformed()** wird das betroffene Objekt jetzt mit Hilfe der Methode **getSource()** ermittelt.
- Diese von **java.util.EventObject** geerbte Methode liefert ein **Object** zurück, das hier in einen **URLButton** gecastet werden muss.

12.7.4 Wiederverwendbarkeit von Applets

Innerhalb eines HTML-Dokumentes kann ein Applet mit verschiedenen Parametern *mehrmals* aufgerufen werden.

Siehe **JavaNet/URL/Applets/URLWahlX3.html**.

12.8 Sockets

Während die Klassen **URL** und **URLConnection** die Netzprogrammierung auf einem relativ hohen Abstraktionslevel erlauben, benötigen wir für *Client-Server-Anwendungen* oft einen tieferen Einstieg.

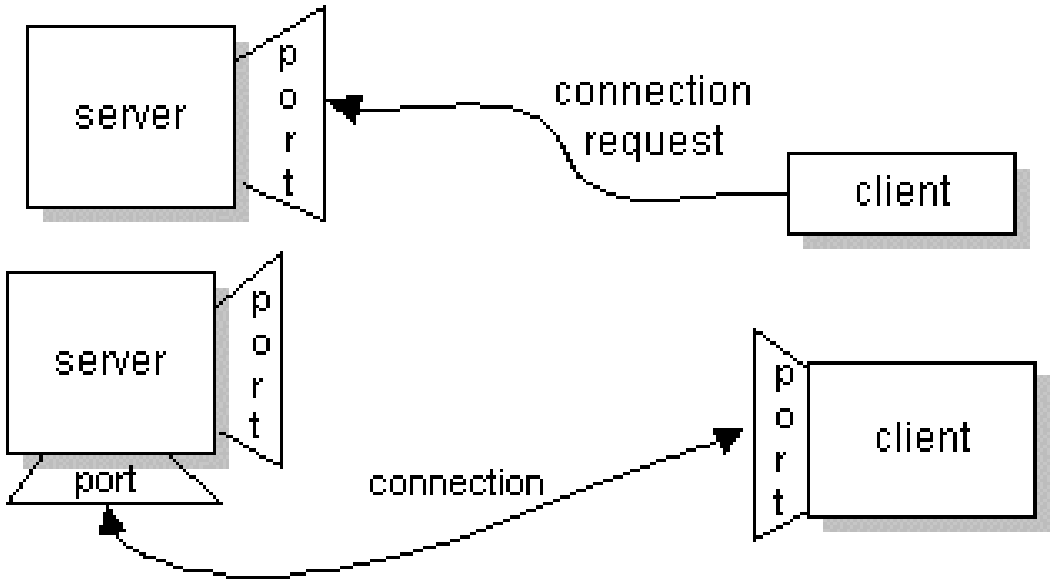
Hier unterstützt uns die Sprache Java mit den Klassen **Socket** und **ServerSocket**.

Ein *Socket* ist ein Endpunkt einer Verbindung, die Kommunikation zwischen zwei Programmen im Netzwerk in beide Richtungen erlaubt.

Sowohl das TCP- als auch das UDP-Protokoll verwenden *Ports*, um ein- und ausgehende Daten dem entsprechenden Prozess zuordnen zu können.

Ein Socket ist an eine spezielle Portnummer gebunden.

Rollenverteilung: Clientprogramme und Serverprogramme



12.9 Ein einfacher Client

Die folgende Anwendung implementiert ein Clientprogramm, das auf den Echo-Service eines Hosts zugreifen will. Diesen Service erreicht man üblicherweise über die Portnummer 7. Es handelt sich um eine TCP-Verbindung.

Welche Schritte muss der Client durchführen, um mit dem Server zu kommunizieren?

1. Socket erzeugen.
2. Verbindung aufbauen.

3. Ausgabestrom auf die Verbindung setzen. Der Ausgabestrom wird verbunden mit dem Eingabestrom des Serverprogramms.

4. Eingabestrom erzeugen, um die Antworten des Serverprogramms zu empfangen.

5. Über den Ein- und Ausgabestrom wird nun kommuniziert. Dazu ist ggf. ein eigenes Protokoll zu implementieren.

6. Ein- und Ausgabestrom schließen.

7. Socket schließen.

JavaNet/Socket/EchoClient.java

```
18          String host = argv[0];
21          Socket echoSocket = new Socket(host, 7);
27          OutputStream socketoutstr = echoSocket.getOutputStream();
29          OutputStreamWriter osr = new OutputStreamWriter(socketoutstr);
31          BufferedWriter bw = new BufferedWriter(osr);
37          InputStream socketinstr = echoSocket.getInputStream();
39          InputStreamReader isr = new InputStreamReader(socketinstr);
41          BufferedReader br = new BufferedReader(isr);
43          String anfrage = "Hallo";
44          String antwort;
50          bw.write(anfrage);
51          bw.newLine();
52          bw.flush();
54          antwort = br.readLine();
63          bw.close();
65          br.close();
67          echoSocket.close();
```

Quelltextanalyse

- Mit **new Socket(host,7)** erzeugen wir eine Instanz von **java.net.Socket**.
- Als Parameter übergeben wir den Namen des Servers, der auf Port 7 den Echoservice anbietet.

- Mit der Instantiierung wird gleichzeitig die Verbindung aufgebaut.
- Mögliche Ausnahmen sind **`UnknownHostException`** und **`IOException`**.
- Wir verschachteln nun wieder einige Ausgabe-Streams, damit der Client die Verbindung komfortabel beschreiben kann.
- Ebenso erzeugen wir Eingabe-Streams.
- Der Echoservice erwartet einen String als Eingabe und sendet ihn umgehend zurück.
- Anschließend werden die Streams und der Socket geschlossen.

12.10 Eine einfache Client-Server-Anwendung

Wir implementieren nun einen eigenen Echoservice, den wir über den Port 1192 laufen lassen. Das Clientprogramm **`MeinEchoClient`** erzeugt nun einen Socket namens **`meinEchoSocket`**, der nun statt Portnummer 7 die Portnummer 1192 wählt.

JavaNet/Socket/MeinEchoClient.java

```
21          Socket meinEchoSocket = new Socket(host, 1192);
```

Als Serverprogramm bilden wir nun das Verhalten des Echoservices nach.

JavaNet/Socket/MeinEchoServer.java

```
20          ServerSocket serverSocket = new ServerSocket(1192);
22          Socket clientSocket = serverSocket.accept();
28          OutputStream socketoutstr = clientSocket.getOutputStream();
66          clientSocket.close();
67          serverSocket.close();
```

Quelltextanalyse

- Mit **`ServerSocket serverSocket = new ServerSocket(1234);`** legen wir eine Instanz der Klasse **`ServerSocket`** an.
- Ein Serversocket horcht einen bestimmten Port ab auf Anfragen, die über das Netzwerk kommen.
- Die Instanz ruft die Methode **`accept()`** auf. Damit wartet das Serversocket auf eine Verbindung.
- Wurde die Verbindung erfolgreich aufgebaut, erhalten wir als Rückgabewert eine Instanz von **`Socket`**. Über dieses Socket wird dann die Kommunikation mit dem Client abgewickelt.
- Die Methode **`accept()`** ist blockierend, bis eine Verbindung aufgebaut wurde.
- Für die Kommunikation mit dem Client über **`clientSocket`** werden nun Server-seitig auch ein Ein- und Ausgabestrom angelegt.

- Das einfache Protokoll lautet hier:
  1. Lese einen String: **`anfrage`**
  2. Verarbeite den String: **`antwort = ...`**
  3. Schreibe Antwortstring
- Für kompliziertere Dienste sind eigene Protokolle zu implementieren.
- Am Ende sind der Ein- und Ausgabestrom, der Socket und der ServerSocket zu schließen.

12.11 Datagramme

- Während TCP ein verbindungsorientiertes Protokoll ist, arbeitet UDP (User Datagram Protocol) als verbindungsloses Protokoll.
- Nachrichten werden nicht mehr über eine vorher geöffnete Verbindung kommuniziert, sondern als *Datagramme* versendet.
- Ein Datagramm enthält neben der eigentlichen Nachricht noch Zusatzinformationen (z. B. Codierung von Sender und Empfänger), die es erlauben, dass es sich seinen Weg durch das Netzwerk alleine bahnen kann.

- Das zugehörige Protokoll UDP garantiert nicht, dass ein Datagramm jemals ankommt oder dass mehrere Datagramme in derselben Reihenfolge beim Empfänger eintreffen, in der sie gesendet wurden.
- Man sollte deshalb mit UDP keine großen Datenmengen versenden.
- Da aber UDP im Vergleich zu TCP wesentlich effizienter (schneller) ist, kann man sich durchaus einige sinnvolle Anwendungen vorstellen, die auf UDP zurückgreifen: Zeitansage, Ping, DNS (Domain Name System) und NFS (Network File System)
- In Java stehen uns die Klassen **`DatagramPacket`** und **`DatagramSocket`** zur Verfügung, um Kommunikation über UDP abzuwickeln.

12.11.1 `Bestellung` — ein UDP-Client

Die folgende Client-Server-Anwendung basiert nun auf dem UDP-Protokoll.

Welche Schritte muss eine Client-Applikation unter UDP durchführen, um mit einem Server zu kommunizieren?

1. Ein **`DatagramPacket`** erstellen. Neben der Nachricht (als **`byte`**-Array) muss Information über die *Länge*, den *Empfänger* und die *Portnummer* des Datagramms bereit gestellt werden.
2. Ein **`DatagramSocket`** erstellen.

3. Das **DatagramPacket** über das **DatagramSocket** versenden.

4. Den **DatagramSocket** schließen.

Der Client wird über ein Applet gesteuert, das in dieses HTML-Dokument eingebettet wird:

JavaNet/Datagramme/Bestellung.html

```
9  <APPLET CODE="Bestellung.class" WIDTH=150 HEIGHT=150>
10   <PARAM NAME="meinPort" VALUE="1193">
11 </APPLET>
```

JavaNet/Datagramme/Bestellung.java A

```
14   String meinHost;
15   int meinPort;
18   public void init()
19   {
26     meinHost = getCodeBase().getHost();
27     meinPort = Integer.parseInt(getParameter("meinPort"));
36   }
42   private void sendeNachricht(String nachricht)
43   {
67         byte[] data = new byte[nachricht.length()];
68         data = nachricht.getBytes();
69         InetAddress addr = InetAddress.getByName(meinHost);
70         DatagramPacket pack = new DatagramPacket(data, data.length, addr,
71                                                  meinPort);
72         DatagramSocket ds = new DatagramSocket();
73         ds.send(pack);
74         ds.close();
```

Quelltextanalyse

- In der Methode **`init()`** werden die Memberattribute **`meinHost`** und **`meinPort`** festgelegt.
- Mit **`meinHost = getCodeBase().getHost();`** wird der Name des Hosts ermittelt, von dem das Applet geladen wurde (und auf dem auch die Serveranwendung läuft).
- Die Portnummer **`meinPort`** wird aus der HTML-Seite ausgelesen.
- Die Methode **`sendeNachricht()`** wird von **`itemStateChanged()`**, **`start()`** und **`stop()`** aufgerufen.
- Sie soll den übergebenen String **`nachricht`** als **`DatagramPacket pack`** an den Server **`meinHost`** schicken.
- Der String **`nachricht`** wird umgewandelt in ein Feld **`data`** vom Typ **`byte`**.
- Es wird ein **`DatagramPacket pack`** erzeugt. In diesem wird festgelegt
  - was geschickt wird: **`data`**
  - wie groß das **`DatagramPacket`** ist: **`data.length`**
  - an welchen Rechner geschickt wird: **`addr`**
  - an welchen Port: **`meinPort`**
- Es wird ein **`DatagramSocket ds`** erzeugt, das das **`DatagramPacket pack`** versenden soll.

- Das **DatagramSocket ds** ist der Client, der die Kommunikation mit dem Server veranlasst.
- Die Kommunikation wird hier mit **ds.send(pack);** angestoßen.
- Schließlich wird das **DatagramSocket ds** wieder geschlossen.

12.11.2 Empfaenger — ein UDP-Server

Das entsprechende Serverprogramm wird durch eine Applikation gesteuert:

Welche Schritte muss eine Server-Applikation unter UDP durchführen, um mit Clients zu kommunizieren?

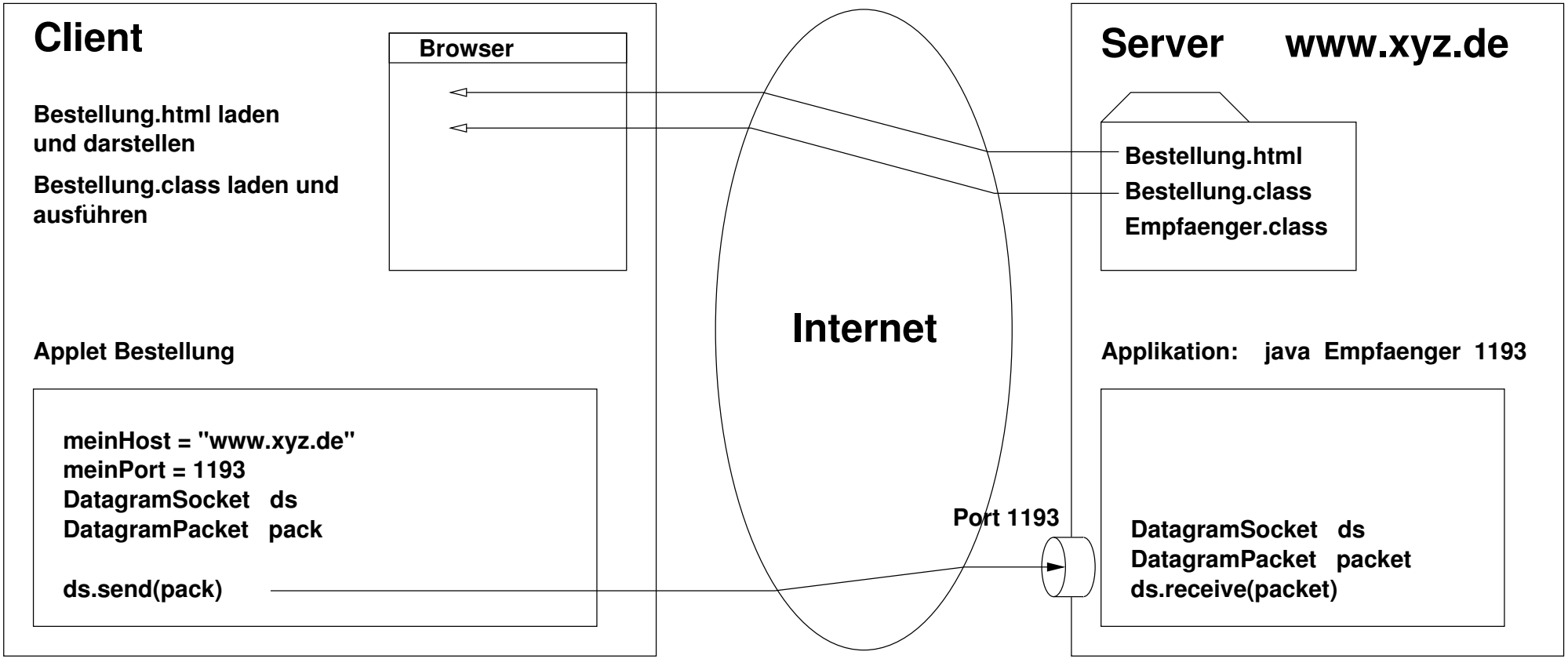
1. Ein **DatagramSocket** erstellen.
2. Ein leeres **DatagramPacket** erstellen.
3. Ein **DatagramPacket** über das **DatagramSocket** empfangen.
4. Das **DatagramPacket** auswerten.
5. Den **DatagramSocket** schließen.

JavaNet/Datagramme/Empfaenger.java

```
33          DatagramSocket ds = new DatagramSocket(Integer.parseInt(argv[0]));
34          while (true)
35            {
36              DatagramPacket pack = new DatagramPacket(new byte[1024], 1024);
37              ds.receive(pack);
38              String nachricht = new String(pack.getData());
39              System.out.println("Bestellung von: "
40                                 + pack.getAddress().getCanonicalHostName()
41                                 + ": "
42                                 + nachricht);
43            } // while
```

Quelltextanalyse

- Mit **`ds = new DatagramSocket(Integer.parseInt(argv[0]));`** erzeugen wir eine Instanz von **`DatagramSocket`**.
- Die Portnummer wird durch den Übergabeparameter **`argv[0]`** festgelegt.
- Das **`DatagramSocket ds`** soll später diesen Port abhorchen.
- Als nächstes erzeugen wir ein leeres **`DatagramPacket pack`**.
- Mit der Anweisung **`ds.receive(pack);`** veranlassen wir das **`DatagramSocket ds`** nun, auf das Eintreffen eines Pakets zu warten.
- Ist dies der Fall, wird das **`DatagramPacket pack`** ausgewertet und eine Meldung gedruckt.



12.12 Zusammenfassende Fragen

- Welche beiden Netzwerkprotokolle werden momentan von Java unterstützt?
- Wo sind sie im OSI-Schichtenmodell angesiedelt?

- Wie lauten die entsprechenden Klassen aus dem Package **java.net**?
- Was sind die für Java wichtigen Unterschiede zwischen TCP und UDP?
- Was sind Ports?
- Wie verwaltet Java URLs?
- Wie kann man eine Verbindung zu einer URL aufbauen?
- Wie erhält man schreibenden Zugriff auf eine Verbindung?
- Welche Schritte muss eine Java-Applikation durchführen, um mit CGI-Skripten arbeiten zu können?
- Welche Zugriffmöglichkeiten auf Netzwerke bieten Applets?
- Was sind Sockets?
- Welche Schritte muss eine Client-Applikation unter TCP durchführen, um mit einem Server zu kommunizieren?
- Wie verwendet man die Klasse **ServerSocket**?
- Was sind Protokolle?
- Welche Klassen stellt das Package **java.net** für die Kommunikation unter UDP bereit?

- Welche Schritte muss eine Client-Applikation unter UDP durchführen, um mit einem Server zu kommunizieren?
- Welche Schritte muss eine Server-Applikation unter UDP durchführen, um mit Clients zu kommunizieren?

13 Swing

Inhalte / Ziele. *Seit Java 1.2 gibt es mit Swing einen Nachfolger für das AWT. Swing bringt nicht nur einen Ersatz für fast alle Elemente des bisherigen AWT, es ist außerdem wesentlich umfangreicher geworden. An Hand mehrerer Beispielprogramme werden die wichtigsten Unterschiede und Neuerungen erklärt.*

13.1 Allgemeines zu Swing

Warum wurde das AWT ersetzt?

- Das AWT verwendete zur Darstellung Komponenten des jeweiligen Betriebssystems.
- Der Umfang des AWT beschränkte sich daher auf den „kleinsten gemeinsamen Nenner“ der unterstützen Plattformen.
- Obwohl AWT-Programme plattformunabhängig sein sollten, gab es mit verschiedenen VMs Unterschiede im Aussehen und Verhalten.

Was ist in Swing anders / besser?

- Swing-Komponenten sind *light-weight (leichtgewichtig)*. Sie sind komplett in Java programmiert und zeichnen sind selbst.
- Die Funktionalität ist stark erweitert und das Aussehen und die Bedienung vereinheitlicht.
- Das konkrete Aussehen *(Look and Feel)* kann zur Laufzeit umgeschaltet werden.

13.2 Von AWT zu Swing

Wir betrachten den Umstieg von AWT zu Swing am Beispiel der Kniffel-Applikation aus Kapitel 8.2.

Swing/Kniffel/Kniffel.java

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
```

- Es muss zusätzlich das Package **`javax.swing`** importiert werden.
- Obwohl AWT- und Swing-Komponenten nicht in einem Fenster gemischt werden sollten, muss **`java.awt`** trotzdem importiert werden. Klassen wie **`Color`** oder **`BorderLayout`** werden auch in Swing weiter verwendet.

```
10 public class Kniffel extends JFrame
```

- Vielen AWT-Klassen wird einfach ein **`J`** vorangestellt.
- Statt **`Frame`** erweitern wir nun die davon abgeleitete Klasse **`JFrame`**.

```
20    getContentPane().setBackground(Color.white);
38    getContentPane().setLayout(new BorderLayout());
39    getContentPane().add(unten, BorderLayout.SOUTH);
```

- Es können keine Komponenten direkt im **`JFrame`** platziert werden. Stattdessen werden die Elemente in einem **`Container`** untergebracht, der über die Methode **`getContentPane()`** erreicht wird.
- Der Versuch, die von **`Frame`** geerbten Methoden **`add()`** oder **`setLayout()`** direkt aufzurufen, löst eine Exception (**`Do not use ...`**) aus.

```
21    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

- Es ist bei einfachen Programmen nicht mehr nötig, selbst auf **`WindowEvent`**s zu reagieren.
- Standardmäßig schließt sich ein **`JFrame`** zwar selbst, aber ohne das ganze Programm zu beenden.
- Obiger Aufruf sorgt dafür, dass das Programm beim Schließen des Fenster beendet wird.

```
23    JPanel unten = new JPanel();
24    JButton k1 = new JButton("Neues␣Spiel");
```

- Auch bei **`JButton`** und **`JPanel`** ist jeweils ein **`J`** voranzustellen.
- Die beiden Klassen sind aber keineswegs von **`Button`** bzw. **`Panel`** abgeleitet.
- Gemeinsamer Vorfahr fast aller Swing-Komponenten (bis auf Top-Level-Container wie **`JFrame`** oder **`JDialog`**) ist die Klasse **`JComponent`**.

```
47      JMenuBar mBar = new JMenuBar();
48      JMenu spielMenu = new JMenu("Spiel");
49      JMenuItem spielNeu = new JMenuItem("Neu");
50      spielNeu.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_N,
51                                                     InputEvent.CTRL_DOWN_MASK));
54      spielMenu.add(spielNeu);
61      mBar.add(spielMenu);
62      setJMenuBar(mBar);
```

- Einfache Menüs programmiert man fast genauso, wie mit AWT. Geändert hat sich die Definition von Tastaturkürzeln.
- Solche **KeyStroke**s sind nicht an die Control-Taste gebunden. Die verschiedenen Masken für Control-, Alt-, Shift- und Meta-Taste können beliebig kombiniert werden.

```
18      //setSize(270, 200);//unnötig wegen pack() und Wuerfelbrett.setPreferredSize
64      pack();               // geerbt von Window
```

Swing/Kniffel/Wuerfelbrett.java

```
10  class Wuerfelbrett extends JPanel
21      setPreferredSize(new Dimension(270, 60));
```

- Auch das **Wuerfelbrett** ist nun von **JPanel** abgeleitet.
- Mit der von **JComponent** geerbten Methode **setPreferredSize()** kann das **JPanel** seine bevorzugte Größe anzeigen.

- Der **`pack()`**-Aufruf am Ende des **`Kniffel`**-Konstruktors sorgt dafür, dass sich das Hauptfenster (der **`JFrame`**) den bevorzugten Größen seiner Komponenten anpasst.
- Der **`setSize()`**-Aufruf wird damit überflüssig.

```
42  public void paintComponent(Graphics g)
43  {
44    super.paintComponent(g); // oder im Konstr. setOpaque(false)
45    // Uebermale alles, was vorher war,
46    // mit einem grossen weissen Rechteck.
47    g.setColor(Color.white);
48    g.fillRect(10, 10, 241, 41);
49    // Male nun die Wuerfel
50    zeichneWuerfel(g);
51  }
```

- Die von **`JComponent`** geerbte **`paint()`**-Methode ruft nacheinander die folgenden drei Methoden auf: **`paintComponent()`**, **`paintBorder()`** und **`paintChildren()`**.
- Wir könnten wie gewohnt **`paint()`** überschreiben, es genügt hier aber, **`paintComponent()`** zu überlagern.
- Viele Swing-Komponenten haben standardmäßig einen transparenten Hintergrund. Deshalb müssen wir mit **`super.paintComponent(g);`** zusätzlich die geerbte Methode aufrufen. Alternativ könnten wir im Konstruktor die Anweisung **`setOpaque(false);`** ergänzen.

13.3 Swing-Applets

Wir wollen das **Duke**-Applet aus Abschnitt 11.7.3 in ein Swing-Applet konvertieren. Zum Testen müssen wir den **appletviewer** verwenden, da die gängigen Browser Swing noch nicht unterstützen. Es sind genau zwei Zeilen zu ändern:

Swing/Duke/5-JApplet/Duke.java A

```
8  import javax.swing.*;
10 public class Duke extends JApplet implements MouseListener
```

- Die Klasse **JApplet** ist von **Applet** abgeleitet, befindet sich aber im Package **javax.swing**.
- Das Applet verhält sich genauso, wie die AWT-Version zuvor.

13.3.1 `paint()` und `update()` in Swing

Swing/Duke/6-JPanel/Duke.java A

```
10 public class Duke extends JApplet
11 {
12   DukePanel dp;
13
14   public void init()
15   {
16     dp = new DukePanel(this);
17     getContentPane().add(dp);
18   }
19
20   class DukePanel extends JPanel implements MouseListener
21   {
34     public void paint(Graphics g)
35     {
36       super.paint(g);
50     public void update(Graphics g)
51     {
52       // wird nie aufgerufen !
```

- Die gesamte Funktionalität lagern wir nun aus in ein **`DukePanel`**.
- Wir bei Applikationen auch wird das **`DukePanel`** dem **`JApplet`** nicht direkt hinzugefügt, sondern in dessen **`contentPane`**.
- Wir beobachten, dass ausschließlich **`paint()`** und niemals **`update()`** aufgerufen wird. Dies ist das Standardverhalten aller Swing-Komponenten.

- Auch **`JComponent.update()`** und **`JApplet.update()`** rufen direkt **`paint()`** auf, ohne den Hintergrund zu löschen.

13.4 Look and Feel

Das *Look and Feel* beschreibt die graphische Darstellung der einzelnene Komponenten sowie deren Reaktion auf Benutzereingaben.

Java unterstützt mehrere Look-and-Feels, je nach Betriebssystem bzw. virtueller Maschine stehen nicht immer alle zur Verfügung.

Das folgende Programm erzeugt automatisch ein Menü, mit dem man zwischen den einzelnen Look-and-Feels umschalten kann.

Swing/LookAndFeel/LAFBsp.java

```
13  public class LAFBsp extends JFrame
14  {
15    UIManager.LookAndFeelInfo[] infoLAF;
19    public LAFBsp()
20    {
21      super("LookAndFeel Beispiele");
90      // JMenu + LookAndFeel
91      setLAF(UIManager.getCrossPlatformLookAndFeelClassName());
92      JMenuBar mBar = new JMenuBar();
93      JMenu menuEinst = new JMenu("Einstellungen");
94      JMenu menuLAF = new JMenu("Look and Feel");
95      ButtonGroup bgLAF = new ButtonGroup();
96      infoLAF = UIManager.getInstalledLookAndFeels();
97      for (int i = 0; i < infoLAF.length; i++)
98        {
99          JRadioButtonMenuItem rbmiLAF
100           = new JRadioButtonMenuItem(infoLAF[i].getName());
101         rbmiLAF.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_0 + i,
102                                                       InputEvent.CTRL_DOWN_MASK));
103         rbmiLAF.setActionCommand(infoLAF[i].getClassName());
104         rbmiLAF.addActionListener(new ActionListener()
105           {
106             public void actionPerformed(ActionEvent ae)
107             { setLAF(ae.getActionCommand()); }
108           });
109         bgLAF.add(rbmiLAF);
110         menuLAF.add(rbmiLAF);
111       }
112     menuEinst.add(menuLAF);
113     mBar.add(menuEinst);
```

```
119
120   private void setLAF(String laf)
121   {
122     // aendert das LookAndFeel
123     // laf: der Name einer LookAndFeel-Klasse
124     try
125       {
126         UIManager.setLookAndFeel(laf);
127         SwingUtilities.updateComponentTreeUI(this);
128         pack();
129       }
130     catch (ClassNotFoundException e) { System.err.println(e); }
131     catch (InstantiationException e) { System.err.println(e); }
132     catch (IllegalAccessException e) { System.err.println(e); }
133     catch (UnsupportedLookAndFeelException e) { System.err.println(e); }
```

- Für die Verwaltung von Look-and-Feels ist die Klasse **`UIManager`** *User Interface Manager* zuständig.
- Dessen Methode **`getInstalledLookAndFeels()`** liefert Informationen über die installierten Look-and-Feels in einem Array vom Typ **`UIManager.LookAndFeelInfo[]`** zurück.
- Die Beschriftung der Menüpunkte wird mit **`getName()`** automatisch erzeugt.
- Um das Look-And-Feel umzuschalten wird der genaue Klassenname benötigt, zu ermitteln über **`getClassName()`**.
- Diesen Namen können wir mit **`setActionCommand()`** dem Menüpunkt zuordnen und später über **`ActionEvent.getActionCommand()`** wieder auslesen.

- Das eigentliche Umschalten erfolgt in drei Schritten:
  1. **`UIManager.setLookAndFeel()`** aktiviert das neue Look-and-Feel.
  2. **`SwingUtilities.updateComponentTreeUI(this)`** informiert alle Komponenten über die Änderung.
  3. **`pack()`** passt das Layout an die veränderten Größen an.
- Die beiden Look-and-Feels „Metal“ und „Motif“ sind immer vorhanden.
- Steht ein Look-and-Feel nicht zur Verfügung (z. B. „Windows“ unter Linux), dann wird eine **`UnsupportedLookAndFeelException`** geworfen.
- Den Klassennamen für „Metal“ erhält man auch über **`UIManager.getCrossPlatformLookAndFeelClassName()`**, den eines systemabhängigen (unter Linux ebenfalls „Metal“, „Windows“ unter Windows) über **`UIManager.getSystemLookAndFeelClassName()`**.

13.5 Ausgewählte Swing-Komponenten

Das Look-and-Feel-Programm enthält zur Demonstration einige ausgewählte Komponenten aus Swing.

```
27      JButton jbut = new JButton("ein JButton");
28      jbut.setToolTipText("Das ist ein JButton");
29      jbut.setMnemonic(KeyEvent.VK_B);
```

- Allen von **`JCompononent`** abgeleiteten Swing-Element kann mit **`setToolTipText()`** ein erklärender Text hinzugefügt werden, der angezeigt wird, wenn der Mauszeiger sich über dem Element befindet.

- Tastenkürzel für Knöpfe und andere beschriftete Dialogelemente werden mit **`setMnemonic()`** gesetzt. Ist der Buchstabe in der Beschriftung vorhanden, so wird sein erstes Vorkommen unterstrichen (unabhängig von Groß- und Kleinschreibung). Ausgelöst wird das Tastenkürzel durch gleichzeitiges Drücken der **`Alt`**-Taste.

```
38      JButton jbutEnde = new JButton("Ende", new ImageIcon("exit.png"));
```

- Zur Beschriftung können zusätzlich (oder auch ausschließlich) Icons verwendet werden.

```
16  JFrame myself;
22    myself = this;
30    jbut.addActionListener(new ActionListener()
31      {
32        public void actionPerformed(ActionEvent ae)
33        { JOptionPane.showMessageDialog(myself, "JButton gedrückt"); }
34      });
```

- Die Klasse **JOptionPane** stellt eine Reihe fertiger Dialogfenster bereit, zum Beispiel dieses einfache Hinweisfenster.

- Der erste Parameter von **showMessageDialog()** muss eine Referenz auf den „Vater-Frame“ enthalten. Da wir in der anonymen Klasse hierfür nicht **this** verwenden können, benötigen wir die Referenz **myself**.

```
44          if (JOptionPane.showOptionDialog(myself, "Programm beenden?", "Frage",
45                                           JOptionPane.YES_NO_OPTION,
46                                           JOptionPane.QUESTION_MESSAGE,
47                                           null, null, null)
48              == JOptionPane.YES_OPTION)
49            System.exit(0);
```

- Die Klasse **JOptionPane** bietet noch weitere Arten von Dialogfenster: **showOptionDialog** für Ja-Nein-(Abbrechen-)Fragen, **showConfirmDialog** für OK-(Abbrechen-)Dialoge und **showInputDialog** für einfache Texteingaben.

```
58    JLabel jlabHTML = new JLabel("<HTML><SMALL>ein␣JLabel␣mit␣<I>HTML</I>" +
59                                 "-Beschriftung</SMALL></HTML>");
```

- Zur formatierten Beschriftung von **JLabel**s, **JButton**s usw. können HTML-Tags verwenden werden. Der Text muss dazu von ***<HTML>*** und ***</HTML>*** eingeschlossen werden.

```
64    JPanel jchbPanel = new JPanel();
65    jchbPanel.setBorder(BorderFactory.createTitledBorder("Checkboxen"));
67    JCheckBox jchb1 = new JCheckBox("eins", true);
69    jchbPanel.add(jchb1);
73    getContentPane().add(jchbPanel);
```

- Zusammengehörige Elemente können in einem eigenen **JPanel** zusammengefasst und mit Hilfe eines Rahmens optisch gruppiert werden.

```
77    ButtonGroup bgr = new ButtonGroup();
80    JRadioButton jrbut1 = new JRadioButton("ett", true);
82    bgr.add(jrbut1);
84    JRadioButton jrbut2 = new JRadioButton("två");
86    bgr.add(jrbut2);
```

- Soll von mehreren **JRadioButton**s immer nur einer ausgewählt sein, müssen diese alle einer gemeinsamen **ButtonGroup** hinzugefügt werden.

13.6 Ein einfacher Texteditor

Wir wollen einen kleinen Texteditor programmieren, mit dem man mehrere Dateien gleichzeitig in verschiedenen Unterfenstern editieren kann. Java unterstützt das Konzept des *MDI (Multiple Document Interface)* mit **`JInternalFrames`**.

Swing/Editor/SwingEditor.java

```
11 public class SwingEditor extends JFrame
12 {
17   public SwingEditor()
18   {
23     desktop = new JDesktopPane();
24     desktop.setPreferredSize(new Dimension(800, 600));
25     setContentPane(desktop);
```

- Unterfenster können nicht einfach dem über **`getContentPane()`** zugänglichen Standardcontainer hinzugefügt werden. Stattdessen benötigen wir ein **`JDesktopPane`**.

```
121   private void open()
122   {
123     JFileChooser chooser = new JFileChooser(new File("."));
124     if (chooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
125       {
126         File f = chooser.getSelectedFile();
127         if (f.canRead())
128           {
129             InternalEditorFrame ief = new InternalEditorFrame(f);
130             desktop.add(ief);
131             desktop.setPosition(ief, 0);    // als oberstes Fenster anzeigen
132             desktop.setSelectedFrame(ief);  // funktioniert z.T. nicht
133           }
134         else
135           JOptionPane.showMessageDialog(this, "Datei " + f.getName()
136                                         + " nicht lesbar");
137       }
138   }
```

- Die Klasse **`JFileChooser`** bietet ein komplettes Dialogfenster zur Dateiauswahl.
- **`showOpenDialog()`** zeigt das Fenster zum Öffnen von Dateien an.
- Ist die ausgewählte Datei lesbar, so wird ein neues internes Fenster erzeugt und dem **`JDesktopPane desktop`** hinzugefügt.
- **`setPosition(ief, 0)`** zeigt das neue Fenster als oberstes Fenster an, **`setSelectedFrame()`** soll dem neuen Fenster den Fokus geben, was leider nicht in allen Java-Version klappt.

```
140   private void speichern()
141   {
142     InternalEditorFrame selFr = (InternalEditorFrame)desktop.getSelectedFrame();
143     if (selFr != null)
144       selFr.speichern();
145   }
```

- Wird der Menüpunkt „Speichern“ angewählt, müssen wir mit **`getSelectedFrame()`** zunächst das gerade aktive Unterfenster bestimmen.

Swing/Editor/InternalEditorFrame.java

```
10 public class InternalEditorFrame extends JInternalFrame
11 {
12   JScrollPane scrpane;
13   JEditorPane editpane;
14   File theFile;
15
16   InternalEditorFrame(File aFile)
17   {
18     super("tmp-Name", true, true, true, true);
19     theFile = aFile;
20     if (theFile != null)
21       setTitle(theFile.getName());
22     else
23       setTitle("unbenannt");
24     setPreferredSize(new Dimension(600, 400));
25     editpane = new JEditorPane();
26     editpane.setAutoscrolls(true);
27     scrpane = new JScrollPane(editpane);
28     getContentPane().add(scrpane);
29     if (theFile != null)
30       {
31         try
32           {
33             editpane.read(new BufferedReader(new FileReader(theFile)),
34                           null);
35           }
36         catch (FileNotFoundException fnfe)
37           { JOptionPane.showMessageDialog(this, fnfe.getMessage()); }
38         catch (IOException ioe)
39           { JOptionPane.showMessageDialog(this, ioe.getMessage()); }
40       }
41     pack();
42     setVisible(true);
43   }
```

- Unterfenster müssen von **`JInternalFrame`** abgeleitet werden.
- Die Klasse **`JEditorPane()`** stellt alle grundlegenden Fähigkeiten eines Editors bereit.
- Letztere packen wir in ein **`JScrollPane`** ein. So werden automatisch vertikale oder horizontale Scrollbalken angezeigt, sobald der Text nicht mehr in das Fenster passt.
- Die Methode **`JEditorPane.read()`** füllt das Textfenster mit Text aus einem Eingabestrom, hier aus einer Datei.

```
45  public void speichern()
46  {
47    if (theFile != null)
51            editpane.write(new BufferedWriter(new FileWriter(theFile)));
58    else
59      speichernUnter();
62  public void speichernUnter()
63  {
64    JFileChooser chooser = new JFileChooser(theFile == null ?
65                                            new File(".") : theFile);
66    if (chooser.showSaveDialog(this) == JFileChooser.APPROVE_OPTION)
67      {
68        File f = chooser.getSelectedFile();
69        try
70          {
71            f.createNewFile(); // hier passiert nichts, falls f schon existiert
72            theFile = f;
73            setTitle(f.getName());
74            editpane.write(new BufferedWriter(new FileWriter(theFile)));
```

- Zum Speichern in einen Ausgabestrom dient die Methode **`JEditorPane.write()`**.
- Ist das Fenster über den Menüpunkt „Neu“ erzeugt worden, rufen wir stattdessen **`speichernUnter()`** auf.
- Dort verwenden wir diesmal **`showSaveDialog()`** aus **`JFileChooser`**.

14 Das Package `java.sql`

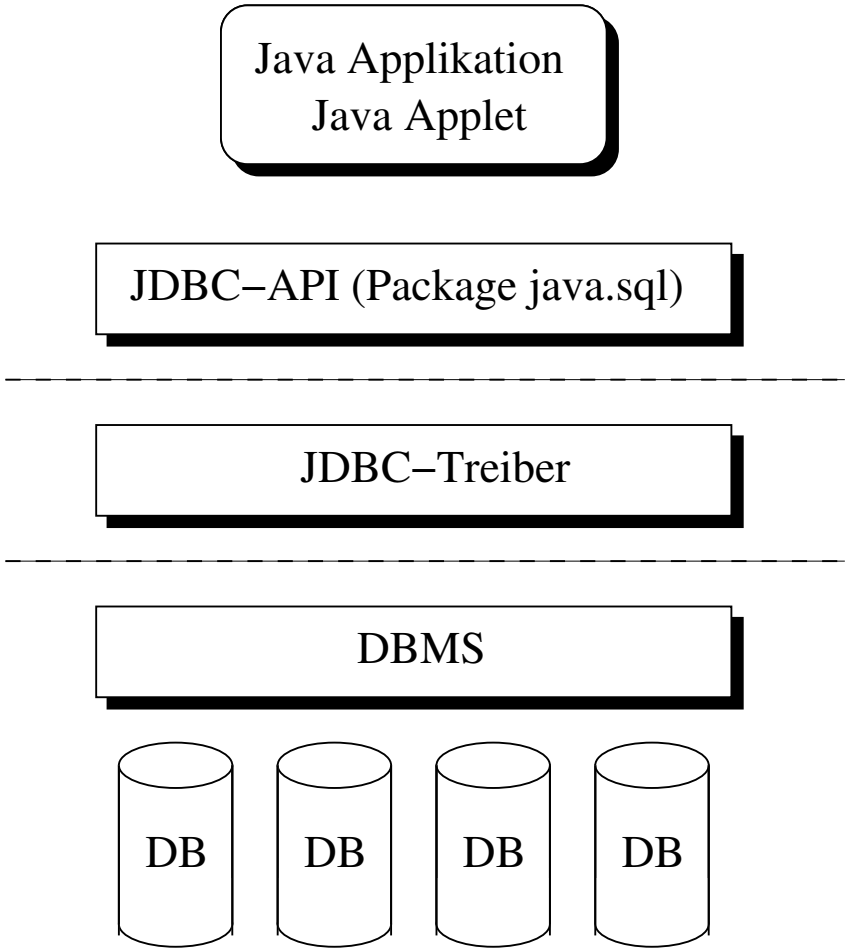
Inhalte / Ziele. *Eines der wichtigsten kommerziellen Einsatzgebiete von Java ist die Entwicklung von Datenbankanwendungen. Die Plattformunabhängigkeit wird durch die Java Database Connectivity (JDBC) gewährleistet. Mit Java entwickelte Software kann also weiterverwendet werden, wenn die zu Grunde liegende Datenbank (z. B. Oracle, MS-Access, MySQL) ausgewechselt wird. Im Gegensatz zu den übrigen Kapiteln benötigen wir hier neben dem JDK (mit dem Package* **`java.sql`***) zusätzlich ein Datenbank-Management-System (DBMS) sowie einen entsprechenden Treiber.*

14.1 Systemarchitektur

Folgende Komponenten benötigen wir für Datenbankzugriffe in Java:

- JDK mit JDBC-API (Package **`java.sql`**)
- eine Datenbank (DB) mit Datenbank-Management-System (DBMS)
- einen JDBC-Treiber für diese Datenbank

Wir skizzieren hier den einfachsten Fall einer Systemarchitektur, d. h. dass sowohl der Datenbank-Server als auch der in Java programmierte Datenbank-Client auf einem Rechner laufen. Es lassen sich auch Datenbankzugriffe unter verteilten Rechnernetzarchitekturen und über das Internet (Web-Browser) realisieren.



14.2 Relationale Datenbanken

Es folgt eine kurz gehaltene Einführung in die grundlegenden Konzepte relationaler Datenbanken.

14.2.1 Grundlagen

- Eine *Datenbank (DB)* ist eine Zusammenstellung von Daten (Tabellen) und Bearbeitungswerkzeugen (Abfragen, Formulare und Berichte), die sich auf einen Themenbereich beziehen oder einen bestimmten Zweck verfolgen.
- Ein *Datenbank-Management-System (DBMS)* ist ein Server-Programm, das Informationen in einer (oder mehreren) Datenbank(en) speichert und den Zugriff auf die enthaltenen Daten regelt.
- *Microsoft Access*, *Oracle*, *DB2* oder *MySQL* gehören zu den *relationalen* Datenbank-Systemen. Solche Systeme speichern Informationen in Tabellen (= Relationen). Verschiedene Tabellen werden durch Beziehungen verknüpft. Es gibt $1:1$, $1:N$ und $M:N$ Relationen.
- *Tabellen* bestehen aus mehreren Zeilen und Spalten. Eine *Spalte* (auch *Field* genannt) enthält dabei immer die gleiche Art von Information, z. B. **`Nachname`**. Spalteninformation bezeichnet man auch als *Attribut*. Die *Zeilen* einer Tabelle enthalten die *Datensätze*. Ein Datensatz setzt sich aus mehreren Attributen zusammen.
- Eine der Spalten kann (und sollte) einen *Primärschlüssel (primary key)* enthalten, der für jeden Datensatz eindeutig sein muss.

Wir werden in den folgenden Beispielen mit einer MySQL-Datenbank arbeiten. Dabei greifen wir nur auf eine einzelne Tabelle **personal** zu. Es ist zu betonen, dass bei einem guten Datenbankdesign weitere wichtige Regeln zu beachten sind, um kritische Probleme wie z. B. *Redundanz*, *Inkonsistenz* und *Einfüge-*, *Pflege-* oder *Lösch-Anomalien* zu vermeiden. Ein Hilfsmittel dazu ist z. B. die *Normalisierung* von Tabellen.

14.2.2 Arbeiten mit Datenbanken am Beispiel von MySQL

Alles Wichtige finden Sie in:

- Krämer / Hofschuster, Kurzanleitung zur Datenbank MySQL:
  http://www.math.uni-wuppertal.de/wrswt/ss03/swt/mysql.pdf

Datenbank-Management-Systeme differenzieren ihre Benutzer typischerweise in Gruppen:

- Datenbankadministratoren (**root**) haben uneingeschränkten Zugriff auf eine Datenbank. Sie starten und beenden den Datenbank-Server und vergeben Zugriffsrechte für die übrigen Benutzer.

  Zuerst initialisieren wir ein Datenbankverzeichnis für MySQL und vergeben ein Passwort für root.

JavaSQL/Vorbereitung/dbinst.sh

```
18      mysql_install_db --datadir=$HOME/Java/Vorlesung/Quelltexte/JavaSQL/db
22  ../StartStop/sqld_start
27  read -p "Passwort: " -s PASS1
33      mysqladmin --socket=$HOME/Java/Vorlesung/Quelltexte/JavaSQL/db/mysql.sock \
34          -u root password $PASS1
```

Danach legen wir eine neue Datenbank **JavaVorl** an und erlauben einem lokalen Benutzer, die Datenbank zu bearbeiten.

JavaSQL/Vorbereitung/createdb.sql

```
4 CREATE DATABASE JavaVorl;
5 GRANT ALL ON JavaVorl.* TO username@localhost;
```

Ferner legt root unter Verwendung der *Data Definition Language (DDL)* Tabellen an. Dazu gehört u. a. die Spezifikation der Attribute.

JavaSQL/Vorbereitung/setupTablePersonal.sql

```
6  USE JavaVorl;

12 CREATE TABLE personal
13   (
14     Name VARCHAR(20) NOT NULL,
15     PersNr INTEGER PRIMARY KEY,
16     StOrt VARCHAR(20),
17     UBereich VARCHAR(20),
18     Abtr VARCHAR(5),
19     GebNr INTEGER,
20     Gehalt DOUBLE
21   );
```

Das SQL-Kommando

```
25 DESCRIBE personal;
```

erzeugt dann folgende Ausgabe:

```
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| Name     | varchar(20) |      |     |         |       |
| PersNr   | int(11)     |      | PRI | 0       |       |
| StOrt    | varchar(20) | YES  |     | NULL    |       |
| UBereich | varchar(20) | YES  |     | NULL    |       |
| Abtr     | varchar(5)  | YES  |     | NULL    |       |
| GebNr    | int(11)     | YES  |     | NULL    |       |
| Gehalt   | double      | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
```

- Datenbankanwender können mit der *Data Manipulation Language (DML)* in solchen Tabellen dann Datensätze einfügen (**INSERT**), pflegen (**UPDATE**) oder löschen (**DELETE**). Außerdem können *Abfragen (queries)* formuliert und ausgeführt werden. Dazu dient die *structured query language (SQL)*.

JavaSQL/Vorbereitung/fillTablePersonal.sql

```
5  USE JavaVorl;

11 INSERT INTO personal
12   VALUES('Frits', 17, 'Aholming', 'Elektro', 'F&E', 11, 22000);
```

Die Zeile

```
25 SELECT * FROM personal;
```

gibt die ganze Tabelle aus:

```
+----------+--------+----------+----------+-------+-------+--------+
| Name     | PersNr | StOrt    | UBereich | Abtr  | GebNr | Gehalt |
+----------+--------+----------+----------+-------+-------+--------+
| Frits    |     17 | Aholming | Elektro  | F&E   |    11 |  22000 |
| Frans    |   9133 | Aholming | Elektro  | Contr |    11 |  44100 |
| Lubbe    |    321 | Aholming | Elektro  | Vertr |     8 |  19800 |
| Enzian   |     18 | München  | Mechanik | F&E   |     2 |  26500 |
| Truhel   |     54 | Karben   | Kfz      | F&E   |     2 |  22800 |
| Jöndhard |    739 | Karben   | Kfz      | F&E   |     2 |  22900 |
+----------+--------+----------+----------+-------+-------+--------+
```

- Mit dieser *SQL-Anweisung (Statement)*

```
SELECT Name, Gehalt, UBereich, GebNr FROM personal WHERE GebNr=2;
```

erhalten wir beispielsweise als Ergebnismenge *(ResultSet)*

```
+----------+--------+----------+-------+
| Name     | Gehalt | UBereich | GebNr |
+----------+--------+----------+-------+
| Enzian   |  53000 | Mechanik |     2 |
| Truhel   |  43500 | Kfz      |     2 |
| Jöndhard |  45300 | Kfz      |     2 |
+----------+--------+----------+-------+
```

14.3 Verbindung zu einer MySQL-Datenbank aufbauen

Für eine sinnvolle Anwendung der folgenden Beispiele setzen wir voraus, dass die Java-Applikationen auf einem Rechner ausgeführt werden, auf dem ein MySQL-Datenbank-System installiert und der entsprechende Dämon aktiviert ist. Ferner soll es eine Datenbank **`JavaVorl`** geben, die zumindest die oben beschriebene Tabelle **`personal`** enthält.

Als Verbindung zwischen dem DBMS und einer Java-Applikation benötigen wir dann noch einen JDBC-Treiber. MySQL AB, der Hersteller der Datenbank MySQL, stellt mit dem MySQL Connector/J einen eigenen JDBC-Treiber zur Verfügung. Aus Sicht des JDK ist ein solcher Treiber ein Package mit einer Sammlung von **`*.class`**-Dateien. Da dieses Package aber nicht zum Standardumfang des JDKs gehört, muss dem Compiler **`javac`** bekannt gegeben werden, wo es zu finden ist.

Eine Möglichkeit besteht darin, die Umgebungsvariable **CLASSPATH** um das entsprechende Verzeichnis oder jar-Archiv zu ergänzen, auf den IT-Rechnern z. B. um

```
/usr/local/sw/mysql-connector-java-3.0.8-stable/mysql-connector-java-3.0.8-stable-bin.jar
```

Details finden sich in den Installationsanweisungen zum Treiber.

Nach diesen Vorbereitungen können wir nun schrittweise eine einfache Java-Applikation erstellen:

JavaSQL/SimpleQuery/1-TreiberLaden/SimpleQuery.java

```
1  import java.sql.*;
12     try
13       {
14         System.out.println("* Treiber laden");
15         Class.forName("com.mysql.jdbc.Driver").newInstance();
18       }
19     catch (ClassNotFoundException cnfe)
20       {
21         System.err.println("Konnte Treiber nicht laden.");
22         cnfe.printStackTrace();
23       }
24     catch (InstantiationException ie)
25       {
26         System.err.println("Konnte Treiber nicht laden.");
27         ie.printStackTrace();
28       }
29     catch (IllegalAccessException iae)
30       {
31         System.err.println("Konnte Treiber nicht laden.");
32         iae.printStackTrace();
33       }
```

Quelltextanalyse

- Wir binden das Package **java.sql** ein.
- Mit der Anweisung

  ```
  Class.forName("com.mysql.jdbc.Driver").newInstance();
  ```

  wird der Treiber in die Java-Applikation geladen. Es wird dazu die statische Methode **forName()** der Klasse **Class** aufgerufen. (Als Parameter ist der volle qualifizierende Name anzugeben, für Details siehe API. Anderes Beispiel: **java.awt.Button**.) Diese gibt eine Referenz auf die Treiberklasse zurück und bewirkt so, dass der Treiber geladen wird.
- Konnte der Treiber nicht geladen werden, wird eine von drei möglichen Exceptions geworfen, die alle gefangen werden müssen.

Nach diesen Vorbereitungen können wir nun schrittweise eine einfache Java-Applikation erstellen:

JavaSQL/SimpleQuery/2-Verbinden/SimpleQuery.java

```
14      final String hostname = "localhost";
15      final String port = "3306";
16      final String dbname = "JavaVorl";
17      final String user = System.getProperty("user.name");
18      final String password = "";

20      Connection conn = null;

42      try
43        {
44          System.out.println("* Verbindung aufbauen");
45          String url = "jdbc:mysql://" + hostname + ":" + port + "/" + dbname;
46          conn = DriverManager.getConnection(url, user, password);
47
48          System.out.println("* Datenbank-Verbindung beenden");
49          conn.close();
50        }
51      catch (SQLException sqle)
52        {
53          System.out.println("SQLException: " + sqle.getMessage());
54          System.out.println("SQLState: " + sqle.getSQLState());
55          System.out.println("VendorError: " + sqle.getErrorCode());
56          sqle.printStackTrace();
57        }
```

Quelltextanalyse

- Mit der Klasse **`Connection`** können wir nun eine Verbindung zur Datenbank erstellen. (Im Package **`java.sql`** ist **`Connection`** als Interface deklariert. Die konkrete Implementierung als Klasse ist Bestandteil des Treibers.) Eine Instanz dieser Klasse wird erzeugt durch Anwendung der statischen Methode **`getConnection()`** der Klasse **`DriverManager`**.

Durch die Anweisung

```
conn = DriverManager.getConnection(url, user, password);
```

werden folgende Schritte durchgeführt:

1. Unter den geladenen Treibern wird ein Passender ausgesucht. Wird keiner gefunden, wird die Ausnahme **SQLException** geworfen.
2. Danach wird versucht, eine Verbindung zur im String **url** spezifizierten Datenbank aufzubauen. Ferner wird geprüft, ob der Benutzer **user** mit dem Passwort **password** zugangsberechtigt ist. Schlägt der Aufbau der Verbindung fehl, wird eine **SQLException** geworfen.

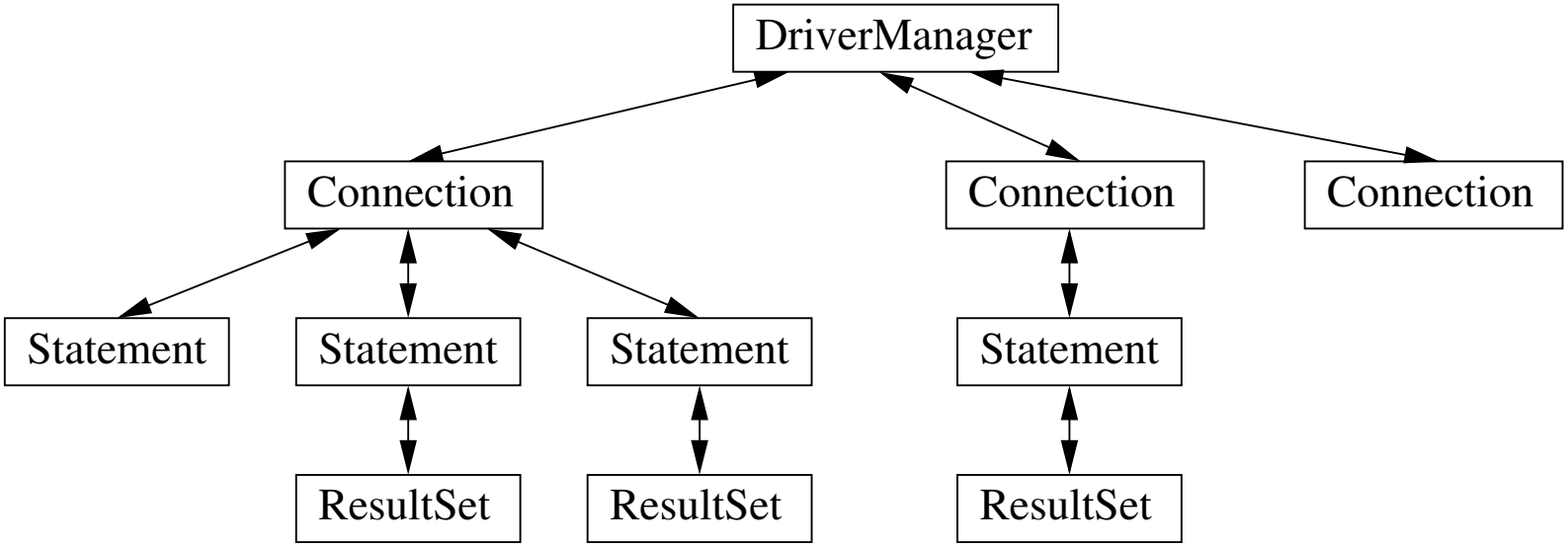
Die Spezifikation von **url** für die Verbindung zu einer Datenbank eröffnet mit **jdbc:**, gefolgt vom Sub-Protokoll (hier **mysql**). Danach sind der Hostname (= Name des Rechners, auf dem der Datenbankserver läuft) und die Portnummer anzugeben. In diesem konkreten Fall wählen wir **localhost**, da sowohl der Datenbankserver als auch die Java-Applikation **SimpleQuery** auf demselben Rechner laufen sollen, und **3306** als Portnummer (Standard-Port für MySQL). Konkret soll mit der Datenbank **JavaVorl** gearbeitet werden.

- Mit der Anweisung **conn.close()** wird die Verbindung zur Datenbank wieder geschlossen.

Die oben angegebene Java-Applikation kann als Rahmenprogramm verwendet werden, um darauf aufbauend Datenbank-Anwendungen zu erstellen. Die Hauptarbeit besteht dabei in der Auswahl eines geeigneten Treibers sowie der korrekten Konfiguration der Datenbank-Verbindung . . .

14.4 Statements, Queries und ResultSets

Mit dem **DriverManager** können mehrere **Connection**s zu Datenbanken betrieben werden. Innerhalb einer **Connection** können mehrere **Statement**s zur (sukzessiven) Ausführung einer oder mehrerer SQL-Anweisungen verwendet werden. Die Ergebnisse solcher SQL-Anweisungen werden als Instanzen der Klasse **ResultSet** zurückgegeben.



Wir ergänzen das Rahmenprogramm nun derart, dass aus der Tabelle **personal** der Datenbank **JavaVorl** die Spalten **Name**, **StOrt** und **Gehalt** ausgegeben werden:

JavaSQL/SimpleQuery/3-Query/SimpleQuery.java

```
48          System.out.println("* Statement beginnen");
49          Statement stmt = conn.createStatement();
50
51          System.out.println("* Abfrage beginnen");
52          String sqlCommand =
53            "SELECT Name, StOrt, Gehalt FROM personal";
54          // "SELECT Name, StOrt, Gehalt FROM personal WHERE Gehalt > 50000";
55          ResultSet rs = stmt.executeQuery(sqlCommand);
56
57          System.out.println("* Ergebnisse anzeigen");
58          while (rs.next())
59            {
60              String name = rs.getString(1);
61              String standort = rs.getString("StOrt");
62              double gehalt = rs.getDouble(3);
63              System.out.println(name + " " + standort + " " + gehalt);
64            }
65
66          System.out.println("* Statement beenden");
67          stmt.close();
```

Quelltextanalyse

- Wir erzeugen eine Instanz der Klasse **`Statement`**.
- Danach spezifizieren wir eine SQL-Anweisung als **`String`**. Dieser dient als Parameter für die Objektmethode **`executeQuery()`**, die lesend auf die Datenbank zugreift und eine Referenz auf ein **`ResultSet`** zurückgibt.

- Ein **ResultSet** ermöglicht den Zugriff auf eine (dynamische) Tabelle, die durch eine **SELECT**-Anweisung generiert wurde. Ein sog. *Cursor* zeigt auf den aktuellen Datensatz (Zeile). Mit der Methode **next()** wird der Cursor auf den folgenden Datensatz verschoben. Der **boolean**-Rückgabewert wird **false**, wenn kein weiterer Datensatz existiert.

- Auf die einzelnen Attribute (Spalten) kann in beliebiger Reihenfolge zugegriffen werden. Dazu verwenden wir die **getXXX()**-Methoden (siehe API-Referenz).

- Mit der Anweisung **stmt.close()** werden sowohl das **Statement** als auch alle erzeugten **ResultSet**s geschlossen.

Beachte: Im Package **java.sql** sind **Connection**, **Statement** und **ResultSet** Interfaces. Deren konkrete Implementierung als Klassen ist im Treiber realisiert.

14.5 Schreibender Zugriff auf die Datenbank

Zum Abschluss demonstriert das folgende Beispiel, wie ein neuer Datensatz in die Datenbank eingefügt werden kann.

JavaSQL/SimpleInsert/SimpleInsert.java

```
48          System.out.println("* Statement beginnen");
49          Statement stmt = conn.createStatement();
50
51          System.out.println("* Einfuegen");
52          String sqlCommand =
53            "INSERT INTO personal " +
54            "VALUES('Heizer', 666, 'Karben', 'Kfz', 'F&E', 2, 34000);";
55          stmt.executeUpdate(sqlCommand);
56
57          System.out.println("* Statement beenden");
58          stmt.close();
```

Quelltextanalyse

- Wir spezifizieren eine entsprechende SQL-Anweisung als String.
- Da wir nun *schreibend* auf die Datenbank zugreifen wollen, verwenden wir nun die Methode **`executeUpdate()`**.
- Danach schließen wir das **`Statement`** und lassen uns mit einem neuen **`Statement`** die aktualisierte Datenbank ausgeben.

Beachte: Starten wir das Programm ein zweites Mal, so wird eine **`SQLException`** geworfen, die anzeigt, dass wir den Schlüssel **`666`** kein zweites Mal einfügen dürfen.

14.6 Zusammenfassende Fragen

- Welche Komponenten benötigen wir, um eine Java-Datenbank-Anwendung betreiben zu können?
- Was bedeuten die Begriffe / Abkürzungen DB, DBMS, JDBC, Tabelle, Relation, DML, DDL und SQL?
- Nennen Sie Synonyme für Zeile und Spalte einer Tabelle.
- Welche Funktion übernimmt das DBMS, der JDBC-Treiber und das JDBC-API?
- Welche Schritte sind notwendig, um in einer Java-Applikation eine Verbindung zu einer Datenbank aufzubauen und wieder zu schließen (Rahmenprogramm)?
- Spezifizieren Sie einen syntaktisch korrekten String für die URL einer Datenbank (siehe API-Referenz).
- Setzen Sie graphisch die Elemente **`DriverManager`**, **`Connection`**, **`Statement`**, **`ResultSet`** zueinander in Beziehung.
- Umschreiben Sie die jeweiligen Aufgaben und Möglichkeiten in eigenen Worten.
- Mit welchen Methoden werden Instanzen der einzelnen Klassen erzeugt?
- Was ist der Unterschied zwischen **`executeQuery()`** und **`executeUpdate()`**? Welche SQL-Anweisungen sind jeweils zulässig?