



- Java Skriptum 2-

Java 2.Jahrgang

Verfasser: Mag. Otto Reichel
HTBLuVA St.Pölten
Abteilung Informatik

Aktuelle Version: 13. Januar 2015

Inhaltsverzeichnis

Inhaltsverzeichnis	i
1 Einführung	1
1.1 Was ist Java?	1
1.1.1 Designmerkmale von Java	1
1.1.2 Eigenschaften der Java Virtual Machine	1
1.1.3 Das Java Development Kit - JDK	2
1.2 Möglichkeiten von Java	2
1.2.1 Konsolenapplikation	2
1.2.2 Graphische Applikation	2
1.2.3 Applet	3
1.2.4 Servlet	3
2 Grundlagen	4
2.1 Sprachelemente	4
2.1.1 Kommentare	4
2.1.2 Quelldateien	4
2.1.3 Reservierte Worte und Bezeichner	5
2.2 Einfache Datentypen	5
2.2.1 Ganzzahlige Datentypen	6
2.2.2 Der Datentyp <code>char</code>	6
2.2.3 Die Fließkommatypen <code>float</code> und <code>double</code>	6
2.2.4 Der logische Typ <code>boolean</code>	7
2.3 Referenztypen	7
2.4 Arrays	7
2.4.1 Erzeugung eines Arrays	7
2.5 Lauffähige Klassen	8
2.5.1 Die Startmethode <code>main()</code>	9
2.6 Variable und ihre Initialisierung	9
2.7 Methoden	10
2.7.1 Übergabe von Argumenten	10
2.7.2 Varargs	11
2.7.3 Methodenüberladung	12
2.8 Garbage Collection	13
3 Sprachsyntax	14
3.1 Operatoren	14
3.1.1 Übersicht und Vorrangregeln	14
3.1.2 Typkonversion und Castoperator	15
3.1.3 Inkrement- und Dekrementoperator	16
3.1.4 Die arithmetischen Operatoren	16
3.1.5 Die logischen Operatoren	17
3.1.6 Die relationalen Operatoren	17
3.1.7 Der Bedingungsoperator	18
3.1.8 Die Bitoperatoren	18
3.2 Kontrollstrukturen	19
3.2.1 Abfragen	20

3.2.2	Schleifen	22
4	Objektorientierte Programmierung	26
4.1	Grundlagen	26
4.1.1	Idee der objektorientierten Programmierung	26
4.1.2	Eigenschaften einer objektorientierten Sprache	26
4.2	Klassen und Objekte	27
4.2.1	Klassen	27
4.2.2	Paketzugehörigkeit	28
4.2.3	Access Modifier	28
4.2.4	Zugriffsfunktionen, Datenkapselung	28
4.2.5	Objekte	29
4.2.6	Instanzmethoden und die <code>this</code> -Referenz	29
4.2.7	Konstruktoren und Destruktor	30
4.2.8	Variable als Klassenmitglieder	31
4.2.9	Methoden als Klassenmitglieder	32
4.2.10	Statische Initialisierer	33
4.2.11	UML-Klassendiagramme	34
4.3	Vererbung	36
4.3.1	Grundlegende Konzepte	36
4.3.2	Ableiten einer Klasse, die "is-a"-Beziehung	36
4.3.3	Konstruktoren und Vererbung	40
4.3.4	Instanzinitialisierer	41
4.3.5	Function Overriding	42
4.3.6	Polymorphie	43
4.4	Abstrakte Methoden und abstrakte Klassen	45
4.4.1	Abstrakte Methoden	45
4.4.2	Abstrakte Klassen	45
4.5	Interfaces	47
5	Exceptionhandling	52
5.1	Grundlagen	52
5.2	Fehlerklassen	52
5.2.1	Die Klasse <code>java.lang.Throwable</code>	52
5.2.2	Die Klasse <code>java.lang.Exception</code>	53
5.2.3	Die Klasse <code>java.lang.RuntimeException</code>	53
5.2.4	Die Klasse <code>java.lang.Error</code>	54
5.2.5	Eigene Exceptionklassen	54
5.3	Auslösen von Ausnahmen	54
5.4	Die <code>try-catch</code> - Anweisung	55
5.4.1	Syntax und einfache Anwendung	55
5.4.2	Mehrere <code>catch</code> - Klauseln	56
5.4.3	Die <code>finally</code> - Klausel	57
5.4.4	Weitergabe von Exceptions	57
6	Grundlegende Klassen	58
6.1	Die Klasse <code>java.lang.Object</code>	58
6.1.1	Die Methode <code>toString()</code>	58
6.1.2	Die Methode <code>equals()</code>	59
6.2	Die Klasse <code>java.lang.String</code>	60
6.2.1	Wichtige Methoden der Klasse <code>java.lang.String</code>	60
6.2.2	Stringpool, Vergleichen von Strings	62
6.2.3	Stringverkettung	63
6.3	Die Klassen <code>StringBuffer</code> und <code>StringBuilder</code>	63
6.3.1	Wichtige Methoden	63
6.3.2	Eigenschaften	64
6.3.3	Beispiel	65
6.4	Die Klasse <code>java.util.Scanner</code>	65
6.4.1	Eingabe von <code>System.in</code>	65

6.4.2	Lesen aus einer Textdatei	67
6.5	Die Wrapperklassen	68
6.5.1	Konstruktion von Wrapperobjekten	68
6.5.2	Vergleich von Wrapperobjekten	69
6.5.3	Auslesen der gespeicherten Werte	69
7	Ergänzungen und Zertifizierungswissen	70
7.1	Assertions	70
7.1.1	Einführung	70
7.1.2	Die <code>assert</code> -Anweisung	70
7.1.3	Verwendung von Assertions	70
7.1.4	Aktivieren und Deaktivieren von Assertions	71
7.2	Innere Klassen	72
7.2.1	Top-Level Nested Classes und Interfaces	72
7.2.2	Nicht statische innere Klassen	72
7.2.3	Lokale Klassen	73
7.2.4	Anonyme innere Klassen	74
7.3	Autoboxing und Unboxing	74
7.3.1	Autoboxing/Unboxing bei Funktionsparametern	75
7.3.2	Autoboxing/Unboxing bei Funktionsüberladung	76
7.3.3	Autoboxing/Unboxing bei Vergleichen	76
7.4	Enums	77
7.4.1	Konzept	77
7.4.2	Definition einer Enum	78
7.4.3	Enums und Switch-Case	80
7.5	Generics	80
8	Erstellung graphischer Benutzerschnittstellen	83
8.1	AWT und Swing	83
8.1.1	Überblick Swing	83
8.2	Grundlegende Komponenten und Klassen	84
8.2.1	Die Klasse <code>java.awt.Component</code>	84
8.2.2	Die Klasse <code>java.awt.Container</code>	85
8.2.3	Die Klasse <code>javax.swing.JComponent</code>	85
8.2.4	Die Klasse <code>javax.swing.JPanel</code>	85
8.2.5	Die Klasse <code>java.awt.Window</code>	85
8.2.6	Die Klasse <code>java.awt.Frame</code>	85
8.2.7	Die Klasse <code>javax.swing.JFrame</code>	85
8.2.8	Die Klasse <code>java.awt.Dialog</code>	85
8.2.9	Die Klasse <code>javax.swing.JDialog</code>	85
8.2.10	Die Klasse <code>java.awt.Panel</code>	85
8.2.11	Die Klasse <code>java.applet.Applet</code>	86
8.2.12	Die Klasse <code>javax.swing.JApplet</code>	86
8.3	Wichtige Klassen aus <code>java.awt</code> und <code>javax.swing</code>	86
8.3.1	Grundgerüst für eine Swingapplikation	86
8.3.2	Die Klasse <code>java.awt.Color</code>	87
8.3.3	Die Klasse <code>java.awt.Font</code>	88
8.3.4	Die Klasse <code>java.awt.Point</code>	89
8.3.5	Die Klasse <code>java.awt.Dimension</code>	89
8.3.6	Die Klasse <code>java.awt.Rectangle</code>	90
8.3.7	Die Klasse <code>java.awt.Polygon</code>	90
8.3.8	Die Klasse <code>java.awt.Component</code>	90
8.3.9	Die Klasse <code>javax.swing.JComponent</code>	92
8.4	LayoutManager	92
8.4.1	Einführung	92
8.4.2	Container	93
8.4.3	Spezielle Layoutmanager	94
8.4.4	Verschachteln von Layoutmanagern	98
8.5	Eventhandling	99

8.5.1	Einführung	99
8.5.2	Eventklassen	100
8.5.3	Ereignisempfänger	100
8.5.4	Konkrete Implementierungsmöglichkeiten	103
8.6	Zeichnen mit <code>java.awt.Graphics</code>	109
8.6.1	Verwendung von Farben	109
8.6.2	Darstellung von Text	110
8.6.3	Zeichnen von Linien	110
8.6.4	Zeichnen von Rechtecken	110
8.6.5	Zeichnen von Kreisen und Ellipsen	111
8.6.6	Zeichnen von Polygonen	111
8.7	Zeichnen mit <code>java.awt.Graphics2D</code>	113
8.7.1	Zeichnen von Objekten	113
8.7.2	Linien	113
8.7.3	Rechtecke	113
8.7.4	Ellipsen	113
8.7.5	Bögen	114
8.7.6	Quadratische Kurven	115
8.7.7	Kubische Kurven	115
8.7.8	Konstruktive Flächengeometrie	116
8.7.9	Pfade	118
8.7.10	Affine Transformationen	119
8.7.11	Darstellungsattribute festlegen	123
8.7.12	Animationen	125
8.8	Das Model-View-Controller-Konzept (MVC)	130
8.8.1	Das Model-View-Controller Konzept	130
8.8.2	Vordefinierte Datenmodelle	132
8.9	Verwendung einer <code>JList</code>	133
8.9.1	Die Klasse <code>DefaultListModel<E></code>	133
8.9.2	Die Klasse <code>AbstractListModel</code>	133
8.9.3	Rendern einer <code>JList</code>	135
8.10	Verwendung einer <code>JTable</code>	137
8.10.1	Das Interface <code>TableModel</code>	137
8.10.2	Die Klasse <code>AbstractTableModel</code>	138
8.10.3	Editieren von Zellen	140
8.10.4	Rendern einer <code>JTable</code>	141
8.10.5	Abschließendes Beispiel	142
8.11	Dialoge	150
8.11.1	Template für einen OK - Cancel - Dialog	151
8.11.2	Beispiel	152
	Abbildungsverzeichnis	157
	Verzeichnis der Listings	158

Kapitel 1

Einführung

1.1 Was ist Java?

Eine vollständig objektorientierte Sprache ohne freie Funktionen, die in den frühen 90er-Jahren von Sun Microsystems entwickelt wurde. Sun beschreibt die Sprache folgendermaßen:

A simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multi-threaded and dynamic language.

1.1.1 Designmerkmale von Java

Plattformunabhängig

- Javacode wird zu plattformunabhängigen Bytecode compiliert:

`Myclass.java` → `Myclass.class`

- Der Bytecode wird dann plattformabhängig von der Java Virtual Machine JVM interpretiert.

Robust

- Voll objektorientiert: jede Zeile Code gehört zu einer Klasse
- Strenge Typprüfung
- Eingebautes und erzwungenes Exception-Handling
- Keine Pointer

Klein und schnell

- Jede Klasse wird nur bei Bedarf geladen
- Eingebautes Multithreading

Sicher

- Die Laufzeitumgebung kann Klassen in ihrer Funktionalität einschränken

1.1.2 Eigenschaften der Java Virtual Machine

- Der Class Loader lädt die erforderlichen Klassen (gemäß dem Suchpfad CLASSPATH)
- Der Verifier überprüft Klassen auf korrekte Verwendung (gültiger Java-Bytecode, keine ungültigen Register etc.) sowie Sicherheitsbeschränkungen bei File- und Netzzugriffen
- Garbage Collector: Der Finalizer-Thread gibt alle nicht referenzierten Speicherbereiche automatisch wieder frei

1.1.3 Das Java Development Kit - JDK

Das JDK liegt aktuell in der Version 1.8 vor, ist frei verfügbar, plattformabhängig und besteht unter anderem aus folgenden Werkzeugen:

- `javac` - Java Compiler
- `java` - Java Interpreter (VM)
- `appletviewer` - Tool zum Testen von Java Applets
- `jdb` - Java Debugger
- `javadoc` - erstellt HTML-Dokumentationen von *.java Dateien
- `jar` - Archivierungswerkzeug
- `javap` - Dissasembler

1.2 Möglichkeiten von Java

Mit Java lassen sich u.a. Konsolenapplikationen, graphische Applikationen, Applets und Servlets entwickeln. Die folgenden Listings geben zu jeder dieser Möglichkeiten je ein Begrüßungsprogramm an.

1.2.1 Konsolenapplikation

```
1 public class HelloKonsole {  
2     public static void main(String args[]) {  
3         System.out.println("Hello Java");  
4     }  
5 }
```

Listing 1.1: Konsolenapplikation

1.2.2 Graphische Applikation

```
1 import java.awt.*;  
2 import javax.swing.*;  
3  
4 public class HelloGUI extends JFrame {  
5     public static void main(String args[]) {  
6         new HelloGUI().setVisible(true);  
7     }  
8  
9     public HelloGUI() {  
10         super("Hello");  
11         this.setBounds(200,100,300,150);  
12         this.getRootPane().setOpaque(false);  
13         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
14     }  
15  
16     public void paint(Graphics g){  
17         g.drawString("Hello Java", 40, 40);  
18     }  
19 }
```

Listing 1.2: Graphische Applikation

1.2.3 Applet

```
1 import java.applet.Applet;
2 import java.awt.*;
3
4 public class HelloApplet extends Applet {
5     public void paint(Graphics g) {
6         g.drawString("Hello Java", 20,20);
7     }
8 }
```

Listing 1.3: Applet

1.2.4 Servlet

```
1 import javax.servlet.*;
2 import java.io.*;
3
4 public class HelloServlet extends GenericServlet {
5     public void service(ServletRequest req, ServletResponse res) throws ServletException,
6         IOException {
7         res.setContentType("text/html");
7         PrintWriter out = res.getWriter();
8         out.println("<head>");
9         out.println("<body>");
10        out.println("<h1>Hello World</h1>");
11        out.println("</body>");
12        out.println("</html>");
13    }
14 }
```

Listing 1.4: Servlet

Kapitel 2

Grundlagen

2.1 Sprachelemente

2.1.1 Kommentare

Es gibt in Java drei Arten von Kommentaren:

- Einzeilige Kommentare werden mit `//` eingeleitet und enden mit der aktuellen Zeile.
- Mehrzeilige Kommentare beginnen mit `/*` und enden mit `*/`. Sie können sich über mehrere Zeilen erstrecken.
- Dokumentationskommentare beginnen mit `/**` und enden mit `*/`. Sie dienen dazu, Programme im Quelltext zu dokumentieren. Mit Hilfe des Tools `javadoc` können sie aus der Quelle extrahiert und in ein HTML-Dokument umgewandelt werden. Beispiel:

```
1  /**
2   * Compares two Objects for equality.
3   * Returns a boolean that indicates whether this Object
4   * is equivalent to the specified Object. This method is
5   * used when an Object is stored in a hashtable.
6   * @param  obj      the Object to compare with
7   * @return          true if these Objects are equal;
8   *                 false otherwise.
9   * @see            java.util.Hashtable
10  */
11
12  public boolean equals(Object obj) {
13      return this == obj;
14  }
```

Listing 2.1: Dokumentationskommentar

Dokumentationskommentare stehen immer vor dem Element, das sie beschreiben, etwa wie oben die Methode `equals()` der Klasse `java.lang.Object`. Der erste Satz ist die Überschrift, dann folgt eine längere Beschreibung der Funktionalität. Die durch `@` eingeleiteten Elemente sind Makros, die eine besondere Bedeutung haben.

`@param` spezifiziert Methodenparameter, `@return` den Rückgabewert und `@see` einen Verweis. Daneben gibt es noch die Makros `@exception`, `@version` und `@author`.

2.1.2 Quelldateien

Alle Quelldateien enden mit der Erweiterung `.java`. Enthält eine Quelldatei eine öffentliche Toplevel-Klasse, so muss sie ohne Erweiterung den gleichen Namen besitzen wie die mit `public` definierte Toplevel-Klasse. Zwei oder mehr Toplevel-Klassen mit öffentlichem Zugriff darf es in einer Java-Quelldatei nicht geben.

In einer Java-Quelldatei können folgende Elemente auftreten:

- Packagedeklaration

- Importstatements
- Klassen- und Interfacedefinitionen, Enums

Diese Toplevel-elemente müssen genau in der obigen Reihenfolge auftreten, wobei alle optional sind (die kleinste gültige Java-Quelldatei ist die leere Datei) und die Packagedeklaration eindeutig sein muss. Kommentare dürfen überall stehen. Beispiel für eine gültige Quelldatei:

```

1  /* TestClass.java */
2
3  package mypack; // Package-Deklaration
4
5  import java.awt.Button; // Import einer speziellen Klasse
6  import java.util.*; // Import eines ganzen Paketes
7
8  // Öffentliche Klasse - bestimmt den Namen der Quelldatei
9  public class TestClass {
10     // Mitglieder der Klasse
11 }
12
13 // Nicht öffentliche Klasse
14 class A {
15     // Mitglieder der Klasse
16 }

```

Listing 2.2: Aufbau einer Quelldatei

2.1.3 Reservierte Worte und Bezeichner

Reservierte Worte

Java kennt in der Version 1.5 53 Reservierte Worte:

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	false	final	finally
float	for	goto	if	implements
import	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	short	static
strictfp	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while		

`assert` ist neu seit Version 1.4, `enum` ist neu seit Version 1.5.
`goto` und `const` sind reserviert, obwohl sie keine Bedeutung haben.

Bezeichner

Ein Bezeichner wird verwendet, um den Namen einer Variablen, einer Methode, einer Klasse oder einer Sprungmarke anzugeben. Das erste Zeichen eines Bezeichners muss ein Buchstabe, ein Unterstrich oder das Dollarzeichen sein. In weiterer Folge können neben diesen Zeichen auch die Ziffern '0' bis '9' verwendet werden. Alle Zeichen sind signifikant und case sensitiv. Es dürfen keine reservierten Worte verwendet werden.

2.2 Einfache Datentypen

Java kennt 8 einfache Datentypen: 4 ganzzahlige Typen, 2 Fließkommatypen, einen Charaktertyp sowie einen logischen Typ:

Typ	Größe in Bit	Standardwert	Typ	Größe in Bit	Standardwert
byte	8	0	float	32	0.0f
short	16	0	double	64	0.0
int	32	0	char	16	\u0000
long	64	0L	boolean	1	false

2.2.1 Ganzzahlige Datentypen

Die 4 ganzzahligen Datentypen `byte`, `short`, `int` und `long` sind vorzeichenbehaftet. Sie sind für jede Umgebung mit fixen Größen erklärt, wodurch die Verwendung eines `sizeof`-Operators wie etwa in C überflüssig wird. Die folgende Tabelle zeigt diese fixen Größen:

Typ	Größe in Bit	Minimum	Maximum
byte	8	$-2^7 = -128$	$2^7 - 1 = 127$
short	16	$-2^{15} = -32768$	$2^{15} - 1 = 32767$
int	32	$-2^{31} = -2147483648$	$2^{31} - 1 = 2147483647$
long	64	$-2^{63} \approx -9,2 \cdot 10^{18}$	$2^{63} - 1 \approx 9,2 \cdot 10^{18}$

Ganzzahlige Literale

Dezimale, oktale (Präfix `0`) und hexadezimale Darstellung (Präfixe `0x`, `0X`) ist möglich. Ganzzahlige Literale sind grundsätzlich vom Typ `int` (32 bit). Das Suffix `L` oder `l` spezifiziert den Typ `long` (64 bit). Ganzzahlige Literale vom Typ `byte` oder `short` sind nicht möglich.

Dezimal:	0	42	-23795
Oktal:	0777	02	0126
Hexadezimal:	0x01	0Xcafe	0x12h
Long:	3L	077L	0x1010L

2.2.2 Der Datentyp char

In Java ist ein Charakter 16 Bit groß und dient zur Speicherung von Unicodezeichen. Der Unicode-Zeichensatz fasst eine große Anzahl internationaler Zeichensätze zusammen. Der Unicoesatz ist mit den ersten 128 Zeichen des ASCII-Zeichensatzes und mit den ersten 256 Zeichen des ISO-8859-1-Zeichensatzes kompatibel.

Charakter-Literale

`char`-Literale werden grundsätzlich in einfache Hochkomma gesetzt. Um auch nicht über die Tastatur erreichbare Unicodezeichen verarbeiten zu können, ist auch die Darstellung durch `\uxxxx` möglich, wobei `xxxx` eine notwendigerweise 4-stellige Hexzahl sein muss:

```
char c='w';
char cl = '\uCAFE';
```

Analog zu C stellt Java eine Reihe von Escapesequenzen zur Darstellung von Sonderzeichen zur Verfügung:

'\b'	Backspace	'\t'	horizontaler Tabulator
'\n'	Zeilenschaltung	'\f'	Formularvorschub
'\r'	Wagenrücklauf	'\''	einfaches Hochkomma
'\"'	doppeltes Hochkomma	'\\'	Backslash

2.2.3 Die Fließkommatypen float und double

Java kennt die beiden Fließkommatypen nach IEEE-754 `float` (4 Byte) und `double` (8 Byte).

Fließkommaliterale

Fließkommaliterale beinhalten entweder einen Dezimalpunkt, einen Exponenten oder ein Suffix (`f`, `F` für `float` bzw. `d`, `D` für `double`).

Ein Fließkommaliteral ohne Suffix ist standardmäßig vom Typ `double`.

```
float:    1f    42f    1.22e19F
double:   .47    5D     1.3E-16
```

2.2.4 Der logische Typ `boolean`

Mit `boolean` besitzt Java einen logischen Typ, der nur die Werte `true` und `false` annehmen kann:

```
boolean isRed    = true;
boolean isBlue   = false;
```

2.3 Referenztypen

Referenztypen stellen neben den einfachen Datentypen die zweite wichtige Gruppe von Variablen in Java dar. Eine Referenz ist ein Aliasname für ein Javaobjekt. Verweist eine Referenz momentan auf kein Objekt, sollte sie mit der Konstanten `null` belegt werden. Referenzen sind auch Aliasnamen für Arrays und Strings, die aber eine Sonderstellung unter den Javaobjekten aufweisen:

- Arrays sind klassenlose Objekte, d.h. sie besitzen keine explizite Klassendefinition. Dennoch werden sie vom Laufzeitsystem (der VM) wie normale Objekte behandelt.
- Die Klasse `java.lang.String` ist zwar eine gewöhnliche Klasse in der Laufzeitbibliothek von Java, der Compiler hat aber Kenntnisse über den inneren Aufbau dieser Klasse und generiert bei Stringoperationen automatisch Code, der auf Methoden der Klasse `java.lang.String` zugreift.

Stringliterale

Ein Stringliteral ist eine Sequenz von Unicodezeichen in doppelten Hochkommas, zum Beispiel:

```
String s = "Java 1.5. hat den Codenamen Tiger, 1.6 Mustang";
s ist eine Referenz, die einen Aliasnamen für das Stringliteral darstellt.
```

2.4 Arrays

Arrays in Java sind geordnete Sammlungen primitiver Variablen, Objektreferenzen oder anderer Arrays¹. Sie sind homogen, d.h. alle Elemente eines Arrays müssen den selben Typ aufweisen. Sie unterscheiden sich von Arrays in vielen anderen Sprachen durch die Tatsache, dass sie Objekte sind. Obwohl dieser Umstand in vielen Fällen vernachlässigt werden kann, bedeutet er konkret:

- Arrayvariablen sind Referenzen
- Arrays besitzen Methoden und Instanzvariable
- Arrays werden dynamisch zur Laufzeit erzeugt

2.4.1 Erzeugung eines Arrays

Die Erzeugung eines Arrays erfordert 3 Schritte:

- (1) Deklaration
- (2) Konstruktion
- (3) Initialisierung

¹Mehrdimensionale Arrays

Deklaration

Bei der Deklaration legt man den Namen und den Datentyp des Arrays fest:

```
int []myInts;      // Feld myInts vom Typ int
String []s;        // Feld s von Stringreferenzen
float [][]twoDim;  // Zweidimensionales Feld twoDim vom Typ float
```

Die obigen Zeilen deklarieren jeweils eine Feldreferenz, legen aber noch keine Größe fest (eine Größenangabe an dieser Stelle führt stets auf einen Compilerfehler) und damit auch keinen Speicherplatz für das Feld an. Dies geschieht bei der Konstruktion.

Konstruktion und Initialisierung

Die Größe eines Arrays wird erst zur Laufzeit benötigt, wenn die VM Speicherplatz für das Feld bereitstellt. Dies geschieht mit dem Schlüsselwort `new`:

```
int size = 1000;      // primitive Integervariable
int [] f1, f2;         // Deklaration
f1 = new int[size];    // Konstruktion und Initialisierung
f2 = new int[]{1,2,3,4,5}; // Konstruktion und Initialisierung
```

Bei der Konstruktion wird das Array am Heap erzeugt und wenn nicht anders angegeben mit Standardwerten gefüllt. Es gibt in Java keine uninitialisierten Objekte und damit auch keine uninitialisierten Arrays. Standardwerte sind `0`, `0f` bzw. `0.0` bei numerischen Arrays, `false` bei Arrays vom Typ `boolean` und `null` bei Arrays von Referenzen.

Deklaration und Konstruktion sind auch in einer Befehlszeile möglich:

```
int [] f1 = new int[2000];
int [] f2 = {1,2,3,4,5};
```

Hier kann wie bei `f2` bei gleichzeitiger Initialisierung des Arrays die Anwendung des Operators `new` auch entfallen. Die Feldgröße wird dabei durch die Anzahl der Initialisierungswerte in den geschweiften Klammern bestimmt.

Zugriff auf Feldelemente

Der Zugriff auf die einzelnen Elemente eines Arrays erfolgt mit dem Subskriptoperator `[index]`, wobei die Indizes bei einem Array mit `n` Elementen von `0` bis `n-1` laufen. Der Index muss vom Typ `int` sein. Jedes Array speichert seine Länge in einer Konstanten `length`. Daraus folgt, dass die Größe eines Arrays nach der Konstruktion nicht mehr veränderbar ist. Indexwerte werden vom Laufzeitsystem auf Einhaltung der Feldgrenzen geprüft, sie müssen also größer gleich `0` und kleiner als `length` sein. Bei falschem Indexwert wirft die VM eine `IndexOutOfBoundsException`.

Das folgende Beispiel demonstriert die Konstruktion und Initialisierung eines Arrays:

```
1 long []squares;
2 squares = new long[60000];
3 for(int i = 0; i < squares.length; i++)
4     squares[i] = i * i;
```

Listing 2.3: Verwendung eines Arrays

2.5 Lauffähige Klassen

Das wichtigste Sprachelement von Java ist die Klasse. Klassen werden später im Rahmen der OOP² ausführlich behandelt, in diesem Abschnitt werden Klassen nur als Container für die Startmethode `main()` einer Applikation verwendet.

²Objektorientierte Programmierung

2.5.1 Die Startmethode `main()`

Diese Methode ist der Einsprungpunkt einer Javaapplikation. Zum Erstellen einer Applikation definiert man eine Klasse, die eine solche `main()`-Methode enthält:

```
1 public class Hello {  
2     public static void main(String args[]) {  
3         System.out.println("Hello Java");  
4     }  
5 }
```

Listing 2.4: Startmethode `main()`

Folgende Schritte sind notwendig, um die Applikation zu compilieren und laufen zu lassen:

- (1) Speichern unter `Hello.java`
- (2) Mit `javac Hello.java` zu `Hello.class` compilieren
- (3) Mit `java Hello` starten

Die Startmethode `main()` muss notwendigerweise öffentlich und statisch sein. Die Modifier `public` und `static` sind also notwendig³. Ebenso sind der Rückgabetyp `void` sowie das Array von Stringreferenzen als Parameter zwingend vorgeschrieben. Die Startmethode `main()` hat also folgende Schnittstelle:

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

Kommandozeilenparameter

Die vom Aufrufer der Applikation übergebenen Kommandozeilenparameter werden über das Array `args` angesprochen.

Beispiel:

```
1 public class Params {  
2     public static void main(String []args) {  
3         for(int i = 0; i < args.length; i++)  
4             System.out.println(args[i]);  
5     }  
6 }
```

Listing 2.5: Kommandozeilenparameter

Beispiel für einen Aufruf:

```
java Params gestern heute morgen  
gestern  
heute  
morgen
```

Zu beachten: Weder `java` noch der Klassenname `Params` werden über das Feld `args` gespeichert. Der Name des Feldes ist natürlich im Rahmen gültiger Bezeichner beliebig wählbar.

2.6 Variable und ihre Initialisierung

Variable dienen dazu, Daten im Hauptspeicher eines Programms abzulegen und gegebenenfalls zu lesen oder zu verändern. In Java gibt es aus Sicht ihrer Lebensdauer und ihres Speicherortes drei Typen von Variablen:

- **Instanzvariable**

Diese werden auf Klassenebene ohne das Schlüsselwort `static` deklariert. Ihre Lebensdauer ist an die des zugehörigen Objekts gebunden, d.h. sie werden mit einem Objekt erzeugt und mit diesem auch wieder zerstört. Instanzvariablen werden mit Standardwerten vorbelegt. Sie sind im Objekt gespeichert und liegen daher wie das gesamte Objekt am Heap.

³Die Reihenfolge ist aber unwesentlich

- **Klassenvariable**

Klassenvariablen werden ebenfalls auf Klassenebene mit dem Modifier `static` deklariert. Sie existieren unabhängig von einem konkreten Objekt und ihre Lebensdauer ist vom Ladezustand der Klasse abhängig. Sie liegen ebenfalls am Heap und werden mit Standardwerten vorbelegt.

- **Lokale Variable**

Lokale Variablen werden innerhalb einer Methode oder eines Blocks definiert und sind ab ihrer Definition bis zum Ende der Methode oder des Blocks sichtbar. Sie liegen am Stack und bleiben uninitialisiert. Der Versuch, auf eine solche uninitialisierte lokale Variable zuzugreifen, führt auf einen Compilerfehler.

Beispiel:

```
1 public class Vars {  
2     int instanceVar;           // mit 0 initialisiert  
3     static boolean classVar;   // mit false initialisiert  
4  
5     static int wrong() {  
6         int localVar;         // bleibt uninitialisiert  
7         return localVar + 5;   // Compilerfehler  
8     }  
9 }
```

Listing 2.6: Variablentypen

Zeile 007 erzeugt den Compilerfehler:

```
variable localVar might not have been initialized
```

Bemerkung:

Es ist in Java nicht erlaubt, lokale Variable zu deklarieren, die gleichnamige lokale Variable eines weiter außen liegenden Blocks verdecken. Das Verdecken von Klassen- oder Instanzvariablen durch lokale Variable ist dagegen zulässig.

2.7 Methoden

Methoden⁴ haben prinzipiell den gleichen Aufbau wie in C und können nur innerhalb einer Klasse definiert werden. Hier unterscheidet man wie bei den Variablen Instanzmethoden (nicht statisch) und Klassenmethoden (statisch). Der technische Unterschied wird im Abschnitt über OOP genau erläutert. Der Unterschied in der Verwendung ist bereits an dieser Stelle wesentlich:

- Instanzmethoden sind an ein Objekt gebunden und können nur von einer Objektreferenz aufgerufen werden.
- Klassenmethoden sind von einem Objekt unabhängig und können ohne Objektreferenz über `Klassenname.methode()` aufgerufen werden.

2.7.1 Übergabe von Argumenten

Die Übergabe von Argumenten an eine Methode erfolgt in Java immer per Value, d.h. im zugehörigen Parameter wird immer der Wert des Arguments gespeichert. Verändert man in der Methode den Parameter, so hat dies keinen Einfluss auf das Argument.

Im Bereich primitiver Typen ist dieses Verhalten auch leicht nachvollziehbar, wie das folgende Beispiel zeigt:

```
1 public class ArgsTest_1 {  
2     public static void main(String []args) {  
3         int myInt = 100; // lokal in main  
4         inc(myInt);  
5         System.out.println("Wert von myInt: " + myInt);  
6     }  
}
```

⁴Funktionen werden in einem OOP-Umfeld Methoden genannt

```

7
8 public static void inc(int n) {
9     n++;
10 }
11 }

```

Listing 2.7: Call by Value - primitiver Typ

Ausgabe:

Wert von myInt: 100

In Zeile 4 wird der Wert des Arguments `myInt` (also 100) an den Parameter `n` übergeben und dieser wird in Zeile 10 inkrementiert. Die Originalvariable `myInt` bleibt dabei unverändert.

Etwas schwieriger ist dieser Effekt im Zusammenhang mit Objektreferenzen zu verstehen. Auch hier wird der Wert der Referenz kopiert, beide Referenzen (Argument und Parameter) sind aber nun Aliasnamen für ein und dasselbe Objekt (es wird nur die Referenz, nicht aber das Objekt kopiert). Solange die Parameterreferenz nicht verbogen wird, arbeitet man mit dem Originalobjekt. Obwohl eine Objektreferenz technisch per Value übergeben wird, bedeutet diese Übergabe für das referenzierte Objekt semantisch ein Call by Reference.

Beispiel:

```

1 import java.awt.Button;
2
3 public class ArgsTest_2 {
4     public static void main(String []args) {
5         Button btn; // Buttonreferenz deklarieren
6         btn = new Button(); // neuen Button erzeugen
7         btn.setLabel("Red"); // Beschriftung "Red" setzen
8         replaceLabel(btn);
9         System.out.println("Neue Beschriftung: " + btn.getLabel());
10    }
11
12    public static void replaceLabel(Button ref) {
13        ref.setLabel("Blue"); // ref und btn referenzieren dasselbe Objekt
14        ref = new Button(); // ref wird auf ein anderes Objekt verbogen
15        ref.setLabel("Green");
16    }
17 }

```

Listing 2.8: Call by Value - Referenz

Ausgabe:

Neue Beschriftung: Blue

2.7.2 Varargs

Seit Java 1.5 sind auch Methoden mit variabler Parameterliste möglich. Der entsprechende Parameter muss aus Gründen der Eindeutigkeit der letzte in der Parameterliste sein. In Wirklichkeit wird ein Array des entsprechenden Typs an die Funktion übergeben, wobei in der Argumentliste auch einzelne, durch Beistriche getrennte Werte angegeben werden dürfen. Das folgende Beispiel demonstriert die Syntax und die Vorgangsweise:

```

1 public class Varargs {
2     public static void main(String[] args) {
3         foo(1,2,3,4,5);
4         foo();
5         int []f = {1,2,3};
6         foo(f);
7     }
8
9     public static void foo(int... x) {
10        System.out.println("Anzahl der Argumente: " + x.length);
11        int sum = 0;
12        for(int i = 0; i < x.length; i++) {
13            sum += x[i];
14        }
15        System.out.println("Summe der Argumente: " + sum);

```



```

16 |     System.out.println("---");
17 | }
18 | }

```

Listing 2.9: Varargs

Ausgabe:

```

Anzahl der Argumente: 5
Summe der Argumente: 15
---
Anzahl der Argumente: 0
Summe der Argumente: 0
---
Anzahl der Argumente: 3
Summe der Argumente: 6
---

```

Wie obiges Beispiel zeigt, dürfen an der Stelle eines Varargs-Parameters 0-n Einzelwerte, alternativ aber auch ein Array übergeben werden.

2.7.3 Methodenüberladung

In Java ist es erlaubt, Methoden zu überladen, d.h. innerhalb einer Klasse mehrere unterschiedliche Methoden mit demselben Namen zu definieren. Der Compiler unterscheidet überladene Methoden an Hand ihrer Signatur. Diese setzt sich aus dem nach außen sichtbaren Namen plus codierter Information über die Reihenfolge und Typen der formalen Parameter zusammen. Die Signaturen zweier gleichnamiger Methoden sind also immer dann verschieden, wenn sie sich wenigstens in einem Parameter voneinander unterscheiden. Mehrere Methoden gelten also als sauber überladen, wenn sie sich entweder in der Anzahl ihrer Parameter oder in wenigstens einem Parametertyp unterscheiden.

Beispiel für eine Klasse mit Methodenüberladung:

```

1 | public class Overloading {
2 |     public static void main(String []args) {
3 |         System.out.println(foo(4));
4 |         System.out.println(foo(1,2));
5 |         System.out.println(foo(1,2L));
6 |         System.out.println(foo(1,2,3,4));
7 |     }
8 |
9 |     public static String foo(int a) { return "1"; }
10 |    public static int foo(int a, int b) { return 2; }
11 |    public static String foo(int a, double b) { return "3"; }
12 |    public static int foo(int... x) {return 4;}
13 | }

```

Listing 2.10: Methodenüberladung

Ausgabe:

```

1
2
3
4

```

Wie obiges Beispiel zeigt, sind beim Überladen von Funktionen die folgenden Regeln zu beachten:

- Funktionen mit exakt definierter Parameterliste haben immer Vorrang gegenüber von Funktionen mit variabler Parameterliste.
- Wird keine Funktion mit exakt passender Parameterliste gefunden, so wird ein implizites Widning⁵ versucht. So wird der Aufruf `foo(int, long)` in Zeile 5 durch Funktion `foo(int, double)` abgearbeitet.
- Rückgabetypen gehören nicht zur Signatur einer Methode und können bei überladenen Funktionen auch verschieden sein. Andererseits sind zwei Funktionen, die sich nur im Rückgabetypp unterscheiden nicht überladen und daher in ein und derselben Klasse unzulässig.

⁵Übergang zu einem größeren Datentyp

2.8 Garbage Collection

Alle Objekte werden zur Laufzeit dynamisch erzeugt. Dies geschieht explizit mit dem Schlüsselwort `new`. Der mit `new` angeforderte Speicher liegt am Heap und wird daher nicht wie der Speicher für die am Stack liegenden lokalen Variablen automatisch freigegeben, wenn die zugehörige Methode terminiert.

Speicher für Objekte, die durch das Schlüsselwort `new` dynamisch erzeugt werden, wird in Java vom sog. Garbage Collector wieder automatisch freigegeben, wenn das Objekt von keinem laufenden Thread mehr angesprochen werden kann. Konkret bedeutet dies, dass es keine lokale Referenz mehr geben darf, die auf das Objekt verweist. Als Programmierer kann man ein Objekt für den Garbage Collector vorbereiten, indem man alle Referenzen, die auf das Objekt verweisen auf `null` setzt.

Da es sich beim Garbage Collector um einen Thread niedriger Priorität handelt, kann der Zeitpunkt der Speicherfreigabe nicht bestimmt werden. Eine wesentliche Aussage in diesem Zusammenhang lautet: *"Garbage collection cannot be forced or scheduled."*

Kapitel 3

Sprachsyntax

3.1 Operatoren

3.1.1 Übersicht und Vorrangregeln

Die folgende Tabelle gibt einen Überblick über alle Operatoren der Sprache Java¹:

Typ	Operatoren	Assoziativität
Unäre Postfix-Operatoren	[] . (A) A++ A-- .	links
Unäre Präfix-Operatoren	++A --A +A -A ~ !	rechts
Objekterzeugung und Cast	new (A)	rechts
Binäre arithmetische Operatoren	* / %	links
Binäre arithmetische Operatoren	+ -	links
Shiftoperatoren	<< >> >>>	links
Relationale Operatoren	< <= > >= instanceof	links
Relationale Operatoren	== !=	links
bitweises/boolsches AND	&	links
bitweises/boolsches XOR	^	links
bitweises/boolsches OR		links
boolsches AND	&&	links
boolsches OR		links
Bedingungsoperator	?:	rechts
Zuweisungsoperatoren	= += -= *= /= <<= >>= >>>= &= ^= =	rechts

Dabei gelten die folgenden Vorrangregeln:

- Weiter oben stehende Operatoren haben Vorrang gegenüber weiter unten stehenden.
- Operatoren in der gleichen Zeile haben gleichen Vorrang. Hier ist die Assoziativität maßgebend. Linksassoziativität bedeutet, dass innerhalb eines Ausdrucks Operatoren mit gleichem Vorrang von links nach rechts abgearbeitet werden. Rechtsassoziativität bedeutet, dass innerhalb eines Ausdrucks Operatoren mit gleichem Vorrang von rechts nach links abgearbeitet werden.
- Runde Klammern durchbrechen immer die Regeln von Vorrang und Assoziativität. Es wird empfohlen, in komplexen Ausdrücken die Hierarchie und Assoziativität auch durch nicht notwendige Klammern hervorzuheben. Die Ausdrücke werden dadurch leichter lesbar.

Beispiele:

- `2 + 3 * 4` wird zu `2 + (3 * 4)` ausgewertet, weil der arithmetische Multiplikationsoperator `*` Vorrang gegenüber dem arithmetischen Additionsoperator `+` hat.
- `1 - 2 + 3` wird zu `((1 - 2) + 3)` ausgewertet, weil die arithmetischen Strichoperatoren linksassoziativ sind.

¹A steht für einen an dieser Stelle gültigen Ausdruck

- `--4` wird zu `(-(4))` ausgewertet, weil die arithmetischen Vorzeichenoperatoren rechtsassoziativ sind.

3.1.2 Typkonversion und Castoperator

Im Bereich primitiver numerischer Datentypen wird ein "kleinerer" Typ immer in einen "größeren" Typ konvertiert. Diesen Vorgang nennt man *widning*. Widning erfolgt ohne Informationsverlust². Widning geschieht automatisch in Pfeilrichtung gemäß der folgenden Abbildung:

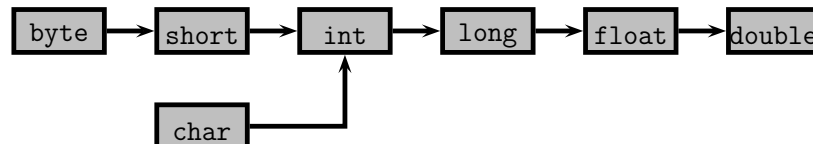


Abbildung 3.1: Widning primitiver Typen

Zuweisung eines größeren auf einen kleineren Typ (also gegen die Pfeilrichtung in obiger Grafik) verlangt immer explizite Typumwandlung mit Hilfe des Castoperators. Dessen Syntax lautet:

```
var2 = (newtype) var1;
```

Fehlt dieser Castoperator, so erzeugt der Compiler eine Fehlermeldung.

Der Datentyp `boolean` ist mit keinem der anderen primitiven Typen verträglich und kann daher auch in keiner Richtung konvertiert werden.

Der folgende Programmabschnitt zeigt die Verwendung des Castoperators:

```

1 public class Widning {
2     public static void main(String []args) {
3         byte b1 = 3;
4         short s1 = 130;
5         b1 = s1;           // Fehler, byte ist kleiner als short
6         b1 = (byte) s1;    // in Ordnung, aber eventuell Datenverlust
7         System.out.println(b1);
8         char ch1 = '\u0020';
9         int i1 = ch1;      // in Ordnung, int ist groesser als char
10        System.out.println(i1);
11    }
12 }

```

Listing 3.1: Widning und casten primitiver Typen

Ein Compilerlauf bringt folgende Fehlermeldung:

```

Widning.java:5: possible loss of precision
found   : short
required: byte
    b1 = s1;           // Fehler, byte ist kleiner als short

```

Entfernt man Zeile 5, so erhält man folgende Ausgabe:

```

-126
32

```

²Bis auf die Konvertierung von `long` in `float`

3.1.3 Inkrement- und Dekrementoperator

Der Inkrementoperator `++` erhöht und der Dekrementoperator `--` vermindert den Wert des Operanden jeweils um 1. Sie existieren sowohl in Präfix- als auch in Postfixnotation. Sie sind mit Ausnahme von `boolean` auf alle primitiven L-Values anwendbar.

Bei Präfixnotation wird der Wert des L-Values vor seiner Verwendung im Ausdruck verändert, bei Postfixnotation erst nach seiner Verwendung im Ausdruck.

Die folgende Tabelle zeigt die Zusammenhänge:

Ausgangswert von a	Ausdruck	Endwert von a	Endwert von b
10	<code>b = a++;</code>	11	10
10	<code>b = ++a;</code>	11	11
10	<code>b = a--;</code>	9	10
10	<code>b = --a;</code>	9	9

3.1.4 Die arithmetischen Operatoren

Die unären Vorzeichenoperatoren `+` und `-`

Java kennt die aus der Mathematik gebräuchlichen unären Vorzeichenoperatoren `+` und `-`. Der folgende Codeausschnitt demonstriert die Verwendung:

```

1 a = -b;
2 b = +10;
3 c = -(a + b);

```

Listing 3.2: Unäre Vorzeichenoperatoren

Die binären arithmetischen Operatoren

Java stellt 5 binäre arithmetische Operatoren zur Verfügung:

Operator	Bedeutung	Beispiel
<code>+</code>	Additionsoperator	<code>c = a + b;</code>
<code>-</code>	Subtraktionsoperator	<code>c = a - b;</code>
<code>*</code>	Multiplikationsoperator	<code>c = a * b;</code>
<code>/</code>	Divisionsoperator	<code>c = a / b;</code>
<code>%</code>	Rest-, Modulooperator	<code>c = a % b;</code>

Bei Verwendung arithmetischen Operatoren sind folgende Regeln zu beachten:

- Es gelten die aus der Mathematik bekannten Hierarchieregeln. Punkt- vor Strichrechnung, unäre Operationen vor binären. Die Modulooperation ist eine Punktoperation (genauer eine Division).
- Der Modulooperator führt eine Division aus und liefert den Rest dieser Division. Er darf im Gegensatz zu anderen Sprachen wie etwa C auch auf Fließkommaoperanden angewendet werden. Bei negativen Operanden und Fließkommaoperanden kann das Ergebnis nach folgender Regel bestimmt werden: vom Absolutbetrag des linken Operanden wird so lange der Absolutbetrag des rechten Operanden subtrahiert, bis das Ergebnis kleiner als der Betrag des rechten Operanden ist. Damit ergibt sich der Absolutbetrag des Resultats, das Vorzeichen des Ergebnisses stimmt mit jenem des linken Operanden überein. Beispiele:
 $17 \% 5 = 2$ $7.6 \% 2.9 = 1.8$
 $-5 \% 2 = -1$ $-5 \% -2 = -1$
- Hat der rechte Operand eines Divisionsoperators (`/` oder `%`) den Wert Null, so wird bei einer Integerdivision eine `ArithmeticException` geworfen (es entsteht also ein Laufzeitfehler), bei einer Fließkommaoperation erhält man ein nicht numerisches Ergebnis³.

³NegativeInfinity, PositiveInfinity bzw. NaN

- Das Ergebnis jeder arithmetischen Operation ist vom Typ `int` oder höher. Der höchste Datentyp der beteiligten Operanden bestimmt dann den Datentyps des Ergebnisses. Wird versucht, das Ergebnis ohne explizitem Typecast in einem niedrigeren Typ zu speichern, so erhält man einen Compilerfehler.

Beispiel:

```
1 byte a = 12, b;
2 b = -a;           // Compilerfehler, Typ von -a ist int
3 b = (byte) -a;    // richtig
```

Listing 3.3: Ergebnistyp `int` oder höher

3.1.5 Die logischen Operatoren

Die logischen Operatoren dürfen nur auf den Typ `boolean` angewendet werden. Java kennt die folgenden logischen Operatoren:

- Das logische AND in 2 Versionen: `&` sowie `&&`. Die Ausdrücke `a & b` bzw. `a && b` liefern genau dann `true`, wenn sowohl `a` als auch `b` den Wahrheitswert `true` haben. Ist wenigstens einer der beteiligten Wahrheitswerte `false`, so ist das Ergebnis `false`. Der Operator `&` wertet dabei stets beide Operanden aus, während der Operator `&&` den rechten Operanden `b` nur dann auswertet, wenn der linke Operand `a` den Wahrheitswert `true` hat. Bei dieser optimierten Auswertung spricht man von short-circuit-Evaluierung.
- Das logische nicht ausschließende OR in 2 Versionen: `|` sowie `||`. Die Ausdrücke `a | b` bzw. `a || b` liefern genau dann `false`, wenn sowohl `a` als auch `b` den Wahrheitswert `false` haben. Ist wenigstens einer der beteiligten Wahrheitswerte `true` so ist das Ergebnis `true`. Der Operator `|` wertet dabei stets beide Operanden aus, während der Operator `||` wieder short-circuit-Evaluierung durchführt. Dabei wird der rechte Operanden `b` nur dann ausgewertet, wenn der linke Operand `a` den Wahrheitswert `false` hat.
- Das logische ausschließende XOR `^`: Der Ausdruck `a ^ b` liefert genau dann `true`, wenn die Wahrheitswerte von `a` und `b` verschieden sind.
- Das logische NOT `!`: Der Ausdruck `!a` liefert den zu `a` entgegengesetzten Wahrheitswert.

Die folgende Tabelle gibt eine Übersicht:

a	b	!a	a b, a b	a & b, a && b	a ^ b
true	true	false	true	true	false
true	false	false	true	false	true
false	true	true	true	false	true
false	false	true	false	false	false

3.1.6 Die relationalen Operatoren

Java kennt 6 Vergleichsoperatoren (relationale Operatoren) im Bereich primitiver Typen⁴. Jeder Vergleich liefert den Ergebnistyp `boolean`, also den Ergebniswert `true` oder `false`. Die folgende Tabelle listet die relationalen Operatoren auf:

Operator	Bedeutung	Beispiel
<code>==</code>	prüft, ob <code>a</code> gleich <code>b</code> ist	<code>a == b</code>
<code>!=</code>	prüft, ob <code>a</code> ungleich <code>b</code> ist	<code>a != b</code>
<code><</code>	prüft, ob <code>a</code> kleiner als <code>b</code> ist	<code>a < b</code>
<code><=</code>	prüft, ob <code>a</code> kleiner oder gleich <code>b</code> ist	<code>a <= b</code>
<code>></code>	prüft, ob <code>a</code> größer als <code>b</code> ist	<code>a > b</code>
<code>>=</code>	prüft, ob <code>a</code> größer oder gleich <code>b</code> ist	<code>a >= b</code>

Bei der Verwendung der relationalen Operatoren sind die folgenden Regeln zu beachten:

⁴Sowie `instanceof` bei Referenzen

- Alle 6 Operatoren dürfen auf zwei beliebige primitive numerische Typen angewendet werden. Bei verschiedenen Typen erfolgt widening auf den höheren der beteiligten Typen, zumindestens jedoch auf `int`.
- Die Operatoren `==` und `!=` dürfen auch auf zwei Werte vom Typ `boolean` und auf Referenztypen angewandt werden. Bei Referenztypen liefert der Ausdruck `a==b` genau dann `true`, wenn `a` und `b` dasselbe Objekt referenzieren. Analog liefert der Ausdruck `a!=b` genau dann `true`, wenn `a` und `b` unterschiedliche Objekte referenzieren.

3.1.7 Der Bedingungsoperator

Der Bedingungsoperator `?` ist der einzige dreistellige Operator in Java. Seine Syntax lautet:

`<x> ? : <c>`

`<x>` ist dabei ein boolescher Ausdruck. Liefert `<x>` den Wert `true`, so ist der Ergebniswert des Gesamtausdrucks gleich dem Wert von ``, sonst dem Wert von `<c>`. Die Ausdrücke `` und `<c>` müssen dabei typverträglich, also beide entweder einfache numerische Typen, boolesche Typen oder Objektreferenzen sein.

Beispiel:

```

1 public class Conditional {
2     public static void main(String []args) {
3         int a = 13, b = 7;
4         int c = a < b ? a : b;
5         System.out.format("Das Minimum von %d und %d ist %d.\n", a, b, c);
6     }
7 }

```

Listing 3.4: Bedingungsoperator

Ausgabe:

Das Minimum von 13 und 7 ist 7.

3.1.8 Die Bitoperatoren

Die Bitoperatoren können nur auf ganzzahlige einfache Datentypen angewendet werden. Das Ergebnis jeder Bitoperation ist von Typ `int` oder `long`. Der Ergebnistyp `long` tritt genau dann auf, wenn wenigstens einer der beteiligten Operanden den Typ `long` aufweist.

Der Einerkomplementoperator

Beim bitweisen Einerkomplementoperator `~` handelt es sich um einen unären Operator, der das Bitmuster des Operanden kippt.

Beispiel:

```

1 byte b = 4;           // b: 0000 0100
2 b = (byte) ~b;        // b: 1111 1011
3 System.out.println(b); // Ausgabe: -5

```

Listing 3.5: Bitweises Einerkomplement

Bitweises AND, OR, XOR

Die binären Bitoperatoren AND (`&`), OR (`|`) und XOR (`^`) arbeiten nach folgenden Regeln:

- Jedes Bit im Ergebnis wird aus den beiden korrespondierenden Bits der Operanden berechnet.
- Beim bitweisen AND (`&`) ergibt sich im Resultat das Bit 1 genau dann, wenn die korrespondierenden Bits in beiden Operanden den Wert 1 haben.

- Beim bitweisen nicht ausschließenden OR (`|`) ergibt sich im Resultat das Bit 1 genau dann, wenn wenigstens eines der beiden korrespondierenden Bits den Wert 1 hat.
- Beim bitweisen ausschließenden XOR (`^`) ergibt sich im Resultat das Bit 1 genau dann, wenn genau eines der beiden korrespondierenden Bits den Wert 1 hat.

Beispiel:

```

1 byte a = 13;           // a: 0000 1101
2 byte b = 7;            // b: 0000 0111
3 byte c = (byte) (a & b) // c: 0000 0101
4 System.out.println(c); // Ausgabe: 5
5 c = (byte) (a | b);     // c: 0000 1111
6 System.out.println(c); // Ausgabe 15
7 c = (byte) (a ^ b);     // c: 0000 1010
8 System.out.println(c); // Ausgabe 10</pre>

```

Listing 3.6: Bitoperatoren

Die binären Bitoperatoren lassen sich auch mit dem Zuweisungsoperator kombinieren:
`&=`, `|=` und `^=`

Die Shiftoperatoren

Die Shiftoperatoren `<<`, `>>` und `>>>` unterliegen folgenden Regeln:

- Sie lassen sich nur auf einfache ganzzahlige Typen anwenden. Es handelt sich dabei um binäre Operatoren. Das Ergebnis ist wenigstens vom Typ `int`. Der Ergebnistyp `long` tritt genau dann auf, wenn der linke Operand den Typ `long` aufweist.
- Im Ausdruck `a << b` shiftet der Operator `<<` das Bitmuster von `a` um `b` Bits nach links, wobei in das niederwertigste Bit von `a` jeweils Nullen nachgeschoben werden. Shiften um 1 Bit nach links bedeutet (solange sich das Vorzeichenbit nicht ändert) eine schnelle Multiplikation von des linken Operanden mit 2.
- Im Ausdruck `a >> b` shiftet der Operator `>>` das Bitmuster von `a` um `b` Bits nach rechts, wobei das Vorzeichenbit von `a` seinen Wert beibehält. Ist `a` positiv und damit das Vorzeichenbit 0, so werden Nullen nachgeschoben, bei negativem `a` (gesetztem Vorzeichenbit) werden Einsen nachgeschoben. Shiften mit diesem Operator (signed-right-shift) um 1 Bit nach rechts bedeutet eine schnelle Integerdivision von `a` durch 2, solange sich das Vorzeichenbit von `a` nicht ändert.
- Im Ausdruck `a >>> b` shiftet der Operator `>>>` das Bitmuster von `a` um `b` Bits nach rechts, wobei in das Vorzeichenbit von `a` immer Nullen nachgeschoben werden.
- Der rechte Operand der Shiftoperatoren wird vor dem Shiften immer modulo `x` ausgewertet, wobei `x` vom Ergebnistyp der Operation abhängt. In der Regel hat `x` den Wert 32, wenn der linke Operand und damit das Ergebnis der Shiftoperation den Typ `long` aufweisen, so ist `x` gleich 64.
- Die Shiftoperatoren lassen sich auch mit dem Zuweisungsoperator kombinieren:
`<<=`, `>>=` und `>>>=`

3.2 Kontrollstrukturen

Java kennt die folgenden Kontrollstrukturen:

- Die bedingte Verzweigung `if-else`
- Die Fallabfrage `switch-case`
- Die fußgesteuerte `do-while` - Schleife
- Die kopfgesteuerte `while` - Schleife
- Die kopfgesteuerte `for` - Schleife

- Die erweiterte `for` - Schleife⁵

Ein wesentlicher Unterschied zu C ergibt sich durch die Tatsache, dass Wahrheitswerte in Java über den Datentyp `boolean` realisiert sind. Daher müssen alle Bedingungen boolsche Ergebnistypen liefern. Numerische Typen wie in C sind hier also nicht erlaubt.

3.2.1 Abfragen

Die bedingte Verzweigung `if-else`

Die einfache `if` - Anweisung

Die einfache `if`-Anweisung hat folgende Syntax:

```
if (<Bedingung>)
    <Anweisung>
```

<Bedingung> muss dabei ein Ergebnis vom Typ `boolean` liefern. Wird <Bedingung> zu `true` ausgewertet, so wird <Anweisung> ausgeführt. Dabei handelt es sich entweder um eine einzelne Anweisung oder um einen mit `{ }` begrenzten Block von Anweisungen.

Die `if-else` - Anweisung

Die `if-else`-Anweisung hat folgende Syntax:

```
if (<Bedingung>)
    <Anweisung 1>
else
    <Anweisung 2>
```

<Bedingung> muss dabei ein Ergebnis vom Typ `boolean` liefern. Wird <Bedingung> zu `true` ausgewertet, so wird <Anweisung 1> ausgeführt, andernfalls <Anweisung 2>. Dabei handelt es sich entweder um einzelne Anweisungen oder um mit `{ }` begrenzte Anweisungsblöcke.

Bemerkungen zur `if-else` - Anweisung

Es ist zu beachten, dass bei Verschachtelung von bedingten Verzweigungen das Setzen der Blockklammern `{ }` für die Semantik wesentlich sein kann. Jedes `else` wird dem letzten nicht abgeschlossenen `if` - Zweig zugerechnet.

Im zweiten Teil des folgenden Beispiels verleitet das Fehlen der Einrückung in Zeile 14 zu dem Irrglauben, dass dieser `else`-Zweig zum `if` in Zeile 11 gehört:

```
1 int a = 10, b = 20;
2 boolean c = false;
3 //-----
4 if (a == b) {                // if (1)
5     if(c)                    // if (2)
6         System.out.println("A");
7 }
8 else                          // else zu if (1)
9     System.out.println("B");
10 //-----
11 if (a == b)                  // if (1)
12     if(c)                    // if (2)
13         System.out.println("A");
14 else                          // else zu if (2)
15     System.out.println("B");
```

Listing 3.7: Falsche Verwendung `if-else`

Das nächste Beispiel zeigt eine gebräuchliche und übersichtliche Schreibweise für verkettete `if-else`-Anweisungen:

```
1 int a = 10, b = 20;
2 if (a < b) {
3     System.out.println("a kleiner b");
```

⁵Neu seit JDK 1.5

```
4 }  
5 else if(a > b) {  
6     System.out.println("a groesser b");  
7 }  
8 else {  
9     System.out.println("a gleich b");  
10 }
```

Listing 3.8: Verschachtelte Verzweigungen

Die switch - Anweisung

Die `switch` - Anweisung erlaubt die Programmierung einer Fallabfrage, wobei die einzelnen Fälle verschiedenen Werten des ganzzahligen Ausdrucks `<int-expr>` zuzuordnen sind. Die vollständige Syntax lautet:

```
switch(<int-expr>) {  
    case label-1:  
        <Anweisung 1>  
    case label-2:  
        <Anweisung 2>  
    ...  
    case label-n:  
        <Anweisung n>  
    default:  
        <Anweisung>  
}
```

Die `switch` - Anweisung wird folgendermaßen abgearbeitet:

1. Der ganzzahlige Ausdruck `<int-expr>` wird ausgewertet. Hier sind die Typen `byte`, `short`, `int`, `char` sowie `Enums`⁶ erlaubt. `Enums` werden später behandelt. Alle anderen Typen (insbesondere der Typ `long` führt auf einen Compilerfehler.
2. Der Wert von `<int-expr>` wird nun der Reihe nach mit den Konstanten `label-1`, `label-2` usw. verglichen. Wird eine Übereinstimmung gefunden, so werden die zur Übereinstimmung gehörige Anweisung und alle folgenden Anweisungen bis zum nächsten `break` abgearbeitet. Jeder Ausdruck `<Anweisung-i>` kann dabei auch aus einer beliebigen Folge von Einzelanweisungen bestehen.
3. Ergibt sich keine Übereinstimmung des Wertes von `<int-expr>` mit einer der verwendeten Konstanten `label-i`, so werden der `default` - Zweig und alle folgenden Anweisungen bis zum nächsten `break` abgearbeitet. Der `default` - Zweig ist optional, eindeutig und muss nicht der letzte Zweig in der `switch` - Anweisung sein.

Das folgende Beispiel demonstriert die Arbeitsweise:

```
1 public class SwitchDemo {  
2     public static void main(String []args) {  
3         int []x = {20,40,50,70};  
4         for(int i : x) {  
5             switch(i) {  
6                 case 10: System.out.println(10);  
7                 case 20: System.out.println(20);  
8                 case 30: System.out.println(30);  
9                     break;  
10                case 40: System.out.println(40);  
11                default: System.out.println("default");  
12                case 50: System.out.println(50);  
13                    break;  
14                case 60: System.out.println(60);  
15                    break;  
16            }  
17            System.out.println("-----");  
18        }  
19    }  
20 }
```

⁶Seit Java 1.5

```

18 |     }
19 | }
20 |

```

Listing 3.9: Demonstration zu switch-case

Ausgabe:

```

20
30
-----
40
default
50
-----
50
-----
default
50
-----

```

Folgende Punkte sind beim Arbeiten mit der `switch` - Anweisung weiters zu beachten:

- Jeder Label `label-i` muss konstant sein bzw. vom Compiler zu einer Konstanten ausgewertet werden können. Erlaubt sind also auch z.B.
`case 2+3:`
oder
`case x:` mit der Definition `final int x = 7;`
- Jeder Label muss eindeutig sein, sonst entsteht ein Compilerfehler.
- Wird ein Label angegeben, dessen Wert nicht in den Wertebereich des Typs von `<int-expr>` passt, so entsteht ein Compilerfehler. Dies passiert z.B. bei einem Label `case 128:`, wenn `<int-expr>` den Typ `byte` hat.
- Syntaktisch richtig ist auch die leere `switch` - Anweisung, also
`switch(i) {}`

3.2.2 Schleifen

Mit Hilfe einer Schleife kann eine einzelne Anweisung oder ein Block von Anweisungen wiederholt abgearbeitet werden. Java kennt nur positiv formulierte Schleifenbedingungen, d.h. eine Schleife wird abgearbeitet, solange eine bestimmte Bedingung erfüllt ist, also den booleschen Wert `true` liefert.

Die `while` - Schleife

Die Syntax der `while` - Schleife lautet:

```

while(<Bedingung>)
    Anweisung

```

`<Bedingung>` muss dabei ein Ergebnis vom Typ `boolean` liefern. Solange `<Bedingung>` den Wert `true` liefert wird der Schleifenkörper `<Anweisung>` ausgeführt. Dabei handelt es sich entweder um eine einzelne Anweisung oder um einen mit `{}` begrenzten Block von Anweisungen.

Die `do-while` - Schleife

Die Syntax der `do-while` - Schleife lautet:

```

do
    Anweisung
while(<Bedingung>);

```

<Bedingung> muss dabei ein Ergebnis vom Typ `boolean` liefern. Solange <Bedingung> den Wert `true` liefert wird der Schleifenkörper <Anweisung> ausgeführt. Da die Bedingung erst am Schleifenende geprüft wird, arbeitet der Schleifenkörper mindestens einmal. Wie bei der `while`-Schleife handelt es sich dabei entweder um eine einzelne Anweisung oder um einen mit `{ }` begrenzten Block von Anweisungen.

Die for - Schleife

Die `for`-Schleife ist die universellste aller Schleifen in Java. Sie wird hauptsächlich für zählergesteuerte Schleifen verwendet. Ihre Syntax lautet:

```
for(<Initialisierung>; <Bedingung>; <Update>)
    <Anweisung>
```

Die `for`-Schleife wird wie folgt abgearbeitet:

1. Zunächst wird <Initialisierung> abgearbeitet. Hier werden üblicherweise eine oder mehrere Variable initialisiert.
2. Danach wird der boolsche Ausdruck <Bedingung> ausgewertet. Solange das Ergebnis `true` liefert wird als nächstes der Schleifenkörper <Anweisung> abgearbeitet.
3. Nach Abarbeitung des Schleifenkörpers wird der Ausdruck <Update> ausgeführt. Hier werden in der Regel ein oder auch mehrere Zählvariable erhöht bzw. vermindert. Nach Abarbeitung von <Update> wird wieder die Schleifenbedingung <Bedingung> ausgewertet und bei `true` die Befehle im Schleifenkörper durchgeführt. Die Schleife endet, sobald die Auswertung von <Bedingung> den Wert `false` liefert.

Die `for` - Schleife entspricht also semantisch folgender `while` - Schleife:

```
<Initialisierung>
while(<Bedingung>) {
    <Anweisung>
    <Update>
}
```

Ein einfaches Beispiel zeigt die Verwendung der `for` - Schleife:

```
1 int x;
2 for(x = 0; x < 10; x++) {
3     System.out.println("Der Wert von x ist " + x);
4 }
```

Listing 3.10: Verwendung der `for`-Schleife

Ausgabe:

```
Der Wert von x ist 0
...
Der Wert von x ist 9
```

Beim Arbeiten mit `for` - Schleifen sind folgende Punkte zu beachten:

- Im Ausdruck <Initialisierung> dürfen auch mehrere Variable initialisiert werden und im Ausdruck <Update> dürfen mehrere Variable hochgezählt werden. Dabei sind die einzelnen Teile dieser Ausdrücke durch Beistriche abzugrenzen. Beispiel:

```
int j, k;
for(j = 3, k = 6; j + k < 20; j++, k += 2) {
    System.out.println("j ist " + j + " und k ist " + k);
}
```

- Im Ausdruck <Initialisierung> dürfen auch Variable deklariert werden. Diese sind dann nur innerhalb der `for` - Schleife sichtbar und verlieren nach der Schleife ihre Gültigkeit:

```
for(int x = 0; x < 10; x++) {
    System.out.println("Der Wert von x ist " + x);
}
System.out.println(x); // Compilerfehler
                        // x nicht mehr sichtbar
```

- Im Ausdruck `<Initialisierung>` dürfen gewöhnliche Ausdrücke und Deklarationen nicht gleichzeitig auftreten und man kann nicht verschiedene Typen deklarieren. Auch dürfen keine Variablen deklariert werden, die andere lokale Variable überdecken:

```
int i = 3;
for(i++, int j = 3; i < 20; j++) {}           // falsch
for(int j = 3, i = 3; i < 20; j++) {}         // falsch
for(int k = 3, long l = 20; k < 20; k++) {}    // falsch
for(int a = 7, b = 3; a > b; a--, b++) {}      // richtig
```

- Eine `for` - Schleife mit leerer Schleifenbedingung ist gültig, es handelt sich dabei um eine Endlosschleife:
`for(;;) {}`

Die erweiterte `for`-Schleife

Diese Schleife ist neu seit Java 1.5 und dient zur Iteration durch ein Array oder eine Collection⁷. Sie wird oft auch als `for-in` - Schleife bezeichnet. Ihre Syntax lautet:

```
for(<Deklaration> : <Expression>)
    <Anweisung>
```

`<Expression>` muss dabei zu einem Array (oder einer Collection) ausgewertet werden. Die Schleife iteriert durch dieses Array. Genauer kann `<Expression>` also eine Arrayreferenz oder ein Methodenaufruf sein, wenn die Methode ein Array liefert. Erlaubt sind Felder beliebigen Typs (also Felder primitiver Typen, Objektreferenzen oder mehrdimensionale Felder).

`<Deklaration>` Hier muss es sich um eine an dieser Stelle neu deklarierte Blockvariable handeln, deren Typ mit jenem der Arrayelemente übereinstimmt. Die Variable verliert nach dem Schleifenblock ihre Gültigkeit. Ihr Wert stimmt mit jenem des aktuellen Arrayelementes überein. Die Arrayelemente können mit Hilfe dieser Schleife nur gelesen, nicht aber verändert werden.

Das folgende Beispiel demonstriert das Arbeiten mit der erweiterten `for` - Schleife:

```
1 public class ForInDemo {
2     public static void main(String[] args) {
3         int []x = {1,2,3,4,5};
4         for(int i : x) {
5             i++; // Veraendert die Arrayelemente nicht
6             System.out.print(i);
7         }
8         System.out.println();
9         for(int j : x) {
10            System.out.print(j);
11        }
12        System.out.println();
13    }
14 }
```

Listing 3.11: Erweiterte `for`-Schleife

Ausgabe:

```
23456
12345
```

⁷Collections werden später behandelt

Verwendung der Schlüsselworte `break` und `continue`

Mit Hilfe des Schlüsselwortes `break` kann die aktuell laufende Schleife oder der aktuelle durch `{ }` begrenzte Block abgebrochen werden. Der Programmfluss setzt mit jener Anweisung fort, die der Schleife oder dem Block unmittelbar folgt.

Mit Hilfe des Schlüsselwortes `continue` kann der aktuell laufende Schleifendurchgang abgebrochen werden. Im Gegensatz zu `break` darf `continue` nur in Schleifen angewendet werden. Der Programmfluss setzt mit (neuerlicher) Prüfung der Schleifenbedingung fort. Bei `for` - Schleifen wird zuvor noch der Ausdruck `<Update>` abgearbeitet.

Bei verschachtelten Kontrollblöcken bzw. Schleifen wirken `break` und `continue` nur auf den innersten Block bzw. die innerste Schleife. Dies lässt sich durch Verwendung einer Sprungmarke ändern. Das folgende Beispiel demonstriert die Vorgangsweise:

```
1 public class LabeledSkip {
2     public static void main(String args[]) {
3         int[][] mat = {{4, 3, 5}, {2, 1, 6}, {9, 7, 8}};
4         int sum = 0;
5         outer:           // label outer
6         for (int i = 0; i < mat.length; i++){
7             for (int j = 0; j < mat[i].length; j++) {
8                 if (j == i)
9                     continue; // Sprung zu j++ in Zeile 7
10                System.out.format("[%d,%d]:%2d ", i, j, mat[i][j]);
11                sum += mat[i][j];
12                if (sum > 10)
13                    break outer; // Sprung zu Zeile 17
14            } // Ende innere Schleife
15            System.out.println();
16        } // Ende aeussere Schleife
17        System.out.format("\nSumme der ausgegebenen Elemente: %d\n", sum);
18    }
19 }
```

Listing 3.12: Verwendung von `break` und `continue`

Ausgabe:

```
[0,1]: 3   [0,2]: 5
[1,0]: 2   [1,2]: 6
Summe der ausgegebenen Elemente: 16
```

Kapitel 4

Objektorientierte Programmierung

4.1 Grundlagen

4.1.1 Idee der objektorientierten Programmierung

Der Grundgedanke der objektorientierten Programmierung (OOP) ist es, Objekte des realen Lebens mit Hilfe von Software abzubilden. Objekte besitzen einen *Zustand* und ein *Verhalten*.

- *Zustand*
Der Zustand eines Objektes wird durch Daten repräsentiert. So kann z.B. der Zustand eines (fahrenden) Motorrades durch Werte wie Geschwindigkeit, eingelegter Gang etc. beschrieben werden.
- *Verhalten*
Ein Objekt kann seinen Zustand ändern. Dies geschieht durch Änderung der Daten, die den momentanen Zustand des Objektes beschreiben. Bei einem (fahrenden) Motorrad sind dies Vorgänge wie bremsen, beschleunigen (ändern der Geschwindigkeit) oder schalten (ändern des eingelegten Ganges). Dieses Verhalten wird softwaremäßig durch Methoden (Funktionen) implementiert. Methoden sind also Funktionen, die auf den Daten (dem Zustand) eines Objektes operieren. Lassen sich die Daten und damit der Zustand des Objektes nur durch diese Methoden ändern, so sind die Daten vor unerlaubtem Zugriff geschützt (ein Motorrad mit einem Fünfganggetriebe kann nicht in den sechsten Gang geschaltet werden).

Klassen

Der Zustand und das Verhalten eines Objektes - also die Datenfelder und die Methoden - werden in der OOP mit Hilfe einer sog. Klassendefinition beschrieben. Eine Klasse stellt also den Bauplan eines Objektes dar. Durch die Definition einer Klasse erhält man einen neuen Datentyp. Mit Hilfe dieses Datentyps können sog. Objekte erzeugt werden. Dieser Vorgang heißt auch Instanzieren der Klasse, ein Objekt wird auch Instanz der Klasse genannt.

4.1.2 Eigenschaften einer objektorientierten Sprache

Eine objektorientierte Programmiersprache muss die folgenden 3 Eigenschaften besitzen:

1. Datenkapselung und Schutz der Daten vor unerlaubten Zugriffen (encapsulation, information hiding)
2. Vererbung (inheritance)
3. Vielgestaltigkeit (Polymorphie)

Datenkapselung und Informationhiding

Wie oben beschrieben bilden die Daten und zugehörige Methoden eine Einheit. Man kapselt also die Daten und Methoden mit Hilfe einer Klassendefinition. Der Schutz der Daten kann durch die Möglichkeit realisiert werden, einzelne Mitglieder einer Klasse (in der Regel die Datenfelder) durch verschiedene Zugriffsbeschränkungen (`private`, `protected`) nach außen unsichtbar zu machen und so vor unerlaubten Zugriffen zu schützen. Die Methoden bilden dann eine Art Schutzmantel um die Daten, die sich nun nur über die Methoden in vordefinierter Art und Weise manipulieren lassen.

Vererbung

Klassen können von anderen Klassen abgeleitet werden und erben damit alle Datenfelder und Methoden der Basis- oder Superklasse. Die Wirkung ist ähnlich, wie wenn die geerbten Informationen direkt in der abgeleiteten Klasse oder Subklasse ausprogrammiert wären. Das Prinzip der Vererbung ermöglicht eine außerordentlich einfache Wiederverwendung fertiger Software.

Polymorphie

Polymorphie oder Vielgestaltigkeit bedeutet, dass Objekte verschiedener Klassen auf die selbe Nachricht (d.h. denselben Methodenaufruf) unterschiedlich reagieren. Näheres dazu im Abschnitt über Polymorphie.

Klassen und Objekte

4.2.1 Klassen

Eine Klassendefinition wird in Java durch das Schlüsselwort `class` eingeleitet. Der Name einer Klasse sollte immer mit einem Großbuchstaben beginnen. Die einfachste gültige Klassendefinition ist die leere Klasse:

```
1 class EmptyClass {  
2 }
```

Listing 4.1: Leere Klasse

Eine Klasse enthält in der Regel Datenfelder und Methoden. Dabei unterscheidet man

- Instanzmethoden
- Klassenmethoden
- Instanzvariable
- Klassenvariable

Klassenmethoden und Klassenvariablen werden durch den Modifier `static` gekennzeichnet. Das folgende einfache Beispiel enthält nur Instanzvariablen und Instanzmethoden:

Listing

```
1 public class Kreis {  
2     private double radius;  
3  
4     public void setRadius(double r) { // Instanzmethode  
5         radius = r > 0.0 ? r : 0.0; // Mutator  
6     }  
7  
8     public double getRadius() { // Instanzmethode  
9         return radius; // Accessor  
10    }  
11  
12    public double umfang() { // Instanzmethode  
13        return 2.0 * radius * Math.PI;  
14    }  
15  
16    public double flaeche() { // Instanzmethode  
17        return radius * radius * Math.PI;  
18    }  
19 }
```

Listing 4.2: Kreis.java Einfache Klasse Kreis

Die Klasse `Kreis` hat eine private Instanzvariable `radius` und 4 öffentliche Instanzmethoden `setRadius`, `getRadius`, `umfang` und `flaeche`.

4.2.2 Paketzugehörigkeit

Javaklassen werden logisch in sogenannte Pakete (packages) eingeteilt. Ein Package wird konkret durch ein Verzeichnis im Dateisystem realisiert. Das Basisverzeichnis dieser Paketstruktur muss im sog. Klassenpfad (`classpath`) liegen. Jedes Paket heißt genauso wie das zugehörige Unterverzeichnis relativ zum Basisverzeichnis. Alle Klassen im aktuellen Arbeitsverzeichnis bilden ebenfalls ein Paket, das sogenannte anonyme Paket.

In jeder Quelldatei (Ausnahme: anonymes Paket) muss als erster Befehl das Paket, zu dem diese Quelldatei gehört angegeben werden:

```
package myPackage;
```

Das wichtigste Paket in der Klassenbibliothek Java heißt `java.lang`. Es ist das einzige Paket, das von anderen Klassen nicht importiert werden muss. Verwendet man in einer Java-Quelldatei Klassen anderer Pakete, so müssen diese über das `import` - Statement bekanntgemacht werden. Dabei kann man das gesamte Paket oder nur einzelne Klassen angeben:

```
import javax.swing.JOptionPane; // einzelne Klasse
import java.util.*;             // gesamtes Paket
```

4.2.3 Access Modifier

Java kennt 4 Zugriffsarten für Klassenmitglieder:

- `private`
Mitglieder, die über das Zugriffsschlüsselwort `private` definiert sind, sind nur innerhalb der eigenen Klasse sichtbar und daher vor direkten Zugriffen von außen geschützt (information hiding).
- default-Zugriff
Der Standardzugriff. Gibt man bei der Definition eines Klassenmitgliedes kein Zugriffsschlüsselwort an, so erhält das entsprechende Mitglied den Default-Zugriff. Dies bedeutet, dass das entsprechende Mitglied in allen Klassen und Methoden des eigenen Pakets sichtbar ist.
- `protected`
Mitglieder mit dieser Zugriffsart sind im eigenen Paket und in allen Subklassen, auch wenn diese außerhalb des eigenen Pakets liegen, sichtbar (vgl. Abschnitt über Vererbung).
- `public`
Mitglieder mit Access-Modifier `public` sind uneingeschränkt überall sichtbar.

Klassen auf Dateiebene¹ dürfen nur die Zugriffe `public` oder Default aufweisen. In jeder Quelldatei darf nur höchstens eine Klasse mit dem Modifier `public` definiert werden. Diese bestimmt dann den Namen der Quelldatei.

4.2.4 Zugriffsfunktionen, Datenkapselung

Da die beiden ersten Methoden `setRadius` und `getRadius` der Klasse `Kreis` den Zugriff auf das private Datenfeld `radius` erlauben, werden sie auch als Zugriffsfunktionen (access functions) bezeichnet. Mit Hilfe des Mutators `setRadius` kann das private Datenfeld `radius` von außerhalb der Klasse verändert werden. Hier wird das Speichern eines ungültigen Wertes (negativer Radius) verhindert. Im Allgemeinen definiert man Methoden als öffentlich und Datenfelder im Sinne der Datenkapselung als privat. Für jede private Variable, auf die von außerhalb der Klasse zugegriffen werden muss, sind dann entsprechende Zugriffsfunktionen zu programmieren. Diese sollten der Namenskonvention für Zugriffsfunktionen folgen:

Die Zugriffsfunktionen für ein Datenfeld `xxx` sollten die Namen `setXxx` (Mutator) und `getXxx` (Accessor) haben. Ist `xxx` vom Typ `boolean`, so nennt man den Accessor `isXxx`.

Eine Klasse, in der alle Datenfelder über den Modifier `private` definiert wurden, heißt vollständig gekapselt. Im Sinne eines guten objektorientierten Designs sollte man Klassen grundsätzlich so gut wie

¹Klassen können auch verschachtelt werden - innere Klassen

möglich kapseln. Dies erhöht die Sicherheit des Codes und erleichtert ein internes Redesign. Alle privaten Mitglieder der Klasse können ohne Probleme verändert werden, solange man die nach außen sichtbare Schnittstelle unverändert lässt.

4.2.5 Objekte

Die Definition einer Klasse erzeugt noch kein Objekt, sondern stellt lediglich eine Art Bauplan für eine solches Objekt dar. Technisch gesehen definiert man mit einer Klasse einen neuen Datentyp. Variable dieses neuen Typs heißen Referenzen und sind Namen für Instanzen bzw. Objekte dieser Klasse. In Java können neue Objekte ausschließlich dynamisch mit Hilfe des Operators `new` erzeugt werden (Ausnahmen: Klasse `java.lang.String` und Felder). Möchte man ein solches Objekte später im Programm ansprechen, so muss es mit Hilfe einer Referenz gespeichert werden. Eine solche Referenz kann man sich als eine Art Zeiger auf ein Objekt vorstellen, in dem aber keine nach außen sichtbare Adresse abgespeichert ist. Objekte, denen keine Referenzen zugeordnet sind, gelten als frei, können vom Programm aus nicht mehr angesprochen werden und werden vom System² bei Speicherbedarf automatisch aus dem Speicher entfernt.

Das folgende Beispiel demonstriert diese Techniken unter Verwendung der weiter oben definierten Klasse `Kreis`:

```

1 package oop;
2
3 public class KreisTest {
4     public static void main(String[] args) {
5         Kreis k1; // Objektvariable oder Referenz deklarieren
6         k1 = new Kreis(); // Neues Objekt erzeugen und k1 zuweisen
7         k1.setRadius(3.8);
8         System.out.format("Umfang von k1: %5.2f\n", k1.umfang());
9         System.out.format("Flaeche von k1: %5.2f\n", k1.flaeche());
10
11        Kreis k2 = new Kreis();
12        k1 = null; // k1 wird die Nullreferenz zugewiesen
13        k2 = new Kreis(); // die Referenz k2 wird "verbogen"
14        k2.setRadius(-2.5);
15        System.out.format("Radius von k2: %5.2f\n", k2.getRadius());
16    }
17 }

```

Listing 4.3: Klasse `KreisTest`

Ausgabe:

```

Umfang von k1:  23,88
Flaeche von k1: 45,36
Radius von k2:  0,00

```

Beim Erzeugen eines neuen Objektes in den Zeilen 6, 11 und 13 arbeitet eine spezielle Methode, der sogenannte Defaultkonstruktor der Klasse `Kreis`. Immer dann, wenn überhaupt kein Konstruktor vom Programmierer definiert wird, ergänzt der Compiler die Klasse um einen solchen Defaultkonstruktor, der allerdings keine Funktionalität aufweist. Dem in Zeile 6 erzeugten Objekt wird die Referenz `k1` zugewiesen. In Zeile 12 erhält `k1` mit `null` die sog. Nullreferenz zugewiesen, das in Zeile 6 erzeugte Objekt ist ab diesem Zeitpunkt unreferenziert und daher bereit für Garbage-Collecting. Das in Zeile 11 erzeugte Objekt ist nach Zeile 13 ohne Referenz, das in Zeile 13 erzeugte Objekt nach Ende der Methode `main`, da `k1` und `k2` lokale Referenzen sind und mit Ende der Methode `main` vom Stack gelöscht werden.

4.2.6 Instanzmethoden und die `this`-Referenz

Alle in der Klasse `Kreis` definierten Methoden sind Instanzmethoden. Solche Methoden definieren das Verhalten von Objekten und haben Zugriff auf alle anderen Variablen und Methoden der Klasse. Der Aufruf einer Instanzmethode ist an die Existenz eines Objekts gebunden und kann auch nur über ein

²genauer vom Garbage Collector

Objekt erfolgen.

Z.B. `k1.flaeche()`

Eine Instanzmethode darf auf die eigenen Instanzvariablen zugreifen, ohne den Punktoperator zu verwenden.

Z.B. `return 2.0 * radius * Math.PI;`

Dies ist möglich, weil der Compiler alle nicht über Punktnotation verwendeten Variablen `x`, die nicht lokale Variablen sind, auf die Referenz `this` bezieht und damit als `this.x` interpretiert. `this` ist eine Referenz, die auf das aktuelle Objekt verweist und dazu verwendet wird, die eigenen Instanzmethoden und Instanzvariablen anzusprechen. Diese `this`-Referenz ist auch explizit verfügbar und kann wie eine ganz normale Referenz verwendet werden. Sie wird als versteckter Parameter an jede Instanzmethode übergeben. Die Methode `flaeche` der Klasse `Kreis` kann also auch so geschrieben werden:

```
public double flaeche() {  
    return this.radius * this.radius * Math.PI;  
}
```

Lokale Variable einer Methode dürfen Instanzvariable überdecken. Beim Zugriff auf Instanzvariable ist dann die `this`-Referenz unbedingt erforderlich. Dies demonstriert die umgeschriebene Methode `setRadius` der Klasse `Kreis`:

```
public void setRadius(double radius) {  
    this.radius = radius > 0.0 ? radius : 0.0;  
}
```

Diese Namensgleichheit von Instanzvariablen und Parametern ist bei `setXxx`-Methoden und Konstruktoren üblich und sollte konsequent verwendet werden.

4.2.7 Konstruktoren und Destruktor

Konstruktoren

Konstruktoren sind spezielle Instanzmethoden, die bei der Initialisierung eines Objektes aufgerufen werden. Konstruktoren werden als Methoden ohne Rückgabotyp definiert, deren Namen mit jenem der Klasse übereinstimmen, zu der sie gehören. Konstruktoren dürfen eine beliebige Anzahl an Parametern haben und können überladen werden. Die Klasse `Kreis` kann sinnvoll um folgende Konstruktoren ergänzt werden:

```
1 public class Kreis {  
2     private double radius;  
3  
4     public Kreis() {  
5         this.radius = 0.0;  
6     }  
7  
8     public Kreis(double radius) {  
9         this.radius = radius;  
10    }  
11  
12    // weitere Klassenmitglieder  
13 }
```

Listing 4.4: Konstruktoren der Klasse `Kreis`

Im Zusammenhang mit Konstruktoren sind die folgenden Regeln zu beachten:

- Falls der Programmierer in einer Klasse überhaupt keinen expliziten Konstruktor definiert, so erzeugt der Compiler automatisch einen parameterlosen (leeren) Defaultkonstruktor. Definiert man in der Klasse dagegen irgendeinen Konstruktor, so erzeugt der Compiler keinen Konstruktor und man muss auch den parameterlosen Konstruktor ausprogrammieren.

- Ein Konstruktor kann einen anderen Konstruktor der eigenen Klasse aufrufen. Diese Technik heißt Konstruktorenverkettung. Dazu wird die formale Methode `this([argumente])` mit der gewünschten Argumentliste aufgerufen. Der Aufruf von `this` muss die erste Anweisung im Konstruktor sein. Der parameterlose Konstruktor der Klasse `Kreis` hätte also auch wie folgt programmiert werden können:

```
public Kreis() {
    this(0.0);
}
```

- Neben den Konstruktoren darf es auch Methoden mit Rückgabotyp geben, die genauso heißen wie die Klasse. Diese gewöhnlichen Methoden unterscheiden sich von einem Konstruktor dann nur durch den Rückgabotyp. Obwohl die Sprachsyntax von Java diesen Programmierstil erlaubt, sollte er unbedingt vermieden werden.

```
class BadClass {
    public BadClass()          {} // Konstruktor
    public BadClass(int i)     {} // Konstruktor
    public void BadClass(int i) {} // Instanzmethode
}
```

- Erzeugt der Compiler einen Defaultkonstruktor, so erhält dieser immer den gleichen Zugriff wie die Klasse. Der Defaultkonstruktor einer öffentlichen Klasse ist also `public`, der einer Klasse mit Defaultzugriff besitzt ebenfalls Defaultzugriff.

Destruktor

Neben Konstruktoren, die während der Initialisierung eines Objekts aufgerufen werden, gibt es in Java auch Destruktoren. Sie werden unmittelbar vor dem Zerstören eines Objekts durch den Garbage Collector aufgerufen.

Ein Destruktor wird als parameterlose Methode mit dem Namen `finalize` und dem Zugriff `protected` definiert:

```
protected void finalize() {}
```

4.2.8 Variable als Klassenmitglieder

Innerhalb einer Klasse können zwei Typen von Variablen definiert werden:

- Instanzvariable
- Klassenvariable

Instanzvariable

Instanzvariable oder Objektvariable sind an ein Objekt gebunden, d.h. jede Instanz einer Klasse hat ihren eigenen Satz von Instanzvariablen. Instanzvariable können bei ihrer Definition auch initialisiert werden. Geschieht dies nicht, so werden Sie beim Erzeugen des Objektes mit Standardwerten vorbelegt (einfache numerische Variable mit 0, boolsche Variable mit `false` und Objektreferenzen mit `null`). Die Verwendung von Instanzvariablen ist an ein Objekt gebunden, d.h. eine Instanzvariable kann nur über ein bestehendes Objekt angesprochen werden.

Beispiel für eine Klasse mit Instanzvariablen:

```
1 class InstanceVars {
2     private int i;           // wird mit 0 vorbelegt
3     private double d = 1.2;
4     private Button b;       // wird mit null vorbelegt
5     private String s = new String("Java");
6 }
```

Listing 4.5: Instanzvariable

Klassenvariable

Im Gegensatz zu Instanzvariablen existiert eine Klassenvariable nicht pro Instanz sondern genau einmal für die gesamte Klasse. Sie beschreibt also nicht den Zustand eines Objekts sondern kann zur Speicherung von Daten verwendet werden, die die gesamte Klasse betreffen. Klassenvariable werden mit dem Modifier `static` definiert. Sie können von außerhalb der Klasse (erlaubter Zugriff vorausgesetzt) über

`Klassenname.Klassenvariable`

angesprochen werden. Mögliche Einsatzgebiete von Klassenvariablen sind:

- Mit Hilfe von Klassenvariablen können Konstante definiert werden. Zu diesem Zweck ergänzt man die Definition mit dem Modifier `final`. Datenfelder mit diesem Modifier können nicht verändert werden, entsprechen also Konstanten in anderen Programmiersprachen. So etwa definiert die Klasse `java.lang.Math` zwei öffentliche finale statische Datenfelder vom Typ `double`, die der Kreiszahl π und der Eulerschen Zahl e möglichst nahe kommen:

```
public static final double E = 2.718281828459045;
public static final double PI = 3.141592653589793;
```

- Mit Hilfe von Klassenvariablen können die einzelnen Objekte einer Klasse miteinander kommunizieren. Verändert eine Instanz der Klasse den Wert einer Klassenvariable, so steht diese Änderung sofort allen anderen Objekten zur Verfügung.
- Ein weiteres Beispiel für die Verwendung einer Klassenvariablen besteht darin, einen Instanzenzähler in eine Klasse einzubauen. Hierzu wird eine Klassenvariable verwendet, die beim Erzeugen eines Objekts (also in jedem Konstruktor) inkrementiert und beim Zerstören (also im Destruktor) dekrementiert wird.

Klassenvariable sind nicht an die Existenz eines Objektes gebunden, ihre Lebensdauer hängt vom Ladezustand der Klasse ab. Wie Instanzvariable werden sie, sofern bei der Definition nichts Gegenteiliges geschieht, mit Standardwerten vorbelegt.

Beispiel für eine Klassendefinition mit Klassenvariablen (es ist üblich, finale Klassenvariable mit Großbuchstaben zu bezeichnen):

```
1 class StaticVars {
2     private static int i; // i wird mit 0 vorbelegt
3     public static final double SCHILLINGKURS = 13.7603;
4 }
```

Listing 4.6: Klassenvariable

4.2.9 Methoden als Klassenmitglieder

Innerhalb einer Klasse können neben Konstruktoren zwei Typen von Methoden definiert werden:

- Instanzmethoden
- Klassenmethoden

Instanzmethoden

Instanzmethoden operieren auf den Daten eines Objekts, d.h. sie lesen oder verändern Instanzvariable. Greift eine Methode nicht auf Instanzvariable und/oder Instanzmethoden zu, so sollte sie als Klassenmethode implementiert werden.

Instanzmethoden können nur über ein Objekt (oder eine Objektreferenz) aufgerufen werden. Eine Referenz auf das aufrufende Objekt wird versteckt an die Instanzmethode übergeben und heißt innerhalb der Instanzmethode `this`. Das aufrufende Objekt kann innerhalb der Instanzmethode also explizit mit Hilfe der `this`-Referenz angesprochen werden.

Klassenmethoden

Neben Instanzmethoden gibt es auch Klassenmethoden, d.h. Methoden, die unabhängig von einer bestimmten Instanz verwendet werden können. Klassenmethoden werden mit Hilfe des Modifiers `static` definiert und wie Klassenvariable durch Voranstellen des Klassennamens aufgerufen.

Da Klassenmethoden unabhängig von konkreten Instanzen ihrer Klasse existieren, ist ein Zugriff auf Instanzvariable nicht möglich. Der technische Grund dafür ist, dass Klassenmethoden keine (versteckte) `this`-Referenz übergeben bekommen.

Daraus ergibt sich, dass von Klassenmethoden aus (etwa der Einsprungmethode `main`) keine Instanzvariablen oder Instanzmethoden der Klasse direkt angesprochen werden können. Dieses direkte Ansprechen setzt die Existenz der `this`-Referenz voraus, die es in Klassenmethoden aber nicht gibt. Der Compiler quittiert einen solchen Versuch etwa mit der Fehlermeldung:

```
non-static variable j cannot be referenced from a static context
```

Die obige Fehlermeldung wurde von folgendem Programm erzeugt:

```

1 public class TestClass {
2     private int j = 10;           // Instanzvariable
3     private static int k = 20;    // Klassenvariable
4
5     public static void main(String []args) {
6         changeVars1();           // richtig, changeVars1() ist static
7         TestClass.changeVars1(); // richtig und besser lesbar
8         changeVars2();           // falsch, in main keine this-Referenz
9         new TestClass().changeVars2(); // richtig
10    }
11
12    private static void changeVars1() {
13        j = 30; // Erzeugt Compilerfehler, keine this-Referenz
14        k = 30; // Richtig, k ist eine Klassenvariable
15        TestClass o = new TestClass();
16        o.j = 30; // richtig
17        o.k = 40; // auch richtig (leider)
18        TestClass.k = 50; // richtig und gut lesbar
19    }
20
21    private void changeVars2() {
22        j = 30; // richtig, this.j = 30
23        k = 30; // Richtig, k ist immer sichtbar
24        TestClass o = new TestClass();
25        o.j = 30; // richtig
26        o.k = 40; // auch richtig (leider)
27        TestClass.k = 50; // richtig und gut lesbar
28    }
29 }

```

Listing 4.7: Zugriff aus statischem Kontext

Wie obiges Beispiel zeigt, können aus einem statischen Kontext keine Instanzelemente der Klasse direkt angesprochen werden. In einem nicht statischen Kontext existiert die `this`-Referenz und es können Instanzelemente implizit über diese `this`-Referenz verwendet werden.

4.2.10 Statische Initialisierer

Ein statischer Initialisierer ist ein statischer Block innerhalb einer Klasse, der beim Laden der Klasse genau einmal aufgerufen wird. Eine Klasse kann mehrere statische Initialisierer haben. Die statischen Initialisierer werden in der Reihenfolge ihrer Codierung abgearbeitet. In einem statischen Initialisierer kann nur auf Klassenvariable zugegriffen werden, die vor dem Initialisierer definiert wurden. Das folgende Beispiel demonstriert die Zusammenhänge:

```

1 public class StaticInitializer {
2     private static int i = 10;
3     private int a = 20;
4
5     static {

```

```

6      i++;
7      // a++; nicht moeglich, a nicht statisch
8      // j++; nicht moeglich, illegale forward-Referenz
9      System.out.format("Stat. Initialsierer 1: i = %d\n", i);
10   }
11
12   private static int j = 30;
13
14   static {
15       i++;
16       // a++; nicht moeglich, a nicht statisch
17       j++;
18       System.out.format("Stat. Initialsierer 2: i = %d, j = %d\n", i, j);
19   }
20
21   public static void main(String[] args) {
22       i++;
23       j++;
24       System.out.format("Main: i = %d, j = %d\n", i, j);
25   }
26 }

```

Listing 4.8: Statische Initialisierer

Ausgabe:

```

Stat. Initialsierer 1: i = 11
Stat. Initialsierer 2: i = 12, j = 31
Main: i = 13, j = 32

```

4.2.11 UML-Klassendiagramme

UML³ - Klassendiagramme stellen ein grafisches Hilfsmittel zur Beschreibung von Objekten und deren Abhängigkeiten dar. Klassendiagramme sind nur ein Teil der UML, weitere Teile sind z.B. Use-Case-Diagramme und Zeit-Sequenzdiagramme.

UML-Diagramme stellen keine Implementierung dar, sondern nur eine allgemeine Sicht auf die Objektmodelle. Die Notation UML ist programmiersprachenunabhängig.

Ein UML-Klassendiagramm gehorcht folgenden Regeln:

- Klassen/Interfaces⁴ werden im UML-Klassendiagramm als Rechtecke mit einem Namen abgebildet. Unterhalb des Klassennamens werden Membervariable beschrieben, darunter die Methoden mit Parameterliste und Ergebnistyp.
- Variablenbeschreibungen haben die folgende Syntax:
`<Sichtbarkeit> <Name> : <Typ>`
- Methodenbeschreibungen haben die folgende Syntax:
`<Sichtbarkeit> <Name> (<Parameterliste>) : <Ergebnistyp>`
- Die Sichtbarkeit wird mit folgenden Zeichen abgekürzt:
+ (public), # (protected), kein Zeichen (default), - (private)
- Statische Elemente werden unterstrichen.
- Abstrakte Klassen und Methoden⁵ schreibt man kursiv.

Wie exakt ein UML-Diagramm spezifiziert ist, hängt von der konkreten Aufgabenstellung ab. Im folgenden Beispiel ist die Detailtiefe sehr hoch, um alle Möglichkeiten zu demonstrieren. Oft beschränkt man sich auf die Darstellung von Kernmethoden, die für das Verhalten eines Objektes wesentlich sind.

UML-Diagramm zu einer Klasse Counter

Im Folgenden werden das UML-Diagramm und die zugehörige Implementierung einer Klasse `Counter` zur Beschreibung eines einfachen Zählers angegeben:

³Unified Modelling Language

⁴werden später behandelt

⁵werden später behandelt

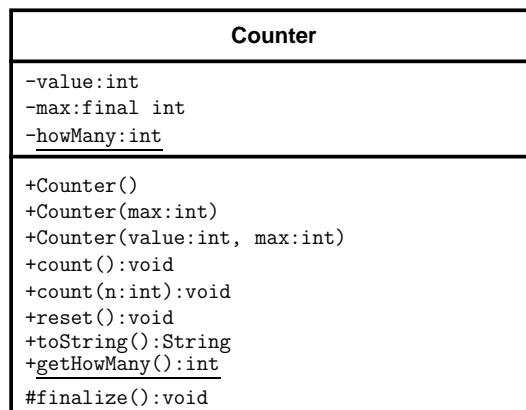


Abbildung 4.1: UML-Diagramm Counter

```

1 public class Counter {
2     private int value;                // Zaehlerstand
3     private final int max;            // Maximalstand
4     private static int howMany        ;    // Instanzzaehler
5
6     public Counter () {                // Standardkonstruktor
7         this(0, 99);                  // Konstruktorenverkettung
8     }
9
10    public Counter (int max) {          // Einparametriger Konstruktor
11        this(0, max);                  // Konstruktorenverkettung
12    }
13
14    public Counter (int value, int max) { // Zweiparamtriger Konstruktor
15        this.value = value;
16        this.max = max;
17        Counter.howMany++;             // Instanzzaehler erhoeihen
18    }
19
20    public void count() {                // Instanzmethode count
21        this.value++;
22        if(this.value > this.max)
23            this.reset();
24    }
25
26    public void count(int n) {
27        while(n-- > 0)
28            this.count();
29    }
30
31    public void reset() {                // Instanzmethode reset
32        this.value = 0;
33    }
34
35    @Override
36    public String toString() {           // Zugriffsmethode fuer value
37        return String.format("%04d", this.value);
38    }
39
40    public static int getHowMany() {     // Zugriffsmethode fuer
41        return Counter.howMany;         // Instanzzaehler
42    }
43
44    @Override
45    protected void finalize() {         // Destruktor
46        Counter.howMany--;              // Instanzzaehler vermindern
47    }
48 }

```

Listing 4.9: Die Klasse Counter

Bemerkung:

Die finale Instanzvariable `max` erhält bei ihrer Definition in Zeile 4 den Defaultwert 0 und darf daher in den Konstruktoren noch gesetzt werden. Initialisiert man sie in Zeile 4 explizit, so darf sie in den Konstruktoren nicht mehr gesetzt werden (in diesem Fall würde man in Zeile 16 einen Compilerfehler erhalten).

4.3 Vererbung

4.3.1 Grundlegende Konzepte

Vererbung⁶ bedeutet, dass eine Klasse die Mitglieder einer anderen Klasse erben kann. Die Vererbung ist das wichtigste Sprachelement der OOP. Jene Klasse deren Eigenschaften vererbt werden, heißt Basisklasse oder Superklasse (baseclass, superclass) und jene Klasse, welche die Eigenschaften der Superklasse erbt, heißt abgeleitete Klasse oder Subklasse (derived class).

Java unterstützt nur die einfache Vererbung, bei der jede Subklasse nur genau eine unmittelbare Superklasse haben kann. Ältere objektorientierte Programmiersprachen, wie z.B. C++ unterstützen auch Mehrfachvererbung, bei der eine Klasse auch mehrere unmittelbare Basisklassen haben kann. Um die Einschränkungen in den Designmöglichkeiten, die durch Ausschluss der Mehrfachvererbung entstehen, zu vermeiden, wurde mit Hilfe der Interfaces eine neue, restriktive Art der Mehrfachvererbung eingeführt.

4.3.2 Ableiten einer Klasse, die "is-a"-Beziehung

Um eine neue Klasse aus einer bestehenden abzuleiten, ist im Kopf der Klasse mit Hilfe des Schlüsselwortes `extends` ein Verweis auf die Basisklasse anzugeben. Dadurch erbt die abgeleitete Klasse mit Ausnahme der Konstruktoren alle Mitglieder der Basisklasse, d.h. alle Variablen und alle Methoden. Durch Hinzufügen neuer Elemente oder überschreiben der geerbten Methoden kann die Funktionalität der abgeleiteten Klasse erweitert werden.

Wird bei der Definition einer neuen Klasse keine Superklasse angegeben, so erbt die neue Klasse automatisch von `java.lang.Object`. Es gibt also keine freien Klassen in Java, jede Klasse hat zumindest `Object` als Superklasse.

Nicht jede Klasse darf zur Ableitung neuer Klassen verwendet werden. Besitzt eine Klasse den Modifizier `final`, ist es nicht erlaubt, eine neue Klasse aus ihr abzuleiten.

Beispiel

Im Folgenden wird die durch das angegebenen UML-Diagramm gegebene Vererbungshierarchie implementiert. Vererbung wird in UML-Klassendiagrammen mit Hilfe eines durchgezogenen Pfeiles veranschaulicht.

Listing

```
1 class BaseClass { // erbt von java.lang.Object
2     private int x, y;
3
4     public void setAll(int x, int y) {
5         this.x = x;
6         this.y = y;
7     }
8
9     public int getX() { return x; }
10    public int getY() { return y; }
11
12    @Override
13    public String toString() { return "x: " + x + " y: " + y; }
14 }
```

Listing 4.10: BaseClass.java Basisklasse

Listing

⁶eng. inheritance

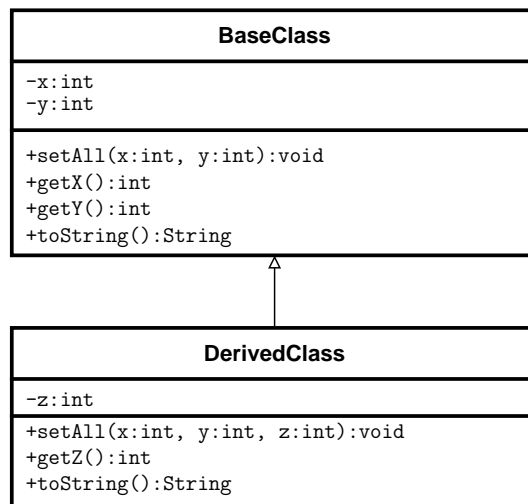


Abbildung 4.2: UML-Diagramm Vererbung

```

1 class DerivedClass extends BaseClass { // erbt von BaseClass
2     private int z; // + 2 weitere Datenmitglieder x,y von BaseClass geerbt
3
4     public void setAll(int x, int y, int z) {
5         this.setAll(x, y); // Aufruf der geerbten Methode setXY
6         this.z = z;
7     }
8
9     public int getZ() {
10        return this.z;
11    }
12
13    @Override
14    public String toString() { // ueberschreibt toString() aus BaseClass
15        return "x: " + this.getX() + " y: " + this.getY() + " z: " + z;
16    }
17 }

```

Listing 4.11: DerivedClass.java Abgeleitete Klasse

Listing

```

1 public class MainClass {
2     public static void main(String args[]) {
3         BaseClass b = new BaseClass();
4         DerivedClass d = new DerivedClass();
5         b.setAll(3,5);
6         d.setAll(3,5); // erlaubt, weil setAll() in BaseClass und
7                       // damit auch in DerivedClass public
8         System.out.println("b >> " + b);
9         System.out.println("d >> " + d);
10        d.setAll(1,2,3);
11        System.out.println("d >> " + d);
12        //-----
13        b = new DerivedClass(); // b referenziert ein DerivedClass Objekt !!
14        b.setAll(10,20); // d.h. hier besteht Typvertraeglichkeit
15        System.out.println("b >> " + b.toString()); // Polymorphie
16    }
17 }

```

Listing 4.12: MainClass.java Applikation

Ausgabe:

```

b >> x: 3 y: 5
d >> x: 3 y: 5 z: 0
d >> x: 1 y: 2 z: 3

```

```
b >> x: 10  y: 20  z: 0
```

Typverträglichkeit, die "is-a" - Beziehung

Wie das obige Beispiel zeigt (Zeile 13) kann ein Objekt der abgeleiteten Klasse `DerivedClass` über die Superklassenreferenz `b` vom Compilertyp `BaseClass` gespeichert werden. Dies ist möglich, weil in der Klassenhierarchie die "is-a"-Beziehung im Sinne der Spezialisierung gilt. Jedes Subklassenobjekt hat alle Eigenschaften der Basisklasse und ist daher ein spezielles Basis-klassenobjekt. Darum ist es auch mit dem Typ der Basisklasse verträglich. Ein Objekt der Klasse `DerivedClass` ist also auch vom Typ `BaseClass` und vom Typ `java.lang.Object`.

Innerhalb der Klassenhierarchie gilt Typverträglichkeit nach folgenden Regeln:

- Ein Subklassentyp kann ohne weiteres (d.h. ohne explizite Typumwandlung) über einen Basisklasstyp gespeichert werden. Insbesondere kann also jedes beliebige Javaobjekt über eine Referenz vom Typ `java.lang.Object` gespeichert werden.
- Ein Superklassentyp kann nach explizitem Typecast über einen Subklassentyp gespeichert werden. Für den Compiler ist diese Typumwandlung in Ordnung, da in dem Superklassentyp ein Objekt der Subklasse gespeichert sein könnte. Ob dies tatsächlich der Fall ist, kann allerdings erst zur Laufzeit geprüft werden. Stellt das Laufzeitsystem fest, dass die "is-a"-Beziehung nicht gegeben ist, so wirft die VM eine `ClassCastException`.
- Typen, die nicht in einer Vererbungslinie liegen, können nie die "is-a"-Beziehung erfüllen und daher verhindert bereits der Compiler hier jede Zuweisung, auch wenn sie mit explizitem Typecast versehen wurde.

Bei der Analyse solcher Zusammenhänge ist immer zweistufig vorzugehen:

1. Es ist zu prüfen, ob die Compilertypen der beteiligten Referenzen und Objekte die oben angegebenen Regeln erfüllen. Bei einem Verstoß gegen eine dieser Regeln entsteht ein Compilerfehler.
2. Es ist zu prüfen, ob für jeden verwendeten Typecast zur Laufzeit die notwendige "is-a"-Beziehung gegeben ist. Ist dies nicht der Fall, so wirft die VM eine `ClassCastException`.

Das folgende Beispiel demonstriert an Hand der gegebenen Klassenhierarchie die beschriebenen Zusammenhänge:

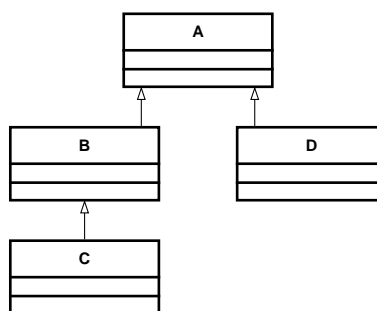


Abbildung 4.3: UML-Diagramm IsA - Beziehung

```

1 class A {}
2 class B extends A {}
3 class C extends B {}
4 class D extends A {}
5
6 public class IsA_Demo {
7     public static void main(String []args) {
8         A a = null;
9         B b = null;
10        C c = null;
11        D d = null;
  
```

```

12 //----- Compiler -----
13 a = b;    // fuer den Compiler ok B "is-a" A
14 b = a;    // fuer den Compiler falsch A "is-not-a" B
15 b = (B)a; // fuer den Compiler ok, in a koennte ein B
16           // gespeichert sein
17 c = (C)d; // mit und ohne Cast falsch, d kann nie ein C
18           // referenzieren
19 //----- Laufzeitsystem -----
20 a = new C(); // zur Laufzeit ok, C "is-a" A
21 d = (D)a;    // fuer den Compiler ok, a koennte ein D
22           // referenzieren
23           // zur Laufzeit falsch, a referenziert ein C und
24           // ein C "is-not-a" D
25 b = (B)a;    // fuer den Compiler ok, a koennte ein B
26           // referenzieren; auch zur Laufzeit ok,
27           // a referenziert ein C und ein C "is-a" B
28 }
29 }

```

Listing 4.13: Is-A Beziehung

Zum besseren Verständnis des oben beschriebenen Sachverhaltes sind folgende Begriffe dienlich:

- Compilertyp, Laufzeittyp
- Upcast, Downcast

Compiler- und Laufzeittyp

Referenzen sind **Compilertypen**. Der Compilertyp einer Referenz wird bei der Definition der Referenz festgelegt und ändert sich nie:

```
A a; // Die Referenz a hat den Compilertyp A
```

Während der Lebensdauer von `a` können der Referenz Objekte verschiedener Typen zugewiesen werden. Das gerade zugewiesene Objekt legt den **Laufzeittyp** von `a` fest:

```
a = new A(); // Laufzeittyp von a ist jetzt A
a = new D(); // Laufzeittyp von a ist jetzt D
```

Hinweis:

Der Compiler arbeitet immer mit Compilertypen. Im Codeabschnitt

```
A a;
a = new B();
a.method();
```

sucht der Compiler die Methode `method()` in der Klasse `A`, da `a` den Compilertyp `A` hat.

Upcast- und Downcast

Speichert man in einem Superklassentyp einen Subklassentyp, so spricht man von **Upcast**. Dieser ist immer ohne explizitem Cast möglich und funktioniert zur Laufzeit sicher.

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

Weist man einem Subklassentyp einen Superklassentyp zu, so spricht man von **Downcast**. Hier ist immer ein expliziter Cast notwendig und es kann erst zur Laufzeit festgestellt werden, ob der Cast tatsächlich möglich ist. Ist der Laufzeittyp von `a` z.B. ein Objekt vom Typ `D`, so wirft die JVM eine `ClassCastException`, da ein `D` im Rahmen der "is-a"-Beziehung nicht vom Typ `B` ist.

```
A a;
B b;
a = new B();
b = (B) a; // kein Problem
a = new D();
b = (B) a; // ClassCastException
```

4.3.3 Konstruktoren und Vererbung

Konstruktoren sind die einzigen Methoden, die nicht vererbt werden. Das bedeutet, dass jede Klasse in der Klassenhierarchie ihre eigenen Konstruktoren hat.

Bei der Erzeugung eines Objektes mit Hilfe des Operators `new` wird der zur angegebenen Parametrisierung passende Konstruktor aufgerufen. Dieser ruft, wenn er nicht über Konstruktorenverkettung einen anderen Konstruktor der eigenen Klasse aufruft, einen Konstruktor der Superklasse auf. Dieser Aufruf kann entweder explizit oder implizit erfolgen.

Falls als erste Anweisung innerhalb eines Konstruktors die Anweisung

```
super([Parameterliste]);
```

steht, wird dies als Aufruf des Superklassenkonstruktors interpretiert. Der Aufruf muss natürlich zu einem in der Superklasse definierten Konstruktor passen, sonst erhält man einen Compilerfehler.

Falls als erste Anweisung im Konstruktor weder über

```
super([Parameterliste]);
```

ein Superklassenkonstruktor explizit aufgerufen wird noch mit Hilfe von

```
this([Parameterliste]);
```

Konstruktorenverkettung durchgeführt wird, so setzt der Compiler an dieser Stelle den impliziten Aufruf `super();` ein und ruft damit den parameterlosen Konstruktor der Superklasse auf. Falls ein solcher Konstruktor in der Superklasse nicht existiert, entsteht ein Compilerfehler.

Durch diese Regel wird bei jeder Instanziierung eines Objekts eine ganze Kette von Konstruktoren aufgerufen. Jeder Konstruktor ruft, bevor sein eigentlicher Methodenrumpf abgearbeitet wird einen Superklassenkonstruktor auf. Die Konstruktoren arbeiten in der Vererbungshierarchie logisch also von oben nach unten. Zuerst wird der Konstruktor der Klasse `java.lang.Object` ausgeführt, dann jener der darunterliegenden Klasse usw., bis zuletzt der Konstruktor der zu instanzierenden Klasse ausgeführt wird.

Im Gegensatz zu den Konstruktoren werden die Destruktoren eines Ableitungszweiges nicht automatisch verkettet. Falls ein Aufruf des Destruktors der Superklasse erforderlich ist, kann er mit Hilfe der Anweisung `super.finalize()` erzwungen werden.

Beispiel

Das folgende Beispiel veranschaulicht die oben vorgestellten Regeln:

```

1 class BaseClass_1 {
2     protected int x, y;
3     public BaseClass_1() {
4         System.out.println("Parameterloser Konstruktor von BaseClass_1");
5     }
6
7     public BaseClass_1(int x, int y) {
8         this.x = x;
9         this.y = y;
10        System.out.println("Zweiparametriger Konstruktor von BaseClass_1");
11    }
12
13    @Override
14    public String toString() {
15        return "x: " + x + " y: " + y;
16    }
17 }
18
19 class DeriveClass_1 extends BaseClass_1 {
20     protected int z;
21
22     public DeriveClass_1() {
23         System.out.println("Parameterloser Konstruktor von DeriveClass_1");

```

```

24 }
25
26 public DeriveClass_1 (int x, int y, int z) {
27     super(x, y);    // expliziter Aufruf des Superklassenkonstruktors
28     this.z = z;
29     System.out.println("Dreiparametriger Konstruktor von DeriveClass_1");
30 }
31
32 @Override
33 public String toString() {
34     return "x: " + x + " y: " + y + " z: " + z;
35 }
36 }
37
38 public class MainClass_1 {
39     public static void main(String args[]) {
40         DeriveClass_1 d1 = new DeriveClass_1();
41         DeriveClass_1 d2 = new DeriveClass_1(1,2,3);
42
43         System.out.println("d1 >> " + d1);
44         System.out.println("d2 >> " + d2);
45     }
46 }

```

Listing 4.14: Konstruktoren und Vererbung

Ausgabe:

```

Parameterloser Konstruktor von BaseClass_1
Parameterloser Konstruktor von DeriveClass_1
Zweiparametriger Konstruktor von BaseClass_1
Dreiparametriger Konstruktor von DeriveClass_1
d1 >> x: 0 y: 0 z: 0
d2 >> x: 1 y: 2 z: 3

```

4.3.4 Instanzinitialisierer

Neben statischen Initialisierern gibt es auch Instanzinitialisierer. Diese werden als Blöcke innerhalb der Klasse ohne jeden Modifier implementiert. Es kann in einer Klasse mehrere Instanzinitialisierer geben, sie werden in der Reihenfolge ihrer Programmierung jedesmal, wenn ein neues Objekt der Klasse instanziiert wird, abgearbeitet. In einem Instanzinitialisierer hat man Zugriff auf alle Datenfelder (statisch und nicht statisch) der Klasse, die vor dem Instanzianialisierer bereits definiert wurden.

Konkret arbeiten die Instanzinitialisierer genau dann, wenn alle Superklassenkonstruktoren bereits abgearbeitet sind und bevor die erste echte Anweisung im aktuellen Konstruktor bearbeitet wird.

Beispiel:

```

1 public class InstanceInitializer {
2     private int i = 10;
3     private static int k = 20;
4     private int j = 30;
5
6     { // Instanzinitialisierer 1
7         inc();
8         System.out.format("Instanzinitialiserer 1: i = %d, j = %d, k = %d\n", i, j, k);
9     }
10
11     private void inc() {
12         i++; j++; k++;
13     }
14
15     { // Instanzinitialisierer 2
16         inc();
17         System.out.format("Instanzinitialiserer 2: i = %d, j = %d, k = %d\n", i, j, k);
18     }
19
20     private InstanceInitializer() {
21         System.out.println("Konstruktor");
22     }
23 }

```

```

24 public static void main(String... args) {
25     new InstanceInitializer();
26     new InstanceInitializer();
27 }
28 }

```

Listing 4.15: Instanz - Initialisierer

Ausgabe:

```

Instanzinitialisierer 1: i = 11, j = 31, k = 21
Instanzinitialisierer 2: i = 12, j = 32, k = 22
Konstruktor
Instanzinitialisierer 1: i = 11, j = 31, k = 23
Instanzinitialisierer 2: i = 12, j = 32, k = 24
Konstruktor

```

4.3.5 Function Overriding

Neben den Membervariablen erbt eine Klasse auch die Methoden ihrer Superklasse. Diese von der Superklasse geerbten Methoden dürfen in der Subklasse neu definiert werden. Diese Technik heißt Überschreiben von Funktionen oder Function-Overriding.

Wird eine Superklassenmethode `foo()` in einer Subklasse überschrieben, so verdeckt die Methode `foo()` der Subklasse jene der Superklasse. Aufrufe von `foo()` in der Subklasse beziehen sich immer auf die überschriebene Variante. Oft ist es allerdings nützlich, aus einer Methode der Subklasse die verdeckte Superklassenmethode aufrufen zu können. Dies ist mit Hilfe von `super.foo()` möglich. Der verkettete Aufruf von Superklassenmethoden (wie z.B. `super.super.foo()`) ist nicht erlaubt.

In folgendem Beispiel überschreibt `foo()` in B die aus A geerbte Methode `foo()`:

```

1 class A {
2     public char foo() { return 'A'; }
3 }
4
5 class B extends A {
6     @Override
7     public char foo() { return 'B'; } // Überschreibt A.foo()
8
9     public char foo(boolean b) { // Ueberladet B.foo()
10         return b ? foo() : super.foo();
11     }
12 }
13
14 public class Overriding {
15     public static void main(String []args) {
16         A a = new A();
17         B b = new B();
18
19         System.out.format("a.foo() liefert %c\n", a.foo());
20         System.out.format("b.foo() liefert %c\n", b.foo());
21         System.out.format("b.foo(true) liefert %c\n", b.foo(true));
22         System.out.format("b.foo(false) liefert %c\n", b.foo(false));
23     }
24 }

```

Listing 4.16: Function Overriding

Ausgabe:

```

a.foo() liefert A
b.foo() liefert B
b.foo(true) liefert B
b.foo(false) liefert A

```

Damit overriding vorliegt, muss die Signatur (d.h. Funktionsname und Parameterliste) exakt mit jener der Superklassenmethode übereinstimmen.

Beim Überschreiben von Methoden sind die folgenden Regeln zu beachten:

- Der Rückgabotyp muss mit jenem der Superklassenmethode exakt übereinstimmen. Seit Java 1.5 sind hier auch sog. Covariant>Returns erlaubt, d.h. bei Referenztypen darf der Rückgabotyp der Subklassenmethode im Rahmen der "is-a" Beziehung ein speziellerer Typ als jener der Superklassenmethode sein.
- Methoden mit dem Modifier `final` können nicht überschrieben werden.
Ausnahme: ist die Superklassenmethode `final` und `private`, so ist sie in der Subklasse unsichtbar und darf mit gleicher Signatur neu definiert werden. Man spricht in diesem Fall aber nicht von overriding.
- Die Subklassenmethode darf keinen restriktiveren Zugriff als die Superklassenmethode haben. In diesem Sinn gilt:

`public > protected > default > private`

- Die Subklassenmethode darf nicht andere oder generellere checked Exceptions⁷ werfen als die Superklassenmethode.
- Klassenmethoden dürfen nur von Klassenmethoden, Instanzmethoden nur von Instanzmethoden überschrieben werden.

4.3.6 Polymorphie

Dynamische Methodensuche, Polymorphie

Da eine Superklassenreferenz zuweisungskompatibel zu Objekten aller Subklassen ist, kann der Compiler oft nicht entscheiden, welche Variante einer überschriebenen Methode er aufrufen soll. Er muss also Code generieren, der dies erst zur Laufzeit entscheidet. Man spricht vom dynamischen Binden bzw. von der dynamischen Methodensuche (im Gegensatz zur statischen Methodensuche, bei der die für einen Funktionsaufruf notwendige Einsprungadresse bereits zur Compilierungszeit feststeht). Das folgende Beispiel demonstriert die Zusammenhänge:

```

1 class ClassA {
2     public int foo() { return 1; }
3 }
4
5 class ClassB extends ClassA {
6     @Override
7     public int foo() { return 2; }
8 }
9
10 public class TestClass {
11     public static void main(String []args) {
12         ClassA a;
13         int i;
14         a = new ClassA();
15         i = a.foo();
16         System.out.format("Wert von i: %d\n", i);
17         a = new ClassB();
18         i = a.foo();
19         System.out.format("Wert von i: %d\n", i);
20     }
21 }

```

Listing 4.17: Dynamisches Binden

Ausgabe:

```

Wert von i: 1
Wert von i: 2

```

Obiges Beispiel demonstriert Polymorphie. In den beiden identischen Zeilen 14 und 17 liefern die Aufrufe von `a.foo()` unterschiedliche Ergebnisse. In Zeile 14 verweist die Referenz `a` auf eine Instanz der Klasse `ClassA` und daher ruft das Laufzeitsystem die Methode `foo()` der Klasse `ClassA` auf (Ergebnis 1). In Zeile 17 verweist `a` hingegen auf ein Objekt der Klasse `ClassB`, wodurch nun die Methode `foo()`

⁷vgl. Abschnitt über Exceptionhandling

der Klasse `ClassB` mit dem Ergebnis 2 aufgerufen wird. Welche Version der überschriebenen Methoden `foo()` in einer bestimmten Situation wirklich aufgerufen wird, entscheidet das System erst zur Laufzeit an Hand der Tatsache, welcher Typ aktuell über die Referenz `a` gespeichert ist.

In vielen objektorientierten Programmiersprachen muss man dieses Verhalten (dynamische Bindung) durch unterschiedliche Techniken erst ermöglichen, in Java werden Methodenaufrufe grundsätzlich dynamisch interpretiert.

Es gibt allerdings 3 Situationen, in denen die dynamische Methodensuche nicht arbeitet. In diesen Situationen werden die Methoden nicht über den Laufzeittyp, sondern über den Compilertyp aufgerufen. Daher kommt hier statische Bindung zum Einsatz:

- Methoden mit Zugriffsmodifizier `private` sind in abgeleiteten Klassen nicht sichtbar und können daher nicht polymorph aufgerufen werden.
- Bei (nicht privaten) Methoden mit Modifizier `final` wird explizit festgelegt, dass sie nicht überschrieben werden können. Damit kann eine finale Methode nicht polymorph aufgerufen werden.
- Auch Klassenmethoden, die ja unabhängig von einer Instanz aufgerufen werden, werden nicht über den Laufzeittyp, sondern über den Compilertyp aufgerufen.

Wesentlich ist auch, dass Instanzvariable stets über den Compilertyp aufgerufen werden.

Das folgende Beispiel demonstriert die oben beschriebenen Sachverhalte. Bei der nicht privaten Instanzmethode `foo1()` kommt Polymorphie zum Tragen, alle anderen Elemente werden wie oben beschrieben über den Compilertyp aufgerufen.

```

1 class A {
2     public int x = 10;
3     public void foo1() {
4         System.out.println("foo1 von A");
5     }
6     public static void foo2() {
7         System.out.println("foo2 von A");
8     }
9     public final void foo3() {
10        System.out.println("foo3 von A");
11    }
12    private void foo4() {
13        System.out.println("foo4 von A");
14    }
15    public void testFoo4() {
16        A a = new A();
17        a.foo4();
18        a = new B();
19        a.foo4();
20    }
21 }
22
23 class B extends A {
24     public int x = 20;
25     public void foo1() {
26         System.out.println("foo1 von B");
27     }
28     public static void foo2() {
29         System.out.println("foo2 von B");
30     }
31     //foo3 in A final, kann nicht ueberschrieben werden
32     private void foo4() {
33         System.out.println("foo4 von B");
34     }
35 }
36
37 public class Polymorphie {
38     public static void main(String []args) {
39         A a = new A();
40         // Compilertyp von a ist A
41         // Laufzeittyp von a ist A
42         System.out.println(a.x);
43         a.foo1();
44         a.foo2();
45         a.foo3();
46         a.testFoo4();

```

```

47
48     a = new B();
49     // Compilertyp von a ist A
50     // Laufzeittyp von a ist B
51     System.out.println(a.x);
52     a.foo1();
53     a.foo2();
54     a.foo3();
55     a.testFoo4();
56 }
57 }

```

Listing 4.18: Verhindern von Polymorphie

```

10
foo1 von A
foo2 von A
foo3 von A
foo4 von A
foo4 von A
10
foo1 von B
foo2 von A
foo3 von A
foo4 von A
foo4 von A

```

4.4 Abstrakte Methoden und abstrakte Klassen

4.4.1 Abstrakte Methoden

Fallweise ist es in einer Basisklasse nicht sinnvoll, eine Methode auszuprogrammieren, da die eigentliche Funktionalität erst in den abgeleiteten Klassen feststeht. Oft benötigt man die Methode aber bereits in der Basisklasse, da sie polymorph über eine Referenz der Basisklasse aufgerufen werden soll. Eine Lösung wäre, die Methode in der Basisklasse mit leerem Methodenrumpf auszuprogrammieren. Die bessere Technik ist, diese Methode in der Basisklasse abstrakt zu definieren.

Im Gegensatz zu konkreten Methoden enthält eine abstrakte Methode nur die Deklaration des Methodenkopfes, aber keine Implementierung des Methodenrumpfes. Anstelle des Funktionsrumpfes steht lediglich ein Semikolon. Zusätzlich wird die Definition mit dem Modifier `abstract` versehen. Beispiel:

```
public abstract void foo();
```

Abstrakte Methoden definieren nur eine Schnittstelle, die durch Überschreiben in einer abgeleiteten Klasse konkretisiert werden kann. Das abstrakte Methoden nur im Zusammenhang mit polymorphen Methodenaufrufen Sinn machen, verhält sich der Modifier `abstract` nicht mit `private`, `final` und `static`.

4.4.2 Abstrakte Klassen

Eine Klasse, die mindestens eine abstrakte Methode enthält, ist selbst abstrakt und muss ebenfalls mit dem Schlüsselwort `abstract` definiert werden. Abstrakte Klassen können nicht instanziiert werden. Statt dessen stellen sie Superklassen dar, die ihre Subklassen zum Konkretisieren der abstrakten Methoden zwingen. Sollte eine Subklasse nicht alle abstrakten Methoden der Superklasse konkretisieren, bleibt sie abstrakt und kann daher auch nicht instanziiert werden. Die Ableitung einer abstrakten Klasse wird erst dann konkret, wenn alle ihre abstrakten Methoden konkretisiert sind. Die Konkretisierung kann also auch schrittweise über mehrere Vererbungsstufen erfolgen.

Beim Arbeiten mit abstrakten Methoden und Klassen sind die folgenden Punkte zu beachten:

- Eine Klasse kann auch als abstrakt definiert werden, wenn sie keine abstrakten Methoden enthält.

- Eine konkrete Methode kann in der nächsten Vererbungsstufe wieder als abstrakt definiert werden.

Beispiel

Das folgende Beispiel demonstriert das Arbeiten mit abstrakten Methoden und Klassen:

```

1 import java.util.Random;
2
3 abstract class Base { // Abstrakte Klasse
4     protected int x;
5     public static Random rd = new Random();
6
7     public Base() {
8         this.x = (int) (Math.random() * 10.0);
9     }
10    // Abstrakte Funktion
11    public abstract String whoAmI();
12 }
13
14 class Derive_1 extends Base {
15     // Funktion wird konkret ueberschrieben
16     @Override
17     public String whoAmI() {
18         return "Klasse Derive_1 - Wert von x: " + x;
19     }
20 }
21
22 class Derive_2 extends Base {
23     // Funktion wird konkret ueberschrieben
24     @Override
25     public String whoAmI() {
26         return "Klasse Derive_2 - Wert von x: " + x;
27     }
28 }
29
30 public class AbstractDemo {
31     public static void main(String args[]) {
32         // Base b = new Base();
33         // Fehler: Base ist abstrakt und kann nicht instanziiert werden
34
35         Base []f = new Base[5]; // Array von 5 Base - Referenzen
36         for(int i = 0; i < f.length; i++) {
37             if(Base.rd.nextBoolean()) {
38                 f[i] = new Derive_1();
39             } else {
40                 f[i] = new Derive_2();
41             }
42         }
43         for(Base b : f) {
44             System.out.println(b.whoAmI()); // Polymorphie
45         }
46     }
47 }

```

Listing 4.19: Abstrakte Klassen AbstractDemo.java

Mögliche Ausgabe:

```

Klasse Derive_2 - Wert von x: 2
Klasse Derive_2 - Wert von x: 0
Klasse Derive_2 - Wert von x: 4
Klasse Derive_1 - Wert von x: 7
Klasse Derive_2 - Wert von x: 3

```

Das obige Beispiel zeigt noch einmal polymorphe Methodenaufrufe:

Den einzelnen Elementen `f[i]` des Feldes `f` vom Typ `Base_2` werden in der ersten `for`-Schleife ab Zeile 32 zufällig Objekte vom Typ `D_1` oder `D_2` zugewiesen. In der erweiterten `for`-Schleife (Zeilen 39 und 40) ruft dann jedes Objekt die Methode `whoAmI()` auf, und je nachdem, welcher Laufzeittyp wirklich hinter der `Base_2`-Referenz `f[i]` steckt, wird über dynamische Methodensuche die richtige Methode `whoAmI()` aufgerufen.

Polymorphie wird oft im Zusammenhang mit abstrakten Basisklassen eingesetzt. Das folgende Szenario beschreibt einen typischen Einsatz:

In einem Zeichenprogramm werden von einer Basisklasse `Shape` z.B. die Klassen `Line`, `Circle`, `Rectangle` usw. abgeleitet. Die Basisklasse `Shape` stellt eine Grundfunktionalität zur Verfügung, beinhaltet aber noch keine Informationen, wie die Figur konkret aussieht. Es ist daher nicht möglich, ein Objekt der Klasse `Shape` zu zeichnen. Nimmt man nun in die Klasse `Shape` eine abstrakte Methode

```
public abstract void draw();
```

auf, so erreicht man damit zwei Dinge:

1. Für den Compilertyp `Shape s` kann nun die Methode `s.draw()` aufgerufen werden, da der Compiler in `Shape` die (abstrakte) Methode `draw()` findet.
2. Durch die abstrakte Methode `draw()` in `Shape` wird man als ProgrammiererIn gezwungen, in den abgeleiteten Klassen die Methode `draw()` konkret zu überschreiben, wenn man diese abgeleiteten Klassen instanziiert will.

Nach den oben beschriebenen Vorbereitungen ist nun folgendes Design möglich: man kann nun in einem Feld von `Shape`-Referenzen verschiedene Zeichenobjekte speichern und dann das Feld über eine Schleife abarbeiten, um für jedes Zeichenobjekt die Methode `draw()` polymorph aufzurufen:

```
1 Shape []z = new Shape[3];
2 z[0] = new Line();
3 z[1] = new Circle();
4 z[2] = new Rectangle();
5 for(Shape s : z) {
6     s.draw();
7 }
```

caption=Polymorphie

4.5 Interfaces

Java verbietet die Mehrfachvererbung, d.h. eine Klasse kann nicht zwei oder mehrere direkte Superklassen haben. Die Klassendefinition

```
public class C extends A, B {}
```

führt auf einen Compilerfehler.

Jede Klasse darf aber beliebig viele Interfaces implementieren. Ein Interface wird ähnlich wie eine Klasse definiert, darf aber nur abstrakte Methoden und Konstante enthalten.

Definition eines Interfaces:

```
1 public interface I {
2     // Mitglieder
3 }
4
5 public class A implements I {
6     // Mitglieder
7 }
```

Listing 4.20: Interfacedefinition

Beim Arbeiten mit Interfaces sind die folgenden Regeln zu beachten:

- Ein Interface hat keinen Konstruktor und kann daher nicht instanziiert werden. Von einem Interface können also nur Referenzen, aber keine Objekte angelegt werden.

- In einem Interfacetyp können zur Laufzeit Objekte beliebiger Klassen gespeichert werden, die das Interface implementieren:

```
I i = new A();
```

- Ein Interface ist implizit abstrakt.

Mitglieder eines Interfaces

In einem Interface können nur abstrakte Methoden und Konstante stehen. Alle Mitglieder eines Interfaces sind implizit `public`:

- Alle Methoden eines Interfaces sind implizit `abstract` und `public`, sie sollten ohne Modifier definiert werden. Natürlich dürfen sie keinen Funktionsrumpf besitzen. Methoden eines Interfaces dürfen auch nicht `static` sein (`static` stellt einen Widerspruch zu `abstract` dar).
- Jede Klasse, die ein Interface implementiert, kann entweder alle oder nur einen Teil der in diesem Interface definierten Methoden konkretisieren. Werden nicht alle Methoden konkretisiert, so ist die Klasse selbst abstrakt und kann nicht instanziiert werden.
- Ein Interface darf auch Datenfelder enthalten. Diese sind implizit `public`, `static` und `final`, d.h. diese Modifier müssen (und sollen) nicht angegeben werden. Der Versuch, einen widersprüchlichen Modifier zu verwenden führt auf einen Compilerfehler. Konstante, die in Interfaces definiert sind, können bei voller Qualifikation ihres Namens auch von Klassen verwendet werden, die dieses Interface nicht implementieren.

Auch innerhalb von Interfaces gibt es Vererbung. Diese lässt sogar Mehrfachvererbung zu, d.h. ein Interface kann von mehreren anderen Interfaces erben. Zusammenfassend gilt also (A, B sind Klassen, X, Y und Z Interfaces):

- Eine Klasse darf von genau einer anderen Klasse erben:

```
class A extends B {}
```

- Ein Interface darf von mehreren Interfaces erben:

```
interface X extends Y, Z {}
```

- Eine Klasse darf mehrere Interfaces implementieren:

```
class A implements X, Y {}
```

- Auch Kombinationen sind zulässig:

```
class A extends B implements X, Y, Z {}
```

Das folgende Beispiel demonstriert das Arbeiten mit Interfaces:

```

1 interface I1 {
2     int TEST = 5;    // public static final int TEST = 5;
3 }
4
5 interface I2 {
6     int TEST = 3;    // public static final int TEST = 3;
7     void foo();      // public abstract void foo();
8 }
9
10 public class InterfaceDemo implements I2 {
11     public static void main(String []args) {
12         new InterfaceDemo().foo();
13         System.out.println("TEST von I2 : " + TEST);
14     }
15
16     public void foo() {
17         System.out.println("TEST von I1 : " + I1.TEST);
18     }
19 }

```

Listing 4.21: Interfaces Beispiel 1

Ausgabe:

TEST von I1 :5
 TEST von I2 :3

Mit Hilfe eines Interfaces kann gewährleistet werden, dass ein Objekt einer Klasse, die das entsprechende Interface implementiert, ein bestimmtes Verhalten aufweist. Die Klasse kann ja nur dann instanziiert werden, wenn alle abstrakten Methoden des Interfaces konkretisiert wurden. Darüber hinaus stellt ein Interface auch einen neuen Referenztyp dar, jedes Objekt einer Klasse *A*, die das Interface *I* implementiert ist im Rahmen der "is-a"-Beziehung auch vom Typ *I*. Interfaces können auch als formale Methodenparameter angegeben werden.

Casting innerhalb der Vererbungshierarchie ist im Bereich der Klassen nur innerhalb einer Vererbungslinie möglich und sinnvoll. Der Typecast auf ein Interface ist dagegen immer möglich. Das folgende Beispiel demonstriert die Sinnhaftigkeit dieser Regel. Dem Beispiel liegt die folgende Vererbungshierarchie zugrunde:

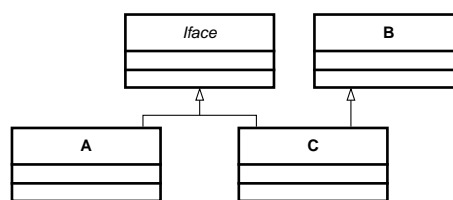


Abbildung 4.4: Casten auf Interfacetypen

```

1 interface Iface {}
2 class A implements Iface {}
3 class B {}
4 class C extends B implements Iface {}
5
6 public class IfaceCast {
7     public static void main(String []args) {
8         Iface i = null;
9         B b = null;
10        b = new C();
11        i = (Iface)b;
12    }
13 }

```

Listing 4.22: Casten auf Interfacetypen

Der Cast des Compilertyps *B* auf den nicht mit *B* in einer Vererbungslinie liegenden Interfacetyp *Iface* in Zeile 11 ist zur Laufzeit in Ordnung, da in *b* der Laufzeittyp *C* gespeichert ist und dieser Typ im Rahmen der "is-a"-Beziehung auch vom Typ *Iface* ist.

Beispiel

Das folgende Beispiel zeigt an Hand der folgenden Vererbungshierarchie das Arbeiten mit Interfaces.

```

1 interface IStack {
2     void push(Object item);
3     Object pop();
4 }
5
6 class StackImpl implements IStack {
7     protected Object[] stackArray;
8     protected int tos;
9
10    public StackImpl(int capacity) {
11        stackArray = new Object[capacity];
12        tos = -1;
13    }
14
15    public void push(Object item) {
16        stackArray[++tos] = item;
17    }

```

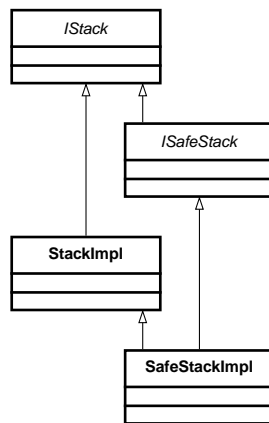


Abbildung 4.5: Vererbungshierarchie Stackbeispiel

```

18
19 public Object pop() {
20     Object objRef = stackArray[tos];
21     stackArray[tos] = null;
22     tos = tos -1;
23     return objRef;
24 }
25 }
26
27 interface ISafeStack extends IStack {
28     boolean isEmpty();
29     boolean isFull();
30 }
31
32 class SafeStackImpl extends StackImpl implements ISafeStack {
33     public SafeStackImpl(int capacity) { super(capacity); }
34     public boolean isEmpty() { return tos < 0; }
35     public boolean isFull() { return tos >= stackArray.length-1; }
36     public void push(Object item) {
37         if(!isFull())
38             super.push(item);
39         else
40             throw new RuntimeException("Stack is full");
41     }
42     public Object pop() {
43         if(!isEmpty())
44             return super.pop();
45         else
46             throw new RuntimeException("Stack is empty");
47     }
48 }
49
50 public class StackDemo {
51     public static void main(String args[]) {
52         SafeStackImpl safeStackRef = new SafeStackImpl(10);
53         StackImpl stackRef = safeStackRef;
54         ISafeStack isafeStackRef = safeStackRef;
55         IStack istackRef = safeStackRef;
56         Object objRef = safeStackRef;
57
58         safeStackRef.push("Dollar");
59         stackRef.push("Euro");
60         System.out.println(isafeStackRef.pop());
61         System.out.println(istackRef.pop());
62         System.out.println(istackRef.pop());
63         System.out.println(objRef.getClass());
64     }
65 }

```

Listing 4.23: Anwendungsbeispiel Interfaces

Ausgabe:

```
Euro
Dollar
Exception in thread "main" java.lang.RuntimeException:
Stack is empty
    at SafeStackImpl.pop(StackDemo.java:46)
    at StackDemo.main(StackDemo.java:62)
```

Kommentiert man die Zeile 62 aus, so erhält man die folgende Ausgabe:

```
Euro
Dollar
class SafeStackImpl
```

Bemerkungen zu obigem Beispiel:

- Die Tatsache, dass das Interface `ISafeStack` vom Interface `IStack` erbt, ist nur für die Zeile 60 wichtig. Über die Referenz `isafeStackRef` könnte man sonst nur die in `ISafeStack` definierten Methoden `isEmpty()` und `isFull()` aufrufen. Ein Objekt der Klasse `SafeStackImpl` wäre auf jeden Fall vom Typ `IStack`, da die Basisklasse `StackImpl` das Interface `IStack` implementiert.
- In Zeile 63 wird die Instanzmethode `getClass()` von `java.lang.Object` aufgerufen. Diese liefert das Klassenobjekt des Laufzeityps von `objRef`, also das Klassenobjekt der Klasse `SafeStackImp`. Zu jeder geladenen Klasse existiert zur Laufzeit ein eindeutiges Objekt⁸, das alle Klassenvariablen speichert. Für dieses Klassenobjekt der Klasse `SafeStackImpl` wird nun implizit die Methode `toString()` aufgerufen, welche den Text `class` gefolgt vom Namen der Klasse ausgibt.
- Es stellt kein Problem dar, dass die Klasse `SafeStackImp` das Interface `IStack` über zwei Vererbungslinien (einmal direkt und einmal indirekt) implementiert.

⁸Gemäß dem Designpattern Singleton

Kapitel 5

Exceptionhandling

5.1 Grundlagen

Exceptions sind Ausnahmen, die durch ein Programm zur Laufzeit verursacht werden können. Das Auslösen einer Ausnahme wird im Java-Sprachgebrauch als "throwing" bezeichnet, das Behandeln einer Ausnahme, also die explizite Reaktion auf das Eintreten einer Ausnahme, als "catching". Das Grundprinzip des Java-Exception-Mechanismus kann wie folgt beschrieben werden:

- Innerhalb einer bestimmten Methode wird eine Exception ausgelöst (geworfen).
- Diese kann nun entweder in jener Methode, in der sie ausgelöst wurde, behandelt (gefangen) werden, oder sie kann (an die aufrufende Methode) weitergeworfen werden.
- Wird die Ausnahme weitergeworfen, so hat der Empfänger der Ausnahme erneut die Möglichkeit, sie entweder zu fangen oder weiterzugeben.
- Wird die Ausnahme von keinem Programmteil gefangen, so führt sie zum Abbruch des Programms und zur Ausgabe einer Fehlermeldung.

5.2 Fehlerklassen

Fehler sind Javaobjekte, die der folgenden (unvollständigen) Klassenhierarchie unterliegen. Aus der Vererbungshierarchie sind die folgenden wesentlichen Eigenschaften abzulesen:

- Die Klasse `Throwable` hat zwei Subklassen, nämlich `Error` und `Exception`.
- Alle `Exception`-Klassen, die direkt von `Exception` erben, heißen `checked`, alle, die von `RuntimeException` erben `unchecked`.

Die wichtigsten Klassen dieser Hierarchie werden im Folgenden kurz beschrieben:

5.2.1 Die Klasse `java.lang.Throwable`

Dies ist die Superklasse aller `Error`- und `Exception`-Klassen. Sie erbt direkt von `Object`. Ihre wichtigsten Methoden sind:

```
public Throwable()  
Erzeugt ein Throwable-Objekt mit einer null-Message.
```

```
public Throwable(String message)  
Erzeugt ein Throwable-Objekt mit der Message message.
```

```
public String toString()  
Liefert eine Kurzbeschreibung des Fehlers.
```

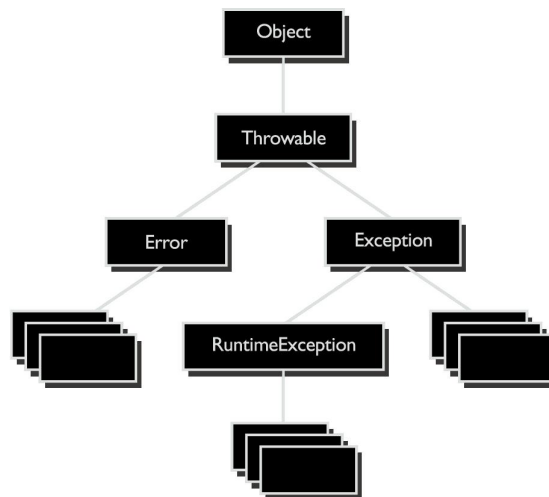


Abbildung 5.1: Vererbungshierarchie für Fehlerklassen

```
public String getMessage()
```

Liefert eine ausführliche Beschreibung des Fehlers.

```
public void printStackTrace()
```

Schreibt einen Auszug des Laufzeitstacks auf `System.err`.

5.2.2 Die Klasse `java.lang.Exception`

Diese Klasse ist die Superklasse aller Exceptionklassen. Exceptions sollten vom Programm gefangen werden. Die wichtigsten Methoden dieser Klasse sind in Analogie zur Klasse `Throwable`:

```
public Exception()
public Exception(String message)
public String toString()
public String getMessage()
public void printStackTrace()
```

Alle Exceptionklassen haben wenigstens die oben beschriebenen Methoden.

5.2.3 Die Klasse `java.lang.RuntimeException`

Alle von dieser Klasse abgeleiteten Exceptionklassen instanziierten sog. unchecked Exceptions, d.h. ihre Behandlung wird vom Compiler nicht überprüft. Programmabschnitte, die nur unchecked Exceptions werfen können, dürfen ohne explizite Fehlerbehandlung ausprogrammiert werden. Damit wird verhindert, dass (fast) jede Zeile Javacode fehlerbehandelt werden muss. Wichtige Beispiele für unchecked Exceptions sind:

- `java.lang.NullPointerException`
Tritt auf, wenn eine Instanzmethode oder eine Instanzvariable über die `null`-Referenz angesprochen wird.
- `java.lang.IndexOutOfBoundsException`
Tritt auf, wenn versucht wird, auf ein Feld- oder Stringelement über einen ungültigen Index zuzugreifen.
- `java.lang.ArithmeticException`
Tritt bei einem arithmetischen Fehler, z.B. einer Integerdivision durch Null auf.
- `java.lang.IllegalArgumentException`
Sollte immer dann von einer Methode geworfen werden, wenn die Methode mit unzulässigen Parameterwerten versorgt wurde.

Alle Exceptions, die nicht vom Typ `RuntimeException` sind, heißen checked Exceptions und unterliegen der "catch-or-throw" Regel. Diese besagt, dass eine checked Exception entweder direkt behandelt (gefangen) oder weitergegeben (geworfen) werden muss. Verstößt man gegen diese Regel, so erhält man einen Compilerfehler.

5.2.4 Die Klasse `java.lang.Error`

Fast alle von dieser Klasse abgeleiteten Fehler zeigen einen abnormalen Zustand der JVM auf und sollten vom Programm nicht behandelt werden, da die weitere problemlose Programmabarbeitung in Frage zu stellen ist. Errors sind unchecked und unterliegen daher nicht der "catch-or-throw"-Regel.

5.2.5 Eigene Exceptionklassen

Oft ist es sinnvoll, selbst Exceptionklassen zu definieren. Diese beinhalten im Wesentlichen zwei Konstruktoren und erben entweder von `java.lang.Exception` (dann handelt es sich um eine checked Exception) oder `java.lang.RuntimeException` (unchecked Exception). Das folgende Listing zeigt eine typische Exceptionklasse:

```
1 public class MyUnchecked extends RuntimeException {  
2     // Parameterloser Konstruktor  
3     public MyUnchecked() {}  
4     // Message-Konstruktor  
5     public MyUnchecked(String message) { super(message); }  
6 }
```

Listing 5.1: Unchecked Exception

5.3 Auslösen von Ausnahmen

Da Fehlerklassen und ihre Instanzen keine Ausnahmestellung besitzen, hat man als Entwickler alle Möglichkeiten der OOP. Es ist also insbesondere möglich, ein Fehlerobjekt zu instanzieren oder eigene Fehlerklassen aus den vorhandenen abzuleiten (siehe oben).

Mit Hilfe der `throw`-Anweisung kann ein Fehlerobjekt zum Auslösen einer Ausnahme verwendet werden. Die Syntax der `throw`-Anweisung ist:

```
throw Ausnahmeobjekt;
```

Das folgende Beispiel definiert ansatzweise eine Klasse `Bruch` zur Verarbeitung von Brüchen. Sollte einem Konstruktor der Nennerwert Null übergeben werden, so erzeugt und wirft dieser Konstruktor eine `IllegalArgumentException`:

```
1 public class Bruch {  
2     private long zaehler = 0L;  
3     private long nenner = 1L;  
4  
5     public Bruch() {}  
6  
7     public Bruch(long zaehler) {  
8         this.zaehler = zaehler;  
9         this.nenner = 1L;  
10    }  
11  
12    public Bruch(long zaehler, long nenner) throws IllegalArgumentException {  
13        if(nenner == 0)  
14            throw new IllegalArgumentException("Nenner gleich Null");  
15        this.zaehler = zaehler;  
16        this.nenner = nenner;  
17        this.kuerzen();  
18    }  
19  
20    // Weitere Klassenmitglieder  
21 }
```

Listing 5.2: Auslösen von Ausnahmen

5.4 Die try-catch - Anweisung

5.4.1 Syntax und einfache Anwendung

Das Fangen einer Exception erfolgt mit Hilfe der try-catch - Anweisung. Ihre Syntax lautet:

```
try {
    <Anweisungen>
}
catch(<Exceptiontyp_1> <Parameter_1>) {
    <Anweisungen>
}
...
catch(<Exceptiontyp_n> <Parameter_n>) {
    <Anweisungen>
}
finally {
    <Anweisungen>
}
```

Der try-Block enthält dabei eine oder mehrere Anweisungen, bei deren Ausführung Fehler der Typen <Exceptiontyp_1> bis <Exceptiontyp_n> auftreten können. In einem solchen Fall wird die Ausführung des try-Blocks abgebrochen und der Kontrollfluss setzt mit den Anweisungen jener catch - Klausel fort, zu der der aufgetretene Fehler passt. Hier kann nun Code untergebracht werden, der eine angemessene Reaktion auf den Fehler realisiert.

Beispiel:

```
1 public class Try_Catch_1 {
2     public static void main(String[] args) {
3         int i, base = 0;
4
5         try {
6             for (base = 10; base >= 2; --base) {
7                 i = Integer.parseInt("40",base);
8                 System.out.format("40 base %2d = %2d\n", base, i);
9             }
10        }
11        catch (NumberFormatException e) {
12            System.out.println("40 ist keine Zahl zur Basis " + base);
13        }
14    }
15 }
```

Listing 5.3: Beispiel 1 zu try-catch

Ausgabe:

```
40 base 10 = 40
40 base  9 = 36
40 base  8 = 32
40 base  7 = 28
40 base  6 = 24
40 base  5 = 20
40 ist keine Zahl zur Basis 4
```

Die Funktion `Integer.parseInt` in Zeile 7 wirft, sollte 40 nicht in eine Zahl im Zahlensystem zur Basis `base` geparkt werden können, eine `NumberFormatException`. Diese erbt von `IllegalArgumentException` und ist damit unchecked.

Während der Entwicklung ist es oft informativ, in der `catch`-Klausel die Fehlermeldung und einen Auszug aus dem Laufzeitstack auszugeben. Dies könnte für obiges Beispiel etwa so aussehen:

```

1 public class Try_Catch_2 {
2     public static void main(String[] args) {
3         int i, base = 0;
4
5         try {
6             for (base = 10; base >= 2; --base) {
7                 i = Integer.parseInt("40",base);
8                 System.out.format("40 base %2d = %2d\n", base, i);
9             }
10        }
11        catch (NumberFormatException e) {
12            System.out.println("***Fehler aufgetreten***");
13            System.out.println("Ursache: " + e.getMessage());
14            e.printStackTrace();
15        }
16    }
17 }

```

Listing 5.4: Beispiel 1 zu try-catch

Ausgabe:

```

40 base 10 = 40
40 base  9 = 36
40 base  8 = 32
40 base  7 = 28
40 base  6 = 24
40 base  5 = 20
***Fehler aufgetreten***
Ursache: For input string: "40"
java.lang.NumberFormatException: For input string: "40"
at java.lang.NumberFormatException.forInputString(Number.java:48)
at java.lang.Integer.parseInt(Integer.java:447)
at Try_Catch_2.main(Try_Catch_2.java:7)

```

5.4.2 Mehrere catch - Klauseln

Werden in einer try-catch-Anweisung mehrere catch-Zweige programmiert, so ist zu beachten, dass jede Exception nur einmal abgefangen werden darf. Daraus folgt auch, dass speziellere Ausnahmen vor generelleren abzufangen sind. Hält man sich nicht an diese Regel, so entsteht ein Compilerfehler.

Das folgende Beispiel veranschaulicht die Problematik an Hand einer `FileNotFoundException`, die eine Subexception der `IOException` ist. Im Sinne der "is-a"-Beziehung ist eine `FileNotFoundException` also eine `IOException`.

```

1 try {
2     FileInputStream fis = new FileInputStream("test.dat");
3     // wirft gegebenenfalls eine FileNotFoundException
4     int i = fis.read(); // aus der Datei lesen
5     fis.close();       // Datei schliessen
6     // read() u. close() werfen ev. eine IOException
7 }
8 catch(FileNotFoundException fnfe) {
9     // Fehlerbehandlung
10 }
11 catch(IOException ioe) {
12     // Fehlerbehandlung
13 }

```

Listing 5.5: Mehrere catch-Klauseln

Würde man die Reihenfolge der catch-Zweige in obigem Beispiel umdrehen, so würde die `FileNotFoundException` bereits im ersten catch-Zweig gefangen und der zweite einen Compilerfehler generieren. Im Prinzip wäre dann der Code der zweiten catch-Klausel `unreachable`, die im Programmfluss immer die erste

passende `catch`-Klausel abgearbeitet wird.

Eine weitere Regel zur `try-catch`-Anweisung besagt, dass nur solche checked Exceptions gefangen werden dürfen, die auch tatsächlich auftreten können. Verstößt man gegen diese Regel, so erhält man einen Compilerfehler. Unchecked Exceptions (und damit auch `Exception` selbst) dürfen immer gefangen werden, unabhängig davon, ob sie auftreten können oder nicht.

5.4.3 Die `finally` - Klausel

Mit Hilfe der optionalen `finally`-Klausel, die als letzte Klausel einer `try-catch`-Anweisung auftreten muss, kann ein Programmfragment definiert werden, das immer dann ausgeführt wird, wenn die zugehörige `try`-Klausel betreten wurde. Dabei spielt es keine Rolle, welches Ereignis dafür verantwortlich war, dass der `try`-Block verlassen wurde. Die `finally`-Klausel wird insbesondere dann ausgeführt, wenn der `try`-Block durch eine der folgenden Anweisungen verlassen wurde:

- Wenn das normale Ende des `try`-Blocks erreicht wurde.
- Wenn eine Ausnahme aufgetreten ist, die durch eine `catch`-Klausel behandelt wurde.
- Wenn eine Ausnahme aufgetreten ist, die nicht durch eine `catch`-Klausel behandelt wurde.
- Wenn der `try`-Block durch eine der Sprunganweisungen `break`, `continue` oder `return` verlassen werden soll.

Die Java-Sprachdefinition gibt nur wenige Fälle an, in denen nach Betreten eines `try`-Blocks eine allenfalls vorhandene `finally`-Klausel nicht abgearbeitet wird. Zwei davon sind:

- Wenn die JVM vorher durch den Aufruf von `System.exit(n)` beendet wird.
- Wenn das System vorher ausfällt.

Die `finally`-Klausel ist also der ideale Ort, um Aufräumarbeiten durchzuführen. Hier können beispielsweise Dateien geschlossen oder Ressourcen freigegeben werden.

5.4.4 Weitergabe von Exceptions

Wird eine (checked) Exception nicht in der Methode, in der sie auftritt, direkt behandelt, so muss sie nach der "catch-or-throw" Regel an die aufrufende Methode weitergegeben werden. Dazu wird der Methodenkopf um eine `throws`-Klausel ergänzt. Die vollständige Syntax einer Methodendefinition lautet also:

```
<Modifier> Typ Name(<[Parameter]>)  
    throws Exception_1, ..., Exception_n {  
    // Methodenrumpf  
}
```

Das folgende Beispiel demonstriert diese Technik. Die Funktion `summe` erhält zwei Strings `str1` und `str2` und versucht diese Strings in Integerwerte zu parsen und die Summe zu berechnen. Dabei kommt wieder die Funktion

`java.lang.Integer.parseInt()`

zum Einsatz, die bei nicht verwandelbarem String eine

`java.lang.NumberFormatException`

wirft. Diese Exception wird in der Funktion nicht gefangen, sondern weitergegeben:

```
1 public int summe(String s1, String s2) throws NumberFormatException {  
2     int i1 = Integer.parseInt(s1);  
3     int i2 = Integer.parseInt(s2);  
4     return i1 + i2;  
5 }
```

Listing 5.6: Werfen von Ausnahmen

Kapitel 6

Grundlegende Klassen

6.1 Die Klasse `java.lang.Object`

Die Klasse `java.lang.Object` ist die Wurzel der Klassenhierarchie. Jede Klasse erbt von dieser Klasse. Alle Javaobjekte (auch Felder) haben die in `Object` definierten Eigenschaften und sind daher vom diesem Typ.

Drei für die Threadkontrolle wichtige finale Methoden in `java.lang.Object` sind `wait()`, `notify()` und `notifyAll()`. Diese werden im Abschnitt über Threads besprochen.

Zwei andere Methoden, `equals()` und `toString()` stellen allgemeine Funktionalität zur Verfügung und können in abgeleiteten Klassen überschrieben werden.

6.1.1 Die Methode `toString()`

```
public String toString()
```

Liefert die Stringdarstellung eines Objekts. Die Methode wird implizit immer dann aufgerufen, wenn diese Stringdarstellung benötigt wird. Die Methode `toString()` aus `java.lang.Object` hat den folgenden Sourcecode:

```
1 public String toString() {  
2     return getClass().getName()+"@"+Integer.toHexString(hashCode());  
3 }
```

Listing 6.1: `toString()` aus `java.lang.Object`

Das folgende Beispiel demonstriert implizite und explizite Aufrufe von `toString()`

```
1 public class ToString {  
2     public static void main(String []args) {  
3         Test t = new Test();  
4         int []f = new int[100];  
5         System.out.println("Test: " + t);      // impliziter Aufruf  
6         t = null;  
7         System.out.println("Nullreferenz: " + t);  
8         System.out.println("Feld f: " + f.toString());  
9     }  
10 }  
11  
12 class Test {}
```

Listing 6.2: Beispiel zu `toString()`

Ausgabe:

```
Test: Test@1ea2dfe  
Nullreferenz: null  
Feld f: [I@17182c1
```

6.1.2 Die Methode `equals()`

Vergleicht man 2 Referenzen mit dem Vergleichsoperator `==`, so erhält man genau dann `true`, wenn die beiden Referenzen ein und dasselbe Objekt referenzieren. Zum Vergleich von Inhalten ist die Methode `equals()` vorgesehen.

```
public boolean equals(Object obj)
```

Die Methode `equals()` aus `java.lang.Object` stellt genau die Funktionalität des Vergleichsoperators `==` zur Verfügung. Ihr Quellcode lautet:

```
1 public boolean equals(Object obj) {
2     return this == obj;
3 }
```

Listing 6.3: `equals()` aus `java.lang.Object`

In abgeleiteten Klassen kann `equals()` so überschrieben werden, dass auch Inhalte verglichen werden. Dabei ist zunächst zu prüfen, ob das übergebene Objekt auch vom Typ des `this`-Objektes ist, andernfalls ist `false` zu retournieren.

Das nächste Beispiel zeigt das Überschreiben der Methode `equals()` an Hand eines einfachen Beispiels:

```
1 class Test {
2     private int a, b;
3
4     public Test(int a, int b) {
5         this.a = a;
6         this.b = b;
7     }
8
9     @Override
10    public boolean equals(Object o) {
11        if(o == null) {
12            return false;
13        }
14        if(this == o) {           // gleiches Objekt, nichts zu vergleichen
15            return true;
16        }
17        if(o.getClass() == this.getClass()) {
18            return a == ((Test)o).a && b == ((Test)o).b;
19        }
20        else {
21            return false;
22        }
23    }
24 }
25
26 public class Equals {
27     public static void main(String []args) {
28         Test t1 = new Test(3,10);
29         Test t2 = new Test(3,10);
30         System.out.println("t1.equals(t2) liefert: " + t1.equals(t2));
31         System.out.println("t1 == t2      liefert: " + (t1 == t2));
32         t2 = new Test(4,10);
33         System.out.println("t1.equals(t2) liefert: " + t1.equals(t2));
34         t2 = t1;
35         System.out.println("t1 == t2      liefert: " + (t1 == t2));
36     }
37 }
```

Listing 6.4: Überschreiben von `equals()`

Ausgabe:

```
t1.equals(t2) liefert: true
t1 == t2      liefert: false
t1.equals(t2) liefert: false
t1 == t2      liefert: true
```


6.2 Die Klasse `java.lang.String`

In Java werden Zeichenketten durch die Klasse `java.lang.String` repräsentiert. Sie bietet Methoden zum Erzeugen von Zeichenketten, zur Extraktion von Teilstrings, zum Vergleich mit anderen Strings sowie zur Erzeugung von Strings aus primitiven Typen. Eine Instanz der Klasse `String` ist prinzipiell eine konstante Kette von Unicode-Zeichen (Feld vom Typ `char`). Nach der Initialisierung eines Strings bleiben Länge und Inhalt konstant. Stringobjekte sind also *immutable*. Zur Verarbeitung von veränderlichen Zeichenketten existieren in Java die Klassen `java.lang.StringBuffer` und `java.lang.StringBuilder`. Alle drei Klassen sind *final*, d.h. von ihnen können keine weiteren Klassen abgeleitet werden.

- Die Klasse `String` ist *final*
- Stringobjekte sind *immutable*

6.2.1 Wichtige Methoden der Klasse `java.lang.String`

Konstruktoren

```
public String()
Erzeugt ein leeres Stringobjekt "".
```

```
public String(String value)
Erzeugt einen neuen String durch Duplizierung von value.
```

```
public String(char[] value)
Erzeugt einen neuen String aus dem Charakterfeld value.
```

```
public String(byte[] value)
Erzeugt einen neuen String aus dem Bytefeld value.
```

Allgemeine Methoden

```
public int length()
Liefert die Länge dieses Stringobjekts.
```

```
public String concat(String s)
Liefert einen neuen String, der entsteht, wenn man den String s an das this-Objekt anhängt.
```

Zeichen- und Stringextraktion

```
public char charAt(int index)
Liefert das Zeichen an der nullbasierten Position index. Bei falschem Index (kleiner als 0 oder >= this.length()) wird die unchecked IndexOutOfBoundsException geworfen.
```

```
public String substring(int begin, int end)
Liefert den Teilstring, der an der Position begin startet und an der Position end-1 endet. Bei falschem Index wird die unchecked IndexOutOfBoundsException geworfen.
```

```
public String substring(int begin)
Liefert den Teilstring von der Position begin bis zum Ende des Strings. Bei falschem Index wird die unchecked IndexOutOfBoundsException geworfen.
```

```
public String trim()
Liefert jenen String, der entsteht, wenn am Beginn und Ende des this-Objekts alle Zeichen mit Unicode
```

kleiner gleich 32 (`'\u0020'`) entfernt werden. Es werden speziell also alle führenden und abschließenden Leerzeichen entfernt.

```
public String[] split(String regex)
```

Zerlegt das `this`-Objekt in Teilstrings, wobei die Begrenzer mit dem gegebenen regulären Ausdruck `regex` matchen. Geliefert wird ein Stringfeld, das alle Teilstrings enthält.

Beispiele:

```
"boo:and:foo".split(":"); liefert { "boo", "and", "foo" }
```

```
"boo:and:foo".split("o"); liefert { "b", "", ":and:f" }
```

Vergleich von Zeichenketten

```
public boolean equals(Object o)
```

Überlagert die Methode `equals()` aus `java.lang.Object`. Prüft das `this`-Objekt und `o` auf inhaltliche Übereinstimmung. Sollte `o` nicht vom Typ `String` sein, wird auf jeden Fall `false` geliefert.

```
public boolean equalsIgnoreCase(String s)
```

Vergleicht den `String s` mit dem `this`-Objekt und ignoriert Unterschiede in der Groß-Kleinschreibung.

```
public int compareTo(String s)
```

Vergleicht das `this`-Objekt alphabetisch mit `s`. Retourniert bei `this < s` einen negativen Wert, bei Gleichheit den Wert 0 und bei `this > s` einen positiven Wert.

```
public boolean startsWith(String s)
```

Testet, ob das `this`-Objekt mit der Zeichenkette `s` beginnt.

```
public boolean endsWith(String s)
```

Testet, ob das `this`-Objekt mit der Zeichenkette `s` endet.

Suchen in Zeichenketten

```
public int indexOf(char ch)
```

Sucht das Zeichen `ch` im `this`-Objekt und liefert die Position der ersten Übereinstimmung bzw. -1, wenn `ch` im `String` nicht vorkommt.

```
public int indexOf(String str, int fromIndex)
```

Sucht das Zeichen `ch` im `this`-Objekt ab der Position `fromIndex` und liefert die Position der nächsten Übereinstimmung bzw. -1, wenn `ch` im `String` nicht mehr vorkommt.

```
public int lastIndexOf(char ch)
```

Wie `indexOf()`, liefert aber die Position der letzten Übereinstimmung.

Ersetzen von Zeichenketten

```
public String toLowerCase()
```

Liefert einen neuen `String`, der entsteht, wenn im `this`-Objekt alle Großbuchstaben durch Kleinbuchstaben ersetzt werden.

```
public String toUpperCase()
```

Liefert einen neuen `String`, der entsteht, wenn im `this`-Objekt alle Kleinbuchstaben durch Großbuchstaben ersetzt werden.

```
public String replace(char oldchar, char newchar)
```

Liefert einen neuen `String`, der entsteht, wenn im `this`-Objekt jedes Zeichen `oldchar` durch das Zei-

chen `newchar` ersetzt wird.

Keine dieser Methoden kann das `this`-Objekt verändern, da ein einmal erzeugtes Stringobjekt immutabel ist. Methoden, die einen String liefern, erzeugen intern ein neues Stringobjekt und liefern eine Referenz auf dieses neue Objekt. Ist das Ergebnis inhaltlich mit dem `this`-Objekt identisch, so wird kein neues Objekt erzeugt, sondern die `this`-Referenz retourniert. Das folgende Beispiel demonstriert diesen Sachverhalt:

```

1 public class ImmutableStrings {
2     public static void main(String []args) {
3         String s = new String("    100 + 100 = 200");
4
5         System.out.println(s.trim()); // Veraendert s nicht
6         System.out.println(s);
7         s.replace('0', 'x');           // Veraendert s nicht
8         System.out.println(s);
9         s = s.replace('0', 'x');       // neues Objekt in s gespeichert
10        System.out.println(s);
11    }
12 }

```

Listing 6.5: Immutable-Eigenschaft von Strings

Ausgabe:

```

100 + 100 = 200
 100 + 100 = 200
 100 + 100 = 200
 1xx + 1xx = 2xx

```

6.2.2 Stringpool, Vergleichen von Strings

Beim Vergleichen von Strings gelten die gleichen Regeln wie beim Vergleich beliebiger Javaobjekte (`s1` und `s2` seien Stringreferenzen):

- `s1 == s2` liefert genau dann `true`, wenn `s1` und `s2` das gleiche Objekt referenzieren.
- `s1.equals(s2)` liefert genau dann `true`, wenn die über `s1` und `s2` gespeicherten Strings gleichen Inhalt haben.

Alle Stringkonstanten innerhalb eines Javaprogrammes bilden den sog. Stringpool. Dieser enthält nur Stringobjekte mit unterschiedlichem Inhalt. Das bedeutet, dass bei mehrfachem Vorkommen einer Stringkonstanten im Quellcode diese nur einmal im Stringpool liegt. Vergleiche von Stringkonstanten mit dem Operator `==` liefern bei gleichem Inhalt also `true`, da es sich um ein und dasselbe Objekt aus dem Stringpool handelt. Das folgende Beispiel demonstriert diesen Sachverhalt:

```

1 public class CompareStrings {
2     public static void main(String []args) {
3         String s1 = "Vergleichsstring";
4         String s2 = "Vergleichsstring";
5         // s1 und s2 verweisen auf den gleichen String im Stringpool
6         System.out.println(" 6: s1 == s2      liefert: " + (s1 == s2));
7         System.out.println(" 7: s1.equals(s2) liefert: " + s1.equals(s2));
8         s1 = new String("Vergleichsstring");
9         // nun wurde eine neues Stringobjekt erzeugt
10        System.out.println("10: s1 == s2      liefert: " + (s1 == s2));
11        System.out.println("11: s1.equals(s2) liefert: " + s1.equals(s2));
12    }
13 }

```

Listing 6.6: Vergleich von Stringobjekten

Ausgabe:

```

 6: s1 == s2      liefert: true
 7: s1.equals(s2) liefert: true
10: s1 == s2      liefert: false
11: s1.equals(s2) liefert: true

```

Jeder Stringkonstante aus dem Stringpool kann wie eine Referenz verwendet werden. Dies bedeutet, dass über Stringlitterale Instanzmethoden der Klasse *String* aufgerufen werden können:

```
System.out.println("Java".charAt(2));
System.out.println("java".toUpperCase());
```

Ausgabe:

```
v
JAVA
```

6.2.3 Stringverkettung

Der Operator `+` ist für die Klasse *String* überladen. Ist im Ausdruck `a+b` wenigstens einer der beiden Operanden ein *String*, so wird auch der zweite in einen *String* konvertiert und die *Strings* werden verkettet. Java gewährleistet, dass es für alle Ausdrücke mit Ausnahme von `void` eine Stringdarstellung gibt. Für Objekte ist dies durch die Methode `toString()` gewährleistet, die Stringdarstellung der `null`-Referenz ist der *String* `"null"`. Das folgende Beispiel veranschaulicht den Sachverhalt:

```
1 System.out.println("abc" + 10 + 20);
2 System.out.println(10 + "abc" + 20);
3 System.out.println(10 + 20 + "abc");
4 System.out.println(10 + (20 + "abc"));
5 System.out.println("" + 10 + 20 + "abc");
```

Listing 6.7: Stringverkettung

Ausgabe:

```
abc1020
10abc20
30abc
1020abc
1020abc
```

6.3 Die Klassen *StringBuffer* und *StringBuilder*

Diese Klassen aus dem Paket `java.lang` dienen zur Verarbeitung von Zeichenketten, die sich dynamisch verändern können. Sie besitzen exakt die gleichen Methoden, wobei *StringBuilder*¹ im Gegensatz zu *StringBuffer* nicht threadsicher ist aber dafür in Singlethread-Umgebungen performanter arbeitet. Die Klassen besitzen nicht so viele Methoden zur Auswertung der Zeichenkette, sondern legen den Schwerpunkt auf Operationen zur Veränderung ihres Inhalts.

6.3.1 Wichtige Methoden

Konstruktoren

```
public StringBuffer()
public StringBuilder()
```

Erzeugt einen leeren *StringBuffer* bzw. *StringBuilder*.

```
public StringBuffer(String s)
public StringBuilder(String s)
```

Erzeugt einen *StringBuffer* bzw. *StringBuilder*, der eine Kopie des *Strings* `s` repräsentiert.

```
public StringBuffer(int capacity)
public StringBuilder(int capacity)
```

Erzeugt einen leeren *StringBuffer* bzw. *StringBuilder* mit der angegebenen Ausgangskapazität.

¹neu seit Java 1.5

Einfügen von Elementen

```
public StringBuffer append(String s)
public StringBuilder append(String s)
```

Hängt den String `s` an das Ende des `this`-Objekts an. Zurückgegeben wird eine Referenz auf das auf diese Weise veränderte `this`-Objekt. Zusätzlich gibt es überladene `append()`-Methoden zum Anhängen aller Arten von primitiven Typen. Anstelle eines String-Objekts wird hier der entsprechende primitive Typ übergeben, in einen String konvertiert und an das Ende des `this`-Objekts angehängt.

```
public StringBuffer append(Object o)
public StringBuilder append(Object o)
```

Hängt die Stringdarstellung `o.toString()` an das Ende des `this`-Objektes an. Zurückgegeben wird eine Referenz auf das auf diese Weise veränderte `this`-Objekt.

```
public StringBuffer insert(int offset, String s)
public StringBuilder insert(int offset, String s)
```

Fügt den String `s` ab der Position `offset` in das `this`-Objekt ein.

Löschen von Elementen

```
public StringBuffer deleteCharAt(int index)
public StringBuilder deleteCharAt(int index)
```

Das an der Position `index` stehende Zeichen wird entfernt.

```
public StringBuffer delete(int start, int end)
public StringBuilder delete(int start, int end)
```

Entfernt jenen Teilstring, der von Position `start` bis Position `end-1` reicht.

Verändern von Elementen

```
public void setCharAt(int index, char ch)
```

Das an der Position `index` stehende Zeichen wird durch `ch` ersetzt.

```
public StringBuffer replace(int start, int end, String s)
public StringBuilder replace(int start, int end, String s)
```

Ersetzt das Teilstück von Position `start` bis Position `end-1` durch den String `s`.

```
public StringBuffer reverse()
public StringBuilder reverse()
```

Dreht die Reihenfolge der Zeichen im `this`-Objekt um und retourniert das `this`-Objekt.

Allgemeine Methoden

```
public int length()
```

Liefert die aktuelle Länge des `this`-Objekts.

```
public String toString()
```

Dient zur Konvertierung eines `StringBuffer`- bzw. `StringBuilder`-Objektes in einen String. Liefert den Inhalt des `this`-Objekts als String.

6.3.2 Eigenschaften

- Die Klassen `StringBuffer` und `StringBuilder` überschreiben die Methode `equals()` aus `java.lang.Object` nicht.
- Alle Methoden verändern das `this`-Objekt und schicken gegebenenfalls eine Referenz auf dieses Objekt zurück.

- Die Klassen `StringBuffer` und `StringBuilder` sind final.

6.3.3 Beispiel

Das folgende Beispiel demonstriert einige der oben beschriebenen Methoden und zeigt, wie die Stringverkettung intern mit Hilfe der Klasse `StringBuffer` implementiert ist:

```

1 public class StringBufferExample {
2     public static void main(String []args) {
3         StringBuffer sbuf = new StringBuffer("12345");
4         sbuf.reverse();
5         System.out.println(sbuf);
6         sbuf.insert(2, "xxx");
7         System.out.println(sbuf);
8         sbuf.append("yyy");
9         System.out.println(sbuf);
10
11        String a = "Hallo", b = "Welt" , c;
12        //Konventionelle Verkettung
13        c = a + ", " + b;
14        System.out.println(c);
15        //Implementierung durch StringBuffer
16        c = new StringBuffer().append(a).append(", ").append(b).toString();
17        System.out.println(c);
18    }
19 }

```

Listing 6.8: Beispiel zu `StringBuffer`

Ausgabe:

```

54321
54xxx321
54xxx321yyy
Hallo, Welt
Hallo, Welt

```

6.4 Die Klasse `java.util.Scanner`

Ein Objekt dieses Typs dient dazu, einen Text in einzelne Token aufzuspalten, wobei die Trennstrings mit Hilfe von regulären Ausdrücken definiert werden können. Standardmäßig wird nach Whitespacezeichen getrennt. Die gelesenen Token können mit Hilfe der verschiedenen `nextXXX()`-Methoden in verschiedene Typen konvertiert werden.

6.4.1 Eingabe von `System.in`

Ein `Scanner`-Objekt kann auch verwendet werden, um Daten von `System.in`² zu lesen.

Das folgende Beispiel demonstriert das Lesen einer Zeile von `System.in`:

```

1 import java.util.*;
2 public class Tastatur {
3     public static void main(String[] args) {
4         Scanner sc = new Scanner(System.in);
5         System.out.print("Eingabe --> ");
6         String s = sc.nextLine();
7         System.out.println("Eingabe: " + s);
8     }
9 }

```

Listing 6.9: `Scanner` zum zeilenorientiertem Lesen von `System.in`

²also von der Konsole

In obigem Beispiel werden die folgenden Methoden verwendet:

```
public Scanner(InputStream source)
Erzeugt einen Scanner zum Lesen von source3.
```

```
public String nextLine()
Liest alle Zeichen bis zum nächsten Zeilentrennzeichen und liefert die gelesene Zeile als String.
```

Zum scannen primitiver Datentypen gibt es diverse `nextXXX()`-Methoden. Diese Methoden existieren für alle Datentypen, exemplarisch werden zwei vorgestellt:

```
public int nextInt()
public double nextDouble()
```

Scannen das nächste Token und verwandeln es in einen `int`- bzw. in einen `double`-Wert. Diese Methoden werfen die unchecked `InputMismatchException`, wenn das nächste Token nicht in den entsprechenden Typ verwandelt werden kann.

Vor Aufruf dieser Methoden kann mit Hilfe entsprechender `hasNextXXX()`-Methoden geprüft werden, ob das nächste Token in den gewünschten Datentyp verwandelt werden kann:

```
public boolean hasNextInt()
public boolean hasNextDouble()
```

Diese Methoden liefern `true`, wenn das nächste Token in den entsprechenden Datentyp verwandelt werden kann.

Das nächste Beispiel liest solange Zeilen über die Tastatur, bis "quit" eingegeben wird. Alle Eingaben, die in einen Integer verwandelt werden können werden summiert und diese Summe wird am Ende ausgegeben.

```

1 import java.util.*;
2 public class Tastatur2 {
3     public static void main(String[] args) {
4         Scanner sc = new Scanner(System.in);
5         int akt = 0, summe = 0;
6         String ein = null;
7         while (!"quit".equals(ein)) {
8             if (sc.hasNextInt()) {
9                 akt = sc.nextInt();
10                summe += akt;
11            }
12            else {
13                ein = sc.nextLine();
14            }
15        }
16        System.out.format("Die Summe aller Werte ist %d\n", summe);
17    }
18 }
```

Listing 6.10: Scanner zum Lesen von `System.in` 2

Das folgende Protokoll zeigt zwei Testläufe:

```

java Tastatur2
123 hallo
17
17
quit
Die Summe aller Werte ist 157
```

```

java Tastatur2
hallo 123
17
```

³`System.in` ist vom Typ `InputStream`

```

17
quit
Die Summe aller Werte ist 34

```

6.4.2 Lesen aus einer Textdatei

In diesem Abschnitt wird an Hand eines konkreten Beispiels demonstriert, wie mit Hilfe der Klasse `java.util.Scanner` aus einer Textdatei (CSV-Datei) gelesen werden kann. Die CSV-Datei speichert Länderinformationen und hat den folgenden Aufbau:

```

Kurzzeichen;Name;Hauptstadt;Flaeche;Einwohnerzahl
AU;Australia;Canberra;7686850;19731984
...

```

```

1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.FileNotFoundException;
4
5 public class ScannerReadFile {
6     public static void main(String[] args) {
7         Scanner sc = null;
8         try {
9             sc = new Scanner(new File("countries.csv"));
10        }
11        catch(FileNotFoundException e) {
12            System.err.println("Datei counries.csv nicht gefunden");
13            System.exit(1);
14        }
15
16        String line = null;
17        if(sc.hasNextLine())
18            sc.nextLine(); // Kopfzeile lesen
19        while(sc.hasNextLine()) {
20            line = sc.nextLine();
21            Scanner sc1 = new Scanner(line).useDelimiter(";");
22            Country c = new Country(sc1.next(), sc1.next(), sc1.next(), sc1.nextLong(), sc1.nextLong());
23            System.out.println(c);
24        }
25    }
26 }
27
28 class Country {
29     private String kz;
30     private String name;
31     private String capital;
32     private long flaeche;
33     private long einwohner;
34
35     public Country(String kz, String name, String capital, long flaeche, long einwohner ) {
36         this.kz = kz;
37         this.name = name;
38         this.capital = capital;
39         this.flaeche = flaeche;
40         this.einwohner = einwohner;
41     }
42
43     public String toString() {
44         return String.format("%2s %-20s %-20s %10d %10d", this.kz, this.name, this.capital, this.flaeche, this.einwohner);
45     }
46 }

```

Listing 6.11: Scanner zum Lesen einer Textdatei

Ausgabe:

AU Australia	Canberra	7686850	19731984
BR Brazil	Brasilia	8511965	182032604
CN China	Beijing	9596960	1286975468

DE Germany	Berlin	357021	82398326
ES Spain	Madrid	504782	40217413

6.5 Die Wrapperklassen

Zu jedem der 8 primitiven Datentypen gibt es im Paket `java.lang` eine zugehörige Wrapperklasse⁴. Jede Wrapperklasse kapselt dabei einen primitiven, nicht veränderbaren Wert. Wrapperobjekte sind also *immutable*, d.h. sie lassen sich nach ihrer Konstruktion nicht mehr verändern. Alle Wrapperklassen sind *final*, d.h. es lassen sich keine Subklassen generieren. Die folgende Tabelle gibt zu jedem primitiven Typ den Namen der entsprechenden Wrapperklasse an:

Primitiver Datentyp	Wrapperklasse
boolean	java.lang.Boolean
char	java.lang.Character
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

6.5.1 Konstruktion von Wrapperobjekten

Jede Wrapperklasse besitzt einen Konstruktor, der einen entsprechenden primitiven Typ erwartet. Außerdem besitzt mit Ausnahme der Klasse `Character` jede Wrapperklasse einen Konstruktor, der einen `String` erwartet. Kann der `String` in den entsprechenden Typ geparkt werden, so wird ein gültiges Objekt erzeugt, sonst wirft der Konstruktor (außer bei `Boolean`) eine `NumberFormatException`. Der Konstruktor von `Boolean` erzeugt statt dessen ein Objekt, das den Wert `false` kapselt.

Konstruktor der Klasse `java.lang.Boolean`

```
public Boolean(boolean value)
public Boolean(String s)
```

Konstruktor der Klasse `java.lang.Character`

```
public Character(char value)
```

Konstruktor der Klasse `java.lang.Byte`

```
public Byte(byte value)
public Byte(String s) throws NumberFormatException
```

Konstruktor der Klasse `java.lang.Short`

```
public Short(short value)
public Short(String s) throws NumberFormatException
```

Konstruktor der Klasse `java.lang.Integer`

```
public Integer(int value)
public Integer(String s) throws NumberFormatException
```

Konstruktor der Klasse `java.lang.Long`

```
public Long(long value)
public Long(String s) throws NumberFormatException
```

Konstruktor der Klasse `java.lang.Float`

```
public Float(float value)
public Float(double value)
```

⁴Hüllenklasse

```
public Float(String s) throws NumberFormatException
```

Konstruktor der Klasse `java.lang.Double`

```
public Double(double value)
```

```
public Double(String s) throws NumberFormatException
```

6.5.2 Vergleich von Wrapperobjekten

Alle Wrapperklassen überschreiben die Methode `equals()` aus `java.lang.Object`. Wrapperobjekte des gleichen Typs lassen sich also auf Inhaltsgleichheit prüfen:

```
1 public class Wrapper {  
2     public static void main(String []args) {  
3         Integer i1 = new Integer(10);  
4         Integer i2 = new Integer("10");  
5         Long l = new Long(10);  
6  
7         System.out.println("i1.equals(i2) liefert: " + i1.equals(i2));  
8         System.out.println("i1 == i2      liefert: " + (i1 == i2));  
9         System.out.println("i1.equals(l)  liefert: " + i1.equals(l));  
10    }  
11 }
```

Listing 6.12: Vergleich von Wrapperobjekten

Ausgabe:

```
i1.equals(i2) liefert: true  
i1 == i2      liefert: false  
i1.equals(l)  liefert: false
```

6.5.3 Auslesen der gespeicherten Werte

Alle numerischen Wrapperklassen (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`) erben von der abstrakten Klasse `java.lang.Number`. Diese ist wie folgt definiert:

```
public abstract class Number {  
    public abstract byte byteValue();  
    public abstract short shortValue();  
    public abstract int intValue();  
    public abstract long longValue();  
    public abstract float floatValue();  
    public abstract double doubleValue();  
}
```

Damit beinhalten alle numerischen Wrapperklassen diese oben definierten Methoden, wodurch der gewappte Wert in jeden der 6 primitiven Datentypen verwandelt werden kann.

Kapitel 7

Ergänzungen und Zertifizierungswissen

7.1 Assertions

7.1.1 Einführung

Mit Hilfe einer Assertion kann man Behauptungen über Programmezustände testen. Die `assert`-Anweisung dient dazu, an einer bestimmten Stelle im Programmfluss einen logischen Ausdruck zu platzieren, von dem der Programmierer annimmt, dass er stets wahr ist. Ist das der Fall, fährt das Programm mit der nächsten Anweisung fort, andernfalls wird eine Ausnahme vom Typ `AssertionError` ausgelöst.

7.1.2 Die `assert`-Anweisung

Die `assert`-Anweisung hat zwei Formen:

- `assert <Ausdruck1>`
Dabei ist `<Ausdruck1>` vom Typ `boolean`. Liefert `<Ausdruck1>` den Wert `false`, so wird ein `AssertionError` ohne detaillierter Fehlermeldung geworfen, bei `true` läuft das Programm normal weiter.
- `assert <Ausdruck1> : <Ausdruck2>`
Dabei hat `<Ausdruck1>` die gleiche Bedeutung wie bei der obigen Form. Wird ein `AssertionError` geworfen, so dient die Stringdarstellung von `<Ausdruck2>` als detaillierte Fehlermeldung. `<Ausdruck2>` darf jeder beliebige Javaausdruck mit Ausnahme eines Funktionsaufrufes mit Rückgabotyp `void` sein.

7.1.3 Verwendung von Assertions

Assertions können bzw. sollen in folgenden Programmsituationen verwendet werden:

- **Interne Invariante**
Assertions können dort verwendet werden, wo eine bestimmte Behauptung über eine Variable immer zutreffen sollte. Das folgende Beispiel zeigt eine verschachtelte `if-else` Struktur, bei der im letzten `else`-Zweig immer die Behauptung `i % 3 == 2` zutreffen sollte:

```
1 if(i % 3 == 0) { /* ... */ }  
2 else if(i % 3 == 1) { /* ... */ }  
3 else { /* ... */ }
```

Mit Hilfe einer `assert`-Anweisung lässt sich diese Behauptung überprüfen:

```

1 if(i % 3 == 0) { /* ... */ }
2 else if(i % 3 == 1) { /* ... */ }
3 else {
4     assert i % 3 == 2 : i;
5     /* ... */
6 }

```

• Flow-Control Invariante

Assertions können vorteilhaft immer an Stellen platziert werden, die der Programmfluss nie erreichen sollte. Das an dieser Stelle zu verwendende Statement lautet:

```
assert false;
```

Ein gutes Beispiel für eine solche Flow-Control Invariante ist eine `switch-case`-Struktur, in der alle möglichen Fälle des `switch`-Arguments über die `case`-Zweige abgedeckt sein sollten. Ein `default`-Zweig wäre also unnötig. Um dies zu prüfen, kann im `default`-Zweig die Behauptung `assert false;` wie folgt platziert werden:

```

switch(x) {
    case 1: /* ... */ break;
    case 2: /* ... */ break;
    case 3: /* ... */ break;
    case 4: /* ... */ break;
    default: assert false : "Wrong x: " + x;
}

```

Es gibt einige Situationen, in denen Assertions nicht verwendet werden sollen. Zwei davon sind:

- Assertions sollen nicht verwendet werden, um Parameterwerte in öffentlichen Methoden zu überprüfen. Die Reaktion einer Funktion auf falsche Übergabewerte wird in der Beschreibung festgelegt und wäre bei Behandlung mit Assertions davon abhängig, ob diese ein- oder ausgeschaltet sind. Richtig ist die Vorgangsweise, falsche Parameterwerte mit Hilfe einer `IllegalArgumentException` zu behandeln.
- Assertions dürfen nicht verwendet werden, um Befehle auszuführen, die für die Programmlogik wesentlich sind. Solche Befehle werden nicht ausgeführt, wenn Assertions abgeschaltet sind.

7.1.4 Aktivieren und Deaktivieren von Assertions

Aus Kompatibilität zu älteren JDK-Versionen sind Assertions sowohl im Compiler als auch in der VM standardmäßig deaktiviert. Um einen Quellcode zu übersetzen, der Assertions enthält, kann dem Java-Compiler ab der Version 1.4 die Option

```
-source <release>
```

übergeben werden, wobei für `release 1.4` bzw. `1.5` einzusetzen ist. Andernfalls gibt es eine Fehlermeldung, nach der `assert` ab der Version 1.4 ein Schlüsselwort ist und nicht mehr als Bezeichner verwendet werden darf. Ältere Compilerversionen melden einen Syntaxfehler. Verwendet man in der Quelldatei `AssertionTest.java` Assertions, so kann sie also mit folgendem Kommando übersetzt werden, wenn der Compiler mindestens die Version 1.4 hat:

```
javac -source 1.4 AssertionTest.java
```

Um Assertions zur Laufzeit zu aktivieren, kennt die JVM ab der Version 1.4 die Kommandozeilenoptionen `-enableassertions` und `-disableassertions`, die mit `-ea` bzw. `-da` abgekürzt werden können. Ihre Syntax lautet:

```

java [ -enableassertions | -ea ] [:PaketName... | :KlassenName ]
java [ -disableassertions | -da ] [:PaketName... | :KlassenName ]

```

Werden die Optionen ohne nachfolgenden Paket- oder Klassennamen angegeben, werden die Assertions für alle Klassen mit Ausnahme der Systemklassen `java.*` an- bzw. ausgeschaltet. Wird, durch einen Doppelpunkt getrennt, ein Klassenname angegeben, gilt die jeweilige Option nur für diese Klasse. Wird ein Paketname angegeben (von einem Klassennamen durch drei angehängte Punkte zu unterscheiden), erstreckt sich die Option auf alle Klassen innerhalb des angegebenen Pakets. Die Optionen können mehrfach angegeben werden, sie werden der Reihe nach (von links nach rechts) ausgewertet. Wird keine dieser Optionen angegeben, sind alle Assertions deaktiviert. Das folgende Beispiel zeigt einen Kommandozeilenaufzuruf zur Ausführung der Klasse `Test`, die in allen Klassen des Pakets `com.baz` Assertions enabled, dann aber die Klasse `com.baz.Foo` davon ausnimmt:

```
java -ea:com.baz... -da:com.baz.Foo Test
```

Werden die oben angegebenen Schalter ohne Argumente verwendet, so wirken sie nicht auf Systemklassen. Um Assertions in allen Systemklassen zu aktivieren bzw. zu deaktivieren, gibt es zwei eigene Schalter: `-enablesystemassertions` oder `-esa` zum Aktivieren bzw. `-disablesystemassertions` bzw. `-dsa` zum Deaktivieren.

7.2 Innere Klassen

Klassen können neben Datenfeldern und Methoden auch wieder Klassendefinitionen beinhalten. In diesem Fall spricht man von inneren Klassen. Java kennt vier unterschiedliche Typen:

- Top-Level Nested Classes und Interfaces
- Nicht statische innere Klassen
- Lokale Klassen
- Anonyme Klassen

7.2.1 Top-Level Nested Classes und Interfaces

Dabei handelt es sich um statische innere Klassen oder um innere Interfaces. Top-Level Nested Classes und Interfaces können in beliebiger Tiefe verschachtelt werden, aber nur in anderen statischen Top-Levelklassen und Interfaces. Sie dürfen jede beliebige Zugriffsart haben.

Statische Memberklassen (nested Top-Level Classes) haben wie andere statische Klassenmitglieder keine `this`-Referenz und daher nur Zugriff auf statische Mitglieder der umgebenden äußeren Klasse. Zum Erzeugen einer Instanz einer statischen inneren Klasse benötigt man auch keine Instanz der umgebenden äußeren Klasse. Es ist nur der voll qualifizierte Name zu verwenden.

Beispiel:

```

1 public class Outer {
2     protected static class Inner {}      // Nested Top-Level-Class
3
4     public static void main(String args[]) {
5         Inner i = new Outer.Inner();
6     }
7 }
```

Listing 7.1: Statische innere Klasse

Der Compiler erzeugt aus obiger Quelldatei die folgenden `class`-Dateien:

```
Outer.class
Outer$Inner.class
```

7.2.2 Nicht statische innere Klassen

Nicht statische innere Klassen sind vergleichbar mit anderen nicht statischen Mitgliedern einer Klasse.

- Eine Instanz kann nur mit Hilfe einer existierenden Instanz der umgebenden äußeren Klasse erzeugt werden.

- Eine nicht statische innere Klasse darf selbst keine statischen Mitglieder haben.
- Methoden einer nicht statischen inneren Klasse haben direkten Zugriff auf alle Mitglieder aller umgebenden äußeren Klassen (auch auf private), wobei keine explizite Referenz notwendig ist. In einer inneren Klasse spricht man das zugehörige Objekt der umgebenden äußeren Klasse `Outer` über `Outer.this` an.
- Eine nicht statische innere Klasse darf jeden Zugriffsmodifizier haben.

Beispiel:

```

1 class Outer {
2     private int i = 10;
3     static int k = 5;
4
5     protected class Inner {
6         int i = 30;
7         // static int l = 20; - Compilerfehler
8
9         public void printVars() {
10             System.out.println(Outer.this.i + " " + i + " " + k);
11         }
12     }
13
14     public Inner makeInstanceInner() {
15         return new Inner(); // return this.new Inner();
16     }
17 }
18
19 class Main {
20     public static void main(String []args) {
21         Outer outRef = new Outer();
22         Outer.Inner inRef1 = outRef.makeInstanceInner();
23         inRef1.printVars();
24         Outer.Inner inRef2 = new Outer().new Inner();
25         inRef2.printVars();
26     }
27 }

```

Listing 7.2: Nicht statische innere Klasse

Ausgabe:

```

10 30 5
10 30 5

```

7.2.3 Lokale Klassen

Lokale Klassen sind innere Klassen, die innerhalb einer Methode oder eines Blocks der umgebenden Klasse definiert sind. Solche Klassen sind lokal bezüglich der Methode oder des Blocks und haben daher keinen Zugriffsmodifizier und können auch nicht explizit als `static` definiert werden. Sie sind aber implizit `static`, wenn sie sich in einer statischen Methode befinden. In einer lokalen inneren Klasse darf es keine statischen Mitglieder geben und es darf auch nur auf finale Datenfelder der umgebenden Methode zugegriffen werden.

Lokale Klassen können nur in dem Block oder der Methode, in der sie definiert wurden, instanziiert werden. Die Methode kann aber die Instanz der lokalen Klasse retournieren. Der Rückgabebetyp dieser Methode muss dann mit dem Typ der lokalen Klasse verträglich sein (z.B. der Typ einer Superklasse oder eines Interfaces).

Beispiel:

```

1 interface I {
2     void foo();
3 }
4
5 class Outer {
6     private int i = 10;
7     static int j = 5;
8
9     public I fooOuter(final int k) {

```

```

10  int l = 20;
11  class Lokal implements I {
12      public void foo() {
13          i = 100;        // ok
14          j = 105;        // ok
15          // l = 115;      Compilerfehler
16          System.out.println(i + " " + j + " " + k);
17      }
18  }
19  return new Lokal();
20  }
21  }
22
23  class Main {
24      public static void main(String []args) {
25          I iRef = new Outer().fooOuter(30);
26          iRef.foo();
27      }
28  }

```

Listing 7.3: Lokale innere Klasse

Ausgabe:

100 105 30

7.2.4 Anonyme innere Klassen

Anonyme innere Klassen sind immer lokal in einem Block oder in einer Methode. Es handelt sich dabei um Klassen ohne Namen, die genau einmal instanziiert werden können. Sie können entweder von einer Superklasse abgeleitet werden oder ein vorgegebenes Interface implementieren. Beides gleichzeitig verbietet die Syntax.

Klassendefinition und Instanziierung erfolgen genau an der Stelle, an der das Objekt benötigt wird. Lokale Klassen sind implizit `final`. Sie können nicht `abstract` sein, da auf jeden Fall genau ein Objekt instanziiert wird. Das folgende Beispiel erzeugt ein Objekt einer anonymen lokalen Klasse, die das Interface `ActionListener` implementiert.

```

1  public void aMethod () {
2      button.addActionListener(new ActionListener() {
3          public void actionPerformed(ActionEvent e) {}
4      });
5  }

```

Listing 7.4: Anonyme innere Klasse

Die Definition und Instanziierung einer anonymen lokalen Klasse erfolgt also mit Hilfe der Syntax

```
new XXX() { /* Klassenkoerper */ };
```

wobei `xxx` entweder die Superklasse oder das zu implementierende Interface angibt.

Ist `xxx` eine Superklasse, so können dem Superklassenkonstruktor mit Hilfe der folgenden Syntax auch Argumente übergeben werden:

```
new <superclass-name> ([<argument list>]) { <class definition> };
```

Ist `xxx` ein Interface, so kann nur die folgende Syntax verwendet werden. Die anonyme Klasse erbt dabei implizit von `java.lang.Object`:

```
new <interface-name> () { <class definition> };
```

7.3 Autoboxing und Unboxing

Bis Java 1.4 waren primitive Typen und ihre Wrapperobjekte prinzipiell unverträglich und konnten nur durch geeignete Methoden ineinander übergeführt werden. Wollte man z.B. den in einem Objekt vom Typ `java.lang.Integer` gespeicherten Wert um 1 erhöhen, so war das nur mit folgenden Befehlen möglich:

```
Integer io = new Integer(8);
int ip = io.intValue();           // gewrappten Wert auslesen
ip++;                             // primitiven Wert erhöhen
io = new Integer(ip);             // neuen Wrapper erzeugen
```

Dies lässt sich auch kürzer mit Hilfe einer Zeile programmieren:

```
io = new Integer(io.intValue()+1);
```

In beiden Versionen wird zunächst der im Wrapperobjekt `io` gespeicherte Wert in einen primitiven Integer verwandelt, dieser wird dann inkrementiert und aus dem so um 1 erhöhten primitiven Wert wird ein neues Wrapperobjekt erzeugt. Seit Java 1.5 werden diese notwendigen Umwandlungen durch Autoboxing/Unboxing automatisch durchgeführt und brauchen daher nicht mehr explizit ausprogrammiert zu werden.

- **Autoboxing**

Autoboxing bedeutet, dass aus dem Wert eines primitiven Datentyps automatisch ein entsprechendes Wrapperobjekt erzeugt wird.

- **Unboxing**

Unboxing bedeutet, dass aus einem Wrapperobjekt automatisch ein primitiver Datentyp erzeugt wird.

Das oben angegebene Beispiel lässt sich seit Java 1.5 also wie folgt programmieren:

```
Integer io = new Integer(8);
io++;
```

Bemerkung:

Ein Nachteil dieser Technik ist, dass aus dem Code nicht mehr immer erkennbar ist, ob es sich bei einer Variablen um einen primitiven Typ oder um eine Referenz auf ein Wrapperobjekt handelt.

7.3.1 Autoboxing/Unboxing bei Funktionsparametern

Das folgende Beispiel zeigt, dass Autoboxing/Unboxing auch bei Funktionsaufrufen verwendet werden kann:

```
1 public class Autoboxing_1 {
2     public static void main(String[] args) {
3         int primitiveValue = 9;
4         Integer wrapper = new Integer(8);
5         autobox(primitiveValue);
6         unbox(wrapper);
7         wrapper = null;
8         unbox(wrapper);
9         // unbox(null);    -->    Compilerfehler
10    }
11    static void autobox(Integer val) {
12        System.out.println(val);
13    }
14
15    static void unbox(int val) {
16        System.out.println(val);
17    }
18 }
```

Listing 7.5: Autoboxing/Unboxing

Ausgabe:

```
9
8
Exception in thread "main" java.lang.NullPointerException
    at Autoboxing_1.main(Autoboxing_1.java:8)
```

Das Unboxing in Zeile 8 führt auf eine `NullPointerException`, da zu diesem Zeitpunkt der Referenz `wrapper` der Wert `null` zugewiesen ist. Der Versuch, die Methode direkt mit dem Wert `null` aufrufen (Zeile 9) führt auf einen Compilerfehler.

7.3.2 Autoboxing/Unboxing bei Funktionsüberladung

Um *legacy code*¹ robuster zu machen, wird beim Überladen von Funktionen das Widning dem Autoboxing vorgezogen. Außerdem gilt Widning vor Varargs:

- Widning vor Autoboxing
- Autoboxing vor Varargs

Das folgende Beispiel demonstriert diese Regeln:

```

1 public class Autoboxing_2 {
2
3     static void go1(Integer x) {
4         System.out.println("go1 Integer");
5     }
6     static void go1(long x) {
7         System.out.println("go1 long");
8     }
9
10    static void go2(int x, int y) {
11        System.out.println("go2 int,int");
12    }
13    static void go2(byte... x) {
14        System.out.println("go2 byte... ");
15    }
16    static void go2(Byte x, Byte y) {
17        System.out.println("go2 Byte, Byte");
18    }
19
20    public static void main(String [] args) {
21        int i = 5;
22        go1(i); // which go() will be invoked?
23
24        byte b = 5;
25        go2(b, b);
26    }
27 }

```

Listing 7.6: Overloading und Autoboxing

Ausgabe:

```

go1 long
go2 int,int

```

Da die einzelnen numerischen Wrappertypen alle direkt von `java.lang.Number` erben findet in diesem Bereich kein Widning statt. Das bedeutet, dass z.B. die Methode

```
public static void foo(Long l) { }
```

nicht mit dem Aufruf

```
foo(2);
```

verwendet werden kann. Der Integer 2 kann mit Autoboxing zwar in ein `Integer`-Objekt gewrappt werden, dieser Typ wird wegen der oben beschriebenen Unverträglichkeit aber nicht in den Typ `Long` erweitert.

7.3.3 Autoboxing/Unboxing bei Vergleichen

Hier gelten die folgenden Regeln:

- Vergleicht man ein Wrapperobjekt und einen primitiven Typ mit den Operatoren `==` bzw. `!=` so findet immer Unboxing statt, d.h. der Wrappertyp wird vor dem Vergleich in einen primitiven Typ verwandelt. Der Vergleich von zwei Wrapperobjekten bzw. zwei primitiven Typen unterliegt keinen neuen Regeln.

¹Code vor Java 1.5

- Die Operatoren `<` `<=` `>` `>=` können nun auch auf ein oder zwei Wrapperobjekte angewendet werden. Hier findet immer Unboxing statt.
- Die Vergleichsfunktion `equals()` kann nur über einen Wrappertyp und nicht über einen primitiven Typ aufgerufen werden, da in Java der Punktoperator nicht auf einen primitiven Typ angewendet werden kann. Wird der Methode `equals()` ein primitiver Typ übergeben, so findet wie oben beschrieben Autoboxing statt.

Die ersten beiden Vergleiche in obigem Beispiel demonstrieren diese Regeln:

```

1 public class Autoboxing_3 {
2     public static void main(String[] args) {
3         int primitiveValue = 9;
4         Long wrapper = new Long(9);
5         System.out.println("1: " + (primitiveValue == wrapper));
6         System.out.println("2: " + (primitiveValue < wrapper));
7         //Compilerfehler:
8         //System.out.println(primitiveValue.equals(wrapper));
9         System.out.println("3: " + wrapper.equals(primitiveValue));
10
11        Integer i1 = 100;
12        Integer i2 = 100;
13        System.out.println("4: " + (i1 == i2));
14        Integer j1 = 1000;
15        Integer j2 = 1000;
16        System.out.println("5: " + (j1 == j2));
17    }
18 }

```

Listing 7.7: Autoboxing/Unboxing bei Vergleichen

Ausgabe:

```

1: true
2: false
3: false
4: true
5: false

```

Die Ergebnisse der Vergleiche 4 und 5 sind auf den ersten Blick verblüffend. Hier werden jeweils zwei Wrapperobjekte verglichen, der Operator `==` liefert also genau dann `true`, wenn es sich um ein und dasselbe Objekt handelt. Um dieses Verhalten zu verstehen, muss man wissen, dass Wrapperobjekte für gewisse Wertebereiche von der VM automatisch erzeugt werden und beim Wrappen von Literalen primitiver Typen Verwendung immer auf diese automatisch erzeugten Objekte² zugegriffen wird. Die Wertebereiche der so generierten und damit in einer VM eindeutigen Wrapperobjekte sind:

- Boolean-Objekte für `true` und `false`
- Alle Objekte vom Typ `Byte`
- Short-Objekte im Bereich von -128 bis 127
- Integer-Objekte im Bereich von -128 bis 127
- Charakter-Objekte im Bereich von `\u0000` bis `\u007F`

7.4 Enums

7.4.1 Konzept

Bis Java 1.4 wurden Aufzählungen in Java hauptsächlich über Konstante definiert, denen einzelne Integerwerte zugewiesen wurden:

²Wrapperpool

```

1 public class Marks1 {
2     public static final int EXCELLENT = 1;
3     public static final int GOOD = 2;
4     public static final int SATISFACTORY= 3;
5     public static final int SUFFICIENT = 4;
6     public static final int INSUFFICIENT= 5;
7 }

```

Diese Vorgangsweise ist nicht typsicher, da einem primitiven Integer neben den oben definierten Noten auch andere Werte zugewiesen werden können:

```

int mark1 = Marks1.GOOD;
int mark2 = 7;

```

In Java 1.5 wurden mit Enums typsichere Aufzählungen eingeführt. Diese arbeiten prinzipiell nach dem in folgendem Beispiel vorgestellten Konzept:

```

1 public class Marks2 {
2     private String markName;
3     public static final Marks2 EXCELLENT = new Marks2("sehr gut");
4     public static final Marks2 GOOD = new Marks2("gut");
5     public static final Marks2 SATISFACTORY= new Marks2("befriedigend");
6     public static final Marks2 SUFFICIENT = new Marks2("ausreichend");
7     public static final Marks2 INSUFFICIENT= new Marks2("mangelhaft");
8
9     private Marks2(String name) {
10         this.markName = name;
11     }
12
13     public String toString() {
14         return this.markName;
15     }
16 }

```

Listing 7.8: Konzept einer Enum

Die Typsicherheit ist durch den privaten Konstruktor (Zeilen 9-11) gewährleistet. Dadurch wird verhindert, dass (von außerhalb der Klasse `Marks2`) weitere Objekte erzeugt werden können.

7.4.2 Definition einer Enum

In Java 1.5 wurde mit dem Aufzählungstyp `enum`³ ein Konzept implementiert, das der oben vorgestellten Klasse `Marks2` entspricht. Die Syntax zur Erzeugung einer Enum lautet:

```
[<Modifier>] enum <Bezeichner>{<Wert1>, <Wert2>, ...};
```

Das folgende Beispiel definiert wieder Noten:

```

public enum Marks3 {EXCELLENT, GOOD, SATISFACTORY,
                    SUFFICIENT, INSUFFICIENT};

```

Im Folgenden werden die wichtigsten Eigenschaften von Enums zusammengestellt:

- `enum`-Aufzählungen sind Klassen
Dadurch wird die Typsicherheit gewährleistet. Die Syntax zur Verwendung einer Enum entspricht jener einer Klasse. Ein Aufzählungstyp kann überall dort verwendet werden, wo ein Klassentyp erlaubt ist. Eine Enum ist dabei eine spezielle Klasse und wird auch zu einer `*.class`-Datei kompiliert. Sie darf statische Methoden enthalten, im Speziellen auch eine Startmethode `main()`.
- Enums erben von der Klasse `java.lang.Enum`
Die Klasse `java.lang.Enum` gibt es seit Java 1.5. Sie ist selbst keine Aufzählung, definiert aber das Verhalten jeder Enum.
- Enums sind standardmäßig `final`. Damit wird verhindert, dass man Aufzählungen erweitern kann. Der Modifier `final` braucht (und darf nicht) angegeben werden.

³neues Schlüsselwort in 1.5

- Enums haben standardmäßig einen privaten Konstruktor
Damit ist es unmöglich, Objekte zu erzeugen, die nicht von der Klasse selbst zur Verfügung gestellt werden.
- Alle Aufzählungsobjekte besitzen die Modifier `public`, `static` und `final`.

Von der Klasse `java.lang.Enum` erben Enums die folgenden Eigenschaften:

- `public final boolean equals(Object other)`
Dient zum Vergleich zweier Aufzählungsobjekte. Liefert genau dann `true`, wenn es sich um ein und dasselbe Objekt handelt.
- `implements java.lang.Comparable`
Damit erbt jeder Aufzählungstyp die Methode
`public final int compareTo(E o)`
Mit dieser Methode lassen sich Aufzählungsobjekte auf kleiner, gleich und größer vergleichen. Die dabei verwendete natürliche Reihenfolge entspricht jener, die bei der Definition der einzelnen Objekte angegeben wurde.
- `implements java.io.Serializable`
Jedes Aufzählungsobjekt ist also serialisierbar.
- `public String toString()`
Diese Methode aus `java.lang.Enum` liefert den Namen eines Aufzählungsobjektes. Zum Beispiel liefert der Aufruf `Marks3.GOOD.toString()` den String `"GOOD"`.
- `public static <T extends Enum<T>> T valueOf(String name)`
Bietet die umgekehrte Funktionalität von `toString()`. Es wird das Aufzählungsobjekt zum angegebenen Namen `name` geliefert. Gibt es den angegebenen Namen nicht, so wird eine `IllegalArgumentException` geworfen. Der Aufruf `Marks3.valueOf("GOOD")` retourniert das Aufzählungsobjekt `Marks3.GOOD`.
- `public final int ordinal()`
Diese Methode liefert die nullbasierte Position des Aufzählungsobjektes in der Aufzählung.
- `public static <T extends Enum<T>> T[] values()` Diese Methode liefert alle Aufzählungsobjekte einer Aufzählung als Feld. Damit ist es möglich, über alle Aufzählungswerte zu iterieren.

Das folgende Beispiel demonstriert die angegebenen Methoden an Hand der Enum `Marks3`:

```

1 public enum Marks3 {
2     EXCELLENT, GOOD, SATISFACTORY, SUFFICIENT, INSUFFICIENT;
3
4     public static void main(String[] args) {
5         Marks3 m1 = Marks3.EXCELLENT;
6
7         // equals()
8         boolean b1, b2;
9         b1 = m1.equals(Marks3.EXCELLENT);
10        b2 = m1.equals(Marks3.GOOD);
11        System.out.format("%b,%b\n", b1, b2);
12
13        // compareTo()
14        int i1, i2;
15        i1 = Marks3.SATISFACTORY.compareTo(m1);
16        i2 = Marks3.SATISFACTORY.compareTo(SUFFICIENT);
17        System.out.format("%d,%d\n", i1, i2);
18
19        // valueOf(), toString()
20        m1 = Marks3.valueOf(Marks3.class, "GOOD");
21        System.out.format("%s\n", m1.toString());
22
23        // values(), ordinal()
24        for(Marks3 akt : Marks3.values()) {
25            System.out.format("%d: %-15s\n", akt.ordinal(), akt);
26        }
27    }
28 }

```

Listing 7.9: Grundfunktionalität einer Enum

Ausgabe:

```
true, false
2, -1
GOOD
0: EXCELLENT
1: GOOD
2: SATISFACTORY
3: SUFFICIENT
4: INSUFFICIENT
```

7.4.3 Enums und Switch-Case

Die switch-case-Struktur wurde in Java 1.5 so erweitert, dass als Argumente auch Enum-Typen akzeptiert werden:

```
1 public String verbal(Marks3 m) {
2     switch(m) {
3         case EXCELLENT : return "hervorragend";
4         case GOOD      : return "immer noch super";
5         case SATISFACTORY: return "durchschnittlich";
6         case SUFFICIENT : return "nicht so toll";
7         case INSUFFICIENT: return "mal wieder was lernen";
8         default:        return null;
9     }
10 }
```

Listing 7.10: Enum in einer switch-case

7.5 Generics

Mit den seit Java 1.5 eingeführten Generics können Schablonen für Klassen und Methoden geschrieben werden, um ein und denselben Code für mehrere Datentypen verwenden zu können.

Das folgende Beispiel zeigt eine einfache generische Klasse `Box`, in die spezielle Objekte gespeichert werden können. Dabei wird bei der Klassendefinition in spitzen Klammern ein Typ-Stellvertreter angegeben. Dieser Typ-Stellvertreter wird bei der Instanziierung mit einem speziellen Typ überschrieben. Innerhalb der Klasse kann nun dieser Typ-Stellvertreter wie ein normaler Referenztyp verwendet werden:

```
1 class Box<T> {
2     private T val;
3
4     public Box() {
5     }
6
7     public Box(T val) {
8         this.val = val;
9     }
10
11    public void setValue(T val) {
12        this.val = val;
13    }
14
15    public T getValue() {
16        return val;
17    }
18 }
```

Listing 7.11: Einfache generische Klasse

In der folgenden lauffähigen Klasse wird die generische Klasse `Box` verwendet.

```
1 public class SimpleGeneric {
2     public static void main(String []args) {
3         Box objectBox = new Box();
4         objectBox.setValue("hallo");
```

```

5   String s = (String) objectBox.getValue();
6   System.out.println(s);
7
8   Box<String> stringBox = new Box<String>();
9   Box<Integer> intBox = new Box<Integer>();
10  stringBox.setValue(new String("hallo"));
11  intBox.setValue(12); // Autoboxing
12  System.out.println(stringBox.getValue().toUpperCase());
13  System.out.println(intBox.getValue().toString());
14  }
15  }

```

Listing 7.12: Verwendung einer generischen Klasse

Ausgabe:

```

hallo
HALLO
12

```

- Versorgt man wie in Zeile 3 den Typ-Stellvertreter nicht, so erhält man eine nicht typsichere Instanz der Klasse `Box`, in der beliebige Javaobjekte gespeichert werden können. Der Typ-Stellvertreter wird implizit durch den Typ `Object` ersetzt. Beim Auslesen eines Objektes mit Hilfe der Methode `getValue()` erhält man nun den Rückgabotyp `Object` und muss das Ergebnis z.B. auf den Typ `String` casten, damit Stringmethoden angewendet werden können. Verwendet man eine generische Klasse ohne Versorgung des Typ-Stellvertreters durch einen speziellen Typ, so erzeugt der Compiler die Warnung:

```

SimpleGeneric.java uses unchecked or unsafe operations
Recompile with -Xlint:unchecked for details.

```

Aus Gründen der Aufwärtskompatibilität ist die Unterdrückung eines angegebenen Typ-Stellvertreters prinzipiell erlaubt. Die richtige Verwendung von Generics wird nur vom Compiler geprüft, das Laufzeitsystem kennt Generics nicht. Ignoriert man die oben angegebene Warnung, kann zur Laufzeit die korrekte Verwendung von Generics nicht garantiert werden.

- In Zeile 8 wird eine typsichere `Box`-Instanz erzeugt. In dieser Instanz können nur Objekte vom Typ `String` gespeichert werden. Jeder andere Versuch, z. B. `stringBox.setValue(new Double(3.8));` führt auf einen Compilerfehler.
- Die Methode `getValue()` des typsicheren Objekts `stringBox` liefert den Typ `String`, wodurch die erhaltene Referenz direkt (d.h. ohne Cast) Stringmethoden aufrufen darf.

Die Vorteile bei der Verwendung generischer Klassen sind also:

1. Bessere Wiederverwertbarkeit durch generische Programmierung
2. Die Möglichkeit, Containerobjekte typsicher zu machen
3. Die Vermeidung sonst notwendiger Casts bei der Programmierung

Das Konzept von Generics kann auch auf Methodenebene herabgebrochen werden. Hier muss der Typ-Stellvertreter in der Reihe der Modifier angegeben werden. Die generische Methode `create()` im folgenden Beispiel erzeugt ein typsicheres `Box`-Objekt, wobei der gewünschte Typ über das Argument festgelegt wird.

```

1 public class GenericMethods {
2
3     public static void main(String[] args) {
4         Box<String> b1= createBox("xxx");
5         Box<Integer> b2 = createBox(13);
6         //Box<String> b3= createNumberBox("xxx");
7         Box<Integer> b4 = createNumberBox(13);
8
9         System.out.println(b1);
10        System.out.println(b2);

```

```
11     System.out.println(b4);
12 }
13
14 public static<T> Box<T> createBox(T f) {
15     return new Box<T>(f);
16 }
17
18 public static<T extends Number> Box<T> createNumberBox(T f) {
19     return new Box<T>(f);
20 }
21 }
```

Listing 7.13: Generische Methode

Ausgabe:

```
java.lang.String
java.lang.String
java.lang.Integer
```

Die Methode `createNumberBox` zeigt eine weitere Sprachmöglichkeit bei der Verwendung von Typ-Stellvertretern. Diese können mit `extends` so eingeschränkt werden, dass nur Typen erlaubt sind, die im Rahmen der "is-a"-Beziehung mit `Number` verträglich sind. Zeile 6 würde nun auf einen Compilerfehler führen, da das an die Methode `createNumberBox` gesendete Objekt vom Typ `String` und damit nicht vom Typ `Number` ist. Alternativ ist auch die Einschränkung mit `super` möglich.

Das nächste Beispiel zeigt, dass die Technik auch auf Interfaces anwendbar ist. Die generische Methode `max` bestimmt mit Hilfe der Methode `compareTo` das größere von zwei übergebenen Javaobjekten, die (1) beide vom gleichen Typ sein müssen und (2) beide das Interface `java.lang.Comparable` implementieren müssen:

```
1 // Generische Methode mit Typ-Einschraenkung
2 public <C extends Comparable<C>> C max(C a, C b) {
3     return a.compareTo(b) > 0 ? a : b;
4 }
5 System.out.println(max("A", "Z"));
```

Listing 7.14: Generische Methode mit Typ-Einschränkung

Kapitel 8

Erstellung graphischer Benutzerschnittstellen

Objektorientierte Sprachen wie JAVA stellen zur Erstellung von Programmen mit grafischem Userinterface (GUI) in der Regel umfangreiche Klassenbibliotheken zur Verfügung. Die Fähigkeiten einer solchen Bibliothek lassen sich grob in folgende 4 Gruppen unterteilen:

- Elemente der Benutzeroberfläche: Fenster, Menüs, Schaltflächen, Kontrollfelder, Textfelder, Bildlaufleisten und Listfelder und ihre Anordnung mit Hilfe von Containern und Layout-Managern.
- Steuerung des Programmablaufs auf der Basis von Nachrichten für die Handhabung von System- und Benutzerereignissen.
- Primitivoperationen zum Zeichnen von Linien oder Füllen von Flächen und zur Grafikausgabe von Text.
- Fortgeschrittene Grafikfunktionen zur Darstellung und Manipulation von Bitmaps und zur Ausgabe von Sound.

8.1 AWT und Swing

Zur Erstellung von Programmen mit grafischer Benutzerschnittstellen gibt es in Java mehrere Möglichkeiten. Die beiden wichtigsten sind:

1. Abstract Windowing Toolkit - AWT

Das Paket `java.awt` stellt Klassen für die Gestaltung grafischer Oberflächen zur Verfügung. Dabei verwendet das AWT wo immer möglich Elemente des zugrundeliegenden Fenstersystems (sogenannte schwergewichtige Komponenten). Das Look and Feel eines AWT-Programmes entspricht also jenem des verwendeten Fenstersystems (Windows, Linux, ...). Das AWT stellt auch nur die einfachsten Steuerelemente (Controls) zur Verfügung, da die AWT-Controls in ähnlicher Form und Funktionalität auf allen unterstützten Plattformen verfügbar sein müssen.

2. Swing

Swing gehört zu den sogenannten Java-Foundation-Classes (Grundpaket `javax.swing`) und verwendet im Gegensatz zum AWT wo immer möglich sogenannte leichtgewichtige Komponenten, zeichnet also alle Controls selbst. Nur die Fenster selbst sind schwergewichtig und werden vom verwendeten Fenstersystem zur Verfügung gestellt. Damit kann Swing im Gegensatz zum AWT eigene komplexe Controls zur Verfügung stellen. Beispiele dafür sind etwa Tabellensteuerelemente und Tree-Controls.

In diesem Skriptum werden ausschließlich Swing-Applikationen vorgestellt.

8.1.1 Überblick Swing

Swing ist kein Ersatz für das AWT sondern baut auf diesem auf. Es ist seit dem JDK 1.1 verfügbar. Im Folgenden werden die wichtigsten Konzepte von Swing kurz vorgestellt.

8.2.2 Die Klasse `java.awt.Container`

Objekte der Klasse `Container` sind in der Lage, andere Komponenten aufzunehmen. Die Klasse `Container` stellt Methoden zum Hinzufügen und Entfernen von Komponenten zur Verfügung. Diese Vorgänge werden in Zusammenarbeit mit den Layout-Manager-Klassen realisiert, welche die Positionierung und Anordnung der Komponenten steuern.

8.2.3 Die Klasse `javax.swing.JComponent`

Diese Klasse ist Basisklasse aller Swingkomponenten mit Ausnahme der Top-Level-Container `JFrame`, `JDialog` und `JApplet` und der Menükomponenten.

8.2.4 Die Klasse `javax.swing.JPanel`

Ein `JPanel` ist ein generischer leichtgewichtiger Container und kann zur Verschachtelung von Swing-Komponenten und als Basisklasse für die Entwicklung eigener Swingkomponenten herangezogen werden.

8.2.5 Die Klasse `java.awt.Window`

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

8.2.6 Die Klasse `java.awt.Frame`

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

8.2.7 Die Klasse `javax.swing.JFrame`

Stellt die Swing-Variante eines vollwertigen Fensters des jeweiligen Betriebssystems dar. Es handelt sich um eine schwergewichtige Komponente.

8.2.8 Die Klasse `java.awt.Dialog`

Die Klasse `Dialog` ist dafür vorgesehen, modale oder nicht-modale Dialoge zu realisieren. Ein modaler Dialog ist ein Fenster, das immer im Vordergrund des Fensters bleibt, von dem es erzeugt wurde, und das alle übrigen Fensteraktivitäten und Ereignisse so lange blockiert, bis es geschlossen wird. Ein nicht-modaler Dialog erlaubt es, im aufrufenden Fenster weiterzuarbeiten, auch wenn der Dialog noch nicht geschlossen wurde.

8.2.9 Die Klasse `javax.swing.JDialog`

Diese Klasse kann als Basisklasse für der Erstellung eines Swing-Dialoges verwendet werden. Es handelt sich um eine schwergewichtige Komponente.

8.2.10 Die Klasse `java.awt.Panel`

`Panel` wird in der Praxis vor allem verwendet, um AWT-Container zu verschachteln, damit aus einfachen Layouts komplexe Oberflächen aufgebaut werden können.

8.2.11 Die Klasse `java.applet.Applet`

Die für die Entwicklung von Applets grundlegende Klasse `Applet` ist eine direkte Subklasse von `Panel`. Sie erweitert die Funktionalität der Klasse `Panel` um Methoden, die für das Ausführen von Applets von Bedeutung sind, beschreibt aber letztlich ein Programmelement, das eine Größe und Position hat, auf Ereignisse reagieren kann und in der Lage ist, andere Komponenten aufzunehmen.

8.2.12 Die Klasse `javax.swing.JApplet`

Diese Klasse kann als Basisklasse für der Erstellung von Swing-Applets verwendet werden. Es handelt sich um eine schwergewichtige Komponente.

8.3 Wichtige Klassen aus `java.awt` und `javax.swing`

In diesem Abschnitt werden wichtige Klassen aus den beiden Paketen `java.awt` und `javax.swing` vorgestellt und die wichtigsten Methoden erläutert. Eine vollständige Übersicht liefert die API-Dokumentation. Zunächst wird ein Grundgerüst für eine Swingapplikation erstellt, das in den folgenden Beispielen konsequent herangezogen wird.

8.3.1 Grundgerüst für eine Swingapplikation

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 public class SwingMain extends JFrame {
5     private JPanel p = null;
6
7     public SwingMain() {
8         super("Swing - Grundgerüst");
9         this.p = new JTestPanel();
10        this.setLocation(50, 50);
11        this.getContentPane().add(this.p, BorderLayout.CENTER);
12        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13        this.pack();
14    }
15
16    public static void main(String... args) {
17        EventQueue.invokeLater(new Runnable() {
18            public void run() {
19                new SwingMain().setVisible(true);
20            }
21        });
22    }
23 }
24
25 class JTestPanel extends JPanel {
26     public JTestPanel() {
27         this.setPreferredSize(new Dimension(400, 170));
28         this.setBackground(Color.RED);
29     }
30 }
```

Listing 8.1: Grundgerüst für eine Swingapplikation

Erklärung des Beispiels:

- In den Zeilen 25 - 29 wird eine eigene Komponente `JTestPanel` definiert. Diese Komponente erbt von `JPanel`. In Zeile 27 wird die bevorzugte Größe dieser Komponente mit 400 mal 170 Pixel definiert und in Zeile 28 wird die Hintergrundfarbe der Komponente auf rot gesetzt.
- In den Zeilen 4 - 23 wird Klasse `SwingMain` definiert. Diese erbt von `JFrame` und beschreibt das Hauptfenster der Applikation. Der Konstruktor setzt in Zeile 8 über den Konstruktoraufwurf der Superklasse den Titel und in Zeile 10 die Startposition des Fensters am Desktop. In Zeile 9 wird eine Komponente vom Typ `JTestPanel` erzeugt und in Zeile 11 wird diese Komponente in das

Fenster gelegt. auf Grund des `pack()`-Aufrufes in Zeile 13 wird die Startgröße des Fensters so gewählt, dass die preferred Size der Komponente `p` berücksichtigt wird. Zeile 12 legt die Verhalten des Fensters fest, wenn es geschlossen werden soll. In diesem Fall wird die VM und damit die Applikation beendet.



Abbildung 8.2: Wichtige AWT- und Swing-Klassen

- Die Startmethode `main()` hat die Aufgabe ein Objekt vom Typ `SwingMain` anzulegen und sichtbar zu machen. Da Swingkomponenten aus Performancegründen nicht threadsafe sind, muss gewährleistet werden, dass alle Zugriffe auf Swingkomponenten von einem einzelnen Thread aus getätigt werden - dem sog. Event-Dispatcher. Die statische Methode `invokeLater()` der Klasse `java.awt.EventQueue` gewährleistet, dass die `run()`-Methode des übergebenen `Runnable`-Objektes in diesem Event-Dispatcherthread abgearbeitet wird.

8.3.2 Die Klasse `java.awt.Color`

Das Java-Farbmodell basiert auf dem RGB-Farbmodell. Dieses stellt Farben mit 24 Bit Tiefe dar und setzt jede Farbe aus einer Mischung der drei Grundfarben Rot, Grün und Blau zusammen. Da jede dieser Grundfarben einen Wert von 0 bis 255 haben kann und so mit jeweils 8 Datenbits darstellbar ist, ergibt sich eine gesamte Farbtiefe von 24 Bit. Ein Wert von 0 für eine der Grundfarben bedeutet dabei, dass diese Grundfarbe nicht in das Ergebnis eingeht, ein Wert von 255 stellt die maximale Intensität dieser Grundfarbe ein. RGB-Farben werden üblicherweise durch ganzzahlige Tripel (r, g, b) dargestellt, die den Anteil an der jeweiligen Grundfarbe in der Reihenfolge Rot, Grün und Blau angeben.

Farben werden in Java durch die Klasse `java.awt.Color` repräsentiert. Jedes `Color`-Objekt repräsentiert dabei eine Farbe, die durch ihre RGB-Werte eindeutig bestimmt ist. Farben können durch Instanzieren eines `Color`-Objekts und Übergabe des gewünschten RGB-Wertes an den Konstruktor erzeugt werden:

```
public Color(int r, int g, int b)
```

Erzeugt ein RGB-Farbojekt mit den angegebenen rot-grün-blau Werten im Bereich von jeweils 0-255.

```
public Color(float r, float g, float b)
```

Erzeugt ein RGB-Farbojekt mit den angegebenen rot-grün-blau Werten im Bereich von jeweils $0.0F - 1.0F$

Alternativ stellt die Klasse `Color` ein Reihe von statischen finalen `Color`-Objekten für die Hauptfarben zur Verfügung, die direkt verwendet werden können:

```
Color.BLACK, Color.BLUE, Color.DARK_GRAY, Color.GRAY,
Color.GREEN, Color.MAGENTA, Color.ORANGE, Color.PINK,
Color.RED, Color.WHITE und Color.YELLOW.
```

Um von einem bestehenden Farbojekt die RGB-Werte zu ermitteln, stellt die Klasse `Color` die folgenden Methoden zur Verfügung:

```
public int getRed()
public int getGreen()
```

```
public int getBlue()
```

Liefern die rot-grün-blau Werte des aktuellen `Color`-Objektes im Bereich von jeweils 0-255.

Die Klasse `java.awt.SystemColor`

Um ihren Anwendungen ein einheitliches Look and Feel zu geben, definieren grafische Oberflächen in der Regel eine Reihe von Systemfarben. Diese können im Programm verwendet werden, um beispielsweise die Farbe des Hintergrundes oder die von Dialogelementen konsistent festzulegen. Ab der JDK-Version 1.1 gibt es mit Hilfe der finalen Klasse `java.awt.SystemColor` (Subklasse von `Color`) die Möglichkeit, auf diese Systemfarben zuzugreifen. Einige der wichtigsten Farbkonstanten der Klasse `SystemColor` sind in folgender Tabelle zusammengestellt. Es handelt sich dabei um öffentliche finale statische Datenfelder vom Typ `SystemColor`:

Name	Bedeutung
<code>SystemColor.desktop</code>	Hintergrundfarbe des Desktops
<code>SystemColor.activeCaption</code>	Hintergrundfarbe für die Titelleiste des aktiven Fensters
<code>SystemColor.activeCaptionText</code>	Schriftfarbe für die Titelleiste des aktiven Fensters
<code>SystemColor.window</code>	Hintergrundfarbe für Fenster
<code>SystemColor.windowBorder</code>	Farbe für Fensterrahmen
<code>SystemColor.windowText</code>	Farbe für Fenstertext
<code>SystemColor.menu</code>	Hintergrundfarbe für Menüs
<code>SystemColor.menuText</code>	Textfarbe für Menüs
<code>SystemColor.text</code>	Hintergrundfarbe für Textfelder
<code>SystemColor.textText</code>	Textfarbe für Textfelder
<code>SystemColor.control</code>	Hintergrundfarbe für Dialogelemente
<code>SystemColor.controlText</code>	Textfarbe für Dialogelemente

8.3.3 Die Klasse `java.awt.Font`

Zur Kapselung von Schriftarten dient die Klasse `java.awt.Font`. Das Erzeugen eines neuen Fontobjektes wird über die drei Parameter `name`, `style` und `size` des Konstruktors der Klasse `Font` gesteuert:

```
public Font(String name, int style, int size)
```

Konstruktor der Klasse `java.awt.Font`. Dabei haben die Parameter die folgende Bedeutung:

- Der Parameter `name` gibt den Namen des gewünschten Fonts an. auf allen Javasystemen sollten die Namen `SansSerif` (Helvetica, Arial), `Serif` (TimesRoman) und `Monospaced` (Courier) unterstützt werden.
- Der Parameter `style` wird verwendet, um auszuwählen, ob ein Font in seiner Standardausprägung, fett oder kursiv angezeigt werden soll. Möglich sind folgende Integerkonstante, die auch durch das bitweise OR kombiniert werden können:
 - `Font.PLAIN`, Standardschrift
 - `Font.BOLD`, Fett
 - `Font.ITALIC`, Kursiv
- Der dritte Parameter `size` gibt die Größe der gewünschten Schriftart in Punkt an.

Die Klasse `Font` besitzt auch Methoden, um Informationen über ein bestehendes Fontobjekt zu gewinnen:

```
public String getFamily()
```

Liefert den systemspezifischen Namen des Fontobjektes.

```
public int getStyle()
```

Liefert den Stil des Fontobjektes.

```
public int getSize()
```

Liefert die Größe des Fontobjektes.

Das folgende Beispiel zeigt, wie man sich eine Liste der am aktuellen System verfügbaren Fonts besorgen kann:

```
1 import java.awt.*;
2
3 public class Fontlist {
4     public static void main(String args[]) {
5         GraphicsEnvironment ge;
6         ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
7         String arfonts[] = ge.getAvailableFontFamilyNames();
8         for(String fontname : arfonts) {
9             System.out.println(fontname);
10        }
11    }
12 }
```

Listing 8.2: Liste der Systemfonts

Mögliche (systemabhängige) Ausgabe:

```
arial
arial Black
...
Wingdings 2
Wingdings 3
```

8.3.4 Die Klasse `java.awt.Point`

Ein Objekt dieser Klasse kapselt x- und y-Koordinate eines Punktes, wobei die beiden Koordinaten öffentliche Instanzvariable sind:

```
public int x, y;
```

Konstruktoren:

```
public Point()
public Point(Point p)
public Point(int x, int y)
```

Der parameterlose Konstruktor erzeugt ein Punktobjekt mit den Koordinaten (0/0), die beiden anderen aus den jeweils angegebenen Argumenten.

8.3.5 Die Klasse `java.awt.Dimension`

Ein Objekt dieser Klasse kapselt Länge und Breite eines Rechtecks, wobei die beiden Längenangaben öffentliche Instanzvariable sind:

```
public int width, height;
```

Negative Werte für `width` und `height` sind zulässig. Positive Breitenangaben werden nach rechts, negative nach links aufgetragen. Positive Höhenangaben werden nach unten, negative nach oben gerechnet.

Konstruktoren:

```
public Dimension()
public Dimension(Dimension d)
public Dimension(int width, int height)
```

Der parameterlose Konstruktor erzeugt ein Dimensionsobjekt mit der Ausdehnung 0/0, die beiden anderen aus den jeweils angegebenen Argumenten.

8.3.6 Die Klasse `java.awt.Rectangle`

Ein Objekt dieser Klasse kapselt die Koordinaten des linken oberen Eckpunktes sowie Länge und Breite eines Rechtecks, wobei alle Datenfelder öffentliche Instanzvariable sind:

```
public int width, height;
public int x, y;
```

Die wichtigsten Konstruktoren sind:

```
public Rectangle()
public Rectangle(Point p, Dimension d)
public Rectangle(int x, int y, int width, int height)
```

Der parameterlose Konstruktor setzt alle 4 Instanzvariablen auf 0, die beiden anderen erzeugen ein Rechtecksobjekt aus den jeweils angegebenen Argumenten.

8.3.7 Die Klasse `java.awt.Polygon`

Ein Objekt dieser Klasse kapselt einen geschlossenen Streckenzug, der durch eine geordnete Folge von Punkten definiert wird.

Konstruktoren:

```
public Polygon()
Erzeugt ein leeres Polygon.
```

```
public Polygon(int[] xpoints, int[] ypoints, int npoints)
Erzeugt ein Polygon aus npoints vielen Punkten, wobei die Koordinaten der Punkte aus den synchronisierten Feldern xpoints (x-Koordinaten) und ypoints (y-Koordinaten) stammen.
```

Weitere Methoden:

```
public void addPoint(int x, int y)
Fügt diesem Polygon den Punkt mit den angegebenen Koordinaten hinzu.
```

```
public Rectangle getBounds()
Liefert das kleinste Rechteck, in das dieses Polygon vollständig hineinpasst.
```

```
public boolean contains(Point p)
public boolean contains(int x, int y)
Diese Methoden liefern true, wenn der angegebene Punkt innerhalb des geschlossenen Streckenzuges liegt.
```

8.3.8 Die Klasse `java.awt.Component`

Diese Klasse ist eine abstrakte Klasse und stellt die Basisklasse aller graphischen AWT- und Swing-Komponenten (mit Ausnahme von Menükomponenten) dar. Die Klasse stellt eine Basisfunktionalität zur Verfügung, die von allen abgeleiteten Klassen verwendet werden kann. Im Folgenden werden die wichtigsten Methoden dieser Klasse vorgestellt:

Größe und Position

```
public Dimension getSize()
Liefert die Größe dieser Komponente.
```

```
public void setSize(int width, int height)
public void setSize(Dimension d)
Diese Methoden dienen zum Setzen der Größe dieser Komponente.
```

```
public Point getLocation()  
public Point getLocationOnScreen()
```

Diese beiden Methoden liefern die Position des linken oberen Eckpunktes dieser Komponente. Die Methode `getLocation()` liefert dabei die Position relativ zur übergeordneten Komponente (zur `Parent-Component`), während `getLocationOnScreen()` die Desktopkoordinaten liefert.

```
public void setLocation(int x, int y)  
public void setLocation(Point p)
```

Diese Methoden dienen zum Setzen der Position (des linken oberen Eckpunktes) dieser Komponente relativ zur `Parent-Component`.

```
public Rectangle getBounds()
```

Liefert die Größe und Position dieser Komponente.

```
public void setBounds(int x, int y, int width, int height)
```

Diese Methode dient zum Setzen der Größe und Position dieser Komponente.

```
public Container getParent()
```

Liefert die `Parent-Component` dieser Komponente.

Jede Komponente hat eine sog. *Preferred Size*, also eine bevorzugte Größe. Diese Größe wird von den `LayoutManagern` benötigt, um den von der Komponente beanspruchten Platz zu bestimmen.

```
public Dimension getPreferredSize()  
public void setPreferredSize(Dimension d)
```

Getter und Setter für die bevorzugte Größe dieser Komponente. Die `set`-Methode sollte in eigenen Komponenten überlagert werden.

Farben und Fonts

```
public void setForeground(Color c)  
public void setBackground(Color c)
```

Diese Methoden setzen Vordergrund- und Hintergrundfarbe dieser Komponente. Primitive Grafikoperationen wie direkte Textausgabe oder Zeichnen auf der Komponente verwenden die Vordergrundfarbe, während die Hintergrundfarbe zum Füllen der Komponente dient.

```
public Color getBackground()  
public Color getForeground()
```

Liefern Vordergrund- bzw. Hintergrundfarbe dieser Komponente.

```
public void setFont(Font f)  
public Font getFont()
```

Diese Methoden setzen bzw. liefern jenen Font, der bei Textausgabe auf die Komponente verwendet wird.

Allgemeine Methoden

```
public void setEnabled(boolean b)
```

Aktiviert oder deaktiviert diese Komponente. Eine aktivierte Komponente kann vom Benutzer bedient werden, d.h. sie reagiert auf Usereingaben und sendet gegebenenfalls Ereignisse. Eine deaktivierte Komponente wird grau dargestellt und verarbeitet keinen Userinput. Alle Komponenten werden aktiviert initialisiert.

```
public void setVisible(boolean b)
```

Dient zum Sichtbar- bzw. Unsichtbarmachen dieser Komponente. Alle Komponenten mit Ausnahme von `Toplevelkomponenten` (`Window`, `JWindow`, `Frame`, `JFrame`, `Dialog` bzw. `JDialog`) werden sichtbar initialisiert.

Die Methoden für primitive Grafikoperationen

Jedesmal, wenn das System eine AWT-Komponente ganz oder teilweise neu darstellen muss, wird implizit die Methode `paint(Graphics g)` aufgerufen. Diese Methode erhält ein Objekt vom Typ `java.awt.Graphics`. Dieses Objekt stellt den Grafikkontext (das Zeichenwerkzeug) dar und erlaubt das Ausführen von graphischen Primitivoperationen (zeichnen einfacher geometrischer Figuren und zeichnen von Text) auf dieser AWT-Komponente:

```
public void paint(Graphics g)
```

Zeichnet diese Komponente neu. Subklassen von `java.awt.Component` können diese Methode überschreiben, wobei es nicht notwendig ist `super.paint(g)` aufzurufen.

```
public void repaint()
```

Diese Methode ruft bei AWT-Komponenten in dieser Reihenfolge `update()` und `paint()` auf.

Möchte man in anderen Methoden als `paint()` auf der Komponente zeichnen, so kann man sich mit Hilfe der Methode `getGraphics()` das Zeichenwerkzeug für diese Komponente holen:

```
public Graphics getGraphics()
```

Liefert den Grafikkontext für diese Komponente oder `null`, wenn kein Grafikkontext verfügbar ist. Seit Java 1.5 liefert diese Methode in Wirklichkeit ein `Graphics2D`-Objekt, die gelieferte Referenz kann also auf `Graphics2D` gecastet werden, wodurch die erweiterten Grafikfähigkeiten der Klasse `java.awt.Graphics2D` verwendet werden können.

8.3.9 Die Klasse `javax.swing.JComponent`

Entwickelt man eigene Swing-Komponenten, die direkt oder indirekt aus `JComponent` abgeleitet wurden, so unterscheidet sich das Neuzeichnen von einer AWT-Komponente. Die Methode `paint()` hat in leichtgewichtigen Swing-Komponenten `JComponent` eine recht aufwändige Implementierung und wird normalerweise nicht mehr überlagert. Im Wesentlichen ruft sie nacheinander die folgenden drei Methoden auf:

1. `paintComponent(Graphics g)` - zeichnet die Komponente selbst
2. `paintBorder(Graphics g)` - zeichnet den Rahmen der Komponente
3. `paintChildren(Graphics g)` - zeichnet die Kindkomponenten

Im Allgemeinen reicht es, wenn man für das Zeichnen einer eigenen Komponente die Methode `paintComponent()` überschreibt:

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

8.4 Layoutmanager

8.4.1 Einführung

Java verwendet Layoutmanager, um GUI-Komponenten in Containern zu platzieren. Diese Layoutmanager definieren die Anordnung der Komponenten im Container und legen fest, wie sich die Größe der einzelnen Komponenten ändert, wenn sich die Größe des Containers ändert. Die wichtigsten Layoutmanager sind:

- `FlowLayout`
- `GridLayout`
- `BorderLayout`

- CardLayout
- BoxLayout
- GridBagLayout

8.4.2 Container

Die Basisklasse aller Container ist die Klasse `java.awt.Container`. Wichtige Subklassen sind `java.awt.Window`, `javax.swing.JPanel` und `javax.swing.JFrame`. Jedem Container kann ein Layoutmanager zugeordnet werden. Mit den folgenden Zugriffsmethoden lässt sich der aktuelle Layoutmanager abfragen bzw. setzen:

```
public LayoutManager getLayout()  
Liefert den aktuellen Layoutmanager dieses Containers.
```

```
public void setLayout(LayoutManager mgr)  
Setzt den aktuellen Layoutmanager dieses Containers.
```

Mit Hilfe der mehrfach überladenen Methode `V` lässt sich eine Komponente zu einem Layoutmanager hinzufügen. Der Layoutmanager speichert die Komponenten mit Hilfe einer Liste. Die Ordnung innerhalb der Liste bestimmt die Reihenfolge der Komponenten im Container. Ist der Container bereits sichtbar, so muss die Methode `validate()` aufgerufen werden, um die neue Komponente sichtbar zu machen.

```
public Component add(Component comp)  
Fügt die Komponente comp am Ende des Containers an.
```

```
public Component add(Component comp, int index)  
Fügt die Komponente comp an der Position index im aktuellen Container ein.
```

```
public void add(Component comp, Object constraints)  
Fügt die Komponente comp am Ende des Containers an. Bei der Darstellung des Containers wird das constraints-Objekt herangezogen, um die Position der Komponente zu bestimmen. Dieses Argument hat nur bei bestimmten Layoutmanagern Bedeutung.
```

```
public void add(Component comp, Object constraints, int index)  
Fügt die Komponente comp an der Position index im Container ein. Bei der Darstellung des Containers wird das constraints-Objekt herangezogen, um die Position der Komponente zu bestimmen. Dieses Argument hat nur bei bestimmten Layoutmanagern Bedeutung.
```

Komponenten können aus einem Container auch wieder entfernt werden. Dazu dienen die folgenden Methoden:

```
public void remove(int index)  
Entfernt die Komponente an der Position index.
```

```
public void remove(Component comp)  
Entfernt die Komponente comp aus diesem Container.
```

```
public void removeAll()  
Entfernt alle Komponenten aus diesem Container.
```

Mit Hilfe der folgenden Methoden kann veranlasst werden, dass ein Container seine Komponenten neu zeichnet:

```
public void validate()  
Erklärt alle Komponenten dieses Containers für ungültig und ruft die Methode doLayout() auf, die bewirkt, dass der Container seine Komponenten neu darstellt.
```

Eine Komponente, die zu einem Container hinzugefügt wird, kann selbst wieder ein Container sein, d.h. Container können verschachtelt werden. Diese Technik stellt die Basis für das Design komplexer GUI's dar. Man unterscheidet also zwei Arten von Containern:

1. Container, die in einem Parentcontainer liegen müssen. Typische Beispiele sind `javax.swing.JPanel` und `javax.swing.JApplet`.
2. Container, die unabhängig existieren und nicht in andere Container gelegt werden können, sog. Top-Level-Windows. Beispiele sind `javax.swing.JFrame` und `javax.swing.JDialog`.

Die Klasse `java.awt.Window` (und damit alle abgeleiteten Klassen) stellt eine Methode `pack()` zur Verfügung, die bewirkt, dass alle in diesem Container liegenden Komponenten wenn möglich in ihrer sogenannten "Preferred Size" dargestellt werden. Das Fenster erscheint also in optimaler Größe.

```
public void java.awt.Window.pack()
```

Die Größe aller enthaltenen Komponenten wird wenn möglich auf die preferred Size gesetzt und die Größe des Fensters wird neu berechnet.

8.4.3 Spezielle Layoutmanager

Im Folgenden werden die wichtigsten speziellen Layoutmanager vorgestellt. Zur Demonstration der Eigenschaften werden ausschließlich Komponenten vom Typ `javax.swing.JButton` in die Container gelegt. Alle Layoutmanager-Klassen implementieren eines der Interfaces `java.awt.LayoutManager` bzw. `java.awt.LayoutManager2`. Diese Interfaces schreiben eine gemeinsame Basisfunktionalität vor.

Das folgende Listing gibt ein Grundgerüst für ein Programm zur Demonstration der wichtigsten Layoutmanager an, in dem dann jeweils nur die Methode `customizeLayout()` angepasst werden muss:

```

1 import java.awt.*;
2 import javax.swing.*;
3
4 public class Layouts_1 extends JFrame {
5
6     private JButton b[] = new JButton[5];
7
8     public Layouts_1() {
9         super("Layouts");           // Konstruktoraufruf von JFrame
10        for(int i = 0; i < b.length; i++) {
11            b[i] = new JButton("Button " + (i + 1));
12        }
13        this.customizeLayout();
14        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15        this.pack();
16    }
17
18    private void customizeLayout() {
19        // spezielle Implementierung
20    }
21
22    public static void main(String[] args) {
23        new Layouts_1().setVisible(true);
24    }
25 }

```

Listing 8.3: Grundgerüst für Layouts

FlowLayout

Das `FlowLayout` ordnet die Komponenten zeilenweise von links nach rechts an. Wenn keine weiteren Elemente in die Zeile passen, wird mit der nächsten Zeile fortgefahren.

Auf einem `FlowLayout` lassen sich die zeilenweise Ausrichtung (linksbündig, rechtsbündig bzw. zentriert) der Komponenten sowie die horizontalen und vertikalen Abstände (`hgap` und `vgap`) zwischen den

einzelnen Komponenten einstellen. Diese Einstellungen werden in der Regel bereits mit Hilfe der Konstruktoren getroffen:

```
public FlowLayout()
```

Erzeugt ein `FlowLayout` mit `hgap = vgap = 5` sowie zentrierter Ausrichtung.

```
public FlowLayout(int align)
```

Erzeugt ein `FlowLayout` mit `hgap = vgap = 5` und der Ausrichtung `align`.

Für `align` stehen dabei die Konstanten `FlowLayout.CENTER`, `FlowLayout.LEFT` und `FlowLayout.RIGHT` zur Verfügung.

```
public FlowLayout(int align, int hgap, int vgap)
```

Erzeugt ein `FlowLayout` mit horizontalen Abständen `hgap`, vertikalen Abständen `vgap` und der Ausrichtung `align`.

Eigenschaften des `FlowLayouts`

- Das Flow-Layout berücksichtigt die preferred Size aller hinzugefügten Komponenten, d.h. es stellt alle Komponenten immer in ihrer optimalen Größe dar.
- Der Flow-Layout-Manager ist der Standard-Layoutmanager für ein `JPanel`.

Das folgende Beispiel zeigt für die angegebene Methode `customizeLayout()` den Container unmittelbar nach seiner Erzeugung sowie nach einer Veränderung der Größe durch den User.

```
1 private void customizeLayout() {  
2     this.setLayout(new FlowLayout());  
3     for(int i = 0; i < b.length; i++) {  
4         this.add(b[i]);  
5     }  
6 }
```

Listing 8.4: Flow Layout

- Nach der Erzeugung:

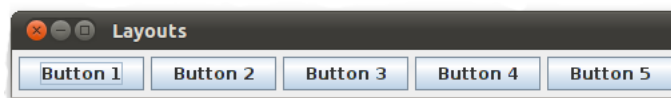


Abbildung 8.3: Flow Layout 1

- Nach Größenänderung

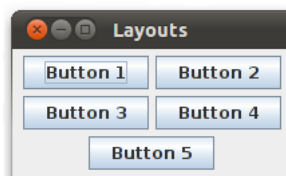


Abbildung 8.4: Flow Layout 2

GridLayout

Das `GridLayout` ordnet die Dialogelemente in einem rechteckigen Gitter an, dessen Zeilen- und Spaltenzahl beim Erstellen des Layoutmanagers angegeben wird. Die einzelnen Zellen des `GridLayouts` haben immer gleiche Größe.

Die Klasse `GridLayout` stellt u.a. die folgenden Konstruktoren zur Verfügung:

```
public GridLayout()
```

Entspricht dem Aufruf `GridLayout(1, 0)`, also mit einer Zeile und dynamischer Spaltenanzahl.

```
public GridLayout(int rows, int columns)
```

Erzeugt ein `GridLayout` mit `rows` Zeilen und `columns` Spalten. Einer der beide Werte darf Null sein. Die Geometrie des `GridLayouts` wird dann durch die von Null verschiedene Größe und der Anzahl der hinzugefügten Komponenten bestimmt. `hgap` und `vgap` werden standardmäßig auf 0 Pixel eingestellt.

```
public GridLayout(int rows, int columns, int hgap, int vgap)
```

Erzeugt ein `GridLayout` mit den übergebenen Einstellungen.

Das `GridLayout` ignoriert die preferred Size der Komponenten, er stellt jede Komponenten so dar, dass sie genau in eine Zelle passt.

Das folgende Beispiel zeigt für die angegebene Methode `customizeLayout()` den Container unmittelbar nach seiner Erzeugung sowie nach einer Veränderung der Größe durch den User.

```
1 private void customizeLayout() {
2     this.setLayout(new GridLayout(2, 3, 5, 2));
3     for(int i = 0; i < b.length; i++) {
4         this.add(b[i]);
5     }
6 }
```

Listing 8.5: Grid Layout

- Nach der Erzeugung:

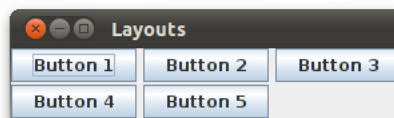


Abbildung 8.5: Grid Layout 1

- Nach Größenänderung



Abbildung 8.6: Grid Layout 2

BorderLayout

Das `BorderLayout` teilt den Container in die vier Randbereiche (North, South, East, West) und den Mittelbereich (Center). Es gibt die folgenden Konstruktoren:

```
public BorderLayout()
```

```
public BorderLayout(int hgap, int vgap)
```

Der parameterlose Konstruktor erzeugt ein `BorderLayout` ohne Zwischenräume der Komponenten, der

zweite Konstruktor definiert den horizontalen Abstand `hgap` und den vertikalen Abstand `vgap`.

Zum Einfügen von Elementen benötigt man die Methode

```
public void add(Component comp, Object constraints)
```

der Klasse `java.awt.Container`. Als zweites Argument wird eine der folgenden Konstanten vom Typ `String`, definiert in der Klasse `java.awt.BorderLayout` übergeben:

- `BorderLayout.NORTH`
- `BorderLayout.WEST`
- `BorderLayout.EAST`
- `BorderLayout.SOUTH`
- `BorderLayout.CENTER`

Lässt man das zweite Argument weg, so wird die Komponente standardmäßig in den Zentrumsbereich gelegt.

Das folgende Beispiel zeigt für die angegebene Methode `customizeLayout()` den Container unmittelbar nach seiner Erzeugung sowie nach einer Veränderung der Größe durch den User.

```
1 private void customizeLayout() {  
2     String []p = { BorderLayout.NORTH, BorderLayout.WEST,  
3                   BorderLayout.SOUTH, BorderLayout.EAST,  
4                   BorderLayout.CENTER };  
5     this.setLayout(new BorderLayout());  
6     for(int i = 0; i < b.length; i++) {  
7         this.add(b[i], p[i]);  
8     }  
9 }
```

Listing 8.6: BorderLayout

- Nach der Erzeugung:

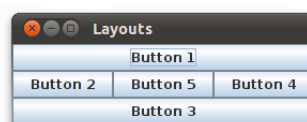


Abbildung 8.7: Border Layout 1

- Nach Größenänderung

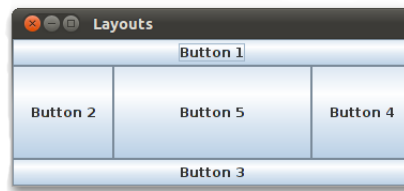


Abbildung 8.8: Border Layout 2

Bezüglich der Skalierung der Komponenten verfolgt das BorderLayout einen Mittelweg zwischen Flow-Layout und GridLayout. Während das FlowLayout die Komponenten immer in ihrer natürlichen Größe (preferred Size) belässt und das GridLayout sie immer skaliert, geschieht dies beim BorderLayout nach folgenden Regeln:

- Nord- und Südelement behalten immer ihre preferred Height, werden aber auf die volle Containerbreite skaliert.
- Ost- und Westelement behalten immer ihre preferred Width, ihre Höhe wird aber so skaliert, dass sie genau zwischen Nord- und Südelement passen.
- Das Zentrumsэлеment wird in der Höhe und Breite so skaliert, dass es den gesamten verbleibenden Platz einnimmt.

Der BorderLayout-Manager ist der Standardlayoutmanager aller Container vom Typ

`java.awt.Window`, also insbesondere von `java.awt.JFrame`.

8.4.4 Verschachteln von Layoutmanagern

Layouts können verschachtelt werden. So kann z.B. im Zentrumsteil eines BorderLayouts ein GridLayout untergebracht werden usw. Diese Technik gestattet die Realisierung komplexer, portabler Dialoge. Um Layoutmanager zu schachteln, wird an der Stelle, die ein Sublayout erhalten soll, ein Objekt der Klasse `javax.swing.JPanel` eingefügt, das einen eigenen Layoutmanager zugeordnet bekommt. Dieses `JPanel` kann mit Dialogelementen bestückt werden, die entsprechend dem zugeordneten Sublayout formatiert werden. Das nachfolgende Beispiel demonstriert diese Technik und erzeugt das dargestellte Layout:

```

1 public void customizeLayout() {
2     // Border-Layout ist Standard
3     JPanel pW = new JPanel(new GridLayout(1,2));
4     JPanel pWW = new JPanel();
5     pWW.setLayout(new GridLayout(3,1));
6     for(int i = 0; i < 3; i++) {
7         pWW.add(b[i]);
8     }
9     pW.add(pWW);
10    pW.add(b[3]);
11    this.add(pW, BorderLayout.WEST);
12    this.add(b[4], BorderLayout.CENTER);
13 }

```

Listing 8.7: Verschachtelte Layouts

Bschreibung:

- Die Zeilen 3 und 4 zeigen 2 verschiedene Konstruktoraufrufe der Klasse `JPanel`. In Zeile 4 wird der Standardkonstruktor verwendet (das Panel arbeitet zunächst mit dem Flow-Layout-Manager) und dann wird mit Hilfe der Methode `setLayout()` ein neuer Layoutmanager zugewiesen. In Zeile 03 wird jener Konstruktor verwendet, der als Argument bereits ein Objekt vom Typ `java.awt.LayoutManager` erwartet. Damit wird bereits beim Erzeugen des Panels der gewünschte Layoutmanager zugewiesen.

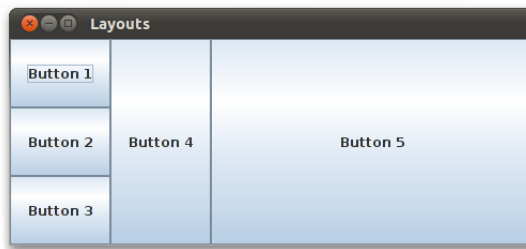


Abbildung 8.9: Verschachtelte Layouts

- Das Layout ist wie folgt aufgebaut: In den Westbereich des BorderLayouts wird ein Panel `pW` gelegt, das einen GridLayout-Manager mit einer Zeile und zwei Spalten verwendet. In die linke Zelle dieses Gridlayouts wird wieder ein Panel `pWW` gelegt, das ein GridLayout mit drei Zeilen und einer Spalte verwendet.

8.5 Eventhandling

8.5.1 Einführung

Beim Ablauf grafisch gesteuerter Programme entstehen durch Benutzeraktionen (Maus- und Tastatureingaben, Bedienung von Steuerelementen) sog. Ereignisse (Events), welche die Applikation von diesen Benutzeraktionen informieren. Aus diesem Grund werden Programme mit grafischem Userinterface auch *ereignisgesteuerte Programme* genannt. An diesem Nachrichtenverkehr sind drei verschiedene Arten von Objekten beteiligt:

- **Ereignisquellen (event sources)**
Diese erzeugen Ereignisse (wie. z.B. ein Mausereignis) und senden diese Ereignisse zu den registrierten Ereignisempfängern. Eine Ereignisquelle kann z.B. ein Button oder ein Fenster sein.
- **Events**
Diese kapseln Informationen über die Art des Events, über die Ereignisquelle sowie weitere nützliche Informationen über das Ereignis.
- **Ereignisempfänger**
Diese werden beim Auftreten eines entsprechenden Ereignisses informiert und können in angemessener Weise reagieren.

Dieses Kommunikationsmodell nennt sich Delegation Event Model oder Delegation Based Event Handling und wurde mit der Version 1.1 des JDK eingeführt. Das folgende einfache Beispiel zeigt die prinzipiell notwendigen Schritte bei der Verwendung dieses Eventmodells. Ein Fenster erzeugt in bestimmten Situationen Mausereignisse, auf die reagiert werden soll.

1.Schritt - Programmierung des Ereignisempfängers

Der Ereignisempfänger oder Eventlistener ist zur Verfügung zu stellen. Es handelt sich dabei um eine Klasse, die das entsprechende Listener-Interface implementiert. Benötigt man einen Empfänger für Mausereignisse, so hat man das Interface `java.awt.event.MouseListener` zu implementieren. Dieses Interface beinhaltet 5 abstrakte Methoden, die konkret überlagert werden müssen.

```

1 import java.awt.event.MouseListener;
2 import java.awt.event.MouseEvent;
3
4 public class MyMouseListener implements MouseListener {
5
6     public void mouseEntered(MouseEvent e) {
7         System.out.println("Mouse entered");
8     }
9
10    public void mouseExited(MouseEvent e) {

```



```

11     System.out.println("Mouse exited");
12 }
13
14 public void mouseClicked(MouseEvent e) {
15     System.out.println("Mouse clicked");
16 }
17
18 public void mousePressed(MouseEvent e) {
19     System.out.println("Mouse pressed");
20 }
21
22 public void mouseReleased(MouseEvent e) {
23     System.out.println("Mouse released");
24 }
25 }

```

Listing 8.8: Eventhandler

2.Schritt - Registrieren des Ereignisempfängers bei der Quelle

Damit die Ereignisquelle (in diesem Fall das Hauptfenster) die Ereignisse an den Empfänger sendet, muss sich dieser bei der Quelle registrieren. Dies geschieht durch die Registrierungsmethode `addMouseListener()`.

Diese Methode erwartet ein Objekt vom Typ `MouseListener`, wodurch gewährleistet ist, dass jeder Empfänger für Mausereignisse das Interface `MouseListener` implementiert und daher das in diesem Interface definierte Verhalten aufweist. Tritt nun im Sender ein Mausereignis auf, so wird ein entsprechendes Objekt vom Typ `MouseEvent` generiert und an den Empfänger gesendet, d.h. es wird die zugehörige Methode im Empfänger aufgerufen.

```

1 import javax.swing.*;
2 import java.awt.event.*;
3
4 public class Events_1 extends JFrame {
5     public static void main(String[] args) {
6         new Events_1().setVisible(true);
7     }
8
9     public Events_1() {
10         super("Eventhandling 1");
11         this.setBounds(100, 100, 300, 200);
12         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         this.addMouseListener(new MyMouseListener());
14     }
15 }

```

Listing 8.9: Eventhandling Schritt 2

8.5.2 Eventklassen

Ereignisse wie z.B. Mausereignisse sind Javaobjekte. Die Superklasse aller Ereignisklassen ist `java.util.EventObject`. Diese Klasse enthält eine Methode zur Feststellung des Senders:

```
public Object getSource()
```

Diese Methode aus `java.util.EventObject` liefert den Sender des Events.

8.5.3 Ereignisempfänger

Damit ein Objekt Nachrichten empfangen kann, muss es eine Reihe von Methoden implementieren, die von der Nachrichtenquelle, bei der es sich registriert hat, aufgerufen werden können. Um sicherzustellen, dass diese Methoden vorhanden sind, müssen die Ereignisempfänger bestimmte Interfaces implementieren, die von `java.util.EventListener` abgeleitet sind. Diese `EventListener`-Interfaces befinden sich im Paket `java.awt.event`. Jedes Interface definiert eine eigene Methode für jede Ereignisart dieser Ereignisklasse. (Siehe unten).

Eventtypen

Die Eventklassen lassen sich in 2 Gruppen einteilen:

- Semantic- oder Highlevel events
- Lowlevel events

Highlevel events

Diese Events werden vom Benutzer durch Bedienung von Steuerelementen (Buttonclicks, ändern des Status einer Checkbox, bedienen eines Schiebereglers, Texteingabe in ein Textfeld etc.) ausgelöst. Sie werden im Abschnitt über Steuerelemente ausführlich besprochen. Einige wichtige Semantic events sind:

- `ActionEvent`

Listener-Interface:

```
public interface ActionListener extends EventListener {  
    void actionPerformed(ActionEvent e);  
}
```

- `AdjustmentEvent`

Listener-Interface:

```
public interface AdjustmentListener extends EventListener {  
    void adjustmentValueChanged(AdjustmentEvent e);  
}
```

- `ItemEvent`

Listener-Interface:

```
public interface ItemListener extends EventListener {  
    void itemValueChanged(ItemEvent e);  
}
```

- `TextEvent`

Listener-Interface:

```
public interface TextListener extends EventListener {  
    void textValueChanged(TextEvent e);  
}
```

Lowlevel events

Diese Events werden direkt oder indirekt vom Betriebssystem ausgelöst. Beispiele sind Mausereignisse, Tastaturereignisse oder Fensterereignisse. Die wichtigsten Low-Level-Events sind:

- `ComponentEvent`

Listener-Interface:

```
public interface ComponentListener extends EventListener {
    void componentHidden(ComponentEvent e);
    void componentMoved(ComponentEvent e);
    void componentResized(ComponentEvent e);
    void componentShown(ComponentEvent e);
}
```

- ContainerEvent

Listener-Interface:

```
public interface ContainerListener extends EventListener {
    void componentAdded(ContainerEvent e);
    void componentRemoved(ContainerEvent e);
}
```

- FocusEvent

Listener-Interface:

```
public interface FocusListener extends EventListener {
    void focusGained(FocusEvent e);
    void focusLost(FocusEvent e);
}
```

- KeyEvent

Listener-Interface:

```
public interface KeyListener extends EventListener {
    void keyPressed(KeyEvent e);
    void keyReleased(KeyEvent e);
    void keyTyped(KeyEvent e);
}
```

- MouseEvent

Hier gibt es 2 Listener-Interfaces (eines für gewöhnliche Mausevents und eines für Mausbewegungen):

```
public interface MouseListener extends EventListener {
    void mouseEntered(MouseEvent e);
    void mouseExited(MouseEvent e);
    void mousePressed(MouseEvent e);
    void mouseReleased(MouseEvent e);
    void mouseClicked(MouseEvent e);
}
```

```
public interface MouseMotionListener extends EventListener {
    void mouseDragged(MouseEvent e);
    void mouseMoved(MouseEvent e);
}
```

- WindowEvent

Listener-Interface:

```

public interface WindowListener extends EventListener {
    void windowActivated(WindowEvent e);
    void windowClosed(WindowEvent e);
    void windowClosing(WindowEvent e);
    void windowDeactivated(WindowEvent e);
    void windowDeiconified(WindowEvent e);
    void windowIconified(WindowEvent e);
    void windowOpened(WindowEvent e);
}

```

8.5.4 Konkrete Implementierungsmöglichkeiten

In diesem Abschnitt werden am Beispiel von ActionEvents verschiedene Möglichkeiten demonstriert, Eventhandling zu implementieren. Dabei wird eine Applikation mit 2 Button vorgestellt, wobei bei Betätigung des ersten Button das Fenster zentriert und bei klicken des zweiten Button das Fenster auf eine zufällige Bildschirmposition verschoben wird. Außerdem werden alle auftretenden Fensterereignisse auf der Konsole mitprotokolliert. Dabei werden folgende API-Elemente verwendet:



Abbildung 8.10: Events implementieren

- Ereignisklassen: `ActionEvent`, `WindowEvent`
- Listener-Interfaces: `ActionListener`, `WindowListener`
- Registrierungsmethoden: `addActionListener()`, `addWindowListener()`
- Ereignisquellen: Die Button bzw. das Fenster

Möglichkeit 1 - Interface implementieren

Die Fensterklasse selbst implementiert die erforderlichen `EventListener`-Interfaces, stellt alle erforderlichen Ereignismethoden zur Verfügung und registriert sich selbst bei der Ereignisquelle.

```

1 package awt_swing;
2
3 import java.awt.Dimension;
4 import java.awt.FlowLayout;
5 import java.awt.Toolkit;
6 import java.awt.event.ActionEvent;
7 import java.awt.event.ActionListener;
8 import java.awt.event.WindowEvent;
9 import java.awt.event.WindowListener;
10 import java.util.Random;
11 import javax.swing.JButton;
12 import javax.swing.JFrame;
13
14 public class Events_01 extends JFrame implements ActionListener,
15                                     WindowListener {
16
17     private static Random rd = new Random();
18
19     private JButton buttonCenter;
20     private JButton buttonRandom;
21
22     public Events_01() {
23         super("Events - Moeglichkeit 1");
24         // Layout erstellen
25         buttonCenter = new JButton("Center position");
26         buttonRandom = new JButton("Random position");
27         this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

```

```
28     this.setLayout(new FlowLayout());
29     this.add(buttonCenter);
30     this.add(buttonRandom);
31     // Registrieren der Ereignisempfaenger
32     this.addWindowListener(this);
33     buttonCenter.addActionListener(this);
34     buttonRandom.addActionListener(this);
35     // Fenster packen
36     pack();
37 }
38
39 @Override
40 public void actionPerformed(ActionEvent e) {
41     // Eventquelle bestimmen
42     Object source = e.getSource();
43     if(source == buttonCenter) {
44         // Fenster zentrieren
45         this.setLocationRelativeTo(null);
46     }
47     else if(source == buttonRandom) {
48         // Dimension des Desktop bestimmen
49         Dimension scr = Toolkit.getDefaultToolkit().getScreenSize();
50         // Zufaelliche Position bestimmen
51         int x = rd.nextInt(scr.width - this.getWidth());
52         int y = rd.nextInt(scr.height - this.getHeight());
53         // Zufaelliche Position setzen
54         this.setLocation(x, y);
55     }
56 }
57
58 @Override
59 public void windowOpened(WindowEvent e) {
60     System.out.println("Window opened");
61 }
62
63 @Override
64 public void windowClosing(WindowEvent e) {
65     System.out.println("Window closing");
66     // Fenster beim Betriebssystem abmelden und zerstören
67     this.dispose();
68     // Applikation beenden
69     System.exit(0);
70 }
71
72 @Override
73 public void windowClosed(WindowEvent e) {
74     System.out.println("Window closed");
75 }
76
77 @Override
78 public void windowIconified(WindowEvent e) {
79     System.out.println("Window iconified");
80 }
81
82 @Override
83 public void windowDeiconified(WindowEvent e) {
84     System.out.println("Window deiconified");
85 }
86
87 @Override
88 public void windowActivated(WindowEvent e) {
89     System.out.println("Window activated");
90 }
91
92 @Override
93 public void windowDeactivated(WindowEvent e) {
94     System.out.println("Window deactivated");
95 }
96
97 public static void main(String []args) {
98     new Events_01().setVisible(true);
99 }
100 }
```

Listing 8.10: Eventhandler - Konkrete Implementierung 01 (Interfaces implementieren)

Bei dieser Implementierung gibt es nur eine einzige Klasse `Layouts_01.java`. Sie ist einerseits eine Subklasse von `javax.swing.JFrame`, um ein Fenster darstellen zu können. Andererseits implementiert sie die Interfaces `ActionListener` und `WindowListener`. Jedes Objekt dieser Klasse kann daher auf Actionevents und Windowevents reagieren (ist also ein Empfänger für solche Events). Der eigentliche Code zur Reaktion auf diese Events steckt in den überschriebenen Ereignismethoden. Diese Implementierung ist sehr naheliegend, denn sie ist einfach und erfordert keine weiteren Klassen. Nachteilig ist dabei allerdings, dass alle Methoden aller implementierten Interfaces überschrieben werden müssen.

Zusätzlich zu den Kommentaren sind folgende Punkte zu beachten:

- In den Zeilen 32-34 werden die Ereignisempfänger bei den Sendern registriert. Als Empfänger wird immer das `this`-Objekt angegeben, das ja selbst die entsprechenden Listener-Interfaces implementiert. Das Fenster registriert sich bei sich selbst, die Sender für die Actionevents sind die beiden Button.
- In Zeile 27 wird das Fenster so konfiguriert, dass es nichts tut, wenn der User das Fenster schließen möchte. Es wird aber in diesem Fall ein `WindowClosing`-Event, auf das in der Ereignismethode `windowClosing()` entsprechend reagiert wird.
- In der Ereignismethode `actionPerformed()` muss zunächst der Sender des Action-Events festgestellt werden, da beide Button ihre Action-Events an denselben Empfänger (nämlich das Fenster) senden.

Möglichkeit 2 - Innere Klassen

Programmierung eigener Empfängerklasse. Diese Art der Implementierung entspricht der Möglichkeit 1, allerdings werden eigene Empfängerklassen programmiert. Diese werden in der Regel als innere Klassen implementiert, damit man vom Empfänger aus bequem auf die privaten Datenelemente der umgebenden äußeren Klasse zugreifen kann.

```

1 package awt_swing;
2
3 import java.awt.Dimension;
4 import java.awt.FlowLayout;
5 import java.awt.Toolkit;
6 import java.awt.event.ActionEvent;
7 import java.awt.event.ActionListener;
8 import java.awt.event.WindowEvent;
9 import java.awt.event.WindowListener;
10 import java.util.Random;
11 import javax.swing.JButton;
12 import javax.swing.JFrame;
13
14 public class Events_02 extends JFrame {
15
16     private static Random rd = new Random();
17     private JButton buttonCenter;
18     private JButton buttonRandom;
19
20     public Events_02() {
21         super("Events - Möglichkeit 1");
22         // Layout erstellen
23         buttonCenter = new JButton("Center position");
24         buttonRandom = new JButton("Random position");
25         this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
26         this.setLayout(new FlowLayout());
27         this.add(buttonCenter);
28         this.add(buttonRandom);
29         // Registrieren der Ereignisempfaenger
30         this.addWindowListener(new WindowHandler_02());
31         buttonCenter.addActionListener(new CenterActionHandler_02());
32         buttonRandom.addActionListener(new RandomActionHandler_02());
33         // Fenster packen

```

```
34     pack();
35 }
36
37 // Innere Klasse fuer ActionEvents des Button-Center
38 private class CenterActionHandler_02 implements ActionListener {
39
40     @Override
41     public void actionPerformed(ActionEvent e) {
42         Events_02.this.setLocationRelativeTo(null);
43     }
44 }
45
46 // Innere Klasse fuer ActionEvents des Button-Random
47 private class RandomActionHandler_02 implements ActionListener {
48
49     @Override
50     public void actionPerformed(ActionEvent e) {
51         Dimension scr = Toolkit.getDefaultToolkit().getScreenSize();
52         // Zufaelliche Position bestimmen
53         int x = rd.nextInt(scr.width - Events_02.this.getWidth());
54         int y = rd.nextInt(scr.height - Events_02.this.getHeight());
55         // Zufaelliche Position setzen
56         Events_02.this.setLocation(x, y);
57     }
58 }
59
60 // Innere Klasse fuer WindowEvents
61 private class WindowHandler_02 implements WindowListener {
62
63     @Override
64     public void windowOpened(WindowEvent e) {
65         System.out.println("Window opened");
66     }
67
68     @Override
69     public void windowClosing(WindowEvent e) {
70         System.out.println("Window closing");
71         // Fenster beim Betriebssystem abmelden und zerstueren
72         Events_02.this.dispose();
73         // Applikation beenden
74         System.exit(0);
75     }
76
77     @Override
78     public void windowClosed(WindowEvent e) {
79         System.out.println("Window closed");
80     }
81
82     @Override
83     public void windowIconified(WindowEvent e) {
84         System.out.println("Window iconified");
85     }
86
87     @Override
88     public void windowDeiconified(WindowEvent e) {
89         System.out.println("Window deiconified");
90     }
91
92     @Override
93     public void windowActivated(WindowEvent e) {
94         System.out.println("Window activated");
95     }
96
97     @Override
98     public void windowDeactivated(WindowEvent e) {
99         System.out.println("Window deactivated");
100     }
101 }
102
103 public static void main(String[] args) {
104     new Events_02().setVisible(true);
105 }
106 }
```

Listing 8.11: Eventhandler - Konkrete Implementierung 02 (Innere Klassen)

In obigem Beispiel sind folgende Punkte zu beachten:

- Zu jedem Button wurde eine eigene Empfängerklasse programmiert, welche das Interface `ActionListener` implementiert. Damit ist es nicht mehr notwendig, in den HandlerMethoden `actionPerformed()` den Sender abzufragen, da jeder Sender seinen eigenen Empfänger zugeordnet hat.
- Verwendet man in einer inneren Klasse das Schlüsselwort `this`, so spricht man damit die Instanz der inneren Klasse an. Die zugeordnete Instanz der äußeren Klasse erhält man, indem man dem Schlüsselwort `this` den Namen der äußeren Klasse voranstellt. Z.B: `Events_02.this.setLocation(x, y)`
- Da die Empfängerklassen als eigene (innere) Klassen implementiert wurden, muss beim Registrieren der einzelnen Empfänger bei den einzelnen Sendern (Zeile 30-32) jeweils eine neue Instanz der Empfängerklasse erzeugt und der Registrierungsmethode übergeben werden.

Auch bei dieser Möglichkeit der Implementierung sammeln sich schnell viele nicht benötigte Methoden an. Diese Problematik kann mit Hilfe von Adapterklassen vermieden werden.

Möglichkeit 3 - Adapterklassen

Eine Adapterklasse ist eine Klasse, die ein (oder mehrere) Interface(s) implementiert und die geerbten abstrakten Methoden mit leeren Methodenrumpfen konkretisiert. Adapterklassen können verwendet werden, wenn aus einem Interface lediglich ein Teil der Methoden benötigt wird, der Rest aber unwesentlich ist. In diesem Fall leitet man eine neue Klasse aus der Adapterklasse ab, anstatt das (die) zugehörige(n) Interface(s) zu implementieren.

In dieser neuen Klasse überschreibt man nur die benötigten Methoden. Alle anderen Methoden des (der) Interfaces werden von der Basisklasse (also der Adapterklasse) ohne Funktionalität zur Verfügung gestellt.

Die folgende Tabelle zeigt die wichtigsten Adapterklassen aus `java.awt.event` mit den entsprechenden Interfaces:

ListenerInterface	Adapterklasse
FocusListener	FocusAdapter
KeyListener	KeyAdapter
MouseListener	MouseAdapter
MouseMotionListener	MouseAdapter, MouseMotionAdapter
ComponentListener	ComponentAdapter
ContainerListener	ContainerAdapter
WindowListener	WindowAdapter

Das nächste Beispiel verwendet als innere Klasse eine von `WindowAdapter` abgeleitete innere Klasse. Bei dieser Art der Implementierung ist nur mehr jene Ereignismethode zu überschreiben, die wirklich benötigt wird (hier `windowClosing()`). Außer der inneren Klasse für die Behandlung von Window-Events gibt es zur Möglichkeit 2 keine wesentlichen Änderungen.

```

1 // Innere Klasse fuer WindowEvents
2 private class WindowHandler_03 extends WindowAdapter {
3
4     @Override
5     public void windowClosing(WindowEvent e) {
6         System.out.println("Window closing");
7         // Fenster beim Betriebssystem abmelden und zerstören
8         Events_03.this.dispose();
9         // Applikation beenden
10        System.exit(0);
11    }
12 }

```

Listing 8.12: Eventhandler - Konkrete Implementierung 03 (Adapterklassen)

Möglichkeit 4 - Anonyme innere Klassen

Die letzte Möglichkeit zeigt die Implementierung des Ereignisempfängers mit Hilfe von anonymen inneren Klassen. Hier wird der Empfänger genau an der Stelle implementiert und instanziiert, an der er benötigt wird, nämlich beim Aufruf der Registrierungsmethode. Bei dieser Möglichkeit wird jedem Sender ein eigener eindeutiger Empfänger zugeordnet, wodurch das Eventhandling systematischer und überschaubarer wird. Viele Codegeneratoren (z.B. der GUI-Builder von Netbeans) arbeiten mit dieser Methode.

```
1 package awt_swing;
2
3 import java.awt.Dimension;
4 import java.awt.FlowLayout;
5 import java.awt.Toolkit;
6 import java.awt.event.ActionEvent;
7 import java.awt.event.ActionListener;
8 import java.awt.event.WindowAdapter;
9 import java.awt.event.WindowEvent;
10 import java.awt.event.WindowListener;
11 import java.util.Random;
12 import javax.swing.JButton;
13 import javax.swing.JFrame;
14
15 public class Events_04 extends JFrame {
16
17     private static Random rd = new Random();
18
19     private JButton buttonCenter;
20     private JButton buttonRandom;
21
22     public Events_04() {
23         super("Events - Möglichkeit 4");
24         // Layout erstellen
25         buttonCenter = new JButton("Center position");
26         buttonRandom = new JButton("Random position");
27         this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
28         this.setLayout(new FlowLayout());
29         this.add(buttonCenter);
30         this.add(buttonRandom);
31         // Registrieren der Ereignisempfaenger
32         this.addWindowListener(new WindowAdapter() {
33             @Override
34             public void windowClosing(WindowEvent e) {
35                 System.out.println("Window closing");
36                 // Fenster beim Betriebssystem abmelden und zerstören
37                 Events_04.this.dispose();
38                 // Applikation beenden
39                 System.exit(0);
40             }
41         });
42         buttonCenter.addActionListener(new ActionListener() {
43             @Override
44             public void actionPerformed(ActionEvent e) {
45                 Events_04.this.centerWindow();
46             }
47         });
48         buttonRandom.addActionListener(new ActionListener() {
49             @Override
50             public void actionPerformed(ActionEvent e) {
51                 Events_04.this.randomizeWindow();
52             }
53         });
54         // Fenster packen
55         pack();
56     }
57
58     private void centerWindow() {
59         this.setLocationRelativeTo(null);
60     }
61
62     private void randomizeWindow() {
```

```

64 Dimension scr = Toolkit.getDefaultToolkit().getScreenSize();
65 // Zufaelliche Position bestimmen
66 int x = rd.nextInt(scr.width - Events_04.this.getWidth());
67 int y = rd.nextInt(scr.height - Events_04.this.getHeight());
68 // Zufaelliche Position setzen
69 Events_04.this.setLocation(x, y);
70 }
71
72 public static void main(String[] args) {
73     new Events_04().setVisible(true);
74 }
75 }

```

Listing 8.13: Eventhandler - Konkrete Implementierung 04 (Anonyme innere Klassen)

8.6 Zeichnen mit `java.awt.Graphics`

Die abstrakte Klasse `java.awt.Graphics` stellt eine geräteunabhängige Schnittstelle für die Darstellung von Grafiken zur Verfügung. Konkrete Subklassen bieten Implementierungen für spezielle Plattformen und Geräte. Wie man ein Objekt vom Typ `java.awt.Graphics` erhält wurde im obigen Abschnitt beschrieben. In diesem Abschnitt werden die wichtigsten Methoden und Möglichkeiten bei der Verwendung von `Graphics` vorgestellt. Allen Beispielen liegt das folgende Gerüst zugrunde, wobei die Methode `paintComponent()` beliebig ausgetauscht werden kann:

```

1 import java.awt.*;
2 import javax.swing.*;
3 import java.awt.geom.*;
4
5 public class PaintMain extends JFrame {
6     private JPanel p = null;
7
8     public PaintMain() {
9         super("Demo zu Graphics - Grundgerüst");
10        this.p = new JPanel();
11        this.setLocation(50, 50);
12        this.getContentPane().add(this.p, BorderLayout.CENTER);
13        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14        this.pack();
15    }
16
17    public static void main(String... args) {
18        EventQueue.invokeLater(new Runnable() {
19            public void run() {
20                new PaintMain().setVisible(true);
21            }
22        });
23    }
24 }
25
26 class JPanel extends JPanel {
27     public JPanel() {
28         this.setPreferredSize(new Dimension(600, 400));
29         this.setBackground(Color.WHITE);
30     }
31
32     public void paintComponent(Graphics g) {
33         super.paintComponent(g);
34         // weitere Implementierung
35     }
36 }

```

Listing 8.14: Demo zu `java.awt.Graphics` - Grundgeruest

8.6.1 Verwendung von Farben

Dem Grafikkontext ist zu jeder Zeit eine aktuelle Zeichenfarbe zugeordnet. Diese kann mit den beiden folgenden Methoden manipuliert werden:

```
public Color getColor()
```

Liefert die aktuelle Farbe dieses Grafikkontextes.

```
public void setColor(Color c)
```

Setzt die aktuelle Farbe dieses Grafikkontextes.

8.6.2 Darstellung von Text

Zur Darstellung von Text verwendet man eine der folgenden Methoden der Klasse `Graphics`:

```
public void drawString(String str, int x, int y)
```

```
public void drawChars(char data[], int offset, int length, int x, int y)
```

```
public void drawBytes(byte data[], int offset, int length, int x, int y)
```

Mit `drawString()` wird der String `str` im Grafikfenster an der Position `(x, y)` ausgegeben. Das Koordinatenpaar `(x, y)` bezeichnet dabei das linke Ende der Basislinie des ersten Zeichens in `str`. Die Methoden `drawChars()` und `drawBytes()` sind Variationen von `drawString()`. Anstelle eines Strings erwarten sie ein Array von Zeichen bzw. Bytes als Quelle für den auszugebenden Text. Mit `offset` und `length` stehen zwei zusätzliche Parameter zur Verfügung, die zur Angabe der Startposition bzw. der Anzahl der auszugebenden Zeichen verwendet werden können.

Dem Grafikkontext ist zu jeder Zeit ein aktueller Font zugeordnet. Dieser kann mit den beiden folgenden Methoden manipuliert werden:

```
public Font getFont()
```

Liefert den aktuellen Font dieses Grafikkontextes.

```
public void setFont(Font f)
```

Setzt den aktuellen Font dieses Grafikkontextes.

8.6.3 Zeichnen von Linien

Zum Zeichnen von Linien steht die folgende Methode aus `Graphics` zur Verfügung:

```
public void drawLine(int x1, int y1, int x2, int y2)
```

Zeichnet eine Linie vom Punkt `(x1/y1)` zum Punkt `(x2/y2)`.

```
1 public void paintComponent(Graphics g) {  
2     super.paintComponent(g);  
3  
4     int x = 80;  
5     for(int i = 0; i < 60; i++) {  
6         g.drawLine(x, 40, x, 100);  
7         x += 1+3*Math.random();  
8     }  
9 }
```

Listing 8.15: `java.awt.Graphics` - Zeichnen von Linien

Ausgabe:

8.6.4 Zeichnen von Rechtecken

Zum Zeichnen von Rechtecken stehen die folgenden Methoden aus `Graphics` zur Verfügung:

```
public void drawRect(int x, int y, int width, int height)
```

```
public void fillRect(int x, int y, int width, int height)
```

Zeichnet bzw. füllt ein Rechteck der Breite `width` und der Höhe `height`, dessen linke obere Ecke an



Abbildung 8.11: Graphics - Zeichnen von Linien

der Position (x/y) liegt.

```
public void drawRoundRect(int x, int y, int width, int height,
                        int arcWidth, int arcHeight)
public void fillRoundRect(int x, int y, int width, int height,
                        int arcWidth, int arcHeight)
```

Zeichnet bzw. füllt ein Rechteck mit abgerundeten Ecken, wobei `arcWidth` und `arcHeight` den horizontalen und vertikalen Radius des Ellipsenabschnitts bestimmen, der zur Darstellung der "runden Ecke" verwendet wird.

8.6.5 Zeichnen von Kreisen und Ellipsen

```
public void drawOval(int x, int y, int width, int height)
public void fillOval(int x, int y, int width, int height)
```

Zeichnen bzw. füllen Ellipsen (Kreise). Die übergebenen Parameter spezifizieren ein Rechteck der Größe `width` und `height`, dessen linke obere Ecke an der Position (x/y) liegt. Gezeichnet wird die größte Ellipse, die vollständig in das Rechteck hineinpasst.

```
public void drawArc(int x, int y, int width, int height,
                  int startAngle, int arcAngle)
public void fillArc(int x, int y, int width, int height,
                  int startAngle, int arcAngle)
```

Zeichnen bzw. füllen Ellipsenbogen. Dabei handelt es sich um einen zusammenhängenden Abschnitt der Umfangslinie einer Ellipse. Die ersten 4 Parameter stimmen dabei mit jenen von `drawOval()` überein. Mit `startAngle` wird der Winkel angegeben, an dem mit dem Kreisabschnitt begonnen werden soll, und `arcAngle` gibt den zu überdeckenden Bereich an. Dabei bezeichnet ein Winkel von 0 Grad die 3-Uhr-Position, und positive Winkel werden entgegen dem Uhrzeigersinn gemessen. Als Einheit wird Grad verwendet und nicht das sonst übliche Bogenmaß.

Beispiel:

```
1 public void paintComponent(Graphics g) {
2     super.paintComponent(g);
3
4     g.setColor(Color.GREEN);
5     g.drawRect(10, 40, 50, 50);
6     g.setColor(Color.RED);
7     g.drawArc(10, 40, 50, 50, 30, 300);
8     g.setColor(Color.GREEN);
9     g.drawRect(70, 40, 50, 50);
10    g.setColor(Color.RED);
11    g.fillArc(70, 40, 50, 50, 30, 300);
12 }
```

Listing 8.16: `java.awt.Graphics` - Zeichnen von Ellipsen

8.6.6 Zeichnen von Polygonen

Zum Zeichnen von Polygonen (geschlossenen Streckenzügen) stehen die folgenden Funktionen aus `Graphics` zur Verfügung:

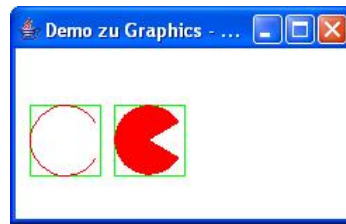


Abbildung 8.12: Graphics - Zeichnen von Ellipsen

```
public void drawPolygon(int []x, int []y, int n)
```

Zeichnet ein geschlossenes Polygon mit den synchronisierten Koordinatenfeldern `x` und `y`. Die Anzahl der gültigen Koordinatenpaare wird durch `n` festgelegt.

```
public void fillPolygon(int []x, int []y, int n)
```

Füllt das Polygon mit der aktuellen Zeichenfarbe, sonst wie oben.

```
public void drawPolygon(Polygon p)
```

```
public void fillPolygon(Polygon p)
```

Zeichnet bzw. füllt das Polygon `p`.

Zum Zeichnen von nicht geschlossenen Streckenzügen steht die Funktion `drawPolyline()` zur Verfügung:

```
public void drawPolyline(int []x, int []y, int n)
```

Zeichnet einen offenen Polygon mit den synchronisierten Koordinatenfeldern `x` und `y`. Die Anzahl der gültigen Koordinatenpaare wird durch `n` festgelegt.

Beispiel:

```

1 public void paintComponent(Graphics g) {
2     super.paintComponent(g);
3
4     int []x = {10, 30, 40, 50, 70, 50, 40, 30};
5     int []y = {70, 60, 40, 60, 70, 80, 100, 80};
6
7     Polygon star = new Polygon(x, y, x.length);
8     g.drawPolygon(star);
9     Rectangle bounds = star.getBounds();
10    g.drawRect(bounds.x, bounds.y, bounds.width, bounds.height);
11    for(int i = 0; i < x.length; i++) {
12        x[i] += 80;
13    }
14    g.fillPolygon(x, y, x.length);
15    for(int i = 0; i < x.length; i++) {
16        x[i] += 80;
17    }
18    g.drawPolyline(x, y, 5);
19 }
```

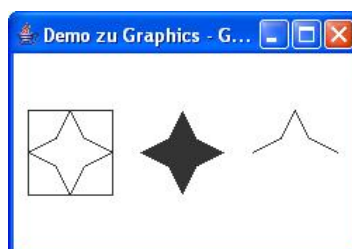
Listing 8.17: `java.awt.Graphics` - Zeichnen von Polygonen

Abbildung 8.13: Graphics - Zeichnen von Polygonen

8.7 Zeichnen mit `java.awt.Graphics2D`

Die Klasse `Graphics2D` stellt bei weitem bessere Grafikmöglichkeiten als die Klasse `Graphics` zur Verfügung. `Graphics2D` erbt von `Graphics` und die Methoden `paint(Graphics g)` bzw. `paintComponent(Graphics g)` bekommen in Wirklichkeit ein `Graphics2D`-Objekt übergeben, d.h. man kann das übergebene Objekt ohne Probleme auf `Graphics2D` casten:

```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;
    // weitere Implementierung
}
```

Im Folgenden werden die erweiterten Möglichkeiten von `Graphics2D` im Überblick dargestellt. Da `Graphics2D` von `Graphics` erbt, sind natürlich auch alle im obigen Abschnitt vorgestellten Methoden verfügbar.

8.7.1 Zeichnen von Objekten

Zum Zeichnen von Objekten gibt es im Wesentlichen die beiden Methoden

```
public void draw(Shape s)
public void fill(Shape s)
```

Zeichnet bzw. füllt das Objekt `s`.

Bei `Shape` handelt es sich um ein Interface in `java.awt`. Im Folgenden werden die wichtigsten Klassen angegeben, die dieses Interface implementieren und deren Instanzen daher mit Hilfe der oben angegebenen Methoden `draw()` bzw. `fill()` gezeichnet werden können.

Alle diese Klassen gibt es in 2 Versionen, nämlich `XXX2D.Float` und `XXX2D.Double`. Dabei handelt es sich um innere Kindklassen der jeweiligen abstrakten Klasse `XXX2D`. Die beiden inneren Kindklassen unterscheiden sich nur durch die Genauigkeit der gespeicherten Werte und können sonst genau gleich verwendet werden.

8.7.2 Linien

Linien sind Objekte der der Klassen `Line2D.Float` bzw. `Line2D.Double`:

```
public Line2D.Float(float x1, float y1, float x2, float y2)
public Line2D.Float(Point2D p1, Point2D p2)
public Line2D.Double(double x1, double y1, double x2, double y2)
public Line2D.Double(Point2D p1, Point2D p2)
```

Erzeugt ein `Line2D`-Objekt mit Anfangspunkt `(x1/y1)` bzw. `p1` und Endpunkt `(x2/y2)` bzw. `p2`.

8.7.3 Rechtecke

Rechtecke sind Objekte der der Klassen `Rectangle2D.Float` bzw. `Rectangle2D.Double`:

```
public Rectangle2D.Float(float x, float y, float w, float h)
public Rectangle2D.Double(double x, double y, double w, double h)
```

Erzeugt ein `Rectangle2D`-Objekt mit linker oberer Ecke `(x/y)` und der Breite `w` und Höhe `h`.

8.7.4 Ellipsen

Ellipsen sind Objekte der der Klassen `Ellipse2D.Float` bzw. `Ellipse2D.Double`:

```
public Ellipse2D.Float(float x, float y, float w, float h)
public Ellipse2D.Double(double x, double y, double w, double h)
```

Erzeugt ein `Ellipse2D`-Objekt, dessen umschließendes Rechteck die linke oberer Ecke `(x/y)` sowie der Breite `w` und die Höhe `h` hat.

Beispiel:

```

1 public void paintComponent(Graphics g) {
2     super.paintComponent(g);
3
4     Graphics2D g2d = (Graphics2D)g;
5     g2d.draw(new Rectangle2D.Float(30, 30, 100, 50));
6     g2d.setColor(Color.RED);
7     g2d.fill(new Ellipse2D.Float(30, 30, 100, 50));
8 }

```

Listing 8.18: `java.awt.Graphics2D` - Zeichnen von Rechtecken und Ellipsen

Ausgabe:

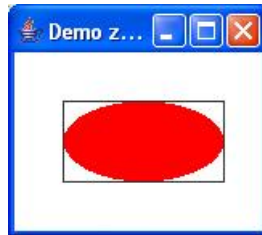


Abbildung 8.14: Graphics - Zeichnen von Rechtecken und Ellipsen

8.7.5 Bögen

Bögen sind Objekte der Klassen `Arc2D.Float` bzw. `Arc2D.Double`:

```

public Arc2D.Float(float x, float y, float w, float h,
                  float start, float extent, int type)
public Arc2D.Float(Rectangle2D ellipseBounds,
                  float start, float extent, int type)
public Arc2D.Double(double x, double y, double w, double h,
                   double start, double extent, int type)
public Arc2D.Double(Rectangle2D ellipseBounds,
                   double start, double extent, int type)

```

Erzeugt ein `Arc2D`-Objekt, wobei der Bogen auf einer Ellipse liegt, dessen umschließendes Rechteck die linke obere Ecke (x/y) sowie der Breite w und die Höhe h hat. Alternativ kann dieses umschließende Rechteck auch über `ellipseBounds` angegeben werden. Mit `start` wird der Winkel angegeben, an dem mit dem Bogen begonnen werden soll, und `extent` gibt den zu überdeckenden Bereich an. Dabei bezeichnet ein Winkel von 0 Grad die 3-Uhr-Position, und positive Winkel werden im entgegen dem Uhrzeigersinn gemessen. Als Einheit wird Grad verwendet und nicht das sonst übliche Bogenmaß. Der Integerwert `type` gibt an, wie der Bogen geschlossen wird. Mögliche Werte sind:

- `Arc2D.OPEN` - der Bogen wird nicht geschlossen
- `Arc2D.CHORD` - der Bogen wird durch eine Sehne geschlossen
- `Arc2D.PIE` - die Endpunkte des Bogens werden mit dem Mittelpunkt verbunden

Beispiel:

```

1 public void paintComponent(Graphics g) {
2     super.paintComponent(g);
3
4     Graphics2D g2d = (Graphics2D)g;
5     g2d.draw(new Arc2D.Float(30, 30, 100, 100, 225, 90, Arc2D.OPEN));
6     g2d.draw(new Arc2D.Float(130, 30, 100, 100, 225, 90, Arc2D.CHORD));
7     g2d.draw(new Arc2D.Float(230, 30, 100, 100, 225, 90, Arc2D.PIE));
8 }

```

Listing 8.19: `java.awt.Graphics2D` - Zeichnen von Bögen

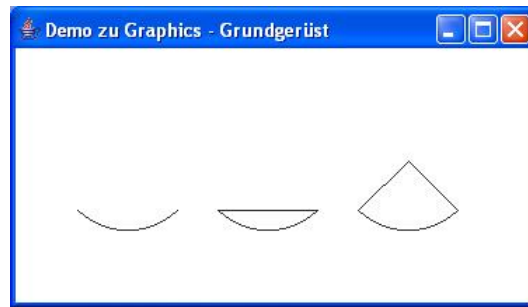


Abbildung 8.15: Graphics - Zeichnen von Bögen

8.7.6 Quadratische Kurven

Quadratische Kurven sind Objekte der Klassen `QuadCurve2D.Float` bzw. `QuadCurve2D.Double`:

```
public QuadCurve2D.Float(float x1, float y1,
                        float ctrlx, float ctrly,
                        float x2, float y2)
public QuadCurve2D.Double(double x1, double y1,
                        double ctrlx, double ctrly,
                        double x2, double y2)
```

Erzeugt eine quadratische Kurve mit den Endpunkten $(x1/y1)$ und $(x2/y2)$. Der Kontrollpunkt $(ctrlx/ctrly)$ ist der Schnittpunkt der Tangenten in diesen Endpunkten.

8.7.7 Kubische Kurven

Kubische Kurven sind Objekte der Klassen `CubicCurve2D.Float` bzw. `CubicCurve2D.Double`:

```
public CubicCurve2D.Float(float x1, float y1,
                        float ctrlx1, float ctrly1,
                        float ctrlx2, float ctrly2,
                        float x2, float y2)
public CubicCurve2D.Double(double x1, double y1,
                        double ctrlx1, double ctrly1,
                        double ctrlx2, double ctrly2,
                        double x2, double y2)
```

Erzeugt eine kubische Kurve mit den Endpunkten $(x1/y1)$ und $(x2/y2)$. Die Kontrollpunkte $(ctrlx1/ctrly1)$ bzw. $(ctrlx2/ctrly2)$ liegen auf den Tangenten in $(x1/y1)$ bzw. $(x2/y2)$.

Beispiel:

```
1 public void paintComponent(Graphics g) {
2     super.paintComponent(g);
3
4     Graphics2D g2d = (Graphics2D) g;
5     Point2D.Float p1 = new Point2D.Float(20, 20);
6     Point2D.Float p2 = new Point2D.Float(100, 50);
7     Point2D.Float c1 = new Point2D.Float(40, 100);
8     Point2D.Float c2 = null;
9
10    g2d.setColor(Color.RED);
11    g2d.draw(new Line2D.Float(p1, c1));
12    g2d.draw(new Line2D.Float(p2, c1));
13    g2d.setColor(Color.BLACK);
14    g2d.setStroke(new BasicStroke(2));
15    g2d.draw(new QuadCurve2D.Double(p1.getX(), p1.getY(), c1.getX(), c1.getY(), p2.getX(), p2.
        getY()));
16    p1 = new Point2D.Float(130, 100);
17    p2 = new Point2D.Float(200, 30);
18    c1 = new Point2D.Float(110, 20);
```



```

19  c2 = new Point2D.Float(240, 110);
20  g2d.setColor(Color.RED);
21  g2d.setStroke(new BasicStroke(1));
22  g2d.draw(new Line2D.Float(p1, c1));
23  g2d.draw(new Line2D.Float(p2, c2));
24  g2d.setColor(Color.BLACK);
25  g2d.setStroke(new BasicStroke(2));
26  g2d.draw(new CubicCurve2D.Double(p1.getX(), p1.getY(), c1.getX(), c1.getY(),
27                                   c2.getX(), c2.getY(), p2.getX(), p2.getY()));
28  }

```

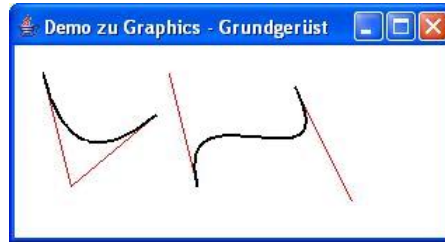
Listing 8.20: `java.awt.Graphics2D` - Zeichnen von Quadratischen und Kubischen Kurven

Abbildung 8.16: Graphics - Zeichnen von Quadratischen und Kubischen Kurven

8.7.8 Konstruktive Flächengeometrie

Eine Möglichkeit, komplexe Shapes zu erzeugen ist das Kombinieren existierender Shapes. Zu diesem Zweck gibt es die Klasse `java.awt.geom.Area`:

```

public class Area extends Object
implements Shape, Cloneable

```

Es gibt zwei Möglichkeiten, Objekte zu erzeugen:

```

public Area()
Erzeugt ein leeres Area-Objekt.

```

```

public Area(Shape s)
Erzeugt ein Area-Objekt aus dem Shape s.

```

Mit Hilfe der folgenden Operationen können zwei Area-Objekte kombiniert werden:

```

public void add(Area rhs)
Fügt dieser Area die Area rhs hinzu (Vereinigungsmenge).

```

```

public void intersect(Area rhs)
Bildet die Durchschnittsmenge des this-Objektes und der Area rhs.

```

```

public void subtract(Area rhs)
Subtrahiert die Area rhs vom this-Objekt (Differenzmenge).

```

```

public void exclusiveOr(Area rhs)
Subtrahiert von der Vereinigungsmenge die Durchschnittsmenge und speichert das Ergebnis im this-Objekt.

```

Das folgende Beispiel demonstriert das Arbeiten mit den oben vorgestellten Techniken:

```

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

```

```

5
6 public class AreaGeometry extends JApplet {
7     public static void main(String s[]) {
8         JFrame frame = new JFrame();
9         frame.setTitle("Constructive Area Geometry");
10        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        JApplet applet = new AreaGeometry();
12        applet.init();
13        frame.getContentPane().add(applet);
14        frame.pack();
15        frame.setVisible(true);
16    }
17
18    public void init() {
19        JPanel panel = new AreaPanel();
20        getContentPane().add(panel);
21    }
22 }
23
24 class AreaPanel extends JPanel {
25     public AreaPanel() {
26         setPreferredSize(new Dimension(760, 230));
27     }
28
29     public void paintComponent(Graphics g) {
30         Graphics2D g2 = (Graphics2D)g;
31         Shape s1 = new Ellipse2D.Double(0, 0, 100, 100);
32         Shape s2 = new Rectangle2D.Double(20, 60, 60, 100);
33         g2.translate(20, 50);
34         g2.draw(s1);
35         g2.draw(s2);
36         g2.translate(150, 0);
37         Area a1 = new Area(s1);
38         a1.add(new Area(s2));
39         g2.fill(a1);
40         g2.translate(150, 0);
41         a1 = new Area(s1);
42         a1.intersect(new Area(s2));
43         g2.fill(a1);
44         g2.translate(150, 0);
45         a1 = new Area(s1);
46         a1.subtract(new Area(s2));
47         g2.fill(a1);
48         g2.translate(150, 0);
49         a1 = new Area(s1);
50         a1.exclusiveOr(new Area(s2));
51         g2.fill(a1);
52     }
53 }

```

Listing 8.21: Konstruktive Flächengeometrie

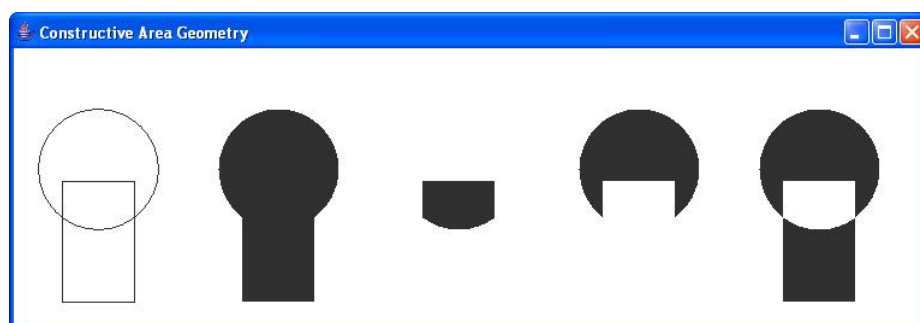


Abbildung 8.17: Graphics - Konstruktive Flächengeometrie

8.7.9 Pfade

Eine weitere Möglichkeit, neu Shapes zu erzeugen, bietet Klasse `java.awt.geom.GeneralPath`. Damit kann man Pfade (und damit von diesen Pfaden umrandete Shapes) erzeugen, die aus den 5 Grundelementen eines Pfades bestehen.

Diese sind im Interface `java.awt.geom.PathIterator` definiert:

- `SEG_MOVETO`
- `SEG_LINETO`
- `SEG_QUADTO`
- `SEG_CUBICTO`
- `SEG_CLOSE`

Passend zu diesen Grundelementen eines Pfades stellt die Klasse `GeneralPath` entsprechende Methoden zur Verfügung, aus denen ein Pfad aufgebaut werden kann:

```
public void moveTo(float x, float y)
```

Fügt den Punkt mit den Koordinaten `x/y` dem Pfad hinzu.

```
public void lineTo(float x, float y)
```

Fügt dem Pfad eine Linie vom aktuellen Punkt zum Punkt mit den Koordinaten `x/y` hinzu.

```
public void quadTo(float x1, float y1, float x2, float y2)
```

Fügt dem Pfad eine quadratische Kurve vom aktuellen Punkt zum Punkt mit den Koordinaten `x2/y2` hinzu, wobei der Punkt `x1/y1` als Kontrollpunkt verwendet wird.

```
public void curveTo(float x1, float y1, float x2,
                   float y2, float x3, float y3)
```

Fügt dem Pfad eine kubische Kurve vom aktuellen Punkt zum Punkt mit den Koordinaten `x3/y3` hinzu, wobei die Punkte `x1/y1` und `x2/y2` als Kontrollpunkte verwendet werden.

Das folgende Programm demonstriert das Arbeiten mit Pfaden:

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4 import java.awt.geom.*;
5
6 public class PathDemo extends JApplet {
7     public static void main(String s[]) {
8         JFrame frame = new JFrame();
9         frame.setTitle("PathDemo");
10        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        JApplet applet = new AreaGeometry();
12        applet.init();
13        frame.getContentPane().add(applet);
14        frame.pack();
15        frame.setVisible(true);
16    }
17
18    public void init() {
19        JPanel panel = new AreaPanel();
20        getContentPane().add(panel);
21    }
22 }
23
24 class AreaPanel extends JPanel {
25     public AreaPanel() {
26         setPreferredSize(new Dimension(350, 100));
27     }
28
29     public void paintComponent(Graphics g) {
30         Graphics2D g2 = (Graphics2D)g;
31     }
```

```

32 //Erzeugen eines Pfades
33 GeneralPath path = new GeneralPath();
34 path.moveTo(-40f, 0f);
35 path.quadTo(0f, 40f, 40f, 0f);
36 path.quadTo(0f, -40f, -40f, 0f);
37 path.moveTo(-20f, 10f);
38 path.lineTo(-20f, -10f);
39 path.lineTo(20f, 10f);
40 path.lineTo(20f, -10f);
41 path.closePath();
42
43 g2.translate(60, 60);
44 g2.draw(path);
45 g2.translate(100, 0);
46 g2.fill(path);
47 path.setWindingRule(GeneralPath.WIND_EVEN_ODD);
48 g2.translate(100, 0);
49 g2.fill(path);
50 }
51 }

```

Listing 8.22: Arbeiten mit Pfaden

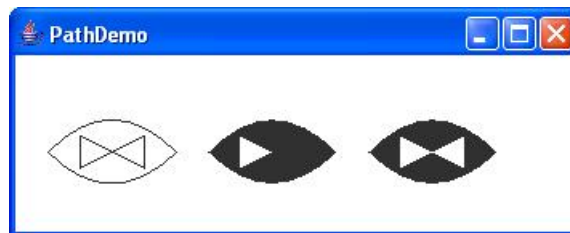


Abbildung 8.18: Graphics - Arbeiten mit Pfaden

Bemerkung:

Welche der inneren Flächen in obigen Beispiel gefüllt werden, lässt sich über die sogenannte Winding-Rule beeinflussen. Für nähere Informationen zu diesem nicht trivialen Thema wird auf die API-Dokumentation verwiesen.

8.7.10 Affine Transformationen

Affine Transformationen sind punkttreue und parallelentreue Abbildungen in der Ebene oder im Raum² und werden in der Computergrafik intensiv eingesetzt. Das Java-API stellt Klassen zur Verarbeitung solcher affinen Transformationen zur Verfügung. Die wichtigsten affinen Transformationen sind:

- Translation (Schiebung)
- Rotation
- Reflection (Spiegelung)
- Scaling (Vergrößerung, Verkleinerung)
- Shearing (Scherung)

Mathematische Grundlagen

Mathematisch werden zweidimensionale affine Transformationen durch 3 x 3 - Matrizen beschrieben. Da eine affine Transformation durch die Gleichungen

$$x_1 = m_{00}x + m_{01}y + m_{02} \quad y_1 = m_{10}x + m_{11}y + m_{12}$$

²hier werden nur 2D-Affinitäten behandelt

beschrieben wird, muss auf 3×3 - Matrizen erweitert werden³, um das Hintereinanderausführen von affinen Transformationen mit Hilfe der Matrizenmultiplikation beschreiben zu können. Obiges Gleichungssystem lässt sich nun wie folgt schreiben:

$$\begin{pmatrix} x1 \\ y1 \\ 1 \end{pmatrix} = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Die Matrizen zu den oben angegebenen affinen Transformationen lauten:

- Translation um den Vektor $\vec{t} = \begin{pmatrix} a \\ b \end{pmatrix}$:

$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix}$$

- Rotation um den Winkel φ mit dem Zentrum $Z(0/0)$:

$$\begin{pmatrix} \cos \varphi & \sin \varphi & 0 \\ -\sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Skalierung mit den Faktoren α in x-Richtung und β in y-Richtung:

$$\begin{pmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Spiegelung an der Geraden $y = kx$:

$$\begin{pmatrix} \frac{2}{1+k^2} - 1 & \frac{2k}{1+k^2} & 0 \\ \frac{2k}{1+k^2} & \frac{2k^2}{1+k^2} - 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Scherung in x-Richtung mit dem Faktor s bzw. in y-Richtung mit dem Faktor t :

$$\begin{pmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 \\ t & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Die Klasse `java.awt.geom.AffineTransform`

Mit Hilfe dieser Klasse werden affine Transformationen definiert:

```
public class AffineTransform extends Object
implements Cloneable, Serializable
```

Die wichtigsten Konstruktoren sind:

```
public AffineTransform()
Erzeugt eine Identität.
```

```
public AffineTransform(double m00, double m10,
                        double m01, double m11,
                        double m02, double m12)
public AffineTransform(float m00, float m10,
                        float m01, float m11,
                        float m02, float m12)
```

³Homogenisierung des Koordinatensystems

Erzeugen eine affine Transformation mit der entsprechenden Matrix.

```
public AffineTransform(double[] flatmatrix)
public AffineTransform(float[] flatmatrix)
```

Erzeugen eine affine Transformation, wobei `flatmatrix` aus 4 oder 6 Werten besteht:

```
{ m00 m10 m01 m11 [m02 m12]}
```

Die folgenden Methoden definieren auf einem bestehenden Objekt vom Typ `AffineTransform` spezielle Affinitäten:

```
public void setToIdentity()
public void setToRotation(double theta)
public void setToRotation(double theta, double x, double y)
public void setToScale(double sx, double sy)
public void setToShear(double shx, double shy)
public void setToTranslation(double tx, double ty)
```

Alle diese Methoden setzen die affine Transformation neu, löschen also vorher definierte Affinitäten. Die überladene Methode `setToRotation` erzeugt eine Rotation mit dem Zentrum $Z(x/y)$.

Transformation eines Shapes

Mit der folgenden Methode der Klasse `AffineTransform` können Shapes transformiert werden:

```
public Shape createTransformedShape(Shape pSrc)
```

Erzeugt und liefert ein neues Shape-Objekt, das durch Transformation von `pSrc` entsteht.

Transformation des Graphics2D-Objektes

Eine AffineTransformation kann auch auf das gesamte `Graphics2D`-Objekt angewendet werden. Zu diesem Zweck stellt die Klasse `Graphics2D` die folgenden Methoden zur Verfügung:

```
public void setTransform(AffineTransform tx)
```

Diese Methode ersetzt eine bestehende Transformation des `Graphics2D`-Objektes durch `tx`.

```
public void transform(AffineTransform tx)
```

Diese Methode verkettet eine bestehende Transformation des `Graphics2D`-Objektes mit `tx`.

Veketten affiner Transformationen

Oft ist es sinnvoll und notwendig, affine Transformationen zu verketten. Sind M_1 , M_2 und M_3 affine Transformationen, so lässt sich die Verkettung von M_3 , M_2 und M_1 in dieser Reihenfolge wie folgt schreiben:

$$\vec{p}_1 = (M_1 \circ M_2 \circ M_3) \vec{p} = M_1 (M_2 (M_3 (\vec{p}))) = M_1 \cdot M_2 \cdot M_3 \cdot \vec{p}$$

Verkettungen werden also immer von rechts nach links durchgeführt, die Verkettung ist nicht kommutativ. Zur Verkettung von affinen Transformationen stellt die Klasse `AffineTransform` die folgenden Methoden bereit:

```
public void rotate(double theta)
public void rotate(double theta, double x, double y)
public void scale(double sx, double sy)
public void shear(double shx, double shy)
public void transform(double tx, double ty)
```

Anders als bei den `setTo`-Methoden wird die entsprechende Transformation an die bereits gespeicherte rechts angehängt. Die Transformationen werden als in umgekehrter Reihenfolge des Methodenaufrufe durchgeführt.

Im Beispiel

```

AffineTransform t = new AffineTransform();
t.rotate(Math.PI / 3.0);
t.scale(2, 0.3);
t.translate(100, 200);

```

wird also eine affine Transformation erzeugt, in der zuerst verschoben, dann skaliert und zum Schluss rotiert wird.

Beispiel:

Das folgende Beispiel demonstriert die Verkettung affiner Transformationen. Dabei wird die Rotation einer Ellipse in Einzelschritte zerlegt. Zuerst wird der Mittelpunkt der Ellipse in den Ursprung verschoben, dann wird die Ursprungsellipse gedreht und zum Schluss wird die gedrehte Ellipse wieder auf ihre Originalposition verschoben:

```

1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.geom.*;
4
5 public class Verkettung extends JApplet {
6     public static void main(String s[]) {
7         JFrame frame = new JFrame();
8         frame.setTitle("Verkettung von Affinitäten");
9         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10        JApplet applet = new Verkettung();
11        applet.init();
12        frame.getContentPane().add(applet);
13        frame.pack();
14        frame.setVisible(true);
15    }
16
17    public void init() {
18        JPanel panel = new CompositionPanel();
19        getContentPane().add(panel);
20    }
21 }
22
23 class CompositionPanel extends JPanel {
24     public CompositionPanel() {
25         setPreferredSize(new Dimension(640, 480));
26         this.setBackground(Color.white);
27     }
28
29     public void paintComponent(Graphics g) {
30         super.paintComponent(g);
31         Graphics2D g2 = (Graphics2D)g;
32         g2.translate(100,100);
33         Shape e = new Ellipse2D.Double(200, 100, 200, 100);
34         g2.setColor(new Color(160,160,160));
35         g2.fill(e);
36         AffineTransform transform = new AffineTransform();
37         transform.translate(-300,-150);
38         e = transform.createTransformedShape(e);
39         g2.setColor(new Color(220,220,220));
40         g2.fill(e);
41         g2.setColor(Color.black);
42         g2.drawLine(0, 0, 150, 0);
43         g2.drawLine(0, 0, 0, 150);
44         transform.setToRotation(Math.PI / 6.0);
45         e = transform.createTransformedShape(e);
46         g2.setColor(new Color(100,100,100));
47         g2.draw(e);
48         transform.setToTranslation(300, 150);
49         e = transform.createTransformedShape(e);
50         g2.setColor(new Color(0,0,0));
51         g2.draw(e);
52     }
53 }

```

Listing 8.23: Verkettung von Affinitäten

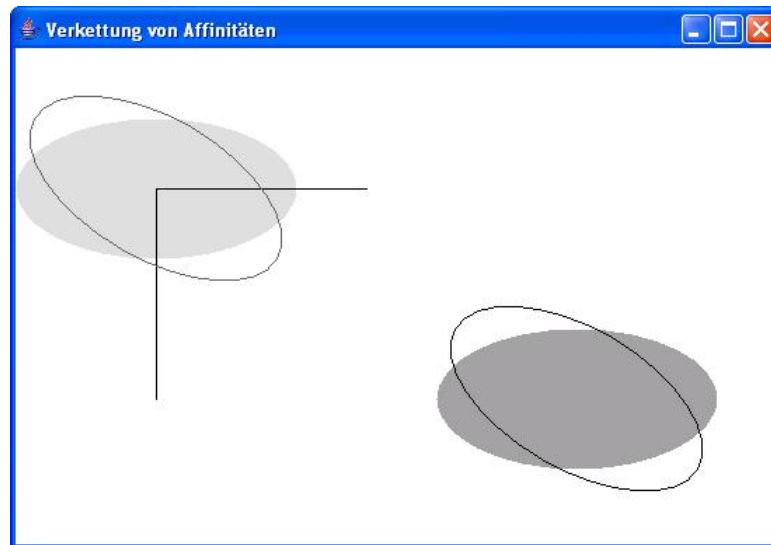


Abbildung 8.19: Graphics - Verkettung von Affinitäten

Bemerkungen:

- Die Verkettung in obigem Beispiel dient nur Demonstrationszwecken. Das gleiche Ziel hätte auch wie folgt erreicht werden können:

```
AffineTransform transform = new AffineTransform();
transform.setToRotation(Math.PI / 6.0, 300, 150);
```

- Eine ausgezeichnete Webseite zum Studium affiner Transformationen findet sich unter:

<http://www.glyphic.com/transform/applet/1intro.html>

8.7.11 Darstellungsattribute festlegen

Anders als beim Arbeiten mit **Graphics** hat man bei **Graphics2D** erweiterte Möglichkeiten um festzulegen, wie ein Zeichenojekt dargestellt werden soll. Unter anderem bietet **Graphics2D** Möglichkeiten zur Festlegung von Linienstärke, Füllmuster und Transparenz.

Antialiasing

Java2D kann Schriften und Grafiken wesentlich weicher zeichnen, indem es Antialiasing verwendet. Das ist eine Rendering-Technik, die scharfe Kanten weichzeichnet, indem sie die Farbe der umgebenden Pixel anpasst.

Antialiasing ist standardmäßig deaktiviert. Man Aktiviert es mit der Methode `setRenderingHint()` des `Graphics2D`-Objekts:

```
g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                     RenderingHints.VALUE_ANTIALIAS_ON);
```

Dieser Code aktiviert Antialiasing bei einem `Graphics2D`-Objekt `g2d`.

Beispiel:


```

1 public void paintComponent(Graphics g) {
2     super.paintComponent(g);
3
4     int x, y;
5     String text;
6     Graphics2D g2d = (Graphics2D)g;
7     Font f = new Font("SansSerif", Font.BOLD, 50);
8     FontMetrics metrics = getFontMetrics(f);
9     g2d.setFont(f);
10    text = "Text ohne Antialiasing";
11    x = (getSize().width - metrics.stringWidth(text)) / 2;
12    y = getSize().height / 3;
13    g2d.drawString(text, x, y);
14    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
15                          RenderingHints.VALUE_ANTIALIAS_ON);
16    text = "Text mit Antialiasing";
17    x = (getSize().width - metrics.stringWidth(text)) / 2;
18    y += y;
19    g2d.drawString(text, x, y);
20 }

```

Listing 8.24: `java.awt.Graphics2D` - Zeichnen von Text mit Antialiasing

Abbildung 8.20: Graphics - Zeichnen von Polygonen

Strichstärke festlegen

`Graphics2D` bietet die Möglichkeit, die verwendete Linienstärke festzulegen. Dazu verwendet man die Methode `setStroke()`:

```
public void setStroke(Stroke s)
```

Setzt das Zeichenwerkzeug für Linien.

Als `Stroke`⁴ übergibt man in der Regel ein `BasicStroke`-Objekt, das man mit folgenden Konstruktoren erzeugt:

```
public BasicStroke(float width)
public BasicStroke(float width, int cap, int join)
```

Erzeugen ein `BasicStroke`-Objekt mit folgenden Attributen:

- `width` - einen `float`-Wert, der die Linienstärke angibt
- `cap` - ein `int`-Wert, der die Art des Linienendes festlegt. Mögliche Werte sind:

⁴Interface in `java.awt`



Abbildung 8.21: Graphics2D - Stile für Linienenden

- `join` - ein `int`-Wert, der den Stil des Verbindungsstücks zwischen zwei Liniensegmenten festlegt. Mögliche Werte sind:



Abbildung 8.22: Graphics2D - Stile für Linienenden

8.7.12 Animationen

Animationen verwenden dynamische Änderungen des graphischen Inhaltes zur Darstellung von Bewegungen. Wenn die Bildwiederholrate eine gewisse Frequenz erreicht (z.B. 60 Bilder pro Sekunde), so wirkt die Bewegung kontinuierlich und damit realistisch. Es gibt zwei Möglichkeiten zur Realisierung einer Animation:

1. Implementierung eines eigenen Threads
2. Verwendung eines Timers vom Typ `javax.swing.Timer`

Implementierung eines eigenen Threads

Steuert man die Animation mit einem eigenen Thread, so ist zu beachten, dass Swing-Komponenten nicht threadsafe sind und daher nicht von einem anderen als dem Event-Dispatcher-Thread aus angesprochen werden sollten. So darf z.B. vom Animationsthread aus nicht die Methode `getGraphics()` aufgerufen werden, um mit dem so erhaltenen `Graphics`-Objekt auf Swing-Komponenten zu zeichnen. Es gibt aber zwei Swingmethoden, die threadsafe sind und daher von anderen Threads aus aufgerufen werden dürfen:

```
public void repaint()
public void revalidate()
```

Ein gute Vorgangsweise zur Erzeugung eines Animationsthreads ist es, die Darstellung von der Änderung der Modelldaten zu trennen. Nachdem die darzustellenden Daten im Animationsthread neu berechnet wurden, kann mit Hilfe der Methode `repaint()` eine Neuzeichnung veranlasst werden. Das folgende Programmfragment demonstriert die Vorgangsweise:

```
public void paintComponent(Graphics g) {
    // Neuzeichnen der Modelldaten
}

// run()-Methode des Animationsthreads
public void run() {
    while(running) {
        // Neuberechnung der Modelldaten
        repaint();
        try {
```

```

        Thread.sleep(delay);
    } catch (InterruptedException e) { /* ... */ }
}

```

Im folgenden Beispiel wird die oben beschriebene Methode verwendet, um Regen zu simulieren:

```

1  import java.awt.*;
2  import java.awt.geom.*;
3  import java.awt.event.*;
4  import java.util.*;
5  import javax.swing.*;
6
7  public class Regen extends JApplet {
8      public static void main(String s[]) {
9          JFrame frame = new JFrame();
10         frame.setTitle("Regen");
11         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12         JApplet applet = new Regen();
13         applet.init();
14         frame.getContentPane().add(applet);
15         frame.pack();
16         frame.setVisible(true);
17     }
18
19     public void init() {
20         JPanel panel = new RegenPanel();
21         getContentPane().add(panel);
22     }
23 }
24
25 class RegenPanel extends JPanel implements Runnable{
26     Point2D.Double[] pts = new Point2D.Double[1200];
27
28     public RegenPanel() {
29         setPreferredSize(new Dimension(640, 480));
30         setBackground(Color.GRAY);
31         for (int i = 0; i < pts.length; i++) {
32             pts[i] = new Point2D.Double(Math.random(), Math.random());
33         }
34         Thread thread = new Thread(this);
35         thread.start();
36     }
37
38     public void paintComponent(Graphics g) {
39         super.paintComponent(g);
40         g.setColor(Color.white);
41         for (int i = 0; i < pts.length; i++) {
42             int x = (int) (640*pts[i].x);
43             int y = (int) (480*pts[i].y);
44             int h = (int) (25*Math.random());
45             g.drawLine(x, y, x, y+h);
46         }
47     }
48
49     public void run() {
50         while(true) {
51             for (int i = 0; i < pts.length; i++) {
52                 double x = pts[i].getX();
53                 double y = pts[i].getY();
54                 y += 0.1*Math.random();
55                 if (y > 1) {
56                     y = 0.3*Math.random();
57                     x = Math.random();
58                 }
59                 pts[i].setLocation(x, y);
60             }
61             repaint();
62             try {
63                 Thread.sleep(100);
64             } catch (InterruptedException ex) {}
65         }
66     }
67 }

```

67 }

Listing 8.25: Animation 1: Regen

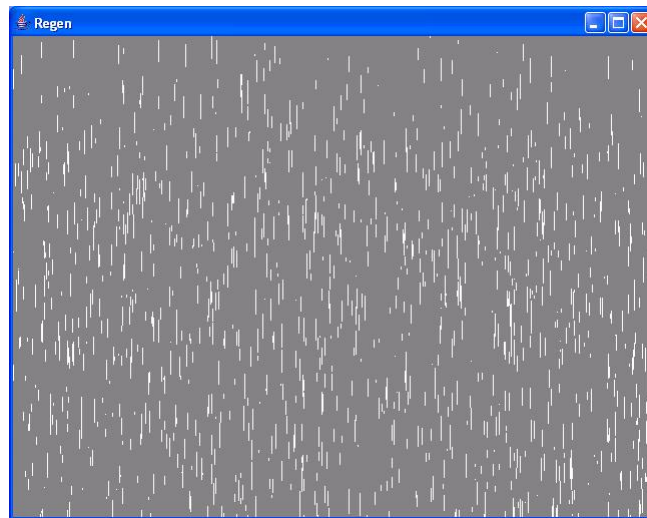


Abbildung 8.23: Graphics - Animierter Regen

Vewendung von `javax.swing.Timer`

Ein Objekt der Klasse `javax.swing.Timer` erzeugt periodisch Action-Events. Damit wird in vordefinierten Zeitintervallen die Methode `actionPerformed()` des zugeordneten Action-Listeners aufgerufen, in der das Bild neu gezeichnet werden kann. Zur Initialisierung des Timers kann dem Konstruktor das Zeitintervall an der ActionListener übergeben werden:

```
public Timer(int delay, ActionListener listener)
```

Erzeugt einen Timer, der alle `delay` Millisekunden ein Action-Event auslöst und damit die Methode `actionPerformed` im Listener `listener` auslöst.

Der Timer kann mit Hilfe der Methode `public void start()` gestartet und mit `public void stop()` gestoppt werden.

Beispiel

Das folgende Beispiel demonstriert eine animierte analoge Uhr, die über ein Timerobjekt gesteuert wird:

```

1 import java.awt.*;
2 import java.awt.geom.*;
3 import java.awt.event.*;
4 import java.util.Calendar;
5 import javax.swing.*;
6
7 public class Clock2D extends JApplet {
8     public static void main(String s[]) {
9         JFrame frame = new JFrame();
10        frame.setTitle("Clock");
11        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12        JApplet applet = new Clock2D();
13        applet.init();
14        frame.getContentPane().add(applet);
15        frame.pack();
16        frame.setVisible(true);
17    }
18
19    public void init() {
20        JPanel panel = new ClockPanel();

```



Abbildung 8.24: Graphics - Animierte analoge Uhr

```

21     getContentPane().add(panel);
22 }
23 }
24
25 class ClockPanel extends JPanel implements ActionListener{
26     AffineTransform rotH = new AffineTransform();
27     AffineTransform rotM = new AffineTransform();
28     AffineTransform rotS = new AffineTransform();
29
30     public ClockPanel() {
31         setPreferredSize(new Dimension(640, 480));
32         setBackground(Color.white);
33         Timer timer = new Timer(500, this);
34         timer.start();
35     }
36
37     public void paintComponent(Graphics g) {
38         super.paintComponent(g);
39         Graphics2D g2 = (Graphics2D)g;
40         g2.translate(320,240);
41         // clock face
42         Paint paint = new GradientPaint(-150,-150,Color.white,150,150,Color.gray);
43         g2.setPaint(paint);
44         g2.fillOval(-190, -190, 380, 380);
45         g2.setColor(Color.gray);
46         g2.drawString("Java 2D", -20, 80);
47         Stroke stroke = new BasicStroke(3);
48         g2.setStroke(stroke);
49         g2.drawOval(-190, -190, 380, 380);
50         for (int i = 0; i < 12; i++) {
51             g2.rotate(2*Math.PI/12);
52             g2.fill3DRect(-3, -180, 6, 30, true);
53         }
54         // clock hands
55         Shape hour = new Line2D.Double(0, 0, 0, -80);
56         hour = rotH.createTransformedShape(hour);
57         Shape minute = new Line2D.Double(0, 0, 0, -120);
58         minute = rotM.createTransformedShape(minute);
59         Shape second = new Line2D.Double(0, 0, 0, -120);
60         second = rotS.createTransformedShape(second);
61         g2.setColor(Color.black);
62         g2.setStroke(new BasicStroke(5, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND));
63         g2.draw(hour);
64         g2.draw(minute);
65         g2.setStroke(new BasicStroke(2));
66         g2.draw(second);
67     }
68 }

```

```
69 public void actionPerformed(ActionEvent e) {
70     int hour = Calendar.getInstance().get(Calendar.HOUR);
71     int min = Calendar.getInstance().get(Calendar.MINUTE);
72     int sec = Calendar.getInstance().get(Calendar.SECOND);
73     rothH.setToRotation(Math.PI * (hour+min/60.0)/6.0);
74     rotM.setToRotation(Math.PI * min /30.0);
75     rotS.setToRotation(Math.PI * sec /30.0);
76     repaint();
77 }
78 }
```

Listing 8.26: Animation 2: Analoge Uhr

8.8 Das Model-View-Controller-Konzept (MVC)

8.8.1 Das Model-View-Controller Konzept

Um graphische Oberflächen zu programmieren genügen prozedurale Konstrukte wie Verzweigungen und Schleifen nicht. Ereignisse (Events) können asynchron auftreten und müssen mit Hilfe einer Warteschlange in definierter Reihenfolge abgearbeitet werden. GUI-APIs benötigen daher ein neues Bearbeitungsmodell mit zusätzlichen Möglichkeiten (vor allem Events, die Reaktion auf Events und Parallelität).

- Das *Model* ist für den Zustand (die Daten) einer Komponente zuständig. Verschiedene Komponenten haben natürlich verschiedene Models. Z. B. kennt das Model eines Scrollbars die aktuelle Position des verschiebbaren Balkens sowie den minimal und den maximal darstellbaren Wert, während das Model zu einem Tabellensteuerelement z.B. ein zweidimensionales Feld sein kann. Das Model und seine Daten sind unabhängig von der visuellen Darstellung der Komponenten und speichern keine Daten, die nur zur Darstellung der Komponente benötigt werden. So ist es nicht Aufgabe des Models einer Textkomponente, die aktuelle Cursorposition zu speichern.
- Die *View* beschreibt, wie die Komponente am Bildschirm dargestellt wird.
- Der *Controller* bestimmt, wie eine Komponente auf Events reagiert. Beispiele für Events sind Mausklick, Fokus bekommen oder verlieren, Tastaturevents und viele andere mehr.

Die nächste Abbildung beschreibt schematisch die Zusammenhänge für einen Scrollbar. Die View stellt

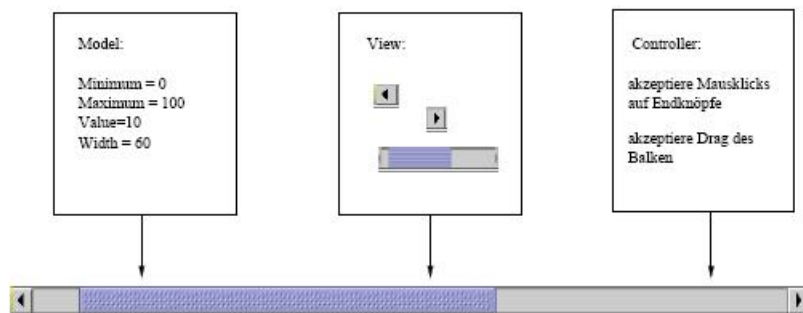


Abbildung 8.25: Model-View-Controller Konzept 1

sich gemäß den Daten im Model dar. So ist z.B. die aktuelle Position des Schiebereglers im Modell gespeichert. Der Controller erkennt, dass das Verschieben der Schiebemarke eine legale Aktion ist. Wenn dieses Event auftritt, korrigiert der Controller die Daten im Model. Swing benutzt eine vereinfachte Form dieses allgemeinen MVC-Konzepts. View und Controller werden zu einem Objekt zusammengefasst, das die Komponente rendert und das auch die Events verarbeitet. Dieses Objekt wird UI-Delegate genannt. Daher besteht eine Swingkomponente also aus einem Model und einem UI-Delegate. Das Model verwaltet den Zustand (die Daten) der Komponente. Das UI-Delegate ist verantwortlich für die Informationen, die benötigt werden, um die Komponente darzustellen. Zusätzlich reagiert sie auf Events.

Ein Model kann auch mehreren Views gleichzeitig zugeordnet werden. Änderungen des Models durch den ersten Controller verändern die Daten im Model und wirken sich daher unmittelbar auf die zweite View aus. Das folgende Beispiel demonstriert dies an einem Scrollbar und einem Slider. Das Standardmodel für die beiden Steuerelemente ist das `DefaultBoundedRangeModel`. Dieses kapselt 4 Integerwerte, die bereits im Konstruktor dieser Klasse gesetzt werden können:

```
public DefaultBoundedRangeModel(int value, // aktueller Wert
                                int extent, // Laenge des Reglers
                                int min,    // Minimalwert
                                int max)   // Maximalwert
```

Konstruktor der Klasse `DefaultBoundedRangeModel`.

```
1 package awt_swing;
2
```

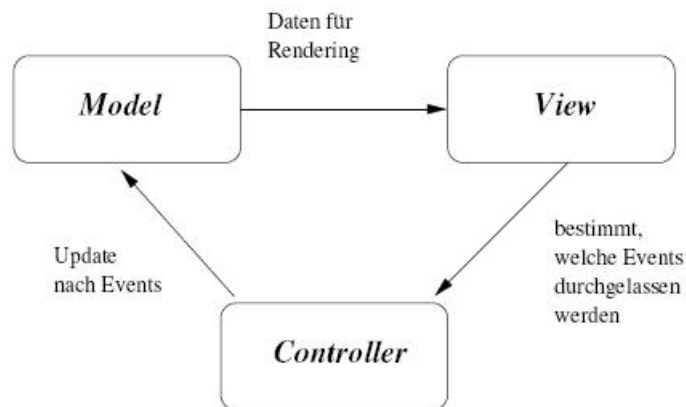


Abbildung 8.26: Model-View-Controller Konzept 2

```

3 import java.awt.*;
4 import javax.swing.*;
5
6 class JScrollbarPanel extends JPanel {
7
8     private JScrollBar sb = null;
9     private JSlider slider = null;
10    private DefaultBoundedRangeModel m = null;
11
12    public JScrollbarPanel() {
13        this.sb = new JScrollBar(JScrollBar.VERTICAL);
14        this.sb.setPreferredSize(new Dimension(25, 150));
15        this.slider = new JSlider(JSlider.HORIZONTAL);
16        // neues Modell erzeugen
17        this.m = new DefaultBoundedRangeModel(30, 5, 0, 90);
18        this.sb.setModel(m); // Scrollbar als View zuweisen
19        this.slider.setModel(m); // Slider als View zuweisen
20        this.setPreferredSize(new Dimension(400, 170));
21        this.add(this.sb);
22        this.add(this.slider);
23    }
24 }
25
26 public class JScrollbarMain extends JFrame {
27     private JScrollbarPanel p = null;
28
29     public JScrollbarMain() {
30         this.p = new JScrollbarPanel();
31         this.setTitle("MVC - Bounded Range Model");
32         this.setLocation(50, 50);
33         this.getContentPane().add(this.p, BorderLayout.CENTER);
34         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
35         this.pack();
36     }
37
38     public static void main(String... args) {
39         java.awt.EventQueue.invokeLater(new Runnable() {
40             public void run() {
41                 new JScrollbarMain().setVisible(true);
42             }
43         });
44     }
45 }

```

Listing 8.27: Ein Model - mehrere Views JScrollbarMain.java

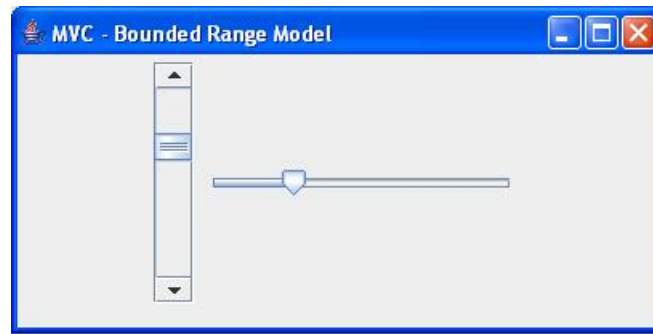


Abbildung 8.27: Ein Model - mehrere Views

8.8.2 Vordefinierte Datenmodelle

Für die diversen Swing-Steuerelemente gibt es vordefinierte Datenmodelle. Jedem Datenmodell liegt ein Interface zugrunde. Die folgende Tabelle bietet eine Aufstellung der wichtigsten Swing-Komponenten, der zugehörigen Interfaces und der speziellen fertigen Implementierungen. Wenn eine Komponente nicht angeführt ist, so erbt die Komponente ihr Datenmodell von der Superklasse, so erbt z.B. die Komponente `JButton` das Datenmodell von `AbstractButton`. In manchen Fällen benötigt man mehrere Interfaces, um das Verhalten einer Komponente zu beschreiben. So sind einer `JTable` z.B. sowohl ein `TableModel`, ein `TableColumnModel` und ein `ListSelectionModel` zugeordnet.

Komponente	Data Model Interface	Implementierung
<code>AbstractButton</code>	<code>ButtonModel</code>	<code>DefaultButtonModel</code>
<code>JColorChooser</code>	<code>ColorSelectionModel</code>	<code>DefaultColorSelectionModel</code>
<code>JComboBox</code>	<code>ComboBoxModel</code>	---
	<code>MutableComboBoxModel</code>	<code>DefaultComboBoxModel</code>
<code>JFileChooser</code>	<code>ListModel</code>	<code>BasicDirectoryModel</code>
<code>JList</code>	<code>ListModel</code>	<code>AbstractListModel</code> <code>DefaultListModel</code>
	<code>ListSelectionModel</code>	<code>DefaultListSelectionModel</code>
<code>JMenuBar</code>	<code>SingleSelectionModel</code>	<code>DefaultSingleSelectionModel</code>
<code>JPopupMenu</code>	<code>SingleSelectionModel</code>	<code>DefaultSingleSelectionModel</code>
<code>JProgressBar</code>	<code>BoundedRangeModel</code>	<code>DefaultBoundedRangeModel</code>
<code>JScrollBar</code>	<code>BoundedRangeModel</code>	<code>DefaultBoundedRangeModel</code>
<code>JSlider</code>	<code>BoundedRangeModel</code>	<code>DefaultBoundedRangeModel</code>
<code>JSpinner</code>	<code>SpinnerModel</code>	<code>AbstractSpinnerModel</code> <code>SpinnerDateModel</code> <code>SpinnerListModel</code> <code>SpinnerNumberModel</code>
<code>JTabbedPane</code>	<code>ListSelectionModel</code>	<code>DefaultListSelectionModel</code>
<code>JTable</code>	<code>TableModel</code>	<code>AbstractTableModel</code> <code>DefaultTableModel</code>
	<code>TableColumnModel</code>	<code>DefaultTableColumnModel</code>
	<code>ListSelectionModel</code>	<code>DefaultListSelectionModel</code>
<code>JTextComponent</code>	<code>Document</code>	<code>AbstractDocument</code> <code>PlainDocument</code> <code>StyledDocument</code> <code>DefaultStyledDocument</code> <code>HtmlDocument</code>
<code>JToggleButton</code>	<code>ButtonModel</code>	<code>JToggleButton</code> <code>ToggleButtonModel</code>
<code>JTree</code>	<code>TreeModel</code>	<code>DefaultTreeModel</code>
	<code>TreeSelectionModel</code>	<code>DefaultTreeSelectionModel</code> <code>JTree.EmptySelectionModel</code>

Modelle können den einzelnen Swingkomponenten entweder bereits im Konstruktor oder mit Hilfe der

Methoden

```
setModel()  
setDocument()
```

zugeordnet werden.

Verwendet man eine konkrete Implementierung eines Datenmodells direkt, so werden bei einer Änderung der Daten im Modell alle registrierten Views automatisch benachrichtigt. Diese kümmern sich dann um ein Refresh ihrer Oberfläche. Diese Automatik ist der größte Vorteil des MVC-Konzeptes. Programmiert man sein eigenes Modell durch Ableiten einer abstrakten Implementierung, so hat man sich im um die Benachrichtigung der View durch Aufruf diverser `fireXXX()`-Methoden selbst zu kümmern. Im Folgenden wird die Verwendung der oben aufgelisteten Model-Implementierungen an Hand einiger Beispiele demonstriert.

8.9 Verwendung einer *JList*

8.9.1 Die Klasse `DefaultListModel<E>`

Diese seit Java 1.7 generische Klasse dient als Standardmodell für eine *JList* verwendet als Datenspeicher die obsoleete Collection `java.util.Vector<E>`. Im Wesentlichen kann man mit diesem Modell die gleichen Operationen wie mit einem Vector durchführen (Hinzufügen und entfernen von Elementen, indizierter Zugriff).

8.9.2 Die Klasse `AbstractListModel`

Die Klasse

```
public abstract class AbstractListModel extends Object  
                                     implements ListModel, Serializable
```

erbt vom Interface `ListModel` die folgenden beiden abstrakten Methoden:

```
public abstract int getSize()  
Liefert die Länge der Liste.
```

```
public abstract Object getElementAt(int index)  
Liefert das Element an der Position index.
```

Darüber hinaus hat Sie zur Kommunikation mit der View die folgenden konkreten Methoden:

```
public void addListDataListener(ListDataListener l)
```

Fügt diesem Listmodell einen Listener `l` hinzu, der jedesmal benachrichtigt wird, wenn sich Daten im Modell ändern.

```
public void removeListDataListener(ListDataListener l)  
Entfernt aus diesem Listmodel den Listener l.
```

```
public ListDataListener[] getListDataListeners()  
Liefert alle bei diesem Modell registrierten Listener.
```

```
protected void fireContentsChanged(Object source, int i0, int i1)  
Subklassen müssen diese Methode aufrufen, wenn sich ein oder mehrere Elemente im Modell verändert haben. Die Position der Änderungen wird durch das Intervall i0 bis i1 (Randpunkte eingeschlossen) angegeben.
```

```
protected void fireIntervalAdded(Object source, int i0, int i1)  
Subklassen müssen diese Methode aufrufen, wenn ein oder mehrere Elemente in das Modell eingefügt wurden. Die Position der neuen Elemente wird durch das Intervall i0 bis i1 (Randpunkte eingeschlossen) angegeben.
```

protected void fireIntervalRemoved(Object source, int i0, int i1)
 Subklassen müssen diese Methode aufrufen, wenn ein oder mehrere Elemente aus dem Modell entfernt wurden. Die Position der entfernten Elemente wird durch das Intervall *i0* bis *i1* (Randpunkte eingeschlossen) angegeben.

Jedem *AbstractListModel* können beliebig viele Instanzen vom Typ *ListDataListener* zugeordnet werden. Dieses Interface im Paket *javax.swing.event* definiert die Methoden

```
public abstract void contentsChanged(ListDataEvent e)
public abstract void intervalAdded(ListDataEvent e)
public abstract void intervalRemoved(ListDataEvent e)
```

Ruft man im *ListModel* eine der obigen *fireXXX()*-Methoden auf, so werden in allen registrierten *ListDataListener* die entsprechenden Methoden aufgerufen.

Beispiel

Das folgende Beispiel demonstriert die Implementierung eines Listmodels, das Integerwerte sortiert speichert. Jeder Interwert wird so in das Modell eingefügt, dass die Ineterwerte immer aufsteigend sortiert sind.

```
1 import java.util.List;
2 import java.util.ArrayList;
3 import java.util.Collections;
4 import javax.swing.AbstractListModel;
5 import javax.swing.event.ListDataListener;
6 import javax.swing.event.ListDataEvent;
7
8 public class SortedIntListModel extends AbstractListModel {
9
10     // Datenspeicher
11     List<Integer> data = new ArrayList<Integer> ();
12
13     // Ueberschreiben der abstrakten Methode getSize()
14     @Override
15     public int getSize() {
16         return data.size();
17     }
18
19     // Ueberschreiben der abstrakten Methode getElementAt(int index)
20     @Override
21     public Object getElementAt(int index) {
22         return data.get(index);
23     }
24
25     // Hinzufuegen eines Elements
26     public void addInteger(Integer val) {
27         int i = Collections.binarySearch(data, val);
28         if(i < 0) {
29             i = -(i + 1);
30         }
31         data.add(i, val);
32         fireIntervalAdded(this, i, i);
33     }
34
35     // Entfernen eines Elements
36     // Liefert das entfernte Element bzw. null, wenn val nicht gefunden wird
37     public Integer removeInteger(Integer val) {
38         int i = Collections.binarySearch(data, val);
39         if(i >= 0) {
40             Integer del = data.remove(i);
41             fireIntervalRemoved(this, i, i);
42             return del;
43         }
44
45         return null;
46     }
47
48     // Ueberschreiben von toString()
```

```

49 @Override
50 public String toString() {
51     return this.data.toString();
52 }
53
54 // main() - Methode zum Testen
55 public static void main(String []args) {
56     // SortedIntListModel instanziiieren
57     SortedIntListModel m = new SortedIntListModel();
58
59     // Mit Hilfe einer anonymen inneren Klasse wird ein ListDataListener erzeugt
60     // und dem SortedIntListModel hinzugefuegt
61     m.addListDataListener(new ListDataListener() {
62         @Override
63         public void intervalRemoved(ListDataEvent e) {
64             System.out.println("IntervalRemoved: " + e.getIndex0() + " - " + e.getIndex1());
65             System.out.println("New Data: " + e.getSource());
66             System.out.println("-----");
67         }
68         @Override
69         public void intervalAdded(ListDataEvent e) {
70             System.out.println("IntervalAdded: " + e.getIndex0() + " - " + e.getIndex1());
71             System.out.println("New Data: " + e.getSource());
72             System.out.println("-----");
73         }
74         @Override
75         public void contentsChanged(ListDataEvent e) {
76             System.out.println("IntervalChanged: " + e.getIndex0() + " - " + e.getIndex1());
77             System.out.println("New Data: " + e.getSource());
78             System.out.println("-----");
79         }
80     });
81
82     // Diveres Operationen auf dem Listmodel durchfuehren
83     m.addInteger(8);
84     m.addInteger(10);
85     m.removeInteger(7);
86     m.addInteger(8);
87     m.removeInteger(10);
88 }
89 }

```

Listing 8.28: Selbst definiertes ListModel `SortetIntListModel.java`

Ausgabe:

```
IntervalAdded: 0 - 0
New Data: [8]
-----
IntervalAdded: 1 - 1
New Data: [8, 10]
-----
IntervalAdded: 0 - 0
New Data: [8, 8, 10]
-----
IntervalRemoved: 2 - 2
New Data: [8, 8]
```

8.9.3 Rendern einer JList

Eine `JList` verwendet zur Darstellung der Daten in der View einen `ListCellRenderer`. Der Standardrenderer stellt die Stringdarstellungen der im Model gespeicherten Daten in Komponenten vom Typ `JLabel` linksbündig dar. Man kann diesen Standardrenderer durch einen selbst programmierten Renderer ersetzen. Dieser muss das Interface `ListCellRenderer` implementieren. Dieses Interface definiert als einzige die folgende Methode:

[illegible]

Die Methode liefert die an der Position `index` darzustellende Komponente. `list` ist eine Referenz auf die darzustellende `JList`, `value` das Objekt aus dem zugehörigen Model, `isSelected` und `cellHasFocus` beschreiben den Zustand der darzustellenden Zelle.

Mit Hilfe der `JList`-Methode

```
public void setCellRenderer(ListCellRenderer cellRenderer)
```

kann einer `JList` dieser selbst programmierte Renderer zugewiesen werden.

Beispiel

Das folgende Beispiel ordnet einer `JList` ein `SortedIntListModel` zu, füllt das Model mit 5 Werten im Bereich von 0 bis 20 und rendert die Liste mit Objekten vom Typ `JSlider`:

```

1 import java.awt.BorderLayout;
2 import java.awt.Component;
3 import java.awt.EventQueue;
4 import javax.swing.JFrame;
5 import javax.swing.JList;
6 import javax.swing.JSlider;
7 import javax.swing.ListCellRenderer;
8 import javax.swing.border.EmptyBorder;
9
10 public class CustomListRenderer extends JFrame {
11
12     private JList jList = null;
13     private SortedIntListModel m = new SortedIntListModel();
14     private int []data = { 12, 3, 17, 5, 3 };
15
16     // Konstruktor
17     public CustomListRenderer() {
18         super("Eigener List-Renderer");
19         for(int i : data) {
20             m.addInteger(i);
21         }
22         this.jList = new JList(m);
23         this.jList.setBorder(new EmptyBorder(20,20,20,20));
24         this.jList.setCellRenderer(new MyCellRenderer());
25         this.setLocation(50, 50);
26         this.getContentPane().add(jList, BorderLayout.CENTER);
27         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28         this.pack();
29     }
30
31     // Startmethode main()
32     public static void main(String... args) {
33         EventQueue.invokeLater(new Runnable() {
34             public void run() {
35                 new CustomListRenderer().setVisible(true);
36             }
37         });
38     }
39 }
40
41 // Selbst programmierter ListCellRenderer
42 class MyCellRenderer extends JSlider implements ListCellRenderer<Integer> {
43
44     public MyCellRenderer() {
45         this.setMinimum(0);
46         this.setMaximum(20);
47         this.setMajorTickSpacing(5);
48         this.setMinorTickSpacing(1);
49         this.setPaintLabels(true);
50         this.setPaintTicks(true);
51         this.setOpaque(true);
52     }
53
54     @Override
55     public Component getListCellRendererComponent(

```

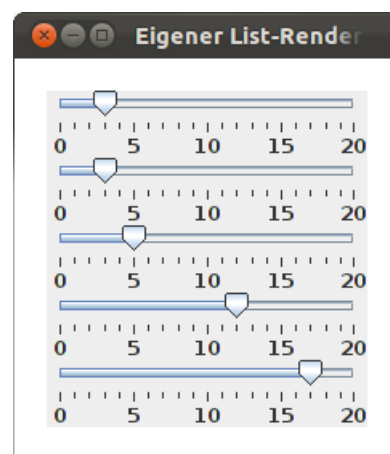
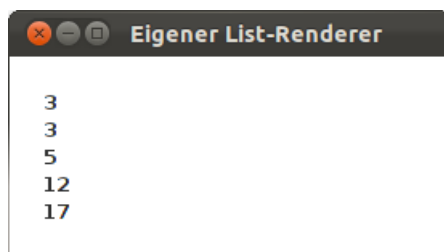
```

56     JList list,           // zugrundeliegende JList
57     Integer value,       // darzustellender Wert aus dem Model
58     int index,           // Index der Zelle
59     boolean isSelected,  // ist die Zelle selektiert?
60     boolean cellHasFocus) { // hat die Zelle den Fokus?
61
62     this.setValue(value);
63     this.setForeground(list.getForeground());
64     this.setEnabled(list.isEnabled());
65
66     return this;
67 }
68 }

```

Listing 8.29: Selbst definierter ListCellRenderer CustomListrenderer.java

Die folgende Ausgabe zeigt das Ergebnis zunächst mit auskommentierter und dann mit einkommentierter Zeile 26:



8.10 Verwendung einer *JTable*

8.10.1 Das Interface *TableModel*

Das Interface `javax.swing.table.TableModel` beschreibt die Schnittstelle eines Modells für das Steuerelement *JTable*. Dieses Interface definiert die folgenden Methoden:

```

public void addTableModelListener(TableModelListener l)
public void removeTableModelListener(TableModelListener l)

```

Diese Methoden dienen zum Hinzufügen bzw. Entfernen eines Table-Model-Listeners. Diese Listener werden benachrichtigt, wenn sich Daten im Table-Model ändern.

```

public int getColumnCount()
public int getRowCount()

```

Liefern die Anzahl der Spalten bzw. Zeilen, die für die Darstellung der Daten im Modell notwendig sind.

```

public Class<?> getColumnClass(int columnIndex)

```

Die hier retournierte Klasse wird für den Defaultrenderer und den Editor dieser Spalte verwendet. Vordefinierte Werte sind:

- `Boolean` - wird als `CheckBox` gerendert
- `Number` - wird mit einem rechtsbündigen Label gerendert
- `Double`, `Float`, `Integer` etc. - wie `Number`, allerdings wird zur Darstellung eine `NumberFormat` - Instanz mit dem Standardformat für die aktuelle Locale verwendet.
- `Date` - zur Darstellung wird eine `DateFormat` - Instanz (`SHORT`-Stil für Datum und Zeit) verwendet.

- ImageIcon, Icon - wird mit einem zentrierten Label gerendert
- Object - wird durch einen Label, der die Stringdarstellung des Objekts zeigt, gerendert

```
public String getColumnName(int columnIndex)
```

Liefert die Überschrift dieser Spalte.

```
public Object getValueAt(int rowIndex, int columnIndex)
```

Liefert den Wert der Zelle rowIndex/columnIndex

```
public void setValueAt(Object aValue, int rowIndex, int columnIndex)
```

Setzt den Wert der Zelle rowIndex/columnIndex.

```
public boolean isCellEditable(int rowIndex, int columnIndex)
```

Liefert true, wenn die Zelle rowIndex/columnIndex editierbar ist. Liefert diese Methode false, so ändert setValueAt() den Wert der Zelle nicht.

8.10.2 Die Klasse AbstractTableModel

Diese Klasse stellt die Basisimplementierung des Interfaces TableModel dar. Die Methoden

```
public int getColumnCount();
```

```
public int getRowCount();
```

```
public Object getValueAt(int row, int column);
```

sind abstrakt und müssen überschrieben werden.

Die Spaltenüberschriften lauten A,B,...,Z, AA,AB,... Die Methode isCellEditable() liefert immer false, das Model ist also read-only, solange diese Methode nicht überschrieben wird.

Verändert man die Daten im Modell von außen, so hat man die zugeordneten TableModelListener mit Hilfe einer der folgenden fireXXX()-Methoden zu benachrichtigen:

```
public void fireTableDataChanged()
```

```
public void fireTableStructureChanged()
```

```
public void fireTableRowsInserted(int firstRow, int lastRow)
```

```
public void fireTableRowsUpdated(int firstRow, int lastRow)
```

```
public void fireTableRowsDeleted(int firstRow, int lastRow)
```

```
public void fireTableCellUpdated(int row, int column)
```

Beispiel

Das folgende Beispiel verwendet eine JTable mit einem selbst definierten TableModel. Die Spalten 2 und 3 werden als Number und als Integer gerendert. Die ersten beiden Spalten sind editierbar.

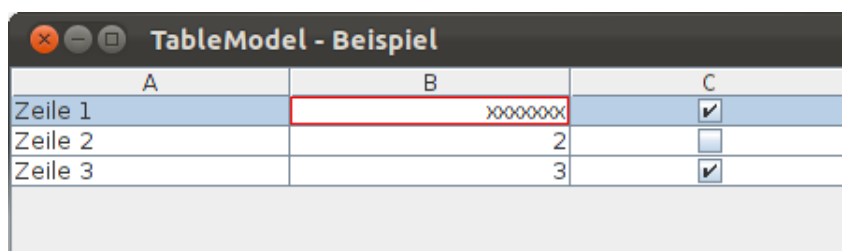
```
1 import java.awt.EventQueue;
2 import java.awt.BorderLayout;
3 import javax.swing.JFrame;
4 import javax.swing.JTable;
5 import javax.swing.JScrollPane;
6 import javax.swing.table.AbstractTableModel;
7
8 public class CustomTableModel extends JFrame {
9
10     private JTable jTable = null;
11     private MyTableModel m = new MyTableModel();
12
13     // Konstruktor
14     public CustomTableModel() {
15         super("TableModel - Beispiel");
16         this.jTable = new JTable(m);           // Table erzeugen - Model zuweisen
17         this.setLocation(50, 50);
18         this.getContentPane().add(new JScrollPane(jTable), BorderLayout.CENTER);
19         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20     }
21 }
```

```

20     this.pack();
21 }
22
23 // Startmethode
24 public static void main(String... args) {
25     EventQueue.invokeLater(new Runnable() {
26         public void run() {
27             new CustomTableModel().setVisible(true);
28         }
29     });
30 }
31 }
32
33 // TableModel - Daten sind eine 2-dimensionales Objectarray
34 class MyTableModel extends AbstractTableModel {
35     private Object [][]data = {"Zeile 1", 1, true},
36                               {"Zeile 2", 2, false},
37                               {"Zeile 3", 3, true}};
38
39     @Override
40     public int getRowCount() {
41         return data.length;
42     }
43
44     @Override
45     public int getColumnCount() {
46         return data[0].length;
47     }
48
49     @Override
50     public Object getValueAt(int rowIndex, int columnIndex) {
51         return data[rowIndex][columnIndex];
52     }
53
54     @Override
55     public Class<?> getColumnClass(int columnIndex) {
56         switch(columnIndex) {
57             case 1: return Integer.class;
58             case 2: return Boolean.class;
59             default: return super.getColumnClass(columnIndex); // Object.class
60         }
61     }
62
63     // 1. und 2.Spalte (Indizes 0 und 1) sind editierbar
64     @Override
65     public boolean isCellEditable(int rowIndex, int columnIndex) {
66         return columnIndex <= 1;
67     }
68
69     // Die editierten Daten werden in das Modell geschrieben
70     @Override
71     public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
72         data[rowIndex][columnIndex] = aValue.toString();
73         System.out.println("Geaendert: " + aValue);
74     }
75 }

```

Listing 8.30: Selbst definierter TableModel CustomTableModel.java



	A	B	C
Zeile 1		xxxxxxx	<input checked="" type="checkbox"/>
Zeile 2		2	<input type="checkbox"/>
Zeile 3		3	<input checked="" type="checkbox"/>

Abbildung 8.28: Selbst definiertes Table-Model

8.10.3 Editieren von Zellen

Macht man eine Zelle editierbar, so wird als Komponente beim Editieren einer Zelle standardmäßig ein `JTextField` verwendet. Spaltenweise kann man auch andere Swingkomponenten als Zelleditor zuweisen. Zu diesem Zweck benötigt man ein Objekt vom Typ `TableCellEditor`. Dabei handelt es sich um ein Interface im Paket `javax.swing.table`. Die einfachste Möglichkeit besteht darin, ein Objekt vom Typ `DefaultCellEditor` zu erzeugen. Diese Klasse implementiert das Interface `TableCellEditor` und hat 3 Konstruktoren:

```
public DefaultCellEditor(JTextField textField)
public DefaultCellEditor(JCheckBox checkBox)
public DefaultCellEditor(JComboBox comboBox)
```

Nun besorgt man sich ein Objekt vom Typ `javax.swing.table.TableColumn`. Jeder Spalte einer `JTable` ist ein solches Objekt zugeordnet. Es speichert alle Attribute einer Spalte, wie z.B. Breite, Überschrift usw. Außerdem kann dem `TableColumn`-Objekt sowohl ein Editor als auch ein Renderer zugewiesen werden:

```
public void setCellEditor(TableCellEditor cellEditor)
public void setCellRenderer TableCellRenderer cellRenderer)
```

Das `TableColumn`-Objekt zu einer bestimmten Spalte besorgt man sich wie folgt (`col` ist dabei der nullbasierte Index der Spalte):

```
TableColumn c = jTable.getColumnModel().getColumn(col);
```

Beispiel

Das folgende Beispiel erweitert das obige Beispiel dahingehend, dass der zweiten Spalte ein Zelleditor in Form einer Combobox zugeordnet wird:

```
1 import java.awt.EventQueue;
2 import java.awt.BorderLayout;
3 import javax.swing.JFrame;
4 import javax.swing.JTable;
5 import javax.swing.JComboBox;
6 import javax.swing.DefaultCellEditor;
7 import javax.swing.JScrollPane;
8 import javax.swing.table.AbstractTableModel;
9 import javax.swing.table.TableColumn;
10 import javax.swing.table.TableCellEditor;
11
12 public class CustomTableModel_1 extends JFrame {
13
14
15     private JTable jTable = null;
16     private MyTableModel m = new MyTableModel();
17
18     public CustomTableModel_1() {
19         super("TableModel - Beispiel");
20
21         // Array der moeglichen Integerwerte
22         Integer [] choices = { 1, 2, 3, 4, 5 };
23         // Combobox fuer TableCellEditor
24         JComboBox<Integer> cBox = new JComboBox<>(choices);
25         // TableCellEditor aus JComboBox erzeugen
26         TableCellEditor editor = new DefaultCellEditor(cBox);
27
28         this.jTable = new JTable(m);
29         this.jTable.setRowHeight(20);
30         // TableColumn fuer 2. Spalte (Index 1) holen
31         TableColumn col = jTable.getColumnModel().getColumn(1);
32         // TableCellEditor zuweisen
33         col.setCellEditor(editor);
34
35         this.setLocation(50, 50);
36         this.getContentPane().add(new JScrollPane(jTable), BorderLayout.CENTER);
37         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```

38     this.pack();
39 }
40
41 // Startmethode main()
42 public static void main(String... args) {
43     EventQueue.invokeLater(new Runnable() {
44         public void run() {
45             new CustomTableModel_1().setVisible(true);
46         }
47     });
48 }
49 }

```

Listing 8.31: Eigener Zelleditor CustomTableModel_1.java

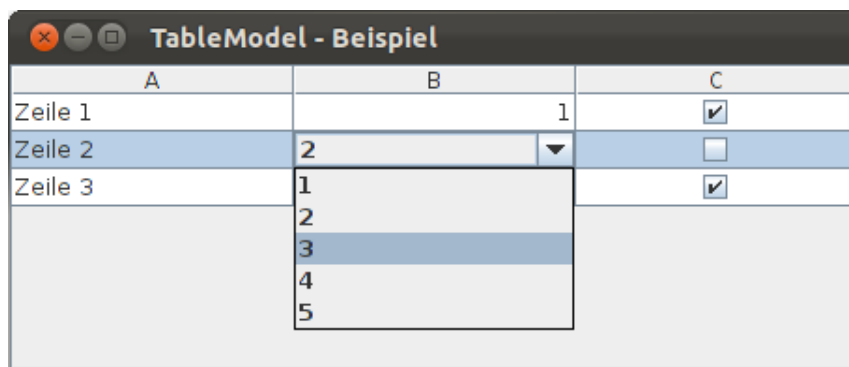


Abbildung 8.29: Eigener Zelleditor

Man kann auch selbst geschriebene Celleditoren verwenden. Dazu erstellt man eine Klasse, die

1. von der Klasse `javax.swing.AbstractCellEditor` erbt.
Dabei hat man die abstrakte Methode
`public abstract Object getCellEditorValue();`
zu überschreiben.
Zwei weitere wichtige Methoden aus dieser Klasse sind:
`protected void fireEditingCanceled()`
Benachrichtigt alle Listener, dass der Editiervorgang abgebrochen wurde.
`protected void fireEditingStopped()`
Benachrichtigt alle Listener, dass der Editiervorgang beendet wurde.
2. das Interface `javax.swing.table.TableCellEditor` implementiert.
Dabei ist die folgende Methode zu implementieren:

```

Component getTableCellEditorComponent(JTable table,
                                       Object value,
                                       boolean isSelected,
                                       int row,
                                       int column);

```

Diese Methode liefert jene grafische Komponente, die den CellEditor visualisiert. `table` referenziert die dabei betroffene Tabelle, `value` ist der zu editierende Wert aus dem Model, die weiteren Parameter sind selbsterklärend.

Die Verwendung von selbst erstellten Celleditoren demonstriert das anschließende Beispiel.

8.10.4 Rendern einer *JTable*

Eine *JTable* verwendet zur Darstellung der Daten in der View einen *TableCellRenderer*. Der Standardrenderer stellt die Daten gemäß der Methode `getColumnClass()` des Models dar. Man kann wie

im folgenden Beispiel demonstriert einen Renderer programmieren, der das Interface `TableCellRenderer` implementiert. Dieses Interface definiert die folgende Methode:

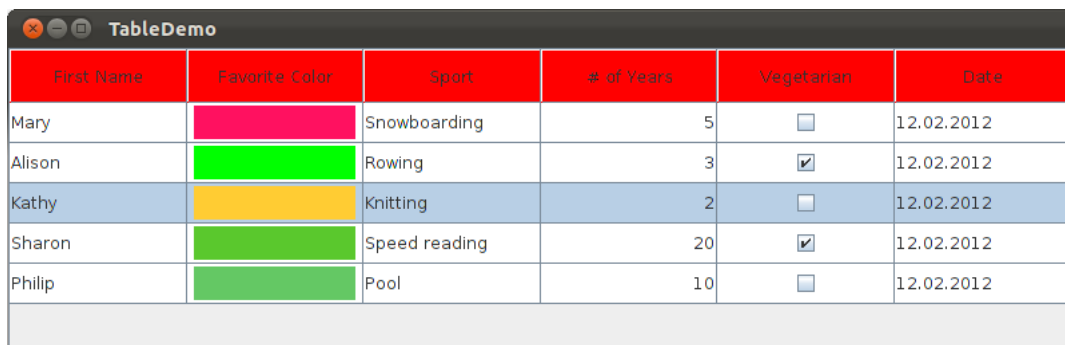
```
public Component getTableCellRendererComponent(JTable table,
                                                Object value,
                                                boolean isSelected,
                                                boolean hasFocus,
                                                int row, int column)
```

Diese Methode liefert die in der Zelle `row/column` darzustellende Komponente. `table` ist eine Referenz auf die darzustellende `JTable`, `value` das Objekt aus dem zugehörigen Model, `isSelected` und `hasFocus` beschreiben den Zustand der darzustellenden Zelle.

Das nächste Beispiel demonstriert die Verwendung.

8.10.5 Abschließendes Beispiel

Das folgende Beispiel demonstriert die wichtigsten Möglichkeiten beim Arbeiten mit einer `JTable`. Es wird die folgende Tabelle generiert:



First Name	Favorite Color	Sport	# of Years	Vegetarian	Date
Mary		Snowboarding	5	<input type="checkbox"/>	12.02.2012
Alison		Rowing	3	<input checked="" type="checkbox"/>	12.02.2012
Kathy		Knitting	2	<input type="checkbox"/>	12.02.2012
Sharon		Speed reading	20	<input checked="" type="checkbox"/>	12.02.2012
Philip		Pool	10	<input type="checkbox"/>	12.02.2012

Abbildung 8.30: Eigener `TableCellRenderer`

Dabei sind folgende Details implementiert:

- In jeder Zeile wird eine Instanz der Klasse `jtable.Data` visualisiert.
- Jede Zelle lässt sich editieren.
- Der zweiten Spalte wurde ein eigener Renderer (`jtable.ColorRenderer`) zugewiesen.
- Der zweiten Spalte wurde ein eigener Editor (`jtable.ColorEditor`) zugewiesen.
- Der letzten Spalte wurde ein eigener Editor (`jtable.DateEditor`) zugewiesen. Dabei wurde das Fremdpaket `jcalendar-1.4.jar` verwendet. Eine Referenz zu diesem Paket findet man unter `JCalendar`.

Die Klasse `jtable.Data`

Jede Zeile der Tabelle visualisiert eine Instanz dieser Klasse.

Listing:

```
1 package jtable;
2
3 import java.awt.Color;
4 import java.util.ArrayList;
5 import java.util.Date;
6 import java.util.List;
7
8 public class Data {
```

```

9  // Instanzvariable
10 private String name;
11 private Color color;
12 private String sport;
13 private Integer number;
14 private Boolean yesno;
15 private Date date;
16
17 // Konstruktor
18 public Data(String name, Color color, String sport, Integer number, Boolean yesno, Date date
19 ) {
20     this.name = name;
21     this.color = color;
22     this.sport = sport;
23     this.number = number;
24     this.yesno = yesno;
25     this.date = date;
26 }
27
28 // Getter und Setter
29 // ...
30
31 // Diese Methode liefert eine Liste von Testdaten
32 public static List<Data> getTestData() {
33     List<Data> testData = new ArrayList<Data>();
34     testData.add(new Data("Mary", new Color(255,17,96), "Snowboarding", 5, false, new Date()))
35     ;
36     testData.add(new Data("Alison", new Color(0,255,0), "Rowing", 3, true, new Date()));
37     testData.add(new Data("Kathy", new Color(17, 36, 78), "Knitting", 2, false, new Date()));
38     testData.add(new Data("Sharon", new Color(90,200,45), "Speed reading", 20, true, new Date
39     ());
40     testData.add(new Data("Philip", new Color(100,200,100), "Pool", 10, false, new Date()));
41     return testData;
42 }
43
44 // Ueberschriften fuer die Tabelle
45 public static final String []HEADER = {"First Name", "Favorite Color", "Sport",
46                                         "# of Years", "Vegetarian", "Date"};
47
48 // Moegliche Sportarten
49 public static final String []SPORTS = {"Snowboarding", "Rowing", "Knitting", "Speed reading"
50                                         ,
51                                         "Pool", "Biking", "Basball", "Soccer" };
52
53 // Klassenobjekte der einzelnen Spalten
54 public static final Class []TYPES = {String.class, Color.class, String.class,
55                                       Integer.class, Boolean.class, Date.class };
56 }

```

Listing 8.32: Datenklasse Data.java

Die Klasse jTable.MyTableModel

Diese Klasse erbt von AbstractTableModel und implementiert das der Tabelle zugeordnete Model.

Listing:

```

1  package jTable;
2
3  import java.awt.Color;
4  import java.util.ArrayList;
5  import java.util.Date;
6  import java.util.List;
7
8  import javax.swing.table.AbstractTableModel;
9
10 public class MyTableModel extends AbstractTableModel {
11
12     // Datenspeicher

```

```
13 private List<Data> data;
14
15 // Konstruktor
16 public MyTableModel() {
17     data = new ArrayList<Data>(Data.getTestData());
18 }
19
20 // Anzahl der Zeilen - muss ueberschrieben werden
21 @Override
22 public int getRowCount() {
23     return data.size();
24 }
25
26 // Anzahl der Spalten - muss ueberschrieben werden
27 @Override
28 public int getColumnCount() {
29     return Data.HEADER.length;
30 }
31
32 // Daten in der Zelle row/col - muss ueberschrieben werden
33 @Override
34 public Object getValueAt(int row, int col) {
35     switch(col) {
36         case 0: return data.get(row).getName();
37         case 1: return data.get(row).getColor();
38         case 2: return data.get(row).getSport();
39         case 3: return data.get(row).getNumber();
40         case 4: return data.get(row).getYesno();
41         case 5: return data.get(row).getDate();
42         default: return null;
43     }
44 }
45
46 // Liefert die Spaltenueberschrift zur Spalte col
47 @Override
48 public String getColumnName(int col) {
49     return Data.HEADER[col];
50 }
51
52 // Alle Zellen sind editierbar
53 @Override
54 public boolean isCellEditable(int row, int col) {
55     return true;
56 }
57
58 // Speichert den vom zugeordneten Celleditor gelieferten Wert value in der Zelle row/col
59 @Override
60 public void setValueAt(Object value, int row, int col) {
61     switch(col) {
62         case 0: data.get(row).setName((String)value);
63                 break;
64         case 1: data.get(row).setColor((Color)value);
65                 break;
66         case 2: data.get(row).setSport((String)value);
67                 break;
68         case 3: data.get(row).setNumber((Integer)value);
69                 break;
70         case 4: data.get(row).setYesno((Boolean)value);
71                 break;
72         case 5: data.get(row).setDate((Date)value);
73                 break;
74     }
75     fireTableCellUpdated(row, col);
76 }
77
78 // Liefert das Klassenobjekt zur Spalte col
79 @Override
80 public Class<?> getColumnClass(int col) {
81     return Data.TYPES[col];
82 }
83
84 }
```

Listing 8.33: MyTableModel MyTableModel.java

Die Klasse jTable.ColorRenderer

Mit Ausnahme der 2.Spalte zur Darstellung der bevorzugten Farbe werden überall die Standardrenderer verwendet. Zum Rendern der 2.Spalten wurde ein eigener Renderer implementiert.

Listing:

```

1 package jTable;
2
3 import java.awt.Color;
4 import java.awt.Component;
5 import javax.swing.BorderFactory;
6 import javax.swing.JLabel;
7 import javax.swing.JTable;
8 import javax.swing.border.Border;
9 import javax.swing.table.TableCellRenderer;
10
11
12 public class ColorRenderer extends JLabel implements TableCellRenderer {
13
14     // Rahmen fuer nicht selektierte Zelle
15     private Border unselectedBorder = null;
16     // Rahmen fuer selektierte Zelle
17     private Border selectedBorder = null;
18     // Soll ein Rahmen gezeichnet werden?
19     private boolean isBordered = true;
20
21     // Konstruktor
22     public ColorRenderer(boolean isBordered) {
23         this.isBordered = isBordered;
24         this.setOpaque(true); // nicht durchsichtig
25     }
26
27     // Liefert die jeweilige grafische Komponente
28     // Hier immer ein JLabel, das die Klasse von JLabel erbt
29     // Muss ueberschrieben werden, kommt aus dem Interface TableCellRenderer
30     @Override
31     public Component getTableCellRendererComponent(JTable table,
32                                                     Object value,
33                                                     boolean isSelected,
34                                                     boolean hasFocus,
35                                                     int row, int column) {
36         Color akt = (Color) value; // Farbwert aus dem Modell
37         this.setBackground(akt); // Hintergrund setzen
38
39         if (isBordered) { // soll der Rahmen gezeichnet werden?
40             if (isSelected) { // ist die Zelle selektiert
41                 if (selectedBorder == null) {
42                     selectedBorder = BorderFactory.createMatteBorder(2, 5, 2, 5, table.
43                         getSelectionBackground());
44                 }
45                 this.setBorder(selectedBorder);
46             }
47             else { // ist die Zelle nicht selektiert
48                 if (unselectedBorder == null) {
49                     unselectedBorder = BorderFactory.createMatteBorder(2, 5, 2, 5, table.getBackground()
50                         );
51                 }
52                 this.setBorder(unselectedBorder);
53             }
54         }
55
56         // Tooltip setzen
57         String tipp = String.format("RGB: [%d/%d/%d]", akt.getRed(), akt.getGreen(), akt.getBlue());
58         this.setToolTipText(tipp);
59     }
60 }

```

```

57     return this;                // das this-Objekt (JLabel) liefern
58 }
59 }

```

Listing 8.34: ColorRenderer ColorRenderer.java

Die Klasse `jtable.ColorEditor`

Für das Editieren von Farben wird ein eigener ColorEditor verwendet. Der eigentliche Editor wird als JButton gerendert, der auf ein Actionevent einen JColorChooser öffnet, mit dessen Hilfe die Farbe gewählt werden kann.

Listing:

```

1 package jtable;
2
3 import java.awt.Color;
4 import java.awt.Component;
5 import java.awt.event.ActionEvent;
6 import java.awt.event.ActionListener;
7 import javax.swing.AbstractCellEditor;
8 import javax.swing.JButton;
9 import javax.swing.JColorChooser;
10 import javax.swing.JTable;
11 import javax.swing.table.TableCellEditor;
12
13 public class ColorEditor extends AbstractCellEditor implements TableCellEditor {
14
15     // Die aktuell zu editierende Farbe
16     private Color currentColor;
17     // Grafische Komponente, die den Editor visualisiert
18     private JButton button;
19
20     // Konstruktor
21     public ColorEditor() {
22         button = new JButton();           // Editor = JButton
23         // Auf einen Click auf den Editor wird ein JColorChooser gerendert
24         button.addActionListener(new ActionListener() {
25             public void actionPerformed(ActionEvent e) {
26                 Color c = JColorChooser.showDialog(button, "Pick a color", currentColor);
27                 if(c != null) {           // Neue Farbe wurde ausgewaehlt
28                     currentColor = c;     // Neue Farbe wird zugewiesen
29                 }
30                 fireEditingStopped();      // Editiervorgang wird beendet
31             }
32         });
33         button.setBorderPainted(false);   // Der Button wird ohne Rahmen gezeichnet
34     }
35
36     // Liefert den durch den Editor eingestellten Wert
37     // Muss ueberschrieben werden - ist in AbstractCellEditor abstrakt
38     @Override
39     public Object getCellEditorValue() {
40         return currentColor;
41     }
42
43     // Liefert die grafische Komponente des Editors
44     // Muss ueberschrieben werden - kommt aus dem Interface TableCellEditor
45     @Override
46     public Component getTableCellEditorComponent(JTable table,
47                                                  Object value,
48                                                  boolean isSelected,
49                                                  int row,
50                                                  int column) {
51         currentColor = (Color) value;
52         button.setBackground(currentColor);
53         return button;
54     }
55 }

```

Listing 8.35: ColorEditor ColorEditor.java

Die Klasse jTable.DateEditor

Auch das Datum wird mit Hilfe eines selbst geschriebenen Editors editiert. Als grafische Komponente wird ein `JDateChooser` aus dem Zusatzpaket `jcalendar-1.4.jar` verwendet. Diese Komponente ist ein vollwertiges JavaBean und stellt damit einen `PropertyChangeListener` zur Verfügung. Damit kann man sich ereignisgesteuert von jeder Änderung einer Eigenschaft benachrichtigen lassen. Hier wird dieser Listener dazu verwendet, um sich von einer Änderung der Property `date` informieren zu lassen. Das Interface `PropertyChangeListener` stammt aus dem Paket `java.beans` und hat folgenden Aufbau:

```
public interface PropertyChangeListener extends EventListener {
    void propertyChange(PropertyChangeEvent evt);
}
```

Die Methode `propertyChangeEvent()` wird immer aufgerufen, wenn sich eine Eigenschaft in dem JavaBean ändert.

Die Eventklasse `java.beans.PropertyChangeEvent` besitzt unter anderem die folgenden Methoden:

```
public String getPropertyName()
Liefert den Namen der Property (JavaBezeichner).
```

```
public Object getOldValue()
Liefert den alten Wert der betroffenen Property.
```

```
public Object getNewValue()
Liefert den neuen Wert der betroffenen Property.
```

Listing von `jtable.DateEditor`:

```
1 package jTable;
2
3 import com.toedter.calendar.JDateChooser;
4 import java.awt.Component;
5 import java.beans.PropertyChangeEvent;
6 import java.beans.PropertyChangeListener;
7 import java.text.DateFormat;
8 import java.util.Date;
9 import javax.swing.AbstractCellEditor;
10 import javax.swing.JTable;
11 import javax.swing.table.TableCellEditor;
12
13 public class DateEditor extends AbstractCellEditor implements TableCellEditor {
14
15     // JDateChooser als grafische Komponente
16     private JDateChooser dateChooser;
17     // Statisches Element zum formatieren des datums
18     private static DateFormat df = DateFormat.getDateInstance();
19
20     // Konstruktor
21     public DateEditor() {
22         dateChooser = new JDateChooser();
23         // Es wird ein PropertyChangeListener registriert, um von jeder Aenderung der
24         // Eigenschaft date in JDateChooser informiert zu werden.
25         dateChooser.addPropertyChangeListener(new PropertyChangeListener() {
26             @Override
27             public void propertyChange(PropertyChangeEvent evt) {
28                 if(evt.getPropertyName().equals("date")) { // es wurde die Prperty date veraendert
29                     Date old = (Date) evt.getOldValue(); // alter Wert
30                     Date neu = (Date) evt.getNewValue(); // neuer Wert
```



```

31         if(old == null || old.equals(neu)) {
32             fireEditingCanceled();
33         }
34         fireEditingStopped();
35     }
36 }
37 });
38 }
39
40 //Liefert den durch den Editor eingestellten Wert
41 // Muss ueberschrieben werden - ist in AbstractCellEditor abstrakt
42 @Override
43 public Object getCellEditorValue() {
44     return dateChooser.getDate();
45 }
46
47 //Liefert die grafische Komponente des Editors
48 // Muss ueberschrieben werden - kommt aus dem Interface TableCellEditor
49 @Override
50 public Component getTableCellEditorComponent(JTable table,
51                                             Object value,
52                                             boolean isSelected,
53                                             int row,
54                                             int column) {
55     dateChooser.setDate((Date) value);
56     return dateChooser;
57 }
58 }

```

Listing 8.36: DateEditor DateEditor.java

Die Applikation jTable.DateEditor

In diesem Abschnitt wird die Applikation vorgestellt.

Listing:

```

1 package jTable;
2
3 import java.awt.Color;
4 import java.awt.Dimension;
5 import java.util.Date;
6
7 import javax.swing.DefaultCellEditor;
8 import javax.swing.JComboBox;
9 import javax.swing.JScrollPane;
10 import javax.swing.JTable;
11 import javax.swing.event.TableModelEvent;
12 import javax.swing.event.TableModelListener;
13 import javax.swing.table.TableModel;
14
15 public class TableDemo extends javax.swing.JFrame {
16
17     // Die Tabelle
18     private JTable jTable1;
19     // Combobox fuer den Celleditor
20     private JComboBox<String> sports;
21
22     // Konstruktor
23     public TableDemo() {
24         initComponents();           // GUI initialisieren
25         registerListener();         // zusaetzliche Listener registrieren
26         // JComboBox fuer DefaultCellEditor (bevorzugter Sport)
27         sports = new JComboBox<String>(Data.SPORTS);
28
29         // DefaultCellEditor auf Spalte 3 setzen
30         jTable1.getColumnModel().getColumn(2).setCellEditor(new DefaultCellEditor(sports));
31
32         // Auf allen Spalten mit der ColumnClass Color den ColorEditor setzen
33         jTable1.setDefaultEditor(Color.class, new ColorEditor());
34     }
35 }

```

```

34 // Alternativ nur auf der Spalte 3
35 //jTable1.getColumnModel().getColumn(1).setCellEditor(new ColorEditor());
36
37 // Auf allen Spalten mit der ColumnClass Color den ColorEditor setzen
38 jTable1.setDefaultEditor(Date.class, new DateEditor());
39 // Alternativ: nur auf Spalte 6 setzen
40 //jTable1.getColumnModel().getColumn(5).setCellEditor(new DateEditor());
41
42
43 jTable1.setRowHeight(30); // Zeilenhoehe auf 30 Pixel
44
45 // PreferredSize und Hintergrundfarbe der Table Header setzen
46 jTable1.getTableHeader().setPreferredSize(new Dimension(100,40));
47 jTable1.getTableHeader().setBackground(Color.RED);
48 }
49
50 // GUI initialisieren
51 private void initComponents() {
52     JScrollPane jScrollPane = new javax.swing.JScrollPane();
53     jTable1 = new javax.swing.JTable();
54
55     this.setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
56     this.setTitle("TableDemo");
57
58     jScrollPane.setPreferredSize(new java.awt.Dimension(800, 300));
59
60     jTable1.setModel(new MyTableModel());
61     jTable1.setDefaultRenderer(Color.class, new ColorRenderer(true));
62     jScrollPane.setViewportView(jTable1);
63
64     this.getContentPane().add(jScrollPane, java.awt.BorderLayout.CENTER);
65
66     this.pack();
67 }
68
69 // Startmethode main()
70 public static void main(String args[]) {
71     java.awt.EventQueue.invokeLater(new Runnable() {
72
73         public void run() {
74             new TableDemo().setVisible(true);
75         }
76     });
77 }
78
79 // Auf der Table einen TableModelListener setzen
80 private void registerListener() {
81     jTable1.getModel().addTableModelListener(new TableModelListener() {
82         @Override
83         public void tableChanged(TableModelEvent e) {
84             int row = e.getFirstRow();
85             int column = e.getColumn();
86             TableModel model = (TableModel) e.getSource();
87             //String columnName = model.getColumnName(column);
88             Object data = model.getValueAt(row, column);
89             System.out.format("Changed: [%d/%d]: %s\n", row, column, data);
90         }
91     });
92 }
93 }

```

Listing 8.37: TableDemo TableDemo.java

In der Methode `registerListener()` wird auf der Tabelle ein `TableModelListner` registriert. Dieses aus dem Paket `javax.swing.event` stammende Interface hat den folgenden Aufbau:

```

public interface TableModelListener extends EventListener {
    void tableChanged(TableModelEvent e);
}

```

Die Methode `tableChanged()` wird immer dann aufgerufen, wenn sich das Modell geändert hat. Damit kann losgekoppelt vom Modell auf eine Änderung der Daten reagiert werden (z.B. können die in der

Tabelle editierten Daten persistiert werden).

Die Eventklasse `javax.swing.event.TableModelEvent` besitzt unter anderem die folgenden Methoden:

```
public int getFirstRow()
```

Liefert den Index der ersten von der Änderung betroffenen Zeile.

```
public int getLastRow()
```

Liefert den Index der letzten von der Änderung betroffenen Zeile.

```
public int getColumn()
```

Liefert den Index der von der Änderung betroffenen Spalte.

```
public Object getSource()
```

Liefert das betroffene `TableModel`.

8.11 Dialoge

Dialoge sind eigene Fenster, in denen Steuerelemente platziert werden und die in der Regel verwendet werden, um Programmeinstellungen vornehmen zu lassen. Swing stellt für Dialoge eine eigene Klasse `javax.swing.JDialog` zur Verfügung.

Modale Dialoge

Modale Dialoge blocken alle anderen Fenster der Applikation solange, bis der Dialog vom Benutzer beendet wurde, nicht modale Dialoge laufen in einem eigenen Thread und blockieren die anderen Fenster der Applikation nicht.

Konstruktion eines Dialoges

Jedem `JDialog` ist ein `ParentFrame` oder ein `ParentDialog` zugeordnet. In den Konstruktoren der Klasse `JDialog` wird dieser Parentkomponente übernommen.

```
public JDialog()
public JDialog(Frame owner)
public JDialog(Frame owner, boolean modal)
public JDialog(Frame owner, String title)
public JDialog(Frame owner, String title, boolean modal)
public JDialog(Dialog owner)
public JDialog(Dialog owner, boolean modal)
public JDialog(Dialog owner, String title)
public JDialog(Dialog owner, String title, boolean modal)
```

Dabei bedeutet `owner` die Parentkomponente, der parameterlose Konstruktor erzeugt einen Dialog mit einem minimalen unsichtbaren Eigentümerframe. `modal` bestimmt die Modalität des Dialoges, beim Fehlen dieses Parameters wird der Dialog nicht modal geöffnet. Im Parameter `title` kann der Dialogtitel übergeben werden.

ContentPane

Einem `JDialog` ist standardmäßig eine `ContentPane` zugeordnet, die ein `BorderLayout` trägt. Weist man einem `JDialog` mit

```
dialog.add(child);
```

eine Kindkomponente zu, so wird `child` in Wirklichkeit der `ContentPane` hinzugefügt. Im Wesentlichen entspricht das Arbeiten mit einem `JDialog` dem eines `JFrame`.

8.11.1 Template für einen OK - Cancel - Dialog

Die Entwicklungsumgebung Netbeans stellt für das Arbeiten mit Dialogen mehrere Templates zur Verfügung. Das folgende Listing zeigt das (vereinfachte) Template für einen solchen OK-Cancel-Dialog. Der Dialog enthält zwei Buttons, die zum Bestätigen bzw. Abbrechen des Dialoges verwendet werden.

Listing

```

1 import java.awt.event.ActionEvent;
2 import java.awt.event.KeyEvent;
3 import javax.swing.AbstractAction;
4 import javax.swing.ActionMap;
5 import javax.swing.InputMap;
6 import javax.swing.JComponent;
7 import javax.swing.KeyStroke;
8
9 public class NewOkCancelDialog extends javax.swing.JDialog {
10
11     // Returnstatus, wenn der Dialog abgebrochen wurde
12     public static final int RET_CANCEL = 0;
13     // Returnstatus, wenn der Dialog mit OK beendet wurde
14     public static final int RET_OK = 1;
15
16     private javax.swing.JButton cancelButton;           // OK - Button
17     private javax.swing.JButton okButton;              // Chancel - Button
18     private int returnStatus = RET_CANCEL;             // Returnsatus zu Beginn auf RET_CANCEL
19
20     // Konstruktor
21     public NewOkCancelDialog(java.awt.Frame parent, boolean modal) {
22         super(parent, modal);                          // Superklassenkonstruktor
23         initComponents();                               // Initialisieren des GUI
24
25         // Der Dialog laesst sich mit ESCAPE abbrechen
26         String cancelName = "cancel";
27         InputMap inputMap = getRootPane().getInputMap(JComponent.
28             WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
29         inputMap.put(KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, 0), cancelName);
30         ActionMap actionMap = getRootPane().getActionMap();
31         actionMap.put(cancelName, new AbstractAction() {
32             @Override
33             public void actionPerformed(ActionEvent e) {
34                 doClose(RET_CANCEL);
35             }
36         });
37
38         // Liefert den Returstatus - entweder RET_OK oder RET_CANCEL */
39         public int getReturnStatus() {
40             return returnStatus;
41         }
42
43         // Initialisieren des GUI
44         private void initComponents() {
45             okButton = new javax.swing.JButton();
46             cancelButton = new javax.swing.JButton();
47
48             // WindowListener fuer windowClosing - Event
49             this.addWindowListener(new java.awt.event.WindowAdapter() {
50                 @Override
51                 public void windowClosing(java.awt.event.WindowEvent evt) {
52                     closeDialog(evt);
53                 }
54             });
55
56             // OK - Button
57             okButton.setText("OK");
58             okButton.addActionListener(new java.awt.event.ActionListener() {
59                 @Override
60                 public void actionPerformed(java.awt.event.ActionEvent evt) {
61                     okButtonActionPerformed(evt);
62                 }

```

```

63     });
64
65     // Cancel - Button
66     cancelButton.setText("Cancel");
67     cancelButton.addActionListener(new java.awt.event.ActionListener() {
68         @Override
69         public void actionPerformed(java.awt.event.ActionEvent evt) {
70             cancelButtonActionPerformed(evt);
71         }
72     });
73
74     // okButton und cancelButton dem Container hinzufuegen
75
76     // OK-Button reagiert auf ENTER
77     getRootPane().setDefaultButton(okButton);
78
79     pack();
80 }
81
82 // Handlermethode fuer den OK - Button
83 private void okButtonActionPerformed(java.awt.event.ActionEvent evt) {
84     doClose(RET_OK);
85 }
86
87 // Handlermethode fuer den Cancel - Button
88 private void cancelButtonActionPerformed(java.awt.event.ActionEvent evt) {
89     doClose(RET_CANCEL);
90 }
91
92 // Handlermethode fuer windowClosing
93 private void closeDialog(java.awt.event.WindowEvent evt) {
94     doClose(RET_CANCEL);
95 }
96
97 // Schliessen des Dialoges
98 private void doClose(int retStatus) {
99     returnStatus = retStatus;
100     setVisible(false);
101     dispose();
102 }
103 }

```

Listing 8.38: Template für OK-Cancel-Dialog NewOkCancelDialog.java

8.11.2 Beispiel

Das folgende Beispiel demonstriert, wie die Daten eines Objektes mit Hilfe eines Dialoges editiert werden können, der das oben angegebene Template verwendet.

Datenklasse

Die Datenklasse kapselt einen Integerwert (im Bereich von `I_MINIMUM` bis `I_MAXIMUM`), einen Doublewert und einen String.

Listing

```

1 package jdiallog;
2
3 public class Data {
4     private int i;                // Integerwert
5     private double d;            // Doublewert
6     private String str;          // Stringwert
7
8     public static final int I_MINIMUM = 10;    // Minimaler Wert fuer i
9     public static final int I_MAXIMUM = 20;    // Maximaler Wert fuer i
10
11     // Konstruktor
12     public Data(int i, double d, String str) {

```

```

13     this.i = i;
14     this.d = d;
15     this.str = str;
16 }
17
18 // Getter und Setter
19
20 // toString
21 @Override
22 public String toString() {
23     return "Data [i=" + i + ", d=" + d + ", str=" + str + "]";
24 }
25 }

```

Listing 8.39: Datenklasse für DialogDemo Data.java

Applikation

Die Applikation verwaltet in einem JFrame ein Objekt `data` vom Typ `Data` und zeigt seine Stringdarstellung auf einem JLabel an. Der Button öffnet einen OK - Cancel - Dialog, mit dessen Hilfe die in `data` gespeicherten Werte editiert werden können. Wird der Dialog abgebrochen, so werden die in `data` gespeicherten Daten nicht verändert.

Listing

```

1 package jdiallog;
2
3 import java.awt.BorderLayout;
4 import java.awt.Dimension;
5 import java.awt.Font;
6 import java.awt.event.ActionEvent;
7 import java.awt.event.ActionListener;
8
9 import javax.swing.JButton;
10 import javax.swing.JFrame;
11 import javax.swing.JLabel;
12
13 public class DialogDemo extends JFrame {
14     private Data data; // Das Datenobjekt
15     private JLabel lbData; // Label zum Darstellen
16     private JButton bEdit; // Button zum Aufruf des
17                             Dialoges
18
19     // Konstruktor
20     public DialogDemo() {
21         super("Dialog Demo"); //
22         data = new Data(15, 12.5, "Testdata"); // Daten initialisieren
23         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24
25
26         this.initComponents();
27     }
28
29     // Initialisieren des GUI
30     private void initComponents() {
31         // Label zum Anzeigen der Daten
32         this.lbData = new JLabel();
33         this.lbData.setHorizontalAlignment(JLabel.CENTER);
34         this.lbData.setFont(new Font("SansSerif", Font.BOLD, 16));
35         this.lbData.setPreferredSize(new Dimension(400, 100));
36         this.lbData.setText(data.toString());
37         this.add(lbData, BorderLayout.CENTER);
38
39         // Button zum Starten des Dialoges
40         this.bEdit = new JButton("Editieren");
41         this.bEdit.addActionListener(new ActionListener() {
42             @Override
43             public void actionPerformed(ActionEvent e) {

```

```

44         DialogDemo.this.editWert();
45     }
46 });
47 this.add(bEdit, BorderLayout.SOUTH);
48
49 this.pack();
50 }
51
52 // Handlermethode fuer den JButton bEdit
53 private void editWert() {
54     DataDialog dlg = new DataDialog(DialogDemo.this, data, true); // Modalen Dialog
55     // erzeugen
56     dlg.setVisible(true); // anzeigen
57     if (dlg.getReturnStatus() == DataDialog.RET_OK) { // Dialog wurde mit OK
58         // beendet
59         data = dlg.getData(); // neue Daten zuweisen
60         lbData.setText(data.toString()); // Label aktualisieren
61     }
62 }
63
64 // Startmethode
65 public static void main(String args[]) {
66     java.awt.EventQueue.invokeLater(new Runnable() {
67         @Override
68         public void run() {
69             new DialogDemo().setVisible(true);
70         }
71     });
72 }

```

Listing 8.40: Applikation DialogDemo DialogDemo.java

Applikation

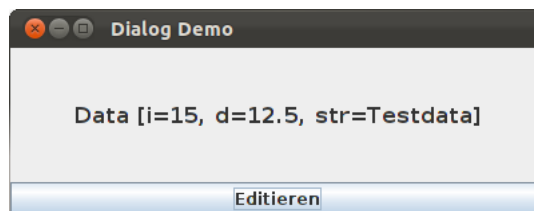


Abbildung 8.31: Applikation DialogDemo

Der Dialog DataDialog.java

Der Dialog rendert die im übergebenen Datenobjekt gekapselten Werte in einem JSpinner und zwei Textfeldern. Beendet der User den Dialog mit OK, so werden die in den Steuerelementen veränderten Daten zunächst validiert und erst bei erfolgreicher Validierung wird der Dialog mit dem Returnstatus RET_OK beendet.

Listing

```

1 package jdiallog;
2
3 import java.awt.BorderLayout;
4 import java.awt.FlowLayout;
5 import java.awt.GridLayout;
6 import java.awt.event.ActionEvent;
7 import java.awt.event.KeyEvent;
8 import javax.swing.AbstractAction;
9 import javax.swing.ActionMap;
10 import javax.swing.InputMap;
11 import javax.swing.JComponent;
12 import javax.swing.JLabel;
13 import javax.swing.JOptionPane;
14 import javax.swing.JPanel;

```

```

15 import javax.swing.JSpinner;
16 import javax.swing.JTextField;
17 import javax.swing.KeyStroke;
18 import javax.swing.SpinnerNumberModel;
19
20 public class DataDialog extends javax.swing.JDialog {
21
22     // Returnstatus, wenn der Dialog abgebrochen wurde
23     public static final int RET_CANCEL = 0;
24     // Returnstatus, wenn der Dialog mit OK beendet wurde
25     public static final int RET_OK = 1;
26
27     private javax.swing.JButton cancelButton;           // Cancel - Button
28     private javax.swing.JButton okButton;              // OK - Button
29     private JSpinner spInteger;                        // Spinner fuer Integer
30     private JTextField tfDouble;                      // Textfeld fuer Double
31     private JTextField tfString;                      // Textfeld fuer String
32     private SpinnerNumberModel m;                     // Model fuer Spinner
33     private int returnStatus = RET_CANCEL;             // Returnstatus zunaechst
34     private Data data;                                // Daten
35
36     // Konstruktor
37     public DataDialog(java.awt.Frame parent, Data data, boolean modal) {
38         super(parent, "Data Dialog", modal);          //
39         // Superklassenkonstruktor
40         this.setLocation(parent.getX() + 20, parent.getY() + 20); // Position relativ zu
41         // parent
42         this.data = data;                             // Daten uebernehmen
43         initComponents();                              // GUI initialisieren
44
45         // Dialog auf ESCAPE abbrechen
46         String cancelName = "cancel";
47         InputMap inputMap = getRootPane().getInputMap(JComponent.
48             WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
49         inputMap.put(KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, 0), cancelName);
50         ActionMap actionMap = getRootPane().getActionMap();
51         actionMap.put(cancelName, new AbstractAction() {
52             @Override
53             public void actionPerformed(ActionEvent e) {
54                 doClose(RET_CANCEL);
55             }
56         });
57     }
58
59     // Liefert den Returnstatus
60     public int getReturnStatus() {
61         return returnStatus;
62     }
63
64     // GUI initialisieren
65     private void initComponents() {
66         // OK - Button
67         this.okButton = new javax.swing.JButton();
68         this.okButton.setText("OK");
69         this.okButton.addActionListener(new java.awt.event.ActionListener() {
70             @Override
71             public void actionPerformed(java.awt.event.ActionEvent evt) {
72                 okButtonActionPerformed(evt);
73             }
74         });
75
76         // Cancel - Button
77         this.cancelButton = new javax.swing.JButton();
78         this.cancelButton.setText("Cancel");
79         this.cancelButton.addActionListener(new java.awt.event.ActionListener() {
80             @Override
81             public void actionPerformed(java.awt.event.ActionEvent evt) {
82                 cancelButtonActionPerformed(evt);
83             }
84         });
85
86         // Spinner fuer Integer

```



```

84  this.m = new SpinnerNumberModel(data.getI(), Data.I_MINIMUM, Data.I_MAXIMUM, 1);
85  this.spInteger = new JSpinner(m);
86
87  // Textfeld fuer Doublewert
88  tfDouble = new JTextField(Double.toString(data.getD()));
89
90  // Textfeld fuer Doublewert
91  tfString = new JTextField(data.getStr());
92
93  // Handlermethode fuer windowClosing
94  addWindowListener(new java.awt.event.WindowAdapter() {
95      @Override
96      public void windowClosing(java.awt.event.WindowEvent evt) {
97          closeDialog(evt);
98      }
99  });
100
101  // GUI designen - Center
102  JPanel pC = new JPanel(new GridLayout(3,2));
103  pC.add(new JLabel("Integer:"));
104  pC.add(spInteger);
105  pC.add(new JLabel("Double:"));
106  pC.add(tfDouble);
107  pC.add(new JLabel("String:"));
108  pC.add(tfString);
109  this.add(pC, BorderLayout.CENTER);
110
111  // GUI designen - South
112  JPanel pS = new JPanel(new FlowLayout(FlowLayout.RIGHT));
113  pS.add(okButton);
114  pS.add(cancelButton);
115  this.add(pS, BorderLayout.SOUTH);
116
117  // OK - Button reagiert auf ENTER
118  this.getRootPane().setDefaultButton(okButton);
119
120  pack();
121  }
122
123  // Handlermethode fuer OK - Button
124  private void okButtonActionPerformed(java.awt.event.ActionEvent evt) {
125      if(validateData()) { // Validieren der Daten
126          OK
127          data = new Data((Integer)m.getValue(), // Daten ver"andern
128                          Double.parseDouble(tfDouble.getText()),
129                          tfString.getText());
130          doClose(RET_OK); // Dialog mit RET_OK
131          schliessen
132      }
133  }
134
135  // Handlermethode fuer Cancel - Button
136  private void cancelButtonActionPerformed(java.awt.event.ActionEvent evt) {
137      doClose(RET_CANCEL); // Dialog mit RET_CANCEL
138      schliessen
139  }
140
141  // Dialog schliessen
142  private void closeDialog(java.awt.event.WindowEvent evt) {
143      doClose(RET_CANCEL);
144  }
145
146  private void doClose(int retStatus) {
147      returnStatus = retStatus;
148      setVisible(false);
149      dispose();
150  }
151
152  // Liefert die Daten
153  public Data getData() {
154      return this.data;
155  }

```

```

154 // Validieren der Daten
155 private boolean validateData() {
156     // HTML-String fuer MessageBox
157     String msg = "<html>Es sind folgende Fehler aufgetreten: ";
158     msg += "<ul>";
159     String in = tfDouble.getText(); // Text fuer Double
160     extrahieren // Kein Text eingegeben
161     if (in.isEmpty()) {
162         msg += "<li>Kein Doublewert eingegeben</li>";
163     }
164     else {
165         try {
166             double d = Double.parseDouble(in); // Versuch Text zu parsen
167         } catch (NumberFormatException e) { // Text kann nicht
168             geparkt werden
169             msg += "<li>Ihre Eingabe (Double) ist keine g&uuml;tige Zahl</li>";
170         }
171     }
172     in = tfString.getText(); // Text fuer String
173     extrahieren // Kein Text eingegeben
174     if (in.isEmpty()) {
175         msg += "<li>Kein Text eingegeben</li>";
176     } else if (!in.matches("[A-Z]?[a-z]+\\s?")) { // Text entspricht nicht
177         dem Format
178         msg += "<li>Kein g&uuml;ltiger Text</li>";
179     }
180     msg += "</ul></html>";
181     if (msg.contains("<li>")) { // msg enthaelt
182         Fehlermeldungen
183         JOptionPane.showMessageDialog(this, msg, "Fehler", JOptionPane.ERROR_MESSAGE);
184         return false;
185     }
186     return true;
187 }

```

Listing 8.41: Dialog Data DataDialog.java

Der Dialog präsentiert sich z.B. wie folgt:



Abbildung 8.32: DataDialog

Die MessageBox mit einer Fehlermeldung könnte wie folgt aussehen:

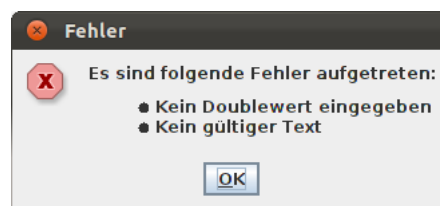


Abbildung 8.33: Fehlermeldung zum DataDialog

Abbildungsverzeichnis

3.1	Widning primitiver Typen	15
4.1	UML-Diagramm Counter	35
4.2	UML-Diagramm Vererbung	37
4.3	UML-Diagramm IsA - Beziehung	38
4.4	Casten auf Interfacetypen	49
4.5	Vererbungshierarchie Stackbeispiel	50
5.1	Vererbungshierarchie für Fehlerklassen	53
8.1	Wichtige AWT- und Swing-Klassen	84
8.2	Wichtige AWT- und Swing-Klassen	87
8.3	Flow Layout 1	95
8.4	Flow Layout 2	95
8.5	Grid Layout 1	96
8.6	Grid Layout 2	96
8.7	Border Layout 1	97
8.8	Border Layout 2	98
8.9	Verschachtelte Layouts	99
8.10	Events implementieren	103
8.11	Graphics - Zeichnen von Linien	111
8.12	Graphics - Zeichnen von Ellipsen	112
8.13	Graphics - Zeichnen von Polygonen	112
8.14	Graphics - Zeichnen von Rechtecken und Ellipsen	114
8.15	Graphics - Zeichnen von Bögen	115
8.16	Graphics - Zeichnen von Quadratischen und Kubischen Kurven	116
8.17	Graphics - Konstruktive Flächengeometrie	117
8.18	Graphics - Arbeiten mit Pfaden	119
8.19	Graphics - Verketteten von Affinitäten	123
8.20	Graphics - Zeichnen von Polygonen	124
8.21	Graphics2D - Stile für Linienenden	125
8.22	Graphics2D - Stile für Linienenden	125
8.23	Graphics - Animierter Regen	127
8.24	Graphics - Animierte analoge Uhr	128
8.25	Model-View-Controller Konzept 1	130
8.26	Model-View-Controller Konzept 2	131
8.27	Ein Model - mehrere Views	132
8.28	Selbst definiertes Table-Model	139
8.29	Eigener Zelleditor	141
8.30	Eigener TableCellRenderer	142
8.31	Applikation DialogDemo	154
8.32	DataDialog	157
8.33	Fehlermeldung zum DataDialog	157

Listings

1.1	Konsolenapplikation	2
1.2	Graphische Applikation	2
1.3	Applet	3
1.4	Servlet	3
2.1	Dokumentationskommentar	4
2.2	Aufbau einer Quelldatei	5
2.3	Verwendung eines Arrays	8
2.4	Startmethode main()	9
2.5	Kommandozeilenparameter	9
2.6	Variablentypen	10
2.7	Call by Value - primitiver Typ	10
2.8	Call by Value - Referenz	11
2.9	Varargs	11
2.10	Methodenüberladung	12
3.1	Widning und casten primitiver Typen	15
3.2	Unäre Vorzeichenoperatoren	16
3.3	Ergebnistyp int oder höher	17
3.4	Bedingungsoperator	18
3.5	Bitweises Einerkomplement	18
3.6	Bitoperatoren	19
3.7	Falsche Verwendung if-else	20
3.8	Verschachtelte Verzweigungen	20
3.9	Demonstration zu switch-case	21
3.10	Verwendung der for-Schleife	23
3.11	Erweiterte for-Schleife	24
3.12	Verwendung von break und continue	25
4.1	Leere Klasse	27
4.2	Kreis.java Einfache Klasse Kreis	27
4.3	Klasse KreisTest	29
4.4	Konstruktoren der Klasse Kreis	30
4.5	Instanzvariable	31
4.6	Klassenvariable	32
4.7	Zugriff aus statischem Kontext	33
4.8	Statische Initialisierer	33
4.9	Die Klasse Counter	35
4.10	BaseClass.java Basisklasse	36
4.11	DerivedClass.java Abgeleitete Klasse	37
4.12	MainClass.java Applikation	37
4.13	Is-A Beziehung	38
4.14	Konstruktoren und Vererbung	40
4.15	Instanz - Initialisierer	41
4.16	Function Overriding	42
4.17	Dynamisches Binden	43
4.18	Verhindern von Polymorphie	44
4.19	Abstrakte Klassen AbstractDemo.java	46
4.20	Interfacedefinition	47
4.21	Interfaces Beispiel 1	48

4.22 Casten auf Interfacetypen	49
4.23 Anwendungsbeispiel Interfaces	49
5.1 Unchecked Exception	54
5.2 Auslösen von Ausnahmen	54
5.3 Beispiel 1 zu try-catch	55
5.4 Beispiel 1 zu try-catch	56
5.5 Mehrere catch-Klauseln	56
5.6 Werfen von Ausnahmen	57
6.1 toString() aus java.lang.Object	58
6.2 Beispiel zu toString()	58
6.3 equals() aus java.lang.Object	59
6.4 Ueberschreiben von equals()	59
6.5 Immutable-Eigenschaft von Strings	62
6.6 Vergleich von Stringobjekten	62
6.7 Stringverkettung	63
6.8 Beispiel zu StringBuffer	65
6.9 Scanner zum zeilenorientiertem Lesen von System.in	65
6.10 Scanner zum Lesen von System.in 2	66
6.11 Scanner zum Lesen einer Textdatei	67
6.12 Vergleich von Wrapperobjekten	69
7.1 Statische innere Klasse	72
7.2 Nicht statische innere Klasse	73
7.3 Lokale innere Klasse	73
7.4 Anonyme innere Klasse	74
7.5 Autoboxing/Unboxing	75
7.6 Overloading und Autoboxing	76
7.7 Autoboxing/Unboxing bei Vergleichen	77
7.8 Konzept einer Enum	78
7.9 Grundfunktionalität einer Enum	79
7.10 Enum in einer switch-case	80
7.11 Einfache generische Klasse	80
7.12 Verwendung einer generischen Klasse	80
7.13 Generische Methode	81
7.14 Generische Methode mit Typ-Einschränkung	82
8.1 Grundgerüst für eine Swingapplikation	86
8.2 Liste der Systemfonts	89
8.3 Grundgerüst für Layouts	94
8.4 Flow Layout	95
8.5 Grid Layout	96
8.6 Border Layout	97
8.7 Verschachtelte Layouts	98
8.8 Eventhandler	99
8.9 Eventhandling Schritt 2	100
8.10 Eventhandler - Konkrete Implementierung 01 (Interfaces implementieren)	103
8.11 Eventhandler - Konkrete Implementierung 02 (Innere Klassen)	105
8.12 Eventhandler - Konkrete Implementierung 03 (Adapterklassen)	107
8.13 Eventhandler - Konkrete Implementierung 04 (Anonyme innere Klassen)	108
8.14 Demo zu java.awt.Graphics - Grundgeruest	109
8.15 java.awt.Graphics - Zeichnen von Linien	110
8.16 java.awt.Graphics - Zeichnen von Ellipsen	111
8.17 java.awt.Graphics - Zeichnen von Polygonen	112
8.18 java.awt.Graphics2D - Zeichnen von Rechtecken und Ellipsen	114
8.19 java.awt.Graphics2D - Zeichnen von Bögen	114
8.20 java.awt.Graphics2D - Zeichnen von Quadratischen und Kubischen Kurven	115
8.21 Konstruktive Flächengeometrie	116
8.22 Arbeiten mit Pfaden	118
8.23 Verkettung von Affinitäten	122
8.24 java.awt.Graphics2D - Zeichnen von Text mit Antialiasing	124

8.25 Animation 1: Regen	126
8.26 Animation 2: Analoge Uhr	127
8.27 Ein Model - mehrere Views JScrollbarMain.java	130
8.28 Selbst definiertes ListModel SortetIntListModel.java	134
8.29 Selbst definierter ListCellRenderer CustomListrendererer.java	136
8.30 Selbst definierter TableModel CustomTableModel.java	138
8.31 Eigener Zelleditor CustomTableModel_1.java	140
8.32 Datenklasse Data.java	142
8.33 MyTableModel MyTableModel.java	143
8.34 ColorRenderer ColorRenderer.java	145
8.35 ColorEditor ColorEditor.java	146
8.36 DateEditor DateEditor.java	147
8.37 TableDemo TableDemo.java	148
8.38 Template für OK-Cancel-Dialog NewOkCancelDialog.java	151
8.39 Datenklasse für DialogDemo Data.java	152
8.40 Applikation DialogDemo DialogDemo.java	153
8.41 Dialog Data DataDialog.java	154