Einführung in die OOP in Java 6
WS 07/08

Ashraf Abu Baker

Zuletzt aktualisiert am: 09.06.2008 15:32
Ich bedanke mich bei Igor Geier und Gerold Kühne für ihre Unterstützung bei der Erstellung dieses Begleitmaterials.
## Inhaltsverzeichnis

<table>
<thead>
<tr>
<th>Inhalt</th>
<th>Seite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inhaltsverzeichnis</td>
<td>3</td>
</tr>
<tr>
<td>1 Einleitung</td>
<td>5</td>
</tr>
<tr>
<td>2 Historie</td>
<td>6</td>
</tr>
<tr>
<td>3 Entwicklung von Java-Programmen</td>
<td>9</td>
</tr>
<tr>
<td>3.1 Vorbereitung</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Klassendefinition</td>
<td>9</td>
</tr>
<tr>
<td>3.3 Kommentare</td>
<td>10</td>
</tr>
<tr>
<td>3.4 Applikationen vs. Applets</td>
<td>11</td>
</tr>
<tr>
<td>4 Grundlagen</td>
<td>12</td>
</tr>
<tr>
<td>4.1 Java-Bezeichner (identifier)</td>
<td>12</td>
</tr>
<tr>
<td>4.2 Variablen und Datentypen</td>
<td>12</td>
</tr>
<tr>
<td>4.2.1 Deklaration und Initialisierung</td>
<td>15</td>
</tr>
<tr>
<td>4.2.2 Wrapper-Klassen</td>
<td>16</td>
</tr>
<tr>
<td>4.2.3 Arithmetik- und Fließkommprobleme</td>
<td>16</td>
</tr>
<tr>
<td>4.3 Typkonvertierung und Casting</td>
<td>18</td>
</tr>
<tr>
<td>4.4 Autoboxing (boxing/unboxing)</td>
<td>20</td>
</tr>
<tr>
<td>5 Arrays</td>
<td>20</td>
</tr>
<tr>
<td>5.1 Definition</td>
<td>21</td>
</tr>
<tr>
<td>5.2 Irreguläre Arrays</td>
<td>22</td>
</tr>
<tr>
<td>5.3 Zugriff auf Arrays</td>
<td>23</td>
</tr>
<tr>
<td>6 Kontrollstrukturen</td>
<td>24</td>
</tr>
<tr>
<td>6.1 if-else</td>
<td>24</td>
</tr>
<tr>
<td>6.2 switch</td>
<td>25</td>
</tr>
<tr>
<td>6.3 Schleifen</td>
<td>26</td>
</tr>
<tr>
<td>6.3.1 while-Schleife</td>
<td>26</td>
</tr>
<tr>
<td>6.3.2 do ... while-Schleife</td>
<td>28</td>
</tr>
<tr>
<td>6.3.3 for-Schleife</td>
<td>29</td>
</tr>
<tr>
<td>6.3.4 for-each-Schleife</td>
<td>30</td>
</tr>
<tr>
<td>6.4 break und continue</td>
<td>31</td>
</tr>
<tr>
<td>7 Grundkonzepte OOP</td>
<td>34</td>
</tr>
<tr>
<td>7.1 Klassen und Objekte</td>
<td>34</td>
</tr>
<tr>
<td>7.2 Klassen in Java</td>
<td>34</td>
</tr>
<tr>
<td>7.3 Methoden und Konstrukturen</td>
<td>35</td>
</tr>
<tr>
<td>7.4 Überladen von Methoden</td>
<td>36</td>
</tr>
<tr>
<td>7.5 Vererbung</td>
<td>38</td>
</tr>
<tr>
<td>7.6 Überschreibung von Methoden</td>
<td>40</td>
</tr>
<tr>
<td>7.7 Polymorphismus</td>
<td>41</td>
</tr>
<tr>
<td>8 Java-spezifische Konzepte</td>
<td>43</td>
</tr>
<tr>
<td>8.1 Referenzen in Java</td>
<td>43</td>
</tr>
<tr>
<td>8.2 this, super, this() und super()</td>
<td>44</td>
</tr>
<tr>
<td>8.3 Klassenvariablen (statische Variablen)</td>
<td>45</td>
</tr>
<tr>
<td>8.4 Statische Methoden (Klassenmethoden)</td>
<td>47</td>
</tr>
<tr>
<td>8.5 Die main-Methode</td>
<td>47</td>
</tr>
<tr>
<td>8.6 Statische Initialisierer</td>
<td>47</td>
</tr>
<tr>
<td>8.7 Nicht statische Initialisierer</td>
<td>48</td>
</tr>
<tr>
<td>8.8 Initialisierungsreihenfolge</td>
<td>48</td>
</tr>
<tr>
<td>8.9 Call-by-reference oder call-by-value?</td>
<td>50</td>
</tr>
</tbody>
</table>
1 Einleitung

Java ist eine von der Firma Sun Microsystems entwickelte Technologie, die aus einer modernen objektorientierten Programmiersprache und einer Plattform besteht.


Abbildung 1: Java-Plattform


Abbildung 2: Dokumentation zur Java-API

Abbildung 3: Bytecode/JVM

Da Java-Programme von der JVM und nicht direkt von der Hardware ausgeführt werden, sind sie plattformunabhängig.
2 Historie


1997 trennte Sun den Interpreter vom JDK und gab eine neue Version (1.1) des JDK und eine Laufzeitumgebung (Java Runtime Environment (JRE)) heraus. Die JRE enthält alle zum Ausführen von Programmen notwendigen Werkzeuge.

Neu in dieser Version war die Entwicklung der Remote Methode Invocation (RMI), die Java Database Connectivity (JDBC) und das AWT. Die RMI ist eine Schnittstelle, die eine einfache Verteilung von Java-Programmen erlaubt. JDBC ist eine Schnittstelle zum Datenbankzugriff. Das AWT ist eine Bibliothek zur Gestaltung von graphischen Oberflächen.

Da das AWT viele Probleme mit sich brachte, entwickelte Sun 1998 eine neue API zur Erstellung von graphischen Oberflächen namens Swing und gab damit die Version 1.2 heraus.

http://java.sun.com/developer/technicalArticles/RoadMaps/Features/

Im Jahre 2000 brachte Sun die Version 1.3 auf den Markt, die u.a. das Servlet Development Kit und die HotSpot Virtual Machine enthielt. Servlets sind Java-Programme, die in einem Web-Server laufen und Web-Seiten dynamisch generieren.

http://java.sun.com/j2se/1.3/docs/relnotes/features.html


http://java.sun.com/j2se/1.4.2/docs/relnotes/features.html

Im JDK 1.5 vom Jahre 2004 wurde die Syntax der Programmiersprache erweitert und neue wichtige Konzepte wie Generics, Boxing/Unboxing sowie Aufzählungstypen und Annotationen eingeführt.

http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html


<table>
<thead>
<tr>
<th>Jahr</th>
<th>Version</th>
<th>Pakete</th>
<th>Klassen</th>
<th>Neue Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996</td>
<td>1.0</td>
<td>7</td>
<td>210</td>
<td>Applets, AWT, I/O, Net</td>
</tr>
<tr>
<td>1997</td>
<td>1.1</td>
<td>22</td>
<td>ca. 600</td>
<td>RMI, JDBC,…</td>
</tr>
<tr>
<td>1998</td>
<td>1.2</td>
<td>59</td>
<td>1524</td>
<td>Swing, IDL, Collections,…</td>
</tr>
<tr>
<td>2000</td>
<td>1.3</td>
<td>76</td>
<td>1840</td>
<td>CORBA ORB, Sound, JNDI, Servlets…</td>
</tr>
<tr>
<td>2002</td>
<td>1.4</td>
<td>135</td>
<td>2723</td>
<td>XML Processing, JNI,…</td>
</tr>
<tr>
<td>2004</td>
<td>1.5</td>
<td>165</td>
<td>3279</td>
<td>Generics, Enhanced for Loop, Autoboxing, Enums, Annotations…</td>
</tr>
<tr>
<td>2006</td>
<td>1.6</td>
<td>202</td>
<td>3777</td>
<td>Scripting, Compiler Access, Desktop Deployment…</td>
</tr>
</tbody>
</table>

Tabelle 1: JDK-Entwicklung
3 Entwicklung von Java-Programmen

3.1 Vorbereitung


Das JDK 6.0 kann von der Sun-Seite unter der folgenden Adresse heruntergeladen werden:


3.2 Klassendefinition

Eine Java-Quellcodedatei besteht aus höchstens einer **public**-Klassendefinition, genannt *public top level class definition*, und einer unbeschränkten Anzahl von nicht **public**-Klassendefinitionen (*non public top level class definitions*), die in einer Java-Quelldatei gespeichert werden. Java-Quelldateien sind Unicode-Dateien, die mit der Erweiterung „.java“ gespeichert werden. Enthält ein Java-Programm eine *public top level*-Klasse, die beispielsweise *MyJavaProgram* heißt, so muss seine Quelldatei *MyJavaProgram.java* heißen:

```java
/**
 * A Java program containing 1 public and 2 non public class
definitions.
*/
public class MyJavaProgram { // public class definition
    public static void main(String [] input) {
        System.out.println("Hi, this is my first Java program!");
    }
}
class MyFirstClass { // non public class definition
}
class MySecondClass { // non public class definition
}
```
Ein Programm kann drei sog. top-level-Kompiliereinheiten enthalten:

1. Paket-Deklaration (*package declaration*)
2. Import-Anweisungen (*import statements*)
3. Klassen- oder Interface-Definitionen


### 3.3 Kommentare

Zur Kommentierung von Java-Programmen unterstützt Java drei Arten von Kommentaren:

1. **Zeilenkommentare:**
   Ein Zeilenkommentar ist genau eine Zeile lang:

   ```java
   // import list
   ```
2. **Blockkommentare:**

Blockkommentare können sich über mehrere Zeilen strecken:

```java
/*
 * A top-level class definition
 */
```

3. **Dokumentationskommentare:**

Das `javadoc`-Tool benutzt Dokumentationskommentare, um eine Dokumentation für Klassen und Interfaces automatisch zu generieren. Sie beginnen mit `/**` und enden mit `*/`

```java
/**
 * This is a class which prints a string on the console.
 */
```

### 3.4 Applikationen vs. Applets

Es gibt zwei Arten von Java-Programmen: Applikationen und Applets.

Eine Applikation ist ein Java-Programm, das als eigenständige Anwendung (*stand-alone application*) von einer auf einem Rechner installierten JVM ausgeführt werden kann.

Applets sind Java-Programme, die in HTML-Seiten eingebettet werden und von einem Java-fähigen Web-Browser als Web-Anwendungen gestartet werden.


Es ist die Aufgabe des Programmierers festzulegen, ob sein Programm als Applikation, als Applet oder auch als beides, funktionieren soll.

Jedes Programm, das als eigenständige Applikation ausgeführt werden soll, muss eine sog. `main`-Methode enthalten, die als Startpunkt der Applikation dient:

```java
//Signatur der main-Methode
public static void main(String [] input) {}
//bzw. (seit JDK 5)
public static void main(String ... input) {}
```

Wenn eine Applikation gestartet werden soll, wird ihre Klasse in die JVM geladen. Die JVM durchsucht dann die Klasse nach der `main`-Methode. Wird die `main`-Methode gefunden, übergibt ihr die JVM ein Parameter-Array und fängt anschließend mit der sequenziellen Ausführung der in ihr
enthaltenen Anweisungen an. Das Parameter-Array enthält ggf. die Eingabeparameter, die an das Programm übergeben werden sollen. Enthält die Klasse keine solche Methode, so wird eine Fehlermeldung ausgegeben.
4 Grundlagen

4.1 Java-Bezeichner (identifier)


int minValue; //legal
int $minValue; //legal
int _minValue; //legal
int $=2; //legal
int $=_5; //legal
int _=3; //legal
int _3=3; //legal
int a3; //legal
int 3a; //illegal
int !=maxValue; //illegal
int -maxValue; //illegal
int min-Value; //illegal
int min!Value; //illegal
int _int; //legal
int $boolean; //legal
int newVariable; //legal
int ifConstant; //legal
int If; //legal
int New; //legal
int Private; //legal

4.2 Variablen und Datentypen

Abbildung 4: Datentypen

Variablen des Datentyps boolean können nur einen booleschen Wert (true, false) speichern. Die Datentypen byte, short, int und long (Integraltypen) repräsentieren Ganzzahlen und unterscheiden sich lediglich in ihren Wertebereichen bzw. in der Anzahl der Bits, die verwendet werden um eine Zahl darzustellen.

Eine char-Variable kann ein Zweibyte Unicode-Zeichen bzw. eine positive Ganzzahl aus dem Intervall [0, 2^{16}-1] speichern.

<table>
<thead>
<tr>
<th>Datentyp</th>
<th>Länge (Bytes)</th>
<th>Wertebereich</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
<td>true, false</td>
</tr>
<tr>
<td>char</td>
<td>2</td>
<td>16-Bit-Unicode-Zeichen (0x0000 ... 0xffff)</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
<td>-2^{7} bis 2^{7}-1 (-128 ... 127)</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>-2^{15} bis 2^{15}-1 (-32768 ... 32767)</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>-2^{31} bis 2^{31}-1 (-2147483648 ... 2147483647)</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>-2^{63} bis 2^{63}-1 (-9223372036854775808...9223372036854775807)</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>1,40239846E-45f ... 3,40282347E+38f</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>4,94065645841246544E-324 ... 1,79769131486231570E+308</td>
</tr>
</tbody>
</table>

Tabelle 2: Datentypen

char ist der einzige numerische Datentyp, der nicht vorzeichenbehaftet ist. Mit char-variablen können ganzzahlige Berechnungen durchgeführt werden:
Variablen vom Typ float bzw. double sind Fließkommazahlen dargestellt nach dem IEEE 754-
Standard. Sie unterscheiden sich lediglich in ihrer Bytelänge und somit im Wertebereich bzw. in der
Genauigkeit (Anzahl der Nachkommastellen) der Zahlen, die sie repräsentieren.

Um Float- von Double-Fließkommazahlen zu unterscheiden, müssen Float-Zahlen mit einem „f“ (oder
groß „F“) am Ende der Zahl gekennzeichnet werden. Eine Fließkommazahl mit einem „d“ (oder groß
„D“) als Suffix wird vom Compiler als Double-Zahl interpretiert. Eine Fließkommazahl ohne Suffix
wird ebenfalls als Double-Zahl interpretiert.
Das folgende Beispiel zeigt, dass Double-Zahlen genauer als Float-Zahlen sind:

```
/*
 * Package: default
 * Class:   Samples1.java
 * Method:  primitiveTypes()
 */
float _22_divided_by_7_as_float=22/7.f;
double _22_divided_by_7_as_double =22/7.d;
System.out.println("_22_divided_by_7_as_float: "+_22_divided_by_7_as_float);
System.out.println("_22_divided_by_7_as_double: "+_22_divided_by_7_as_double);
Ausgabe: 22_divided_by_7_as_float: 3.142857
          22_divided_by_7_as_double: 3.142857142857143
```

Um eine Fließkommazahl von einer Ganzzahl unterscheiden zu können, muss mindestens der
Dezimalpunkt, der Exponent oder der Suffix vorhanden sein.

Alle Zahlen der Integraltypen (char, byte, short, int, long) können als Dezimal-, Hexadezimal-
or Oktalzahlen (Zahlen zur Basis 8) dargestellt werden.
Die Zahlen 281dezimal, 0431oktal und 0x119hexadezimal sind drei unterschiedliche Zahlendarstellungen der
Dezimalzahl 281:
/* Package: default
* Class:   Samples1.java
* Method:  primitiveTypes()
*/

```java
int _281_as_decimal=281;
int _281_as_octal=0431;
int _281_as_hexadecimal=0x119;
System.out.println(_281_as_decimal+ " "+_281_as_octal+ " "+_281_as_hexadecimal);
```

**Ausgabe:** 281 281 281

Zahlen, die mit 0 anfangen, werden in Java als Oktalzahlen interpretiert und dürfen nur Ziffern aus dem Intervall [0,7] enthalten. Alle mit 0x beginnenden Zahlen werden als Hexadezimalzahlen interpretiert.

### 4.2.1 Deklaration und Initialisierung

Bei der Deklaration einer Variablen wird sie an einen Datentyp gebunden:

```java
int a; // Variablendeklaration
```

Bei der Initialisierung einer Variablen wird ihr ein Anfangswert zugeordnet:

```java
int a; // Variablendeklaration
a=2; // Variableninitialisierung
int a=4; // Deklaration und Initialisierung in einem Schritt
```

Bevor auf den Wert einer Variablen zugegriffen werden kann, muss sie mit einem Initialwert belegt werden. Nicht lokale Variablen, die nicht vom Programmierer explizit initialisiert werden, werden vom Java-System mit Standardwerten initialisiert. Der Initialwert hängt vom jeweiligen Typ ab:

<table>
<thead>
<tr>
<th>Datentyp</th>
<th>Initialwert</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>char</td>
<td>'\u0015'</td>
</tr>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td>short</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>0</td>
</tr>
<tr>
<td>float</td>
<td>0.0f</td>
</tr>
<tr>
<td>double</td>
<td>0.0</td>
</tr>
</tbody>
</table>

| Tabelle 3: Standard-Initialwerte |
4.2.2 Wrapper-Klassen

Zu jedem primitiven Datentyp in Java gibt es eine korrespondierende Wrapper-Klasse. Diese kapselt eine primitive Variable in einer objektorientierten Hülle und stellt eine Reihe von Methoden zum Zugriff auf die Variable zur Verfügung. Darüber hinaus verfügen die Wrapper-Klassen über zahlreiche Konstanten, die für viele Berechnungen sehr hilfreich sein können:

<table>
<thead>
<tr>
<th>Typ</th>
<th>Wrapper-Klasse</th>
<th>Konstanten</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Boolean</td>
<td>FALSE, TRUE</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
<td>MIN_VALUE, MAX_VALUE, SIZE</td>
</tr>
<tr>
<td>byte</td>
<td>Byte</td>
<td>MIN_VALUE, MAX_VALUE, SIZE, NaN</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
<td>MIN_VALUE, MAX_VALUE, SIZE, NaN</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
<td>MIN_VALUE, MAX_VALUE, SIZE, NaN</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
<td>MIN_VALUE, MAX_VALUE, SIZE, NaN</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
<td>MAX_EXPONENT, MIN_EXPONENT, MIN_VALUE, MAX_VALUE, SIZE, NaN</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
<td>MIN_EXPONENT, MIN_VALUE, MAX_VALUE, SIZE, NaN</td>
</tr>
</tbody>
</table>

Tabelle 4: Wrapper-Klasse

4.2.3 Arithmetik- und Fließkommaprobleme

In Java können zahlreiche arithmetische Operationen ungültige Werte produzieren bzw. Rundungsfehler verursachen. Tabelle 5 und das nachfolgende Programm zeigen einige typische Fehler, die bei arithmetischen Berechnungen auftreten können:

<table>
<thead>
<tr>
<th>Fehlerart</th>
<th>Beispiel</th>
<th>Ergebnis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Überlauf</td>
<td>10^80 * 10</td>
<td>Infinity</td>
</tr>
<tr>
<td>Rundungsfehler</td>
<td>1.0 / 49 ≠ 1.0</td>
<td>true</td>
</tr>
<tr>
<td>Undefinierte Operation</td>
<td>0.0 / 0.0</td>
<td>NaN</td>
</tr>
<tr>
<td>Rundungsfehler</td>
<td>1.0 / 0.0</td>
<td>Infinity</td>
</tr>
</tbody>
</table>

Tabelle 5: Arithmetik- & Fließkommaprobleme
public static void floatingPointProblems()
{
    // Beispiel für Overflow:
    double d = 1e308;
    System.out.println();
    System.out.println(d + "*10==" + d*10);
    System.out.println("---------------------");
    System.out.println();

    // Beispiel für Underflow:
    d = 1e-305;
    System.out.println("underflow: " + d);
    for (int i = 0; i < 4; i++)
    {
        System.out.println(" " + (d /= 100000));
    }
    System.out.println("---------------------");
    System.out.println();

    // Beispiel für NaN:
    System.out.print("0.0/0.0 ist 'Not-a-Number': ");
    d = 0.0/0.0;
    System.out.println(d);
    System.out.println("---------------------");
    System.out.println();

    // Beispiel für Infinity:
    System.out.print("1.0/0.0 ist 'Infinity': ");
    d = 1.0/0.0;
    System.out.println(d);
    System.out.println("---------------------");
    System.out.println();

    // Beispiel für Rundungsfehler:
    System.out.print("Ungenaue Ergebnisse mit double: ");
    for (int i = 0; i < 100; i++) {
        double z = 1.0 / i;
        if (z * i != 1.0)
        {
            System.out.println("(1.0/" + i+")*"+i+"="+(z * i));
        }
    }
    System.out.print("Wurzel_aus_3="+Math.sqrt(3)+"\n                Wurzel aus 3*Wurzel aus 3= "+
                Math.sqrt(3)*Math.sqrt(3));
    System.out.println("---------------------");
    System.out.println(Long.MAX_VALUE*2);
}
4.3 Typkonvertierung und Casting

Wie bereits erwähnt, besitzt jede Variable in Java einen Typ. Ein wichtiger Aspekt von Java ist, dass der Datentyp einer Variablen in einen anderen Datentyp konvertiert werden kann. Java unterscheidet zwischen zwei Arten von Typkonvertierungen:

1. Die implizite Typkonvertierung, die automatisch vom Compiler durchgeführt wird.
2. Die explizite Typumwandlung (casting), die vom Programmierer durchgeführt wird.

```java
/*
 * Package: default
 * Class: Samples1.java
 * Method: typeConversion()
 */

//Beispiele für implizite Typkonvertierungen
System.out.println(710);//implizite Typumwandlung von int in String
System.out.println(3.5);//implizite Typumwandlung von double in String

int k=5/2;//ganzzahlige Division, k hat den Wert 2
System.out.println("5/2=\"+k);
System.out.println("5/2.0=\"+5/2.0+\" Das Ergebnis ist vom Typ double");
System.out.println("5/2.0f=\"+5/2.0f+\" Das Ergebnis ist vom Typ float");

int x='u';//implizite Typumwandlung von char nach int
System.out.println(x);//liefert 117 zurück

Ausgabe:
710
3.5
5/2=2
5/2.0=2.5 Das Ergebnis ist vom Typ double!
5/2.0f=2.5 Das Ergebnis ist vom Typ float!
117
```

![Abbildung 5: Typkonvertierung](image)

Ob eine Typkonvertierung explizit oder automatisch durchgeführt wird, hängt vom Zieltyp ab. Konvertierungen in Pfeilrichtung werden erweiternde Konvertierungen genannt und müssen nicht explizit durchgeführt werden. Jede Typkonvertierung entgegen der Pfeilrichtung beschreibt eine sog. einschränkende Konvertierung, die explizit angegeben werden muss. Man spricht hier von Casting (explizite Typkonvertierung). Im Folgenden sind alle erlaubten Konvertierungen zwischen primitiven Datentypen aufgelistet:

Erweiternde Typkonvertierungen:
- byte zu short, int, long, float oder double
- short zu int, long, float oder double
char zu int, long, float oder double
int zu long, float oder double
long zu float oder double
float zu double

Einschränkende Typkonvertierungen (erfordern Casting):
short zu byte oder char
char zu byte oder short
int zu byte, short oder char
long zu byte, short, char oder int
float zu byte, short, char, int oder long
double zu byte, short, char, int, long oder float

Sowohl bei erweiternden als auch bei einschränkenden Typkonvertierungen können Daten verloren gehen:

```java
/*
 * Package: default
 * Class: Samples1.java
 * Method: typeConversion()
 */

int i=32;
short s=4;
char c='g';
double d=i; //erweiternde Typumwandlung
float ff=(float)d; //einschränkende Typumwandlung
byte b=(byte)c; //einschränkende
c=(char)s; //einschränkende
c=(char)b; //einschränkende

int k=5/2; // Ganzzahlige Division, Ergebnis ist 2.
5/2.0f; // Das Ergebnis ist 2.5f. Da 2.0f eine float-Zahl ist,
        // findet hier eine implizite Typumwandlung in float statt.
5/2.0; // Das Ergebnis ist 2.5d. Da 2.0 eine double-Zahl ist,
        // findet hier eine implizite Typumwandlung in double statt.
5.0f/2.0; // Das Ergebnis ist 2.5d. Da 2.0 eine double-Zahl ist,
        // findet hier eine implizite Typumwandlung in double statt.

/*
 * Package: default
 * Class: Samples1.java
 * Method: typeConversion()
 */

int big = 1234567890;
float approx = big;
System.out.println(big - (int)approx); //Das Ergebnis sollte 0 sein, //
                                            //ist jedoch -46 --> Datenverlust
double d=10e10;
float f=(float)d;
System.out.println((d-f)); //Das Ergebnis sollte 0 sein, ist
                            //jedoch 2048.0 --> Datenverlust
```
4.4 Autoboxing (boxing/unboxing)

Das Konzept des Autoboxing wurde erst in der Version 5 eingeführt und bezeichnet die automatische Konvertierung zwischen primitiven Typen und Wrapper-Klassen. Vor Version 5 waren die primitiven Datentypen und ihre korrespondierenden Wrapper-Klassen nicht zuweisungskompatibel. So war es z.B. nicht möglich eine primitive Integer-Variable einem Integer-Objekt zuzuweisen oder umgekehrt:

```java
//folgende Zuweisungen waren vor Version 5 nicht erlaubt:
Integer x=2;
int y=new Integer(3);
```

Dank des Autoboxing-Konzepts ist dies nun möglich. Unter Boxing versteht man die automatische Umwandlung eines primitiven Wertes in ein Wrapper-Klassenobjekt. Unboxing ist die Umkehrung, sprich die automatische Umwandlung eines Wrapper-Klassenobjektes in einen Wert:

```java
//boxing/unboxing
Boolean b=false; //boxing
float f=new Float(3.4f); //unboxing
```

Da die Umwandlung automatisch vom Compiler durchgeführt wird, sind primitive Datentypen und ihre korrespondierenden Wrapper-Klassen zuweisungskompatibel. Vor Version 5 hätte man die Umwandlung manuell durchführen müssen:

```java
//Vor Version 5
Boolean b=new Boolean(false);
Float f1=new Float(3.4f);
float f=f1.floatValue();
```
5 Arrays

Ein Array ist eine geordnete Folge von Elementen des gleichen Datentyps. Die Felder eines Arrays können Elemente eines primitiven Datentyps oder Referenzen auf andere Arrays bzw. auf Objekte enthalten.

5.1 Definition

Die Definition eines Arrays erfolgt in drei Schritten:
1. Deklaration
2. Konstruktion
3. Initialisierung

Die Deklaration teilt dem Compiler den Namen der Arrayvariablen und den Typ seiner Elemente mit:

```java
/*
 * Package: default
 * Class: Samples2.java
 * Method: arrays()
 */

int [] list1; // Arraydeklaration
int list2 []; // Arraydeklaration
float [][] twoDimArray; // array of arrays of float
float [][] twoDimArray; // array of arrays of float
float [][] twoDimArray2[]; // array of arrays of float
int [] list3, list4 ; // Deklaration mehrerer Arrays
// Deklaration mehrerer Arrays. list9 und list10
// sind zweidimensionale Arrays
int [] list7, list8, list9[], list10[];
```

Bei Deklarationen ist es erlaubt, den Arraynamen vor nach oder zwischen den eckigen Klammerpaaren zu positionieren. Gleiches gilt für Methoden, die Arrays als Parameter übergeben bekommen bzw. zurück liefern:

```java
public static void main(String [] args)
public static void main(String args []) // ist äquivalent zur
// vorigen Deklaration

public float [] getList();
public float getList () [] ; // beide Deklarationen sind
// äquivalent
```

Bei der Konstruktion eines Arrays wird ein Arrayobjekt mit dem new-Operator instanziert und die Länge des Arrays festgelegt:

```java
list1 = new int[10]; // Arraykonstruktion
```
Immer wenn ein Array generiert wird, werden seine Felder automatisch mit den Standardwerten ihres Datentyps initialisiert:

```java
/*
 * Package: default
 * Class:    Samples2.java
 * Method:   arrays()
 */
byte size=3;
float list6 [] = new float[size];
int [] list5 = new int[3];

for (int i=0;i<list5.length;i++)
    System.out.print(list5[i]+" ");
System.out.println();

for (int i=0;i<list6.length;i++)
    System.out.print(list6[i]+" ");
//Ausgabe 0  0  0 ,
```

Will man während der Definition das Array mit anderen Werten initialisieren, so muss die Deklaration, Konstruktion und Initialisierung zu einem einzelnen Schritt kombiniert werden:

```java
/*
 * Package: default
 * Class:    Samples2.java
 * Method:   arrays()
 */

int [] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };
char ac[] = { 'n', 'o', 't', ' ', 'a', ' ', 'S', 't', 'r', 'i', 'n', 'g' };
float [[[] _3X2Matrix={{1.0f,2.1f},{.5f,2.1f},{3.1f,.72f}};
```

5.2 Irreguläre Arrays

Ein zweidimensionales Array ist ein eindimensionales Array, dessen Felder Referenzen auf andere eindimensionale Arrays enthalten. Es ist irreführend sich derartige Arrays als eine Matrix vorzustellen (umgekehrt kann eine Matrix sehr wohl als ein solches Array von Arrays implementiert werden). In einigen anderen Programmiersprachen haben alle Zeilen eines zweidimensionalen Arrays die gleiche Anzahl von Elementen. In Java ist es möglich, irreguläre mehrdimensionale Arrays zu definieren. Das sind Arrays deren Unterarrays unterschiedliche Längen aufweisen:

```java
/*
 * Package: default
 * Class:    Samples2.java
 * Method:   irregularArray ()
 */

int [][] irregularArray = new int[][]{{3,1},{-1},{5,2,0}};
irregularArray[0] // entspricht {3,1} (erste Zeile)
```
irregularArray[1] // entspricht {-1} (zweite Zeile)
irregularArray[1][0] // entspricht -1 (erstes Element in der zweiten Zeile)
irregularArray[2] // entspricht {5,2,0} (dritte Zeile)

Abbildung 6: Irreguläres Array

5.3 Zugriff auf Arrays

Jedes Arrayelement hat einen numerischen Index. Die Nummerierung der Felder eines Arrays fängt mit dem Index 0 an. So sind die Felder eines Arrays der Länge n von 0 bis n-1 nummeriert. Der Versuch, auf ein Element mit einem falschen Index zuzugreifen, wirft eine sog. ArrayIndexOutOfBoundsException (Exceptions werden in Abschnitt 13 Ausnahmen (Exceptions) behandelt):

```java
/*
 * Package: default
 * Class: Samples2.java
 * Method: wrongIndex()
 */
public static void wrongIndex(){
    int[][] intArray=new int[3][3];
    intArray=new int[][]{{3,1,3},{-1,0,1},{5,2}};
    for(int i=0; i<3; i++)
        for (int j=0; j<3; j++)
            System.out.print(intArray[i][j]+ " ");
}
```

Ausgabe: 3 1 3 -1 0 1 5 2 Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 2
    at Samples.wrongIndex(HelloWorldApp.java:218)
    at SamplesApplication.main(HelloWorldApp.java:19)

Da bei jedem Zugriff auf ein Arrayelement von der JVM ein sog. Bound-check durchgeführt wird, sind Zugriffe auf Arrayelemente langsamer als Zugriffe auf Variablen.
6 Kontrollstrukturen

6.1 if-else

Eine if-Anweisung testet eine boolesche Variable oder einen Ausdruck. Wenn die boolesche Variable oder der Ausdruck den Wert true hat, wird die nachstehende Anweisung oder der nachstehende Anweisungsblock ausgeführt. Ansonsten, wenn der Wert false ist, wird die nachstehende Anweisung oder der nachstehende Anweisungsblock ignoriert und mit dem folgenden Block bzw. der folgenden Anweisung fortgefahren. Eine Erweiterung der if-Anweisung ist die if-else-Anweisung, die einen zusätzlichen else-Teil hat. Der else-Teil wird dann ausgeführt, wenn der boolesche Test im if-Teil der Anweisung den Wert false ergibt:

```java
/*
 * Package: default
 * Class:   Samples2.java
 * Method:  roots()
 */
public static void nullStellen(){
    /*
    * Für eine quadratische Funktion f(X)=aX²+bX+c mit a!=0
    * ist Xi eine Nullstelle falls gilt: f(Xi)=0
    * Die Nullstellen einer quadratischen Funktion lassen sich mithilfe
    * der pq-Formel wie folgt leicht ermitteln:
    * */
    //f(X)=2X²+3X+1
    float a=2f, b=3f, c=1;
    float p=b/a;
    float q=c/a;

    float d=(float)Math.pow(p/2, 2)-q;//d=(p/2)²-q;
    if (d<0)
        System.out.println("Die Funktion f(X)="+a+"X^²+"+b+"X+"+c+" hat keine Nullstellen!");
    else if (d==0)
    {
        float x0=-p/2;
        System.out.println("Die Funktion f(X)="+a+"X^²+"+b+"X+"+c+" hat nur eine Nullstelle: X0="+x0);
    }
    else
    {
        float x0=-p/2+(float)Math.sqrt(d);
        float x1=-p/2-(float)Math.sqrt(d);
        System.out.println("Die Funktion f(X)="+a+"X^²+"+b+"X+"+c+" hat zwei Nullstellen: X0="+x0+" und X1="+x1);
    }
}
```

Der Fragezeichenoperator kann an bestimmten Stellen anstelle einer if-else-Anweisung verwendet werden. Er hat die folgende Syntax:

```java
(boolean_condition) ? expression1 : expression2;
```
expression1 und expression2 müssen entweder numerisch sein, einen Referenztyp oder den Typ boolean haben. Bei der Auswertung wird zunächst der Wert von boolean_condition ermittelt. Ist dieser wahr, so wird expression1, sonst expression2 ausgewertet. Das Ergebnis des Ausdrucks a ? b : c ist also b, falls a wahr ist und c, falls a falsch ist. Der Typ des Rückgabewerts entspricht dem Typ des allgemeineren der beiden Ausdrücke b und c.

6.2 switch


```java
/*
 * Package: default
 * Class: Samples2.java
 * Method: switchSample()
 */

public static void switchSample(int command){

    final int OPEN_FILE=0;
    final int DELETE_FILE=1;
    final int SAVE_FILE=2;
    final int RENAME_FILE=3;

    switch(command){
        case OPEN_FILE: System.out.println("opening file...");break;
        case DELETE_FILE: System.out.println("deleting file...");break;
        case SAVE_FILE: System.out.println("saving file...");break;
        case RENAME_FILE: System.out.println("renaming file...");break;
        default: System.out.println("invalid command !!!!!");
    }
}

Samples.switchSample(2);
Ausgabe: saving file...
```

6.3 Schleifen


6.3.1 while-Schleife

Eine while-Schleife hat die folgende Syntax:

```java
while(boolean_condition)
  repeated_statement_or_block
```

boolean_condition ist ein boolescher Ausdruck, der entweder zu einem false oder true ausgewertet wird.

repeated_statement_or_block ist eine Anweisung bzw. ein Block von Anweisungen, die solange wiederholt werden sollen, bis boolean_condition zu einem false ausgewertet wird:

```java
/*
 * Package: default
 * Class: Samples2.java
 * Method: whileSample()
 */

public static void whileSample() {
    Random rand=new Random();
    int randomSum=0;
    int nextRandInt=0;
    while (randomSum<150) {
        nextRandInt=rand.nextInt(10);// liefert eine zufällige Zahl
        // aus dem geschlossenen [0,10] zurück
        System.out.println("nextInt= "+nextRandInt);
        randomSum+=nextRandInt; // äquivalent zu
        // randomSum=randomSum*nextRandInt
    }
    System.out.println(randomSum); // randomSum>=150
}
```
Eine **while**-Schleife, die die Binärdarstellung einer positiven Zahl berechnet:

```java
/*
 * Package: default
 * Class: Samples2.java
 * Method: whileBinary(int number)
 */

public static void whileBinary(int number) {
    String binary="";
    int div=number;
    int rest;
    while(div>0){
        rest=div%2;
        div=div/2;
        binary=rest+binary;
    }
    System.out.println("Die Binärdarstellung von "+number+
                        " ist "+binary);
}
whileBinary(123);

**Ausgabe:** Die Binärdarstellung von 123 ist 1111011
```

Wenn eine **while**-Schleife nur eine einzige Anweisung wiederholen soll, muss diese Anweisung nicht in geschweifte Klammern gesetzt werden. Man spricht in diesem Fall von einer **single statement while loop**.

```java
/*
 * Package: default
 * Class: Samples2.java
 * Method: singleStatementWhile1()
 */

public static void singleStatementWhile1() {
    byte b=0;
    while(++b<3)
    {
        System.out.println("this statement is printed twice");
    }
    System.out.println("this statement is printed only once");
}

**Ausgabe:** this statement is printed twice
               this statement is printed twice
               this statement is printed only once
```
Beispiele für endlose Schleifen:

```java
public static void endlessWhileLoop() {
    byte b=0;
    while (b<3);
    b++;//this statement will never be executed
}
```

```java
byte y=0;
while (true)
    System.out.println(++y);//overflow after 128 iterations
byte x=1;
byte y=10;
while (Math.min(x,y)<5);// endless loop:
    ++x;
```

6.3.2 do … while-Schleife

Die do-Schleife ist die fußgesteuerte Variante der while-Schleife; d.h. die Schleifenbedingung wird nach der Ausführung des Schleifenkörpers geprüft, wodurch der Schleifenkörper immer mindestens ein mal ausgeführt wird. Die allgemeine Form einer do-Schleife ist

do
    repeated_statement_or_block
while (boolean_condition);

Beispiel:

```java
public static void doWhileSumOfDigits(int number) {
    //Berechnung der Quersumme einer positiven Zahl:
    int sumOfDigits = 0;
    do {
        sumOfDigits += number % 10;// sumOfDigits=sumOfDigits+number%10
    } while (number != 0);
    System.out.println(sumOfDigits);
}
doWhileSumOfDigits(283746);
```

Ausgabe: 30
6.3.3 for-Schleife

Die allgemeine Form einer for-Schleife sieht folgendermaßen aus:

\[
\text{for(} \text{statement; boolean_condition; expression)} \\
\text{repeated_statement_or_block}
\]

Bei einer for-Schleife wird zuerst statement ein einziges Mal ausgeführt. Danach wird boolean_condition ausgewertet. Wenn die Auswertung der boolean_condition wahr ergibt, wird der Anweisungsblock repeated_statement_or_block und anschließend der Ausdruck expression jeweils einmal ausgeführt. Der darauf folgende Schleifendurchlauf beginnt mit einer erneuten Auswertung der Schleifenbedingung boolean_condition.

Beispiel (Fibonacci-Zahlen):

```java
/*
* Package: default
* Class: Samples2.java
* Method: fibonacci(int n)
*/
public static void fibonacci(int n) {
    int fib1 = 0, fib2 = 1;
    int fib = fib1 + fib2;
    for (int i = 3; i <= n; i++) {
        fib1 = fib2;
        fib2 = fib;
        fib = fib1 + fib2;
    }
    System.out.println(fib);
}
fibonacci(11);
```

Ausgabe: 89

Weitere Beispiele:

```java
/*
* Package: default
* Class: Samples2.java
* Method: strangeForLoop()
*/
public static void strangeForLoop() {
    // strange loop
    for (System.out.println("Ich werde ein einziges Mal zu Beginn ausgeführt!");
        i<5;System.out.println("Ich werde am Ende eines Schleifendurchlaufs ausgeführt! "))
    { 
        System.out.println("Ich werde zum "+(i++)+". Mal ausgeführt! ");
    }
}
```
// Die beiden Schleifen sind äquivalent

for(System.out.println("Ich werde ein einziges Mal zu Beginn ausgeführt!");
i<5;)
{
    System.out.println("Ich werde zum "+(i++)+". Mal ausgeführt!");
    System.out.println("Ich werde am Ende eines Schleifendurchlaufs ausgeführt!");
}

// Äquivalent zu:

for(int i=0,j=10; i<5; i++)
    System.out.println(i*j--);

int i=0;
for(;i<5;)
    System.out.println(i++);

// Äquivalent zu:

for(;true;)
    System.out.println("endless loop");

for(int j=0;;j++)
    System.out.println("endless loop");

6.3.4 for-Each-Schleife

Die for-each-Schleife gibt es erst seit Version 5. Sie bietet eine komfortable Möglichkeit über Arrays und Kollektionen (Collections) zu iterieren:

public static void forEachLoop() {
    double [] trigonometry=
    new double[]{Math.PI,Math.sin(Math.PI/2),Math.cos(Math.PI/2)};
    for(double d : trigonometry) // for each element d in
        System.out.println(d); // the trigonometry array
}

// Äquivalent zu:

for(int i=0;i<trigonometry.length;i++)
    System.out.println(trigonometry[i]);
6.4 break und continue

In vielen Fällen ist das Beenden eines Schleifendurchlaufs bzw. das Verlassen des Schleifenrumpfes erforderlich. Hierzu stellt Java zwei Anweisungen zur Verfügung: continue und break.

Die Anweisung break bewirkt, dass mit ihrer Ausführung die Abarbeitung der Schleife sofort beendet wird. Dies geschieht unabhängig davon, ob das Abbruchkriterium der Schleife erfüllt ist oder nicht.

Beispiel:

```
/*
 * Package: default
 * Class: Samples2.java
 * Method: breakSample ()
 */

public static void breakSample()
{
    /* Erstelle eine zufällig generierte
     * Liste aus Ganzzahlen
     */
    int [] numbers = getRandomIntegerList();
    /* Ermittle die Summe aller Zahlen, die vor der ersten
     * negativen Zahl in der Liste vorkommen.
     * Implementierung mit einer for-each-Schleife
     */
    int sum=0;
    for (int number : numbers) {
        if (number<0)
            break;
        sum+=number;
    }
    System.out.println("Die Summe beträgt: "+sum);
}

public static int [] getRandomIntegerList(int arraySize){
    //Erzeuge eine zufällig generierte Liste aus Ganzzahlen
    Random rand=new Random();
    int [] numbers=new int[arraySize];
    for (int i=0;i<numbers.length;i++) {
        //sign ist entweder 1 oder -1
        byte sign=(byte)Math.pow(-1,rand.nextInt(3));
        numbers[i]=rand.nextInt(20)*sign;
        //System.out.print(numbers[i]+" ");
    }
    System.out.println();
    return numbers;
}
```

Durch die Anweisung continue wird im Gegensatz zu break nicht die gesamte Schleife abgebrochen, sondern nur der aktuelle Schleifendurchlauf unterbrochen. Die continue-Anweisung ermöglicht es den aktuellen Schleifendurchlauf zu beenden, wobei danach die Schleifenbedingung erneut ausgewertet wird, und es wird ggf. mit einem neuen Schleifendurchlauf begonnen.
Beispiel:
Wir betrachten wieder eine Liste mit negativen und positiven Zahlen. Für jede positive Zahl soll die Wurzel berechnet und auf dem Bildschirm ausgegeben werden. Die in der Liste enthaltenen negativen Zahlen sollen dabei übersprungen werden. Der folgende Code zeigt wie dies mit Hilfe einer for-Schleife und der continue-Anweisung implementiert werden kann:

```java
public static void continueSample(){
    //Erzeuge eine zufällig generierte Liste aus ganzen Zahlen
    int [] numbers=getRandomIntegerList();

    /* Berechne die Wurzel jeder positiven Zahl in der Liste und überspringe dabei die negativen Zahlen
    * Implementierung mit einer for-each-Schleife
    */
    for (int number : numbers) {
        if (number<0) {
            System.out.println(number+" hat keine Wurzel!");
            continue;
        }
        double sqrt=Math.sqrt(number);
        System.out.println("Wurzel von "+number+"= "+sqrt);
    }
}
```

Neben dieser einfachen Form von break und continue gibt es in Java noch die mit einem Label versehene Form:

```java
//eine mit einem Label versehene break-Anweisung
break LabelName;

//eine mit einem Label versehene continue-Anweisung
continue LabelName;
```

Zu jedem mit einem Label versehenen break oder continue muss es eine mit einem Label versehene Kontrollstruktur (for, if, do, while) geben, die die continue- oder break-Anweisung umschließt. Oft wird ein mit einem Label versehenes break verwendet, um zwei oder mehrere ineinander verschachtelte Schleifen zu beenden.

Das folgende Programm zählt, wie viele Zeilen einer Matrix mindestens eine Null enthalten und gibt diese Zahl auf dem Bildschirm aus:
public static void continueWithLabel(){
  int[][] matrix = getRandomMatrix(5);
  int linesWithZero = 0;
  // ein Label für die continue-Anweisung
  labelForContinue: for(int i=0;i<matrix.length;i++){
    for(int j=0;j<matrix[i].length;j++) {
      if(matrix[i][j]==0){
        // erhöhe Anzahl der Zeilen, die mindestens eine Null enthalten
        linesWithZero++;
        System.out.println();
        // restliche Zahlen ignorieren und nächste Zeile nehmen
        continue labelForContinue;
      }
      // gebe Zahl aus und überprüfe die nächste Zahl in derselben Zeile
      System.out.print(matrix[i][j]+"\t");
    }
    System.out.println();
  }
  if(linesWithZero==1)
    System.out.println("Eine Zeile der Matrix enthält Nullen!");
  else System.out.println(linesWithZero+" Zeilen der Matrix enthalten Nullen!");
}

public static int getRandomMatrix(int matrixSize)[][]{
  Random rand = new Random();
  int [][] matrix = new int[matrixSize][matrixSize];

  for(int i=0;i<matrix.length;i++){
    for(int j=0;j<matrix[i].length;j++) {
      matrix[i][j] = rand.nextInt(20);
      System.out.print(matrix[i][j]+"\t");
    }
    System.out.println();
  }
  return matrix;
}
7 Grundkonzepte OOP

7.1 Klassen und Objekte

Eine Klasse ist ein abstrakter Datentyp, der die Eigenschaften, das Verhalten und somit die Struktur einer Menge gleichartiger Objekte (etwa der realen Welt) festlegt. Die Eigenschaften einer Klasse werden Attribute genannt. Das Verhalten der Objekte einer Klasse legen die Methoden der Klasse fest. Ein Objekt ist eine Instanz (konkrete Ausprägung) einer Klasse, das durch seinen Zustand (Belegung seiner Attribute), sein Verhalten und seine Identität (sein Name) gekennzeichnet ist.

Zum Beispiel kann man sich einen abstrakten Datentyp namens Bankkonto vorstellen und festlegen, dass ein Bankkonto einen Betrag speichert und dass der Betrag über die Operationen abheben und einzahlen verändert werden kann:

<table>
<thead>
<tr>
<th>Klasse</th>
<th>Bankkonto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Kontostand, Kontonummer, Inhaber</td>
</tr>
<tr>
<td>Methoden</td>
<td>einzahlen(), abheben()</td>
</tr>
</tbody>
</table>

7.2 Klassen in Java


Der Klassenrumpf deklariert die Mitglieder (members) einer Klasse. Das sind Felder (fields), Methoden (methods), Konstruktoren (constructors), innere Klassen, Interfaces, Instanz- und statische Initialisierer (instance and static initializers).
Die Implementierung des Bankkontos könnte so aussehen:

```java
/*
 * Package: default
 * Class: Bankkonto.java
 * Method:
 * /

public class Bankkonto { //Klassendeklaration
    //Beginn des Klassenrumpfes

        //Attribute legen den Zustand eines Objektes fest
        public int kontoNummer; //Felddeklaration
        public float kontoStand; //in € // Felddeklaration
        public String inhaber; //Felddeklaration

    //Methoden definieren das Verhalten eines Objektes
    //und ändern seinen Zustand
        public void einzahlen(float betrag){
            kontoStand+=betrag;
        }
        public void abheben(float betrag){
            kontoStand-=betrag;
        }
        public float getKontoStand(){
            return kontoStand;
        }
        public static void main(String [] args ){
            Bankkonto alexKonto=new Bankkonto(); //Objekt (Instanz)
            alexKonto.kontoNummer=2150;
            alexKonto.inhaber="Alex Mustermann";
            alexKonto.kontoStand=12350;
            /*
            * Der Zustand von alexKonto ist durch das
            * Tupel(2150,"Alex M.",12350) bestimmt
            */

            Bankkonto michiKonto=new Bankkonto();
            michiKonto.inhaber="Michaela M.";
            michiKonto.kontoStand=2103;
            michiKonto.kontoNummer=35870;
        }

    //Ende des Klassenrumpfes
}
```

Felddeklarationen beschreiben Variablen, die den Zustand eines Objektes festlegen. Methoden legen fest, wie Variablen verändert werden können. Mit anderen Worten sie legen fest wie der Zustand eines Objektes verändert werden kann.
7.3 Methoden und Konstruktoren

Eine Methode hat eine Signatur (Methodenkopf) und einen Methodenrumpf. Der Methodenkopf besteht aus der Zugriffsart (public, private, protected), einem Rückgabetyp, dem Methodennamen und einer Parameterliste.

```java
public void einzahlen(float betrag) //Methodenkopf (Signatur)
{
    //Methodenrumpf
}
```

Eine Methode muss weder Parameter haben noch einen Rückgabewert liefern. Ein Konstruktor ist eine spezielle Methode, die keinen Rückgabetyp hat (auch nicht void) und so heißt wie die zugehörige Klasse. Konstruktoren werden im Wesentlichen dazu verwendet, die Attribute eines Objektes mit Anfangswerten zu initialisieren.

```java
//Konstruktor
public Bankkonto(){
    kontoNummer=0;
    inhaber="Unbekannt";
    kontoStand=0.f;
}
```

Ein Konstruktor wird immer dann aufgerufen, wenn ein Objekt mit dem new-Operator instanziiert wird:

```java
Bankkonto alexKonto=new Bankkonto(); //Objekt (Instanz)
```

Eine Klasse muss keine Konstruktoren definieren. In diesem Fall fügt der Compiler den parameterlosen Default-Konstruktor in die Klasse ein. Dieser heißt genau so wie die Klasse, hat keine Parameter und enthält keine Anweisungen.

```java
//Der Default-Konstruktor
public KlassenName(){
}
```
7.4 Überladen von Methoden


1. Sie haben eine unterschiedliche Anzahl an Parametern:

   aMethod(float dx, float dy) // 2 Parameter
   aMethod() // keine Parameter
   aMethod(float dx) // 1 Parameter

2. Sie haben die selbe Anzahl an Parametern, jedoch eine unterschiedliche Reihenfolge der Parametertypen:

   aMethod(float dx, int dy)
   aMethod(int dx, float dy)
   aMethod(float dx, float dy)
   aMethod(int dx, int dy)

Die Bezeichnung der Parameter spielt in beiden Fällen keine Rolle. Demnach sind die folgenden Parameterlisten absolut identisch:

   aMethod(float deltaX, int deltaY)
   aMethod(float dx, int dy)

Alle folgenden Methoden stellen Überladungen der Methode translate dar:

   public void translate(float dx, float dy)
   public void translate(int dx, int dy)
   public void translate(float dx, int dy)
   public void translate(int dx, float dy)
   public void translate(int dxy)
   public void translate(float dxy)
   public void translate() 

Keine Überladung:

   public void translate(float distance)
   public void translate(float abstand)

Keine Überladung:
Bemerkung: Konstruktoren können nicht vererbt, jedoch wie Methoden überladen werden. Die Definition mehrerer Konstruktoren kann also als Überladung von Konstruktoren verstanden werden.
### 7.5 Vererbung

Ein wesentliches Konzept in der OOP ist die Vererbung. Vererbung ist ein Ausdruck der Ähnlichkeit zwischen Klassen und bietet die Möglichkeit, die nicht privaten Attribute und Methoden einer existierenden Klasse \( A \) auf eine neue Klasse \( B \) zu übertragen. Man sagt, dass \( B \) die Attribute und Methoden der Klasse \( A \) erbt. \( B \) wird als Unterklasse und \( A \) als Oberklasse bezeichnet. Klassen können also in einer hierarchischen Vererbungsbeziehung stehen. Dieses Konzept unterstützt die Wiederverwendung von bereits existierendem Code. Vererbung macht die in der Vaterklasse definierten Daten für die Unterklasse verfügbar.

Im obigen Beispiel könnte \texttt{Girokonto} eine Unterklasse von \texttt{Bankkonto} sein. Die Klasse \texttt{Girokonto} erbt dann alle Attribute und Methoden von \texttt{Bankkonto} und kann beliebige eigene Attribute und Methoden hinzufügen (z.B. \texttt{dispositionsLimit, dauerAuftragEinrichten()}, etc.)

```java
/*
 * Package: default
 * Class: Bankkonto.java
 * Method:
 */

class Girokonto extends Bankkonto{
    float dispositionsLimit;

    public void dauerAuftragEinrichten(){
        /*
         * Code für die Einrichtung eines Dauerauftrags
         */
    }
}
```
public class Point {
    public float x1;
    public float y1;
    public Point(float x1, float y1){
        this.x1=x1;
        this.y1=y1;
    }
}

class Line extends Point{//Line ist eine Unterklasse von Point,
    float x2;       //die ihrerseits die Oberklasse von Line ist
    float y2;
    public Line(float x1, float y1,float x2, float y2){
        super(x1,y1);//Aufruf des Konstruktors der Oberklasse
        this.x2=x2;
        this.y2=y2;
    }
}

Es ist an dieser Stelle zu betonen, dass nur nicht private Attribute und Methoden geerbt werden; Konstruktoren werden nicht geerbt. Dennoch kann in der Unterklasse jeder Konstruktor der Oberklasse mit super(parameter_list) und den entsprechenden Konstruktor-Parametern aufgerufen werden. Im obigen Beispiel ruft der Konstruktor der Klasse Line den Konstruktor von Point auf.
Hier spricht man von einem expliziten Aufruf des Oberklassenkonstruktors. Dieser Aufruf muss als erste Anweisung innerhalb des eigenen Konstruktors erfolgen.
Findet der Compiler keinen expliziten Aufruf des Oberklassenkonstruktors, so fügt er einen impliziten Aufruf des default-Konstruktors der Oberklasse als erste Anweisung ein.

//Die beiden folgenden Konstruktoren sind äquivalent:
public Line(float x2, float y2){
    this.x2=x2;
    this.y2=y2;
}
public Line(float x2, float y2){
    super();//Falls nicht vorhanden, wird vom
    //Compiler automatisch eingefügt
    this.x2=x2;
    this.y2=y2;
}

Eine Klasse darf nur eine einzige Klasse als direkte Oberklasse haben. Mehrfachvererbung ist in Java nicht erlaubt (im Gegensatz zu z.B. C++).
7.6 Überschreibung von Methoden


```java
/* * Überschreibung der Methode translate(float dx, float dy) */
@Override
public void translate(float dx, float dy){
    super.translate(dx, dy); //Wiederverwendung des Codes der Oberklasse
    x2+=dx;
y2+=dy;
}
/*
* Überschreibung der Methode translate(int dx, int dy)
*/
@Override
public void translate(int dx, int dy){
    super.translate(dx, dy); //Wiederverwendung des Codes der Oberklasse
    x2+=dx;
y2+=dy;
}
/*
* Überschreibung der Methode translate(float distance)
*/
@Override
public void translate(float distance){
    x1+=distance; //Überschreiben ohne Wiederverwendung des Codes
    y1+=distance;
x2+=distance;
y2+=distance;
}
```

Unter Polymorphismus versteht man die Fähigkeit einer Objektreferenz, Objekte seiner Unterklasse zu referenzieren und sich abhängig davon zu verhalten. In Java ist es nämlich erlaubt, einer Oberklassenobjektreferenz ein Unterklassenobjekt \( X \) zuzuweisen. Somit stellt sich die folgende Frage: Verhält sich \( X \) in diesem Fall wie ein Objekt der Ober- oder der Unterklasse? Wird einer Oberklassenobjektreferenz ein Objekt \( X \) einer Unterklasse \( U \) zugewiesen, so verhält sich \( X \) dementsprechend wie alle anderen Objekte von \( U \).

Das folgende Beispiel soll dies verdeutlichen:

```java
/*
* Package: default
* Class: Polymorphism.java
* Method:
*/
class Polygon {  //Oberklasse
    /*
    * Gibt den Flächeninhalt des Polygons zurück.
    */
    public void getSurface(){
        System.out.println("The surface of this polygon is undefined.");
    }
}
class Rectangle extends Polygon{//Unterklasse
    /**
     * Ein Rechteck
     */
    public float width;
    public float height;
    public Rectangle(float width, float height){
        this.width=width;
        this.height=height;
    }
    public void getSurface(){
        System.out.println("The surface of this rectangle is: "+this.width*height);
    }
}
class Triangle extends Polygon{
    float base;
    float height;
    public Triangle(float base, float height){
        this.base=base;
        this.height=height;
    }
    public void getSurface(){
        System.out.println("The surface of this triangle is: "+.5f*base*height);
    }
}
//polygon ist eine Oberklassenobjektreferenz
Polygon polygon=new Polygon();
polygon.getSurface();
//der Oberklassenobjektreferenz wird ein Unterklassenobjekt zugewiesen
```
polygon = new Rectangle(2,3);
// Anhand der Ausgabe verhält sich polygon
// wie ein Objekt der Unterkasse
polygon.getSurface();
polygon = new Triangle(3,2);
polygon.getSurface();

*Auszuge:*
The surface of this polygon is undefined.
The surface of this rectangle is: 6.0
The surface of this triangle is: 3.0

Umgekehrt ist es in Java und in vielen anderen Programmiersprachen nicht erlaubt einer Referenz einer Unterkasse ein Objekt einer Oberklasse zuzuweisen. Folgende Codezeilen lassen sich nicht kompilieren:

```java
Triangle triangle = new Polygon(); // Kompilierfehler
Rectangle rectangle = new Polygon(); // Kompilierfehler
Rectangle rectangle = new Triangle(); // Kompilierfehler
```
8 Java-spezifische Konzepte

8.1 Referenzen in Java

Wie bereits erwähnt, sind Typen in Java in zwei Kategorien unterteilt: Primitive Datentypen und Referentotypen, zu denen Arrays, Klassen und Interfaces zählen. Es gibt einen wesentlichen Unterschied zwischen einer primitiven Variablen (Variable eines primitiven Typs) und einer Referenzvariablen (Variable eines Referentyps):

```java
public class Student {
    public String name; // Referenzvariable (Objektreferenz)
    public Date gebDatum; // Referenzvariable (Objektreferenz)
    public String studiengang; // Referenzvariable (Objektreferenz)
    public int semesteranzahl; // primitive Variable
}

Student student = new Student("Alex", null, "BWL", 3);
```

Abbildung 7: Referenzen

Während eine primitive Variable wie z.B. semesteranzahl ihren tatsächlichen Wert (3) speichert, speichert eine Referenzvariable wie name oder gebDatum eine Referenz auf ein Objekt. Wenn ein Objekt mittels eines Konstruktorsaufrufs erzeugt wird (student = new Student("Alex", null, "BWL", 3)), liefert der new-Operator eine Referenz auf das erzeugte Objekt zurück. Die Referenz wird dann in der Variablen student gespeichert.

8.2 this, super, this() und super()

Neben den eigenen und geerbten Attributen einer Klasse, verfügen die Instanzmethoden und die Konstruktoren einer Klasse über zwei weitere Referenzen this und super. this ist eine Referenz auf das Objekt selbst. super ist eine Referenz auf das Objekt der Oberklasse. Mit this kann eine nicht statische Methode oder ein Konstruktor auf jede Methode oder Variable der Klasse zugreifen. Insbesondere wird die this-Referenz zur Unterscheidung zwischen einer Objektvariablen und einer übergebenen Variablen innerhalb einer Methode oder eines Konstruktors verwendet, wie das folgende Beispiel zeigt:

```java
public Line(float x2, float y2){
    this.x2=x2;
    this.y2=y2;
}
```

Während x2 hier der übergebene Parameter ist, ist this.x2 die Objektvariable, die ebenfalls x2 heißt. Analog kann innerhalb einer Klasse mit super auf die Attribute und Methoden der Oberklasse zugegriﬀen werden.

Wie bereits erwähnt, fügt der Compiler automatisch einen Aufruf des Default-Konstruktors der Oberklasse super() ein, wenn in einem Konstruktor kein Konstruktor der Oberklasse explizit aufgerufen wird. Konstruktoren der eigenen Klasse bzw. der Oberklasse können mit this(parameter_list) bzw. super(parameter_list) aufgerufen werden. Ein solcher Aufruf muss als erste Anweisung im Konstruktor erfolgen:
/*
 * Package: default
 * Class: Point.java
 * Method:
 */

public Line(float x1, float y1, float x2, float y2){
    this(x2,y2); //Aufruf des lokalen Konstruktors
    //Line(float x2, float y2)
    this.x1=x1;
    this.y2=y2;
}

public Line(float x1, float y1, float x2, float y2){
    super(x1,y1); //Aufruf des Konstruktors der Oberklasse
    this.x2=x2;
    this.y2=y2;
}

public Line(float x1, float y1, float x2, float y2){
    this.x1=x1;
    this.y2=y2;
    this(x2,y2); //Fehlermeldung. Konstruktorauftrag steht
    //nicht als erste Anweisung
}

8.3 Klassenvariablen (statische Variablen)

Java unterscheidet zwischen zwei Arten von Variablen: Objekt- und Klassenvariablen.


Während auf eine Klassenvariable sowohl über den Klassennamen mit Mitarbeiter.chef (siehe Beispiel unten) oder auch über eine Instanzreferenz (mitarbeiter1.chef) zugegriffen werden kann, können Instanzvariablen nur über eine Instanzreferenz gelesen oder manipuliert werden. Das folgende Beispiel soll die Unterschiede verdeutlichen:
/*
 * Package: default
 * Class: Mitarbeiter.java
 * Method:
 */

public class Mitarbeiter {
    public static String chef="Oliver P."; //Klassenvariable
    public String name; //Instanzvariable
    public String abteilung; //Instanzvariable

    public static void chefwechsel(String neuerChef){ //statische Methode
        chef=neuerChef;
    }
    public void abteilungwechsel(String neueAbteilung){
        abteilung=neueAbteilung;
    }
    public Mitarbeiter(String name, String abteilung){
        this.name=name;
        this.abteilung=abteilung;
    }
    public void print(){
        System.out.println();
    }

    public static void main(String [] str){
        Mitarbeiter mitarbeiter1=new Mitarbeiter("Nadia O.", "Marketing");
        Mitarbeiter mitarbeiter2=new Mitarbeiter("Klaus M.", "Controlling");
        System.out.println(mitarbeiter1.abteilung);
        System.out.println(mitarbeiter2.abteilung);
        System.out.println(mitarbeiter1.chef);
        System.out.println(mitarbeiter2.chef);
        mitarbeiter1.abteilungwechsel("Finanz");
        mitarbeiter1.chefwechsel("Paul J.");
        System.out.println(mitarbeiter1.abteilung);
        System.out.println(mitarbeiter2.abteilung);
        System.out.println(mitarbeiter1.chef);
        System.out.println(mitarbeiter2.chef);
    }
}

Ausgabe:
Marketing
Controlling
Oliver P.
Oliver P.
Finanz
Controlling
Paul J.
Paul J.

Weitere Beispiele für Klassenvariablen: Math.PI, Integer.MAX_VALUE, Float.MIN_EXPONENT.
8.4 Statische Methoden (Klassenmethoden)

Eine statische Methode, auch Klassenmethode genannt, wird mit dem Schlüsselwort `static` deklariert. Sie ist, wie eine statische Variable an die Existenz eines Objektes nicht gebunden und kann über den Klassennamen aufgerufen werden:

```java
Math.sin(a); Math.cos(a); Math.sqrt(a); Float.parseFloat(s);
```

Statische Methoden dürfen innerhalb einer Klasse nur auf KlassenvARIABLEN oder auf andere Klassenmethoden zugreifen. Sie verfügen nicht über die `this`-Referenz und dürfen sie somit nicht verwenden.

8.5 Die main-Methode

Die am meisten verwendete und bekannteste statische Methode ist die `main`-Methode:

```java
public static void main(String[] str)
//bzw.
public static void main(String ... str)//Signatur der main-
Methode seit Version 5
```


8.6 Statische Initialisierer


```java
/*
 * Package: default
 * Class: InitializationDemo.java
 * Method:
 */
public class InitializationDemo {
  public static Date date; //KlassenvARIABLE
  public static int classLuckyNumber; //KlassenvARIABLE
  public static int numberOfCreatedInstances; //KlassenvARIABLE

  static { //Statischer Initialisierer
    date = Calendar.getInstance().getTime();
    Random rand = new Random();
    classLuckyNumber = rand.nextInt(10);
  }
}
```
8.7 Nicht statische Initialisierer

Nicht-statische Initialisierer unterscheiden sich von den statischen in zwei Punkten: Sie ähneln Konstruktoren und werden unmittelbar vor dem Aufruf eines Konstruktors aufgerufen. Sie können auf die this-Referenz und sowohl auf die statischen als auch auf die Instanzvariablen zugreifen. Eine Klasse kann beliebig viele Initialisierer enthalten.

```
/*
 * Package: default
 * Class: InitializationDemo.java
 * Method:
 */

public class InitializationDemo {
    public int instanceNumber; // Instanzvariable
    public int instanceLuckyNumber; // Instanzvariable

    {// nicht-statischer Initialisierer
        Random rand=new Random();
        instanceNumber=rand.nextInt(10);
        System.out.println();
        this.instanceLuckyNumber=rand.nextInt(10);
    }
}
```

8.8 Initialisierungsreihenfolge

class InitializationDemo {

    //nicht-statischer Initialisierer
    System.out.println("Hi, ich bin ein nicht statischer Initialisierer!");
    System.out.println("Ich werde immer aufgerufen, wenn Du versuchst ein Objekt von meiner Klasse zu instanzieren!");
    System.out.println("Du bist gerade dabei einen Konstruktor meiner Klasse zum "+
    (++InitializationDemo.numberOfCreatedInstances)+". aufzurufen!");
    Random rand=new Random();
    instanceNumber=rand.nextInt(10);
    System.out.println();
    this.instanceLuckyNumber=rand.nextInt(10);
}

public int instanceNumber; //Instanzvariable
public int instanceLuckyNumber; //Instanzvariable

public static Date date; //Klassenvariable
public static int classLuckyNumber; //Klassenvariable
public static int numberOfCreatedInstances; //Klassenvariable

static{//Statischer Initialisierer
    System.out.println("Hi, ich bin ein statischer Initialisierer!");
    System.out.println("Die Klasse "+InitializationDemo.class.getName() +" wird gerade in die JVM geladen!");
    System.out.println("Ich werde nur ein einziges Mal aufgerufen!");
    date=Calendar.getInstance().getTime();
    System.out.println("Heute ist "+date.toString());
    Random rand=new Random();
    classLuckyNumber=rand.nextInt(10);
    System.out.println("Deine Glückszahl ist "+classLuckyNumber);
    System.out.println();
}
public static void changeLuckyNumber(int number){
    InitializationDemo.classLuckyNumber=number;
    System.out.println("----------------------------------------
--");
}

public InitializationDemo( ){
    System.out.println("Hallo, ich bin der Konstruktor Deiner Klasse!");
    instanceNumber=numberOfCreatedInstances;
    System.out.println();
}

public void printLuckyNumber(){
    System.out.println("Hi, ich bin Instanz Nummer "+instanceNumber);
    System.out.println("Mein Glückszahl ist "+instanceLuckyNumber);
    System.out.println("Die Glückszahl meiner Klasse ist "+InitializationDemo.classLuckyNumber);
    System.out.println();
}

public static void main(String ... args){
    InitializationDemo initializer1=new InitializationDemo();
    InitializationDemo initializer2=new InitializationDemo();
    InitializationDemo initializer3=new InitializationDemo();
    initializer1.printLuckyNumber();
    initializer2.printLuckyNumber();
    initializer3.printLuckyNumber();
    InitializationDemo.changeLuckyNumber(11);
    initializer1.printLuckyNumber();
    initializer2.printLuckyNumber();
    initializer3.printLuckyNumber();
}


8.9 Call-by-reference oder call-by-value?

Bevor eine primitive Variable an eine Methode übergeben wird, erstellt die JVM eine Kopie der Variablen, die sie dann an die Methode übergibt. Dies hat zur Konsequenz, dass eine Veränderung der Kopie innerhalb der Methode keine Wirkung auf den Variablenwert hat. Bei Referenzvariablen wird ebenfalls eine Kopie der Referenz an die Methode übergeben, nicht das Objekt selbst. Jede Änderung des Objektes innerhalb der Methode ist nach außen sichtbar wie das unten stehende Beispiel zeigt. Somit werden Parameter in Java per call-by-value an Methoden übergeben. Das folgende Beispiel soll dies verdeutlichen:
```java
/*
* Package: default
* Class: CallByValue.java
* Method: incrementValues
*/

public static void incrementValues(int primitiveVar, MyInteger refVar){
    primitiveVar=primitiveVar+1;
    refVar.value=refVar.value+1;
}

public static void main(String ... arg){
    int primitiveVar=5;
    MyInteger refVar=new MyInteger(5);
    System.out.println("Vor der Inkrementierung:
        primitiveVar="+primitiveVar+
        refVar.value="+refVar.value);
    incrementValues(primitiveVar,refVar);
    System.out.println("Nach der Inkrementierung:
        primitiveVar="+primitiveVar+
        refVar.value="+refVar.value);
}
}

class MyInteger{
    public int value;
    public MyInteger(int value){
        this.value=value;
    }
}

Ausgabe:
Vor der Inkrementierung: primitiveVar=5 refVar.value=5
Nach der Inkrementierung: primitiveVar=5 refVar.value=6
```

### 8.10 Pakete (packages)

Unter einem Paket (package) in Java versteht man eine Sammlung von logisch zusammengehörenden Klassen, Interfaces und andere Typen. Pakete können als eine logische Gruppierung von Klassen aufgefasst werden. In anderen Programmiersprachen werden sie als Programmierbibliotheken bezeichnet. Ein Paketname besteht aus mehreren durch Punkte getrennten Bezeichnern:

```java
//package declaration
package de.frankfurt.uni.cs.gdv.visian3d.datastructures.trees;
```

Der voll qualifizierte Name einer Klasse $A$ innerhalb des Paketes
de.frankfurt.uni.cs.gdv.datastructures.trees ist
de.frankfurt.uni.cs.gdv.datastructures.trees.A

Das Paket de.frankfurt.uni.cs.gdv.datastructures ist ein Unterpaket vom Paket

Klassen, die nicht explizit einem Paket zugeordnet sind, werden im sog. default-package gespeichert. Dies ist der Ordner, in dem die Klasse liegt.

Die Java-API 6.0 enthält etwa 202 Pakete, die 6 377 Klassen und Interfaces umfassen.

8.11 Import

Will eine Klasse auf Bestandteile (z.B. Klassen oder Interfaces) eines Paketes zugreifen, so müssen diese Bestandteile durch eine import-Anweisung importiert werden, ehe sie benutzt werden können. Eine Klasse zu importieren bedeutet, sie samt ihren Attributen und Methoden verfügbar zu machen. Es bedeutet keinesfalls, dass sie in die JVM geladen wird. Ihr Name wird lediglich in den Namensraum der Codedatei eingefügt.

Eine Ausnahme stellen hier die Bestandteile des Pakets java.lang dar. Da dieses Paket eines der wichtigsten Pakete der Java-API ist, werden alle in ihm enthaltenen Klassen vor jedem Compilerlauf automatisch importiert. Das Paket umfasst Klassen, die häufig von Programmen verwendet werden; dazu gehören String, Object, Math, Integer, Boolean, etc.

Die folgende Anweisung importiert die Klasse ArrayList aus dem Paket java.util:

```java
import java.util.ArrayList;
```

Mit der nachstehenden import-Anweisung werden alle im Paket java.util enthaltenen Klassen und Interfaces importiert:

```java
import java.util.*;
```
**8.12 Statischer Import**

Außerhalb einer Klasse können ihre statischen Methoden und Konstanten nur über den Klassennamen angesprochen werden:

```java
long longMaxValue=Long.MAX_VALUE;
```

Seit Version 5 gibt es mit dem statischen Import die Möglichkeit, die statischen Methoden und Konstanten ohne den vorangestellten Klassennamen anzusprechen. Hierzu müssen sie statisch importiert werden. Vergleiche folgende Beispiele:

```java
/*
 * Package: default
 * Class: StaticImport.java
 * Method:
 */

//non static import of java.lang.Math
import java.lang.Math.*; //eigentlich überflüssig

class WithoutStaticImport{
    public static void main(String... args) {
        System.out.println("Wurzel aus " +Math.PI+" ist" +Math.sqrt(Math.PI));
    }
}

/ *
 * Package: default
 * Class: StaticImport.java
 * Method:
 */

//static import of java.lang.Math
import static java.lang.Math.*;

class StaticImport{
    public static void main(String... args) {
        System.out.println("Wurzel aus "+PI+" ist" +sqrt(PI));
    }
}
8.13 Datenkapselung und Modifier


//modifier
public, private, protected, final, abstract, static, native, transient, synchronized, volatile

Die *Modifier* `public`, `private` und `protected` werden als *access-Modifier* (Zugriffsattribute) bezeichnet. Die *access-Modifier* legen fest, wie auf Klassenfeatures zugegriffen werden darf. Unter Klassenfeature ist im Folgenden eine Variable, eine Methode, ein Konstruktor oder die Klasse selbst gemeint.

8.13.1 public

Auf als `public` deklarierte Features kann in jeder Klasse ohne Einschränkung zugegriffen werden.

8.13.2 private

`private`-Features können nur innerhalb ihrer eigenen Klasse verwendet werden.

8.13.3 default

Wird ein Feature ohne *Modifier* deklariert, so kann auf dieses nur innerhalb von Klassen zugegriffen werden, die im eigenen Paket liegen. Man spricht hier von einem *default*-Zugriff.

8.13.4 protected

package modifier1;

public class ModifierClass {
    public byte publicVar = 1;
    protected byte protectedVar = 3;
    byte defaultVar = 4;
    private byte privateVar = 2;

    public byte getPublicVar() {
        return publicVar;
    }

    private byte getPrivateVar() {
        return privateVar;
    }

    protected byte getProtectedVar() {
        return protectedVar;
    }

    byte getDefaultVar() {
        return defaultVar;
    }
}

package modifier1;

public class ForeignClassInTheSamePackage {
    public static void main(String... arg) {
        ModifierClass modifier = new ModifierClass();
        System.out.println(modifier.publicVar);
        System.out.println(modifier.protectedVar);
        System.out.println(modifier.defaultVar);
        System.out.println(modifier.privateVar);//Kompilierfehler
        System.out.println(modifier.getPublicVar());
        System.out.println(modifier.getProtectedVar());
        System.out.println(modifier.getDefaultVar());
        System.out.println(modifier.getPrivateVar());//Kompilierfehler
    }
}
package modifier2;

import modifier1.ModifierClass;

class ForeignClassInDifferentPackage {
    public static void main(String... arg) {
        ModifierClass modifier = new ModifierClass();
        System.out.println(modifier.publicVar);
        System.out.println(modifier.getPublicVar());
        System.out.println(modifier.protectedVar);  // Kompilierfehler
        System.out.println(modifier.getDefaultVar());  // Kompilierfehler
        System.out.println(modifier.defaultVar);  // Kompilierfehler
        System.out.println(modifier.getProtectedVar());  // Kompilierfehler
        System.out.println(modifier.getDefaultVar());  // Kompilierfehler
    }
}

class SubClassOfModifierClass extends ModifierClass {
    byte protected1=protectedVar;  // erlaubter Zugriff auf
    // ein protected Feature;
    byte protected2=super.getProtectedVar();  // erlaubter Zugriff auf
    // ein protected Feature;

    public static void main(String... arg) {
        ModifierClass modifier = new ModifierClass();
        System.out.println(modifier.publicVar);
        System.out.println(modifier.protectedVar);  // Kompilierfehler
        System.out.println(modifier.getDefaultVar());  // Kompilierfehler
        System.out.println(modifier.defaultVar);  // Kompilierfehler
        System.out.println(modifier.getProtectedVar());  // Kompilierfehler
        System.out.println(modifier.getDefaultVar());  // Kompilierfehler
        System.out.println(modifier.getPrivateVar());  // Kompilierfehler
    }
}
8.13.5 final


```java
public class MyMath extends Math{//Kompilierfehler
}
```

**Kompilierfehler:** The Type MyMath cannot subclass the final class Math

final-Variablen sind die Umsetzung von Konstanten in Java. Es gibt kein anderes explizites Schlüsselwort dafür. Der Wert einer final-Variablen kann nach ihrer Initialisierung nicht mehr verändert werden.

Die Klasse Math verfügt über die Konstante PI. Der Versuch, den Wert von PI zu verändern, führt zu einem Kompilierfehler:

```java
Math.PI=22/7f;
```

**Fehlermeldung:** The final field Math.PI cannot be assigned.

final-Methoden dürfen nicht überschrieben werden:

```java
/*
 * Package: modifier1
 * Class: Final.java
 * Method:
 */
public class Final {
   public int x=2;
   public final void setX(int x){
      this.x=x;
   }
   public final void setX(){
      this.x=x;
   }
}

class FinalApplication extends Final{
   public final void setX(){//Kompilierfehler
   }
   public static void main(String ...strings ){
      final Final x=new Final();
      x.x=2;
   }
}
```

**Kompilierfehler:** Cannot override the final method.
9 Abstrakte Klassen und Methoden

Es kommt häufig vor, dass der Programmierer das Verhalten einer Methode in einer Superklasse nicht definieren kann.


public abstract class GeometricObject {

public static final int CIRCLE = 0;
public static final int RECTANGLE = 1;
public static final int TRIANGLE = 2;
public static final int UNDEFINED = -1;

public int type;

public abstract float getSurface();

    public int getType(){
        return type;
    }
}

class Rectangle extends GeometricObject {

    public float width;
    public float height;

    public Rectangle(float width, float height){
        this.width = width;
        this.height = height;
        this.type = GeometricObject.RECTANGLE;
    }

    public float getSurface(){
        return width * height;
    }
}

class Triangle extends GeometricObject {

    public float base;
    public float height;

    public Triangle(float base, float height){
        this.base = base;
        this.height = height;
        this.type = GeometricObject.TRIANGLE;
    }

    public float getSurface(){
        return .5f * base * height;
    }
}

class Circle extends GeometricObject {

    public float radius;

    public float getSurface(){
        return (float)(Math.PI * radius * radius);
    }
}
10 Interfaces

Ein Interface (Schnittstelle) in Java kann als eine abstrakte Klasse aufgefasst werden, die nur abstrakte Methodendeklarationen und statische Konstanten enthält. Sie werden mit dem Schlüsselwort `interface` deklariert und dürfen keine Konstruktoren definieren.

```java
/*
* Package: interfaces
* Class: GeometricObjectInterface.java
* Method:
*/

public interface GeometricObjectInterface { //Interface declaration
    public static final int CIRCLE=0;
    public static final int RECTANGLE=1;
    public static final int SQUARE=2;

    /**
     * Returns the type of the geometric object (circle, square,
     * rectangle, etc. )
     */
    public int getType();

    /**
     * Returns the circumference of the geometric object
     */
    public float getCircumference();
}
```

Alle in einem Interface enthaltenen Attribute sind automatisch statische Konstanten, die initialisiert werden müssen. Alle Methoden sind automatisch `public` auch wenn wie nicht als `public` deklariert sind. Eine Klasse, die ein Interface implementiert, muss alle Methoden des Interfaces überschreiben:

```java
/*
* Package: interfaces
* Class: Circle.java
* Method:
*/

public class Circle implements GeometricObjectInterface{
    public float radius;

    public Circle(float radius){
        this.radius=radius;
    }

    public int getType(){
        return GeometricObjectInterface.CIRCLE;
    }

    public float getCircumference(){
        return (float)(2*Math.PI*radius);
    }
}
```
Eine Klasse, die nicht alle Methoden eines Interfaces implementiert, muss als abstract deklariert werden.
Interfaces können wie Klassen von einander abgeleitet werden. Wie bereits erwähnt, ist die Mehrfachvererbung in Java nicht erlaubt. Interfaces können jedoch als ein restriktiver Mechanismus für Mehrfachvererbung verwendet werden. So kann eine Klasse mehrere Interfaces implementieren. Sie muss dann für jedes Interface alle darin definierten Methoden implementieren.

### 10.1 Das Interface Comparable

Ein Beispiel zur Verdeutlichung:

Um die Objekte der Klasse `Point` paarweise vergleichbar zu machen, muss `Point` das `Comparable`-Interface implementieren und die einzige in ihr enthaltene Methode `compareTo()` überschreiben. `compareTo()` liefert genau dann einen Wert kleiner 0, wenn das Objekt kleiner, größer 0, wenn es größer und gleich 0, wenn es gleich dem als Argument übergebenen Objekt ist:

```java
public class ComparablePoint extends Point implements Comparable<ComparablePoint> {
    public float x1;
    public float y1;

    /* (non-Javadoc)
     * @see java.lang.Comparable#compareTo(java.lang.Object)
     */
    @Override
    public int compareTo(ComparablePoint p) {
        if (x1<p.x1 || this.y1<p.y1) return -1;
        if (x1==p.x1 && this.y1==p.y1) return 0;
        return 1;
    }
}
```

**Bemerkung:** Alle Wrapper-Klassen und viele andere Klassen wie String, `Date`, File, etc. implementieren das `Comparable`-Interface.

11 Namenskonventionen

Eine Namenskonvention ist eine Empfehlung, wie Pakete, Klassen, Variablen, Methoden, Interfaces, etc. sinnvoll bezeichnet werden sollen.

<table>
<thead>
<tr>
<th>Bezeichner -Typ</th>
<th>Konvention</th>
<th>Beispiele</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pakete</td>
<td>Paketnamen bestehen aus Kleinbuchstaben und sollten möglichst kurz sein. Ziffern sind auch erlaubt</td>
<td>de.frankfurt.cs.gdv.visian3d</td>
</tr>
<tr>
<td>Methoden</td>
<td>Methodennamen sollten Verben sein und mit einem Kleinbuchstaben anfangen. Darauf folgende Silben beginnen mit einem Großbuchstaben. Boolesche Funktionen sollten mit „is“, „has“, „can“, etc. anfangen</td>
<td>run(), getLeftChild(), setName(), isReady(), shouldWait(),...</td>
</tr>
<tr>
<td>Variablen</td>
<td>Variablennamen beginnen mit Kleinbuchstaben. Darauf folgende Silben fangen mit einem Großbuchstaben an. Sie sollten kurz und ausdrucksstark sein. Boolesche Variablen sollten mit „is“, „has“, „can“ etc. anfangen</td>
<td>age, userName, isVisible,... hasTicket, canStart,... shouldRestart,...</td>
</tr>
<tr>
<td>Konstanten</td>
<td>Konstanten bestehen aus Großbuchstaben. Wörter werden durch Unterstriche getrennt.</td>
<td>SIZE, MAX_VALUE, EDGE_COLOR,...</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Für Interfaces gilt die gleiche Konvention wie für Klassen</td>
<td></td>
</tr>
</tbody>
</table>

Tabelle 6: Namenskonventionen
12 Object, Math und String

12.1 Object

Die Klasse `java.lang.Object` ist die Oberklasse aller Java-Klassen. Jede Klasse, die nicht explizit von einer anderen Klasse abgeleitet wird, wird implizit von `Object` abgeleitet. Die folgenden beiden Klassendefinitionen sind demnach äquivalent:

```
class A {
}
class A extends Object {
}
```


Die `toString()`-Methode gibt eine String-Darstellung eines Objektes zurück. Z.B. wird die Methode `toString()` automatisch aufgerufen, wenn ein Objekt einer der `print`-Methoden `System.out.print()` oder `System.out.println()` übergeben wird. Sie wandelt dann das Objekt in einen String um.

Wenn die `toString()`-Methode eines Objektes der Klasse `Bankkonto` aufgerufen wird, liefert sie einen String, der etwa so aussehen könnte: `Bankkonto@3e25a5`

```
Bankkonto alexKonto=new Bankkonto(35870,"Alex M.",2103);
System.out.println(alexKonto);
Ausgabe: Bankkonto@3e25a5
```

Die Standardimplementierung der Methode liefert den Klassennamen und den Hash-Code des Objektes im Hexadezimal-Format als String zurück:

```
//Standardimplementierung von toString() in der Klasse Object
public String toString(){
    return getClass().getName() +"@" + Integer.toHexString(hashCode());
}
```

Wenn man eine sinnvolle String-Darstellung des Objektes haben will, dann muss die Methode überschrieben werden:

Die zweite Möglichkeit ist die Methode `boolean equals(Object o)`. Sie ist in der Klasse `java.lang.Object` definiert und wird somit an alle anderen Klassen vererbt. Der eigentliche Sinn der Methode `equals()` ist, zwei verschiedene Objekte auf inhaltliche Gleichheit zu untersuchen. Allerdings ist die Standardimplementierung von `equals()` in `Object` identisch zum Operator `==`, d.h. die von allen Klassen geerbte Methode `equals()` testet nur, ob zwei Referenzen auf dasselbe Objekt zeigen:

```java
//Standardimplementierung von equals(Object o) in der Klasse Object
public boolean equals(Object o){
    return this==o;
}
```

### 12.2 Math


```java
//Kompilierfehler: The constructor Math() is not visible
Math math=new Math();
```

Die Tabelle unten zeigt eine Übersicht der wichtigsten Methoden der Klasse `Math`.

<table>
<thead>
<tr>
<th>Methode</th>
<th>Rückgabe</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs(int i)</code></td>
<td>Absolutbetrag von i</td>
</tr>
<tr>
<td><code>ceil(double d)</code></td>
<td>Nächste Ganzzahl größer gleich i</td>
</tr>
<tr>
<td><code>exp(double a)</code></td>
<td>$e^a$</td>
</tr>
<tr>
<td><code>floor(double d)</code></td>
<td>Nächste Ganzzahl kleiner gleich i</td>
</tr>
<tr>
<td><code>log(double a)</code></td>
<td>$\ln(a)$</td>
</tr>
<tr>
<td><code>max(int i1, i2)</code></td>
<td>Maximum der Zahlen i1 und i2</td>
</tr>
<tr>
<td><code>min(int i1, i2)</code></td>
<td>Minimum der Zahlen i1 und i2</td>
</tr>
<tr>
<td><code>pow(double a, double b)</code></td>
<td>$a^b$</td>
</tr>
<tr>
<td><code>random()</code></td>
<td>Eine Zahl aus dem Intervall $[0.0,1.0]$</td>
</tr>
<tr>
<td><code>sqrt(double d)</code></td>
<td>Wurzel aus d</td>
</tr>
<tr>
<td><code>sin(double d)</code></td>
<td>Sinus-Funktion</td>
</tr>
<tr>
<td><code>cos(double d)</code></td>
<td>Kosinus-Funktion</td>
</tr>
<tr>
<td><code>tan(double d)</code></td>
<td>Tangens-Funktion</td>
</tr>
<tr>
<td><code>round(double d)</code></td>
<td>Rundet d zur nächsten Ganzzahl</td>
</tr>
</tbody>
</table>

**Tabelle 7: Methoden von `Math`**

### 12.3 String

Eine Zeichenkette (String) in Java ist eine indizierte geordnete Folge von Unicode-Zeichen. Strings in Java sind unveränderbar (immutable) und werden durch die Klasse `java.lang.String` repräsentiert. Die Indizierung der Zeichen beginnt mit 0.

Die Klasse ist `final` und kann nicht abgeleitet werden. Sie stellt jedoch zahlreiche Methoden zur Verarbeitung von Zeichenketten zur Verfügung. Ein String-Objekt kann entweder mit einem Konstruktor der Klasse `String` oder als String-Literal erstellt werden:

```java
String s2=new String("strings are immutable!");
String s3="strings are immutable!"; //String-Literal
```
Eine der wichtigsten Methoden der Klasse ist die Methode `equals()`, die `String` von `Object` erbt und überschreibt. Sie wird verwendet um zwei Strings auf inhaltliche Gleichheit zu testen und sollte immer zum Vergleich von Zeichenketten verwendet werden.

Siehe Übung.

```java
/*
 * Package: strings
 * Class: stringCompare.java
 * Method:
 */

public static void stringCompare()
{
    String s1=new String("use the equals()-method to compare strings");
    String s2=new String("use the equals()-method to compare strings");
    String s3="use the equals()-method to compare strings"; //String-Literal
    String s4="use the equals()-method to compare strings"; //String-Literal
    boolean s1_s2;
    boolean s3_s4;
    boolean s2_s3;
    if(s1==s2)
        s1_s2=true;
    else
        s1_s2=false;

    if(s3==s4)
        s3_s4=true;
    else
        s3_s4=false;

    if(s2==s3)
        s2_s3=true;
    else
        s2_s3=false;

    System.out.println(s1_s2+" "+s3_s4+" "+s2_s3);

    s2=s3;
    if(s2==s3)
        s2_s3=true;
    else
        s2_s3=false;

    System.out.println(s2_s3);
}

Ausgabe: false true false true
```

http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht20Ht/java_gebrauch_von_objekten_de

Im Folgenden werden weitere wichtige Methoden der Klasse `String` kurz erläutert.

- **int length()**: Liefert die Länge der Zeichenkette.

```java
String str="strings are immutable!";
System.out.println(str.length()); //liefert 22
```
char `charAt(int index)`:
Liefert das Zeichen an Position `index`. `index` muss ein Wert zwischen 0 und `length()-1` sein. Wird `charAt()` mit einem Index außerhalb dieses Intervalls aufgerufen, so wird eine Exception geworfen (`java.lang.StringIndexOutOfBoundsException`). Exceptions werden im folgenden Kapitel 00 ausführlich behandelt.

```java
String str="strings are immutable!";
System.out.println(str.charAt(str.length()-1)); // liefert '!
```

String `substring(int begin, int end)`:
Liefert den Teilstring, der am Index `begin` beginnt und bei `end-1` einschließlich endet.

```java
String str="strings are immutable!";
System.out.println(str.substring(2,6)); // liefert "ring"
```

String `substring(int begin)`: Liefert den Teilstring von der angegebenen Position bis zum Ende des Strings.

```java
String str="strings are immutable!";
System.out.println(str.substring(12)); // liefert "immutable!"
System.out.println(str.substring(12,str.length())); // liefert "immutable!"
```

String `trim()`: Liefert den String, der entsteht, wenn alle Leerzeichen am Anfang und am Ende des Strings entfernt werden.

```java
String str=" etc. ";
System.out.println(str.trim()); // liefert "etc."
```

boolean `equalsIgnoreCase()`: Vergleicht Strings miteinander ohne Groß-/Kleinschreibung zu beachten.

```java
String str="strings are immutable!";
System.out.println(str.equalsIgnoreCase("Strings are immutable!"))); // liefert true
System.out.println(str.equals("Strings are immutable!"))); // liefert false
```
String `toLowerCase()` bzw. String `toUpperCase()`: Liefert den String zurück, der entsteht, wenn alle Zeichen in Kleinbuchstaben bzw. in Großbuchstaben umgewandelt werden.

```java
String str="Strings Are Immutable!";
System.out.println(str.toLowerCase()); // liefert "strings are immutable!"
System.out.println(str.toUpperCase()); // liefert "STRINGS ARE IMMUTABLE!"
```

String `replace(char oldChar, char newChar)`: Ersetzt alle Vorkommen des Zeichens `oldChar` durch das Zeichen `newChar`.

```java
System.out.println("Kik".replace('k', 'K')); // liefert KiK
```
13 Ausnahmen (Exceptions)

Ausnahmen (Exceptions) sind Java-Klassen zur Erkennung und Behandlung von Laufzeitfehlern. Ein Exception-Objekt repräsentiert einen Fehler, der während der Ausführung eines Programms auftritt und den normalen Ablauf des Programms stört oder gar beendet. Laufzeitfehler können durch den Compiler i.Allg. nicht erkannt werden. Es gibt drei unterschiedliche Arten von Laufzeitfehlern in Java:

1. **Vermeidbare Fehler (Bugs):**
   Das sind Fehler, die durch den Programmierer verursacht werden und vermieden werden können wie z.B. Division durch 0, Zugriff auf einen ungültigen Array-Index, Zugriff auf eine null-Referenz, etc.

   ```java
   public void buggyMethod(){
      Random rand=new Random();
      int x=rand.nextInt(6);
      System.out.println("x=");
      int y=2/x; //Division durch Null -->
      //ArithmeticException:/by zero
      int size=x*2-3;
      int [] list = new int[size]; //negativer Wert für Array-Länge
      //-->NegativeArraySizeException
      float [] array=new float[x];
      array[x-3]=3.5f; //ungültiger Array-Index -->
      //ArrayIndexOutOfBoundsException: 5
      String str="extract @";
      str.charAt(x-4); //ungültiger Index -->
      if(x==4) //StringIndexOutOfBoundsException
      Bugs [] bugs=new Bugs[x];
      bugs[0].buggyMethod(); //Aufruf einer Methode einer
      //null-Referenz --> NullPointerException
      System.out.println("Keine Exceptions");
   }
   ```

2. **Unvermeidbare Fehler:**
   Das sind Fehler, die der Programmierer nicht vermeiden kann. Beispiele hierfür:
   a. Ein Programm versucht auf einen Server zuzugreifen, der gerade ausgeschaltet ist:

   ```java
   URL apacheServer = new URL("http://www.apache.org/");
   connect(apacheServer);
   ```

   b. Ein Programm versucht eine Datei zu öffnen, die beschädigt ist oder gelöscht wurde.

3. **Systemfehler (System Errors):**
Das sind schwerwiegende nicht behebbare Fehler, nach deren Auftreten die Fortsetzung des Programms nicht sinnvoll bzw. gar nicht möglich ist. Sie führen zu einem Programmabbruch und zum Beenden des Interpreters. Beispiele hierfür sind Speicherüberlauf (Memory Overflow) oder Fehler in der JVM selbst.

Wenn ein Programmteil (eine Methode oder Anweisung) einen Fehler verursacht, löst die JVM eine Exception aus (throwing exception), die vom Programm abgefangen (catching exception) und behandelt (handling exception) werden kann. Zur Behandlung einer Ausnahme übergibt die JVM ein Exception-Objekt an das Programm, das Informationen über den Fehler enthält wie z.B. Fehlerart, Fehlerbezeichnung, Programmzustand zum Zeitpunkt des Auftretens des Fehlers, etc. Das Behandeln von Ausnahmen erfolgt mittels einer try-catch-Anweisung:

```java
package exceptions;
class SimpleException {
    public static void convertStringToNumber(String numberString) {
        try {
            // If numberString can’t be converted to a number, the method parseFloat will throw a NumberFormatException.
            float number = Float.parseFloat(numberString);
            System.out.println("Converting "+numberString+" to a number succeeded: "+number);
        } catch (NumberFormatException exceptionObject) {
            // Handling the exception
            System.out.println("Exception: Unable to convert "+numberString+" to a number!");
            System.out.println(exceptionObject.getMessage());
            System.out.println("Printing the stack trace...");
            exceptionObject.printStackTrace();
        }
        convertStringToNumber("549.34");
    }

    public static void main(String[] args) {
        convertStringToNumber("549.34");
        System.out.println("Ausgabe: Converting 549.34 to a number succeeded: 549.34");
        convertStringToNumber("549,34");
        System.out.println("Ausgabe: Exception: Unable to convert 549,34 to a number!");
        System.out.println("For input string: ": "549,34");
        System.out.println("Printing the stack trace...");
        java.lang.NumberFormatException: For input string: "549,34"
at sun.misc.FloatingDecimal.readJavaFormatString(Unknown Source)
at java.lang.Float.parseFloat(Unknown Source)
at exceptions.SimpleException.convertStringToNumber(SimpleException.java:25)
at exceptions.SimpleException.main(SimpleException.java:39)
```

73
13.1 Throwable


```
For input string: "549,34"
```

Die Methode `printStackTrace()` liefert eine Art Momentaufnahme des Programmstacks (`stack trace`) zum Zeitpunkt des Auftretens des Fehlers in Form einer Textnachricht:

```
java.lang.NumberFormatException: For input string: "549,34"
    at sun.misc.FloatingDecimal.readJavaFormatString(Unknown Source)
    at java.lang.Float.parseFloat(Unknown Source)
    at exceptions.SimpleException.convertStringToNumber(SimpleException.java:25)
    at exceptions.SimpleException.main(SimpleException.java:39)
```

Die Textnachricht enthält die Zeilennummer im Quellcode, in der der Fehler auftrat:

```
SimpleException.java:25
```

Sie zeigt auch die Stelle (Klassenname und Zeilennummer), von der aus die Methode aufgerufen wurde, die die Ausnahme ausgelöst hatte:

```
SimpleException.main(SimpleException.java:39)
```

13.2 Ausnahmetypen

Grundsätzlich gibt es in Java zwei Arten von Ausnahmen: geprüfte Ausnahmen (`checked exceptions`) und Laufzeitausnahmen (`runtime exceptions`), auch ungeprüfte Ausnahmen (`unchecked exceptions`) genannt. Wie oben erwähnt gibt es in Java drei Fehlerklassen: vermeidbare Fehler (Bugs), unvermeidbare Fehler und Systemfehler. Laufzeitausnahmen repräsentieren vermeidbare Fehler. So ist z.B. die Ausnahme `ArithmeticException`, die u.a. ausgelöst wird, wenn der Programmierer eine Division durch 0 durchführt. Eine Laufzeitausnahme:

```
int x=1/3;
int y=2/x; // Division durch null -->
// Laufzeitausnahme: ArithmeticException: /by zero
```

Die Ausnahmen `FileNotFoundException` und `IOException` im folgenden Beispiel sind geprüfte Ausnahmen:

```java
/*
 * Package: exceptions
 * Class: SimpleException.java
 * Method: printFile
 */

public void printFile(String fileName){
  //try to read and print the given file
  BufferedReader input = null;
  try {
    input = new BufferedReader(new FileReader(fileName));
    String line = null;
    while ((line = input.readLine()) != null){
      System.out.println(line);
    }
  } catch (IOException ex){
    ex.printStackTrace();
  }
}
```


In Bezug auf die Behandlung von Ausnahmen gibt es einen wesentlichen Unterschied zwischen Laufzeitausnahmen und geprüften Ausnahmen. Laufzeitausnahmen können, müssen jedoch nicht abgefangen und behandelt werden. So könnte man in der Methode `convertStringToNumber()` im obigen Beispiel die `ArithmeticException` abfangen und behandeln, indem man sie mit einer `try-catch`-Anweisung umschließt:

```java
try{
  int x=1/3;
  int y=2/x; //Division durch null -->ArithmeticException: / by zero
} catch(ArithmeticException exceptionObject){
  //Catching the exception
  //handling the exception
  System.out.println("x ist 0. Division wurde nicht ausgeführt!");
}
```

Ähnlich kann man bei der Behandlung der Ausnahmen `ArrayIndexOutOfBoundsException`, `NegativeArraySizeException`, `NullPointerException` und `StringIndexOutOfBoundsException` vorgehen.
13.3 Auslösen einer Exception

Der Programmierer kann Methoden definieren, die selbst Ausnahmen auslösen. Jede Methode kann so deklariert werden, dass sie eine oder mehrere Ausnahmen wirft. Dies erfolgt mithilfe der Anweisung `throws` wie das folgende Beispiel zeigt:

```java
package exceptions;
public class SimpleException {
    public long factorial(int n) throws NegativeValueException {
        if (n < 0) throw new NegativeValueException("invalid parameter: "+n);
        long result = 1;
        for (int i = 2; i <= n; i++)
            result *= i;
        return result;
    }
}
```

Da `factorial()` eine geprüfte Ausnahme wirft (siehe API-Dokumentation), muss jede andere Methode, die `factorial()` aufrufen will, dies innerhalb einer `try-catch`-Anweisung tun:

```java
public long calculator(int n) {
    try {
        return factorial(n);
    } catch (Exception exceptionObject) {
        System.out.println("The input parameter of factorial " +
                        "must be a positive number!");
    }
}
```

13.4 Weitergabe von Exceptions

In vorigen Abschnitt wurde erwähnt, dass geprüfte Ausnahmen immer abgefangen und behandelt werden müssen. Es ist aber:

```java
public long calculator(int n) throws Exception {
    int x = n/2;
    return factorial(x);
}
```
Im obigen Beispiel behandelt die Methode calculator() keine Ausnahmen. Wenn factorial() eine Ausnahme wirft, leitet sie calculator() an die aufrufende Methode (die Methode, die calculator() aufruft) weiter.

### 13.5 Behandlung von mehreren Ausnahmen

Es ist durchaus möglich, dass innerhalb einer try-catch-Anweisung mehrere unterschiedliche Ausnahmen auftreten. Wenn ein Programm mehr als eine Ausnahme behandeln will, dann muss es für jede Ausnahme eine eigene catch-Anweisung bereitstellen, wie das folgende Beispiel zeigt:

```java
public void printFile(String fileName)
    //try to read and print the given file
    BufferedReader input = null;
    try {
        input = new BufferedReader( new FileReader(fileName) );
        String line = null;
        while (( line = input.readLine()) != null){
            System.out.println(line);
        }
    } catch (FileNotFoundException ex) {
        ex.printStackTrace();
    } catch (IOException ex){
        ex.printStackTrace();
    }
}
```

### 13.6 finally

Am Ende einer try-catch-Anweisung kann ein finally-Block stehen, der immer ausgeführt wird, unabhängig davon, ob eine Ausnahme aufgetreten ist oder nicht. Tritt eine Ausnahme auf, so wird der entsprechende catch-Block betreten und ausgeführt. Danach wird der finally-Block ausgeführt. Danach wird der finally-Block ausgeführt. Tritt keine Ausnahme auf, so wird nur der finally-Block ausgeführt:
public void finallyMethod(){
    try{
        Random rand=new Random();
        int x=rand.nextInt(6);
        System.out.println("x="+x);
        int y=2/x; //Division durch null --> ArithmeticException: / by zero
        int [] list = new int[size]; //negativer Wert für Array-Länge --> NegativeArraySizeException
        float [] array=new float[x];
        array[x-3]=3.5f; //ungültiger Array-Index --> ArrayIndexOutOfBoundsException: 5
        String str="extract @";
        str.charAt(x-4); //ungültiger Index --> StringIndexOutOfBoundsException
        Bugs [] bugs=new Bugs[x];
        if(x==4)
            bugs[0].buggyMethod(); //Aufruf einer Methode einer null-Referenz --> NullPointerException
        System.out.println("Keine Exceptions");
    }
    catch (ArithmeticException arithmeticExceptionObject) {
        System.out.println("ArithmeticException...");
    }
    catch (NegativeArraySizeException arithmeticExceptionObject) {
        System.out.println("NegativeArraySizeException...");
    }
    catch (ArrayIndexOutOfBoundsException arrayIndexOutOfBoundsExOb) {
        System.out.println("ArrayIndexOutOfBoundsException...");
    }
    catch (StringIndexOutOfBoundsException stringIndexOutOfBoundsExOb) {
        System.out.println("StringIndexOutOfBoundsException...");
    }
    catch (NullPointerException nullPointerExceptionObject) {
        System.out.println("NullPointerException");
    }
    finally{
        System.out.println("Finally-Block");
    }
}
13.7 Eigene Exception-Klasse deklarieren


Die Klasse sollte zwei Konstruktoren deklarieren: einen parameterlosen Konstruktor und einen Konstruktor, der als Parameter eine Textnachricht (message) übergeben bekommt. Dies ist der Text, der zurückgegeben wird, wenn `getMessage()` aufgerufen wird. Beispiele:

```java
/**
 * Checked exception used in the method factorial of the class
 * SimpleException
 *
 */
public class NegativeValueException extends Exception{

    public NegativeValueException(){
    }

    public NegativeValueException(String message){
        super(message);
    }
}

/**
 * Runtime exception used in the method auszahlen() of the class
 * exceptions.Bankkonto
 */

public class KreditLimitUeberschrittenException extends RuntimeException{

    public KreditLimitUeberschrittenException(){
    }

    public KreditLimitUeberschrittenException(String message){
        super(message);
    }
}
```
Generics sind eine der wichtigsten Erweiterungen von Java in der Version 5. Das Grundprinzip des Konzeptes soll anhand eines einfachen Beispiels erläutert werden. Betrachten wir die folgende Implementierung von Bubble-Sort:

```java
class NonGenericBubbleSort{
    public void bubbleSort (Integer [] A){
        for (int i=0;i<A.length;i++) {
            for(int j=A.length-1;j>=i+1;j--){
                    swap(A, j, j-1);
                }
            }
        }
    }

        Integer temp=A[i];
        A[i]=A[j];
        A[j]=temp;
    }

} // Ende NonGenericBubbleSort

class BubbleSorter {
    public static void main(String[] args) {
        Integer[] A=new Integer[]{-1,0,4,2,1,0,3};
        NonGenericBubbleSort bubbleSorter=new NonGenericBubbleSort();
        bubbleSorter.bubbleSort(A);
        for (Integer element : A)
            System.out.print(element=" ");
    }
} 

Ausgabe: -1 0 0 1 2 3 4
```
Ein Sortieralgorithmus kann Elemente beliebiger Datentypen sortieren, sofern diese paarweise vergleichbar sind (auf den Elementen ist eine Ordnungsrelation definiert; siehe Abschnitt 10.1 Das Interface Comparable). Die obige Implementierung beschränkt sich jedoch nur auf das Sortieren von Integer-Zahlen. Wenn Elemente eines anderen Typs (z.B. Float) sortiert werden sollen, muss der Code an mindestens drei Stellen angepasst und die Methoden bubbleSort() und swap() überladen werden:

```java
/**
 * Package: generics
 * Class:   NonGenericBubbleSort.java
 * Method:
 */
//1. Überladung der Methode bubbleSort()
public void bubbleSort (Float [] A){
    //...
}
public void swapFloat(Float []A, int i, int j){
    Float temp=A[i]; // Anpassung
    A[i]=A[j];
    A[j]=temp;
}
```

Für die sieben unterschiedlichen primitiven Datentypen (int, long, double, ...) müssten dann die beiden Methoden jeweils 6 Mal überladen werden. Darüber hinaus muss die temp-Variable in der swap-Methode angepasst werden. Insgesamt hätte man 14 Methoden, die sich kaum unterscheiden. Dies ist kein guter Stil, einen einfachen Sortieralgorithmus zu implementieren.

Dank des Generics-Konzeptes ist es nun möglich, die nicht generische Klasse NonGenericBubbleSort in eine generische Klasse umzuschreiben, die Elemente beliebiger Datentypen mit einer einzigen Implementierung der Methoden bubbleSort und swap sortieren kann:

```java
/**
 * Package: generics
 * Class:   GenericBubbleSort.java
 * Method:
 */
/**
 * Generische Klasse zur Sortierung von Integer-Arrays
 *
 */
class GenericBubbleSort<T extends Object & Comparable>{
    public void bubbleSort(T [] A){
        for (int i=0; i<A.length; i++) {
            for (int j=A.length-1; j>=i+1; j--){
                }
            }
        }
    }
}
```

Um die generische Klasse GenericBubbleSort in BubbleSorter nutzen zu können, müssen wir sie mit einem konkreten Typ (parametrisierter Typ) etwa Integer, Float oder String instanzieren. Der konkrete Typ steht in spitzen Klammern hinter dem Klassennamen:

Mit der Implementierung von BubbleSort als eine generische Klasse lassen sich Arrays beliebiger Datentypen mit einer einzigen Implementierung der Methoden bubbleSort() und swap() sortieren.

Nicht nur Klassen können als generisch deklariert werden, sondern auch Methoden. Zusätzlich zu den generischen Typen lassen sich auch generische Methoden definieren. Statt die gesamte Klasse BubbleSort als generisch zu deklarieren, könnte man nur die Methoden bubbleSort() und swap() generisch deklarieren:
Auch hier besagen die spitzen Klammern, die hinter dem Modifikator stehen, dass die Methode einen Datentyp namens T verwendet, von dem nur bekannt ist, dass seine Objekte das Comparable-Interface implementieren.

Mehr zum Thema Generics unter:
und
http://angelikalanger.com/Articles/JavaMagazin/Generics/GenericsPart2.html
15 Collections (Sammlungen)


Den Kern des Collections-Frameworks bilden die sechs Interfaces Collection, List, Set, SortedSet, Map und SortedMap:

Abbildung 9: Collection


```java
//Collection-Interface
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
}
```


15.1 Listen

Eine Liste im Collection-Framework ist vergleichbar mit einem dynamisch wachsenden Array, das folgende Eigenschaften aufweist:


Die wichtigsten Listen, die das Collection-Framework implementiert, sind ArrayList, Vector, Stack, und LinkedList. Sie alle sind Klassen, die das List-Interface implementieren:

```java
//List-Interface
interface List<E> extends Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<?> c);
    boolean addAll(int index, Collection<?> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    List<E> subList(int fromIndex, int toIndex);
}
```
List enthält Methoden u.a. zum Suchen, Einfügen und Entfernen von Objekten.

`boolean add(Object obj):`
Fügt das Objekt `obj` am Ende der Liste ein. Die Methode gibt `true` zurück, wenn das Element erfolgreich in die Liste eingefügt wurde.

`void add(int index, Object obj):`
Fügt das Objekt an der Stelle `index` in die Liste ein. Es ist hier zu beachten, dass das Element, welches sich vorher an der Stelle `index` befand und alle Elemente rechts von ihm, eine Stelle nach hinten geschoben werden.

`Object remove(int index):`
Löscht das Objekt an der Stelle `index` und gibt es zurück.

`Object set(int index, Object obj):`
Ersetzt das Objekt an der Stelle `index` durch das Objekt `obj`. Das alte Objekt wird zurückgegeben.

`Object get(int index):`
Gibt das Objekt an der Stelle `index` zurück.

`int size():`
Gibt die Anzahl der Objekte zurück, die in der Liste gespeichert sind.

`boolean contains(Object obj):`
Gibt `true` zurück, wenn `obj` mindestens einmal in der Liste enthalten ist.

`int indexOf(Object obj):`
Gibt die Indexnummer des ersten Vorkommens von `obj` zurück. Wenn `obj` nicht in der Liste vorkommt, wird -1 zurückgegeben.

`int lastIndexOf(Object obj):`
Gibt die Indexnummer des letzten Vorkommens von `obj` zurück. Sollte das Element nicht vorhanden sein, wird -1 zurückgegeben.

`void clear():`
Löscht alle Elemente in der Liste.
15.1.1 ArrayList und Vector


Beispiel:

```java
/*
 * Package: collections
 * Class:   ArrayListSample.java
 * Method:
 */
public class ArrayListSample {
    ArrayList physicalUnits = new ArrayList();

    public ArrayListSample()
        physicalUnits.add("Joule");
        physicalUnits.add("Pascal");
        physicalUnits.add("Volt");
        physicalUnits.add("Coulomb");
        physicalUnits.add(null);//null-Objekte sind erlaubt
        physicalUnits.add("Volt");
        physicalUnits.add("Ohm");
        physicalUnits.add("Newton");
        physicalUnits.add(1,"Celsius");
    }

    public static void main(String ... args) {
        ArrayListSample arrayListSample=new ArrayListSample();
        System.out.print("Physical units: ");
        for (Object unit : arrayListSample.physicalUnits)
            System.out.print(unit+ "");
        System.out.println();
        System.out.println("physicalUnits.size(): "+
            arrayListSample.physicalUnits.size());
        System.out.println("physicalUnits.isEmpty(): "+
            arrayListSample.physicalUnits.isEmpty());
        System.out.println("physicalUnits.contains("celsius")": "+
            arrayListSample.physicalUnits.contains("celsius");
        System.out.println("physicalUnits.remove("Coulomb")": "+
            arrayListSample.physicalUnits.remove("Coulomb");
        System.out.println("physicalUnits.remove(2):
            "+arrayListSample.physicalUnits.remove(2));
        System.out.println("physicalUnits.size()-2): "+
```
Ausgabe:
Physical units: Joule Celsius Pascal Volt Coulomb null Volt Ohm Newton
physicalUnits.size(): 9
physicalUnits.isEmpty(): false
physicalUnits.contains("celsius") = false
physicalUnits.remove("Coulomb") = true
physicalUnits.remove(2): Pascal
physicalUnits.size()-2): Ohm
physicalUnits.lastIndexOf("Pascal") = -1

15.1.2 LinkedList

Eine LinkedList ist eine doppelt verkettete Liste, die neben List das deque-Interface implementiert:

```java
//Das Deque-Interface
public interface Deque<E> extends Queue<E> {
    void addFirst(E e);
    void addLast(E e);
    boolean offerFirst(E e);
    boolean offerLast(E e);
    E removeFirst();
    E removeLast();
    E pollFirst();
    E pollLast();
    E getFirst();
    E getLast();
    E peekFirst();
    E peekLast();
    boolean removeFirstOccurrence(Object o);
    boolean removeLastOccurrence(Object o);
    boolean add(E e);
    boolean offer(E e);
    boolean remove(E e);
    E remove();
    E poll();
    E element();
    E peek();
    void push(E e);
    E pop();
    boolean remove(Object o);
    boolean contains(Object o);
    public int size();
    Iterator<E> iterator();
    Iterator<E> descendingIterator();
}
```
**boolean add(E e)** und **boolean offer(E e)**: Fügen das Element e am Ende der Liste ein.

**E remove()** und **E poll()**: Entfernen das erste Element in der Liste und geben es zurück.

**E element()** und **E peek()**: Liefern das erste Element in der Liste, ohne es zu entfernen.


### 15.2 Mengen (Sets)


#### 15.2.1 HashSet


```java
/*
 * Package: collections
 * Class:   HashSetSample.java
 * Method:
 */

public class HashSetSample {
    Set hashSet=new HashSet();

    public HashSetSample() {
        for(int i=5;i>=0;i--)
            hashSet.add(i*i);
}
```
hashSet.add(3.5f);
hashSet.add(null);//null-Objekte sind erlaubt

public static void main(String ... args) {
    HashSetSample hashSetSample = new HashSetSample();
    System.out.println(hashSetSample.hashSet);
}
Ausgabe:
    [null, 0, 1, 16, 3.5, 4, 25, 9]

15.2.2 LinkedHashSet

Eine java.util.LinkedHashSet hat die gleichen Eigenschaften wie ein HashSet. Sie unterscheidet sich davon lediglich dadurch, dass sie die Elemente in der Reihenfolge speichert, in der sie eingefügt wurden. LinkedHashSet sind als doppelt verkettete Listen implementiert und können null-Objekte aufnehmen.

15.2.3 TreeSet

Ein java.util.TreeSet speichert seine Elemente in einem Rot-Schwarz-Baum (eine effizienter binärer Suchbaum) und lässt weder Duplikate noch null-Objekte zu. TreeSet unterstützen das Einfügen, Finden und Entfernen von Elementen in logarithmischer Zeit. TreeSet implementiert das Interface java.util.SortedSet. SortedSet erweitert Set um zahlreiche Methoden für den Umgang mit sortierten Daten:

interface SortedSet<E> extends Set<E> {
    Comparator<? super E> comparator();
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);
    E first();
    E last();
}

E first() bzw. E last(): Liefert das größte bzw. kleinste Element der Menge zurück.

SortedSet<E> subSet(E fromElement, E toElement):
    Liefert alle Elemente im halboffenen Intervall [fromElement, toElement] zurück.

SortedSet<E> headSet(E toElement):
    Liefert alle Elemente kleiner gleich toElement zurück.
SortedSet<E> tailSet(E fromElement):
Liefert alle Elemente größer gleich fromElement zurück.

TreeSet verlässt sich bei der Suche nach Duplikaten auf die Implementierung der equals()-Methode ihrer Objekte. Damit TreeSet richtig funktionieren kann, sollten ihre Elemente die equals()-Methode der Klasse Object oder das Comparable-Interface implementieren (siehe 10.1 Das Interface Comparable), denn eine Entscheidung, ob ein Element e1 größer oder kleiner als ein Element e2 ist, wird durch Aufruf der Methode equals(), also e1.equals(e2), getroffen.
15.3 Tabellen (Maps)


Die wichtigsten *Maps* sind `HashMap`, `HashTable`, `TreeMap` und `LinkedHashMap`. Sie alle implementieren das *Map*-Interface entweder direkt oder indirekt.

```java
public interface Map<K, V> {
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);
    V put(K key, V value);
    V remove(Object key);
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K, V>> entrySet();
}
```

Die wichtigsten Methoden des *Map*-Interfaces:

*V put*(K key, V value):

*boolean containsKey*(Object key):
  Gibt true zurück, wenn der Schlüssel in der Tabelle existiert.

*V get*(Object key):
  Gibt das dem Schlüssel zugeordnete Objekt zurück, oder null, falls der Schlüssel nicht in der Tabelle enthalten ist.

*Set<K> keySet*():
  Gibt alle Schlüssel der Tabelle als Menge zurück.
Collection<V> values():
Gibt eine Collection der Objekte in der Map zurück. In der Collection können Duplikate enthalten sein.
15.3.1 HashMap und Hashtable


HashSet und Hashtable lassen null-Schlüssel und null-Objekte zu. Ihre Elemente sind nicht geordnet.

15.3.2 TreeMap


Ein Schlüssel kann jedoch mit einem null-Objekt assoziiert sein.

```java
/*
 * Package: collections
 * Class:   TreeMapSample.java
 * Method:
 */

public class TreeMapSample {
    private Map<String, String> treeMap =
        new TreeMap<String, String>();

    public TreeMapSample(){
        treeMap.put("Energy", "Joule");
        treeMap.put("Work", "Joule");
        treeMap.put("Pressure", "Pascal");
        treeMap.put("Voltage", "Volt");
        treeMap.put("Frequency", "Hertz");
        treeMap.put("Resistance", "Ohm");
        treeMap.put(null, "Unknown"); //null-Schlüssel sind nicht erlaubt
        treeMap.put("Power", "Watt");
        treeMap.put("Temperature", "Celsius");
        treeMap.put("Conductance", "Siemens");
        treeMap.put(Unknown, null); //null-Werte sind erlaubt
    }

    public static void main(String[] args) {
        System.out.println("Sorted Map:");
        TreeMapSample treeMapSample = new TreeMapSample();

        for (String key : treeMapSample.treeMap.keySet())
            System.out.println(key + " : " + treeMapSample.treeMap.get(key));
    }
}

Ausgabe:
Sorted Map:
Conductance: Siemens
Energy: Joule
Frequency: Hertz
Power: Watt
Pressure: Pascal
Resistance: Ohm
Temperature: Celsius
Unknown: null
Voltage: Volt
Work: Joule
```
15.4.3 LinkedHashMap

Im Gegensatz zu HashMap und Hashtable, merkt sich eine java.util.LinkedHashMap die Reihenfolge, in der die Schlüssel/Wert-Paare eingefügt werden.

```java
public class MapSample {

    public java.util.Map<Integer, Integer> linkedMap =
        new LinkedHashMap<Integer, Integer>();
    public java.util.Map<Integer, Integer> treeMap =
        new TreeMap<Integer, Integer>();
    public java.util.Map<String, Integer> hashMap =
        new HashMap<String, Integer>();

    public MapSample() {
        int key=11;
        while (key-->0) {
            linkedMap.put(key, key*key);
            treeMap.put(key, key*key);
            hashMap.put(key+"", key*key);
        }
    }

    public static void main(String ... args) {
        MapSample mapSample=
            new MapSample();
        System.out.print("Ordered Map:\t ");
        Iterator<Integer> it=mapSample.linkedMap.keySet().iterator();
        while (it.hasNext())
            System.out.print(mapSample.linkedMap.get(it.next())+" ");
        System.out.println();
        System.out.print("Sorted Map:\t ");
        it=mapSample.treeMap.keySet().iterator();
        while (it.hasNext())
            System.out.print(mapSample.treeMap.get(it.next())+" ");
        System.out.println();
        System.out.print("Unordered Map:\t ");
        Iterator<String> it2=mapSample.hashMap.keySet().iterator();
        while (it2.hasNext())
            System.out.print(mapSample.hashMap.get(it2.next())+" ");
    }
}

Ausgabe:

Ordered Map:   100 81 64 49 36 25 16 9 4 1 0
Sorted Map:    0 1 4 9 16 25 36 49 64 81 100
Unordered Map: 9 4 1 100 0 49 36 25 16 81 64
```
15.4 Generische Collections

Vor JDK 1.5 waren alle Collection-Klassen nicht generisch. Mit der `add`-Methode einer nicht generischen Liste könnte man z.B. Objekte unterschiedlicher Klassen in dieselbe Datenstruktur einfügen:

```java
List list = new ArrayList();
list.add(new Integer(5));
list.add("String-Object");
list.add(new Float(2.3f));
Integer i = (Integer) list.get(0); // Casting erforderlich
String s = (String) list.get(1); // Casting erforderlich
Float f = (Float) list.get(2); // Casting erforderlich
```


Seit Version 5 sind alle Collection-Klassen und -Interfaces generisch und somit homogen. Zugriffe auf Elemente erfordern kein Casting mehr:

```java
List<Integer> list2 = new ArrayList<Integer>();
list2.add(new Integer(5));
list2.add(3);
list2.add(new Integer(2));
Integer ii = list2.get(0); // Casting ist nicht erforderlich
Integer ss = list2.get(1); // Casting ist nicht erforderlich
Integer ff = list2.get(2); // Casting ist nicht erforderlich
```
15.5 Iteratoren (Iterator)

Ein Iterator ist ein Objekt, das das java.util.Iterator-Interface implementiert. Iteratoren bieten einen abstrakten Mechanismus zur Traversierung von Collections. Das Iterator-Interface enthält nur drei Methoden:

```java
//Iterator-Interface
interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Ein Iterator hat einen Zeiger. Wenn ein Iterator initialisiert wird, zeigt dieser Zeiger auf das erste Element in der Sammlung.

```java
E next():
    liefert das Objekt zurück, auf das der Iterator-Zeiger zeigt und bewegt anschließend den Zeiger zum nächsten Element. Wenn der Zeiger beim letzten Element angelangt ist, liefern nachfolgende next()-Aufrufe null zurück. Der Iterator kann dann nicht mehr verwendet werden.
```

```java
boolean hasNext(): liefert genau dann true zurück, wenn die Sammlung nicht leer ist und der Zeiger nicht das Ende der Liste erreicht hat.
```

```java
void remove(): ruft next() auf und entfernt ggf. das Element, das next() zurück liefert.
```

Das Collection-Interface verfügt über die Methode Iterator<E> iterator(), die einen Iterator für die Collection zurück liefert. Der Iterator liefert die Elemente seiner Collection in der Reihenfolge zurück, in der sie gespeichert sind.
public class IteratorSample {
    public List<Integer> list = new ArrayList<Integer>();
    public Set<Integer> set = new HashSet<Integer>();
    public Map<Integer, Integer> map = new HashMap<Integer, Integer>();

    public IteratorSample(){
        int ii=0;
        for(int i=0; i<11; i++){
            ii=i*i;
            list.add(ii);
            set.add(ii);
            map.put(i, ii);
        }
    }

    public static void main(String[] args) {
        IteratorSample is=new IteratorSample();
        Iterator<Integer> it=is.list.iterator();
        System.out.print("List: ");
        while(it.hasNext())
            System.out.print(it.next()+" ");
        System.out.println();

        it=is.set.iterator();
        System.out.print("Set: ");
        while(it.hasNext())
            System.out.print(it.next()+" ");
        System.out.println();

        it=is.map.keySet().iterator();
        System.out.print("Map: ");
        while(it.hasNext()){
            int key=it.next();
            System.out.print("("+key+","+is.map.get(key)+")");
        }
    }
}
16 I/O

16.1 Dateien und Verzeichnisse


Die Klasse verfügt u. a. über den folgenden Konstruktor:
```
File(String pathName):
```

Er erzeugt ein Objekt, das lediglich eine abstrakte Repräsentation der Datei oder des Verzeichnisses kapselt. Es ist an dieser Stelle zu betonen, dass das Instanziieren eines File-Objektes weder das Erstellen einer Datei oder eines Verzeichnisses bedeutet noch deren Existenz nicht voraussetzt. Es handelt sich lediglich um eine abstrakte Beschreibung einer Datei oder eines Verzeichnisses. So ist es zum Beispiel möglich ein File-Objekt zu erstellen, das einen syntaktisch falschen Dateinamen enthält.


Beispiele:
```
File f = new File("c:\daten"); // absolutes Pfad eines Ordners
File f2 = new File("c:\daten\Veranstaltung.java"); // absolutes Pfad einer Datei
File f3 = new File(".\daten\Veranstaltung.java"); // relativer Pfad einer Datei
File f4 = new File(".././beispiele"); // relativer Pfad eines Ordners
```

Die Klasse `File` verfügt u. a. über folgende Methode:
```
public boolean mkdir(): Legt ein Verzeichnis an.
public boolean delete():
public boolean renameTo(File dest):
public String [] list():
```

Die Klasse `File` verfügt u. a. über folgende Methode:
repräsentiert.

**public boolean mkdir():** Legt ein Verzeichnis an.

**public boolean delete():** Löscht die Datei bzw. das Verzeichnis

**public boolean renameTo(File dest):** Die Datei bzw. das Verzeichnis wird in das als Parameter übergebene Objekt umbenannt.

**public String [] list():** Liefert ein Array von String, das für jeden gefundenen Verzeichniseintrag ein Element enthält.

```java
/*
 * Package: io
 * Class:   SimpleFileApplication.java
 * Method:  main()
 */

package io;
import java.io.File;

public class SimpleFileApplication {

    private static File f = new File("SimpleFileApplication.java"); // relativer Pfad einer Datei

    private static File d = new File("./"); // relativer Pfad eines Ordners

    public static void main(String[] args) {
        System.out.println("Die Datei "+f.getName()+" befindet sich im Ordner: "+
                           d.getAbsolutePath());
        System.out.println(f.getName());
        System.out.println(f.getAbsolutePath());
        System.out.println(f.getParent());
        System.out.println(f.exists());
        System.out.println(f.canWrite());
        System.out.println(f.canRead());
        System.out.println(f.isFile());
        System.out.println(f.isDirectory());
        System.out.println(f.lastModified());
        System.out.println(f.length());
        System.out.println(d.getName());
        System.out.println(d.getAbsolutePath());
        System.out.println(d.getParent());
        System.out.println(d.exists());
        System.out.println(d.canWrite());
        System.out.println(d.canRead());
        System.out.println(d.isFile());
        System.out.println(d.isDirectory());
        System.out.println(d.lastModified());
    }
}
```
Das folgende Programm listet alle im angegebenen Verzeichnis enthaltenen Dateien und Verzeichnisse rekursiv auf und gibt sie aus.

```java
/*
 * Package: io
 * Class:   FileLister.java
 * Method:  *
 */
package io;

import java.io.File;

/**
 * Listet alle im angegebenen Verzeichnis enthaltenen Dateien und
 * Verzeichnisse rekursiv auf
 *
 */
public class FileLister {
    public void listRec(File directory, int depth) {
        String[] list = directory.list();
        for (String fileName : list) {
            for (int i = 0; i < depth; i++)
                System.out.print("  ");
            System.out.println(fileName);
            File child = new File(directory, fileName);
```
```java
if (child.isDirectory())
    listRec(child, depth + 1);

// Das Programm erwartet den Namen eines existierenden Verzeichnisses.
public static void main(String[] args) {

    // Wenn das Programm nicht den Namen eines Verzeichnisses als Parameter
    // übergeben bekommt, listet es den Inhalt des aktuellen Verzeichnisses.

    String path = "/"; // Das aktuelle Verzeichnis.
    if (args.length > 0)
        path = args[0];
    File f = new File(path);
    if (!f.isDirectory()) {
        System.out.println(path + " doesn't exist or not a dir.");
        System.exit(0); // Das Programm wird beendet.
    }
    FileLister fl = new FileLister();
    fl.listRec(new File(path), 0);
}
```

### 16.2 Wahlfreier Zugriff auf Dateien (RandomAccessFile)


Die Klasse verfügt u. a. über den folgenden Konstruktor:

```
RandomAccessFile(String file, String mode)
```

Der `mode`-String muss “r”, “rw” sein und gibt an, ob die Datei zum Lesen “r” oder zum Lesen und Schreiben “rw” geöffnet werden soll.


Die Methode `long getFilePointer()` liefert die aktuelle Position des Zeigers innerhalb der Datei. `void seek(long position)` bewegt den Zeiger zur angegeben Position.

Im Gegensatz zu `seek()` positioniert `int skipBytes(int n)` relativ zur aktuellen Position. `n` ist die Anzahl, um die der Dateizeiger bewegt wird. Ist `n` negativ, werden keine Bytes übersprungen. `int read()` liefert das Byte, auf das der Zeiger gerade zeigt, als `int`-Wert zurück. `int read(byte dest[])` versucht so viele Bytes zu lesen, so dass das Array gefüllt wird. Sie liefert die Anzahl der gelesenen Bytes bzw. -1, wenn der Zeiger auf das Ende der Datei zeigt. `int read(byte dest[], int offset, int len)` versucht len Bytes ab Position offset zu lesen und sie in dest zu schreiben.

Alle Methoden erhöhen die Zeigerposition um die Anzahl der gelesenen Bytes.
Die folgenden Schreibmethoden funktionieren analog zu ihren korrespondierenden Lesemethoden, geben jedoch keine Rückgabewerte zurück:
void read(int b), void write(byte dest[]), void write(byte dest[], int offset, int len).

Die folgenden Methoden lesen jeweils ein Element des angegebenen Typs und erwarten, dass es in der Datei in dem durch die korrespondierende write-Methode vorgegebenen binären Format vorliegt:

public final boolean readBoolean()
public final byte readByte()
public final char readChar()
public final double readDouble()
public final float readFloat()
public final int readInt()
public final long readLong()
public final short readShort()
public final String readUTF()
public final void readFully(byte[] b)
public final void readFully(byte[] b, int off, int len)
public final String readLine()
public final int readUnsignedByte()
public final int readUnsignedShort()

readUnsignedByte() Liest ein als vorzeichenlos interpretiertes Byte.
readUnsignedShort() Liest zwei als vorzeichenlos interpretierte Bytes.
readFully(byte[] b) Versucht, den gesamten Puffer b zu füllen.
void close() Schließt eine geöffnete Datei wieder.
Alle oben genannten Methoden werfen eine IOException.

Beispiel:

```java
package io;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Calendar;

/**
 * Das Programm öffnet eine Datei und fügt eine Zeichenkette an das Ende der Datei
 */
public class RandomAccessFileApplication {
    public void writeMySignature(String filename) {
        try {
            File file = new File(filename);
            RandomAccessFile randAccessFile = new RandomAccessFile(file, "rw");
            // Seek to end of file
        }
    }
}
```
randAccessFile.seek(file.length());

    // Append to the end
    randAccessFile.writeBytes("\r\n/*Signed by: "+ System.getProperty("user.name") + " at "+ Calendar.getInstance().getTime().toString() + "*/");
    randAccessFile.close();
}
}

public static void main(String[] args) {
    RandomAccessFileApplication randAccessFileApp = new RandomAccessFileApplication();
    randAccessFileApp
        .writeMySignature("src/io/RandomAccessFileApplication.java");
}

16.3 Streams


Ein Input-Stream (Eingabestrom) ist ein Objekt, das Bytes aus einer Datei liest und sie zurückgibt. Ein Output-Stream (Ausgabestrom) ist ein Objekt, mit dem Bytes in eine Datei sequentiell geschrieben werden können.


Während ByteStreams Bytes lesen und schreiben können, verwenden Character-Streams andere ByteStreams zum Lesen und Schreiben von Zeichen in Dateien

16.3.1 ByteStreams

ByteStreams werden auch Low-Level-Streams genannt, weil sie direkt aus Dateien lesen. Character-Streams dagegen können nicht direkt aus der Dateien lesen oder in sie schreiben. Character-Streams verwenden ByteStreams um schreiben oder lesen zu können.

Beispiel:

```java
import java.io.*;
public class FileCopy {

    public static void main(String[] args) {
        try {
            // Quell-Datei (path muss evtl. angepasst werden)
            FileInputStream in = new FileInputStream("src/io/FileCopy.java");
            // Ziel-Datei (path muss evtl. angepasst werden)
            FileOutputStream out = new FileOutputStream("src/io/CopyOfFileCopy.java");

            int len;
            while ((len = in.read()) > 0) {
                out.write(len);
            }
            out.close();
            in.close();
        } catch (IOException e) {
            System.err.println(e.toString());
        }
    }
}
```
16.3.2 FileReader

Ein FileReader-Objekt liest eine Datei zeichenweise:

```java
/*
 * Package: io
 * Class: CharacterFileReader.java
 * Method:
 */

public class CharacterFileReader {
    public static void main(String[] args) {
        FileReader f;
        int i;

        try {
            // Pfad muss evtl. angepasst werden
            f = new FileReader("src/io/Studenten.csv");
            while ((i = f.read()) != -1) {
                char c = (char) i;
                if (c == ';')
                    c = ' ';
                System.out.print(c);
            }
            f.close();
        }
        catch (IOException e) {
            System.out.println("Error while reading in the file");
        }
    }
}
```

16.3.3 BufferedReader

Dieser Filter dient zur Pufferung von Eingaben und kann verwendet werden, um die Performance beim Lesen von externen Dateien zu erhöhen. Da nicht jedes Byte einzeln gelesen wird, verringert sich die Anzahl der Zugriffe auf den externen Datenträger, und die Lesegeschwindigkeit erhöht sich. Zusätzlich stellt BufferedReader die Methode `readLine` zur Verfügung, die eine komplette Textzeile liest und als String an den Aufrufer zurückgibt:

```java
/*
 * Package: io
 * Class: LineCounter.java
 * Method:
 */

package io;

import java.io.File;
import java.io.IOException;
```
public class LineCounter {

public int countLines(String file) {
    int lines = 0;
    try {
        String line = "";
        while ((line = bufferedReader.readLine()) != null) {
            lines++;
        }
        File f = new File(file);
        System.out.println("Die Datei "+f.getAbsolutePath()+" enthält "+lines+" Zeilen!");
    } catch (IOException e) {
        System.out.println("Can't read File");
    }
    return lines;
}

    public static void main(String[] args) {
        // Pfad muss evtl. angepasst werden
        LineCounter lineCounter = new LineCounter();
        lineCounter.countLines("src/io/LineCounter.java");
    }
}

Character-Streams:

### 16.3.4 FileWriter

Ein FileWriter-Objekt schreibt eine Zeichenfolge in eine Datei.

Beispiel:

```java
/*
 * Package: io
 * Class: CharacterFileWriter.java
 * Method:
 */

package io;

import java.io.FileWriter;
import java.io.IOException;
/**
 * Das Programm erstellt eine CSV-Datei und fügt Studentendaten in sie ein
 */
public class CharacterFileWriter
{
    public static void main(String[] args)
    {

        try {
            String NEW_LINE="\r\n";
            String firstLine = "Vorname;Nachname;Mat-Nr.;Semesteranzahl"+NEW_LINE;
            String student1 = "Markus1;Mustermann1;346572;5"+NEW_LINE;
            String student2 = "Markus2;Mustermann2;23452132;3"+NEW_LINE;
            String student3 = "Markus3;Mustermann3;2209365;4"+NEW_LINE;
            FileWriter f1;
            //Pfad muss evtl. angepasst werden
            f1 = new FileWriter("src/io/Studenten.csv");
            f1.write(firstLine);
            f1.write(student1);
            f1.write(student2);
            f1.write(student3);
            f1.close();
        } catch (IOException e) {
            System.out.println("Error while writing in the file");
        }
    }
}

16.3.5 BufferedWriter


*/
* Package: io
* Class: CharacterFileWriter.java
* Method:
* */
package io;

/**
 * Das Programm fügt 15 Zeilen in eine Datei mittels eines BufferedReader ein.
 */

import java.io.BufferedWriter;
import java.io.FileWriter;

public class BufferedWriterApp {

    public static void main(String args[]) {
        if(!(args.length>0)){
            System.out.println("Das Programm erwartet einen Dateinamen als Eingabeparameter");
            System.exit(0);
        }
        try {
            int LINES=15;
            FileWriter fw = new FileWriter(args[0]);

            BufferedWriter bw = new BufferedWriter(fw);

            for(int i = 0; i < LINES; i++) {
                bw.write("Line " + i + \r\n");
            }
            bw.close();
        }
        catch(Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}
17 Applets


![Diagramm des Lebenszyklus von Applets](image)

Applets werden entweder von javax.swing.JApplet oder von ihrer Oberklasse java.applet.Applet abgeleitet. Der Lebenszyklus eines Applets ist durch die Methoden init(), start(), stop() und destroy() bestimmt. Dies sind die Methoden, die überschrieben werden sollen, wenn ein Java-Programm als Applet laufen soll.

init() wird genau einmal unmittelbar nach Aufruf des Konstruktors automatisch aufgerufen. Sie soll so überschrieben werden, dass sie Initialisierungsarbeiten vornimmt, z.B. Instanzvariablen initialisieren, Objekte erzeugen, die das Applet benötigt, Bilder und Schriften laden, Parameter einlesen etc.

start() wird unmittelbar nach dem Aufruf von init() aufgerufen und immer wieder, wenn das Applet wieder sichtbar wird. Dies ist genau der Fall, wenn der Benutzer zur HTML-Seite des Applets zurück kehrt, nachdem er andere Seiten besucht hat. Dies bewirkt, dass das Applet wieder aktiviert wird.
stop() wird immer aufgerufen, wenn der Benutzer die HTML-Seite des Applets verlässt und andere Seiten besucht. Dies ist genau der Fall, wenn das Applet unsichtbar wird. Diese **Methode** wird typischerweise überschrieben, um alle Aktionen zu beenden, die nicht mehr benutzt werden, wenn das Applet unsichtbar wird, und dient dazu, die CPU nicht unnötig zu belasten. So können gestartete Animationen, Music, Threads usw. gestoppt werden.

destroy() wird nur einmal aufgerufen, wenn der Benutzer das Browser-Fenster schließt, um laufende Threads zu beenden oder belegten Speicherplatz wieder freizugeben, kurz gesagt, um hinter dem Applet "aufzuräumen".

Ein Applet wird während seiner Lebensdauer genau einmal geladen, instanziert und initialisiert (init()), einmal oder mehrere Male gestartet (start()) oder gestoppt (stopp()) und genau einmal „vernichtet“ (destroy()).

Alle genannten Methoden sollen entsprechend überschrieben werden.
 Aus Sicherheitsgründen darf ein Applet in der Regel weder auf Dateien des lokalen Rechners zugreifen noch externe Programme auf diesem starten.

Die Kontrolle der Sicherheit eines Applets ist im Browser implementiert, d.h. es ist auch vom Browser abhängig, welchen Sicherheitsbeschränkungen ein Applet unterliegt. Browser verhindern standardmäßig Folgendes:

- Ausführen eines Programms auf dem Rechner des Benutzers außerhalb der JVM
- Dateizugriff auf dem Rechner des Benutzers (weder Lese- noch Schreibrechte)

Diese Einschränkungen verhindern, dass ein Applet die Datensicherheit verletzt oder das System des Benutzers beschädigt. Die Beschränkung, dass ein Applet keine neuen Verbindungen aufbauen kann, verhindert die Kommunikation mit anderen Programmen, die auf nicht vertrauenswürdigen Rechnern liegen.

Um ein Applet zu schreiben, muss zunächst eine Klasse der folgenden Form erstellt werden:

```java
import java.applet.*;

public class HelloWorld extends Applet {
    ...
}
```
Dabei muss die Applet-Klasse public sein und eine Subklasse der Klasse java.applet.Applet sein.
public class TickerApplet extends JApplet implements Runnable {

    // Die x-Koordinate der aktuellen Textposition
    private int xPos;
    // Die y-Koordinate der Textposition
    private int yPos;
    // Breite des Text-Panners
    private int pannerWidth;
    private Thread thread;
    // Die Laufschrift
    private String text;

    public TickerApplet() {
    }

    @Override
    public void destroy() {
        thread = null;
    }

    @Override
    public void init() {
        // Die Schrift soll in der Mitte des Applets dargestellt werden
        yPos = getSize().height / 2;

        // Das Applet erwartet einen Textparameter namens text
        text = getParameter("text");
        if (text == null)
            text = "Das Applet erwartet einen Textparameter namens Text";

        pannerWidth = getSize().width;
        xPos = Math.min(xPos, pannerWidth);
    }

    @Override
    public void start() {
        thread = new Thread(this);
        thread.start();
        System.out.println("start");
    }

    @Override
    public void stop() {
        thread.stop(); // Die Methode Thread.stop() ist deprecated (wurde verworfen) und sollte in
        // einer richtigen Anwendung nicht verwendet werden!
        // Hinweise wie ein Thread gestoppt werden soll unter:
        // http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html
        thread = null;
        System.out.println("stop");
    }

    @Override
    public void paint(Graphics g) {
        super.paint(g);
        g.drawString(text, xPos, yPos);
    }
}
Java-Programm als Applet und Applikation

Beispiel:
Methoden zum Zeichnen von Graphikprimitiven:


Die Methode public void update(Graphics g) wird von repaint() aufgerufen. Die Standard-update()-Methode löschst den vollständigen Zeichenbereich des Applets bzw. der grafischen Oberfläche einer Applikation (oft ruft das den unangenehmen Flimmereffekt bei schnellen Bildsequenzen hervor) und ruft anschließend die paint()-Methode auf, welche dann das Applet vollständig neu zeichnet. Zum Löschen des Inhaltes des Zeichenbereiches setzt die update-Methode die Farbe jedes Pixels gleich der Hintergrundfarbe. Es ist also genau genommen so, dass nicht die repaint()-Methode direkt die paint()-Methode aufruft, sondern nur indirekt über die update()-Methode.

Graphics-Klasse:

java.awt.Graphics ist eine abstrakte Klasse die zahlreiche Methoden zum Zeichnen von Graphikprimitiven zur Verfügung stellt.


Die Koordinaten im Applet-Fenster werden in der Form (x,y) angegeben. x ist die horizontale Koordinate, y die vertikale. Der Nullpunkt (0,0) befindet sich oben links in der Ecke des Zeichenbereichs. Die Maßeinheit entspricht einem Bildschirmpixel und ist damit geräteabhängig.
package applets;

import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Polygon;

public class Painter extends Applet {

    public void paint(Graphics g) {

        // Erzeugen von Farbobjekten
        Color farbe1 = new Color(255, 100, 92);
        Color farbe2 = new Color(155, 192, 192);
        Color farbe3 = new Color(55, 192, 192);
        Color farbe4 = new Color(5, 12, 120);
        Color farbe5 = new Color(255, 192, 192);

        g.setColor(farbe1);
        g.fillRect(5, 5, 150, 250);

        // Zeichne eine Ellipse
    }
}
g.setColor(farbe2);
g.fillOval(50, 15, 250, 150);
// Zeichne einen Kreis
g.setColor(farbe3);
g.fillOval(250, 150, 150, 150);
// Zeichne einen gefüllten Bogen
g.setColor(farbe4);
g.fillArc(50, 15, 100, 100, 90, 90);
// Zeichne einen weiteren gefüllten Bogen in einer anderen Farbe
g.setColor(farbe5);
g.fillArc(150, 150, 50, 50, 120, 180);
// direkte Verwendung einer Farbkonstanten
g.setColor(Color.green);
g.drawRect(40,40,100,200);   }
}

package applets;

import java.applet.Applet;
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;

/**
 * @author baker
 *
 */
public class DrawColoredStrings extends Applet{
    
    public void paint(Graphics g) {
        
        Font schrift1 = new Font("TimesRoman", Font.PLAIN, 18);
        Font schrift2 = new Font("TimesRoman", Font.BOLD, 14);
        
        }
    }
Font schrift3 = new Font("TimesRoman", Font.ITALIC, 22);
Font schrift4 = new Font("TimesRoman", Font.BOLD + Font.ITALIC, 24);
Font schrift5 = new Font("Arial", Font.PLAIN, 10);
Font schrift6 = new Font("Courier", Font.BOLD, 12);
Font schrift7 = new Font("Arial", Font.ITALIC, 34);
Color farbe1 = new Color(255, 100, 92);
g.setFont(schrift1);
g.drawString("Normaler (plain) Font - TimesRoman", 10, 25);
g.setFont(schrift2);
g.drawString("Fetter (bold) Font - TimesRoman", 10, 50);
g.setFont(schrift3);
g.drawString("Kursiver (italic) Font - TimesRoman", 10, 75);
g.setFont(schrift4);
g.drawString("Fett und kursiv (bold italic) - TimesRoman", 10, 100);
g.setFont(schrift5);
g.drawString("Normaler (plain) Font - Arial", 10, 125);
g.setFont(schrift6);
g.drawString("Fetter Font - Courier", 10, 150);
g.setColor(farbe1);
g.setFont(schrift7);
g.drawString("Farbig, groß und kursiv - Arial", 10, 200);
18 AWT/Swing

Sind Java-API zur Erstellung von graphischen Benutzeroberflächen (GUI) wie Fenster, Buttons, Textfelder, Registerkarten, etc. und zur Zeichnung von graphischen Primitiven.

AWT ist stark abhängig vom darunterliegenden System und mittlerweile veraltet. Swing ist Erweiterung des AWT und plattformunabhängig.