

# I

# nhaltsverzeichnis

<b>1</b>	<b>EINFÜHRUNG .....</b>	<b>7</b>
1.1	OBJEKTORIENTIERUNG.....	7
1.1.1	Klassifizierung.....	7
1.1.2	Klassen .....	8
1.1.3	Objekte .....	9
1.1.4	Attribute.....	9
1.1.5	Botschaften.....	10
1.1.6	Vererbung.....	11
1.2	PROGRAMM.....	11
1.2.1	Sequenz.....	12
1.2.2	Selektion .....	13
1.2.3	Iteration.....	13
1.2.4	Abstraktion .....	14
1.2.5	Benutzerinteraktion .....	15
1.3	JAVA .....	16
1.3.1	Klassenbibliothek .....	16
1.3.2	Portabilität .....	17
1.3.3	Internet .....	20
1.4	ZUSAMMENFASSUNG .....	20
<b>2</b>	<b>KLASSEN, OBJEKTE UND BOTSCHAFTEN .....</b>	<b>22</b>
2.1	PROGRAMMAUFBAU, ATTRIBUTE UND METHODEN .....	22
2.1.1	Zum Programm .....	23
2.1.1.1	Anweisungen, import statement, geschweifte Klammern .....	23
2.1.1.2	TextField, Label, Button und Frame .....	25
2.1.1.3	Attribute und Methoden .....	27
2.1.1.4	Event Handling.....	32
2.1.1.5	String Concatenation .....	32
2.1.1.6	Die main() Methode .....	33
2.1.2	Syntax.....	34
2.1.2.1	Identifizier .....	34
2.1.2.2	Schlüsselwörter .....	35
2.1.2.3	Anweisung.....	35
2.1.2.4	import statement .....	35
2.1.2.5	Klassendeklaration .....	36
2.1.2.6	Application.....	36
2.2	CHECKBOX.....	37
2.2.1	Zum Programm .....	38
2.2.1.1	Checkbox, radio buttons und CheckboxGroup.....	39

2.3	VERERBUNG, KONSTRUKTOR UND REDEFINITION .....	40
2.3.1	<i>Zum Programm</i> .....	42
2.3.1.1	Vererbung .....	43
2.3.1.2	Konstruktor .....	46
2.3.1.3	Graphics und Fonts .....	48
2.3.1.4	paint() und Redefinition .....	49
2.3.2	<i>Syntax</i> .....	50
2.3.2.1	Vererbung .....	50
2.3.2.2	Konstruktor .....	50
2.4	INSTANZVARIABLEN UND KLASSENVARIABLEN .....	51
2.4.1	<i>Zum Programm</i> .....	53
2.4.1.1	Selektion .....	53
2.4.1.2	Instanzvariablen und Klassenvariablen .....	54
2.5	ZUSAMMENFASSUNG .....	56
<b>3</b>	<b>EINFACHE DATENTYPEN, PROGRAMMSTRUKTUREN .....</b>	<b>58</b>
3.1	INT, BOOLEAN UND PROGRAMMSTRUKTUREN .....	58
3.1.1	<i>Zum Programm</i> .....	60
3.1.1.1	Kommentar .....	63
3.1.1.2	Datentypen .....	64
3.1.1.3	Der Datentyp int .....	64
3.1.1.4	Arithmetische Operatoren und Zuweisung .....	65
3.1.1.5	Methoden .....	66
3.1.1.6	Der Datentyp boolean .....	66
3.1.1.7	Vergleichsoperatoren .....	67
3.1.1.8	for Schleife .....	68
3.1.1.9	Strings .....	72
3.1.1.10	if statement .....	72
3.1.1.11	Logische Operatoren und arithmetischer Inkrement-/Dekrement-Operator .....	76
3.1.1.12	switch statement .....	80
3.1.2	<i>Syntax</i> .....	83
3.1.2.1	Kommentar .....	83
3.1.2.2	Variablendeklaration .....	83
3.1.2.3	Gültigkeitsbereich einer Variablen .....	84
3.1.2.4	Instanzvariablen und Klassenvariablen .....	84
3.1.2.5	Lokale Variablen .....	85
3.1.2.6	Zuweisung .....	86
3.1.2.7	Methodendeklaration .....	86
3.1.2.8	Methodenaufruf .....	88
3.1.2.9	Der Datentyp int .....	89
3.1.2.10	Arithmetische Operatoren .....	89
3.1.2.11	Der Datentyp boolean .....	90
3.1.2.12	Vergleichsoperatoren .....	90
3.1.2.13	Logische Operatoren .....	91
3.1.2.14	Schleifen .....	98
3.1.2.15	while Schleife .....	99
3.1.2.16	do Schleife .....	100
3.1.2.17	for Schleife .....	101
3.1.2.18	if statement .....	103
3.1.2.19	switch statement .....	104
3.2	UNICODE UND CHAR .....	105
3.2.1	<i>Zum Programm</i> .....	106
3.2.1.1	TextArea .....	106
3.2.1.2	Konstanten .....	108
3.2.1.3	Der Datentyp char .....	108
3.2.2	<i>Syntax</i> .....	111
3.2.2.1	Konstanten .....	111
3.2.2.2	Der Datentyp char .....	111

3.3	ZUSAMMENFASSUNG .....	112
<b>4</b>	<b>KOMPLEXE DATENTYPEN.....</b>	<b>114</b>
4.1	STRINGS .....	114
4.1.1	Die Klasse String.....	114
4.1.1.1	Methoden.....	114
4.1.1.2	Erzeugung .....	115
4.1.1.3	Eigenschaften .....	115
4.1.1.4	String Concatenation .....	116
4.2	EINFACHE VERSUS KOMPLEXE DATENTYPEN .....	116
4.2.1	Wertsemantik versus Referenzsemantik.....	116
4.2.1.1	Einfache Datentypen .....	116
4.2.1.2	Komplexe Datentypen.....	117
4.2.2	„Pass by Value“ versus „Pass by Reference“ .....	118
4.2.2.1	Einfache Datentypen .....	118
4.2.2.2	Komplexe Datentypen.....	119
4.2.3	Prüfen auf Gleichheit .....	119
4.3	TYPE CASTING .....	120
4.3.1	Einfache Datentypen .....	120
4.3.1.1	Integer-Datentypen.....	121
4.3.1.2	Floating-Point-Datentypen .....	121
4.3.1.3	Datentyp boolean.....	122
4.3.2	Komplexe Datentypen .....	122
4.3.2.1	Klassen .....	122
4.3.2.2	Interfaces .....	122
4.3.3	Implizite Konvertierung.....	123
4.3.4	Syntax .....	123
4.3.4.1	Type Cast .....	123
4.4	ZUSAMMENFASSUNG .....	124
<b>5</b>	<b>EVENT HANDLING.....</b>	<b>125</b>
5.1	ACTIONEVENT UND ACTIONLISTENER.....	125
5.1.1	Zum Programm .....	127
5.1.1.1	Event Handling.....	136
5.1.2	Syntax .....	142
5.1.2.1	Event Handling.....	142
5.2	ZUSAMMENFASSUNG .....	145
<b>6</b>	<b>INSTANZ-/KLASSENMETHODEN.....</b>	<b>147</b>
6.1	INSTANZMETHODE VERSUS KLASSENMETHODE .....	147
6.1.1	Zum Programm .....	148
6.1.1.1	List .....	149
6.1.1.2	Der instanceof-Operator .....	150
6.1.1.3	Instanzmethoden und Klassenmethoden.....	151
6.1.1.4	Satzgenerator.....	152
6.1.2	Syntax .....	154
6.1.2.1	Der instanceof-Operator .....	154
6.1.2.2	Instanzmethoden und Klassenmethoden.....	154
6.1.2.3	Methodenaufruf.....	155
6.1.2.4	Attributzugriff.....	156
6.2	VISIBILITY MODIFIERS .....	156
6.2.1	Zum Programm .....	159
6.2.1.1	List und ItemListener .....	159
6.2.1.2	Menu .....	159
6.2.1.3	Sichtbarkeitsattribuierung und Information Hiding.....	161
6.2.2	Syntax .....	162

6.2.2.1	Visibility Modifiers .....	162
6.3	ZUSAMMENFASSUNG .....	163
<b>7</b>	<b>EXCEPTION HANDLING .....</b>	<b>164</b>
7.1	CALENDAR .....	164
7.1.1	Zum Programm .....	166
7.1.1.1	Zuweisung mit Operation .....	167
7.1.1.2	Information Hiding .....	168
7.1.1.3	Conditional Operator .....	168
7.1.1.4	Method Overloading .....	169
7.1.1.5	Calendar .....	169
7.1.2	Syntax .....	170
7.1.2.1	Zuweisung mit Operation .....	170
7.1.2.2	Conditional Operator .....	171
7.2	GRAPHICS .....	171
7.2.1	Zum Programm .....	173
7.2.1.1	drawOval() und drawLine() .....	174
7.3	ARRAYS .....	178
7.3.1	Zum Programm .....	181
7.3.1.1	Arrays .....	182
7.3.2	Syntax .....	184
7.3.2.1	Arrays .....	184
7.4	EXCEPTION HANDLING .....	186
7.4.1	Zum Programm .....	187
7.4.1.1	String Tokenizer .....	187
7.4.1.2	Exception Handling .....	188
7.4.2	Syntax .....	195
7.4.2.1	Exception Handling .....	195
7.5	ZUSAMMENFASSUNG .....	197
<b>8</b>	<b>ABSTRAKTE KLASSEN .....</b>	<b>199</b>
8.1	ABSTRAKTE METHODEN UND KLASSEN .....	199
8.1.1	Zum Programm .....	202
8.1.1.1	Choice .....	205
8.1.1.2	Abstrakte Methoden .....	206
8.1.1.3	Abstrakte Klassen .....	209
8.1.2	Syntax .....	211
8.1.2.1	Abstrakte Methoden .....	211
8.1.2.2	Abstrakte Klassen .....	211
8.2	ZUSAMMENFASSUNG .....	212
<b>9</b>	<b>INTERFACES .....</b>	<b>213</b>
9.1	INTERFACES UND ADAPTER-KLASSEN .....	213
9.1.1	Zum Programm .....	215
9.1.1.1	Interfaces .....	216
9.1.1.2	Adapter-Klassen .....	217
9.1.2	Syntax .....	219
9.1.2.1	Interfaces .....	219
9.1.2.2	Adapter-Klassen .....	220
9.2	ZUSAMMENFASSUNG .....	221
<b>10</b>	<b>MEHRDIMENSIONALE ARRAYS .....</b>	<b>223</b>
10.1	ZWEIDIMENSIONALER ARRAY .....	223
10.1.1	Zum Programm .....	225
<b>11</b>	<b>LOGO .....</b>	<b>228</b>

11.1	TURTLE-GEOMETRIE.....	228
11.1.1	<i>Zum Programm</i> .....	229
11.2	VERERBUNG.....	231
11.2.1	<i>Zum Programm</i> .....	232
11.3	REKURSION.....	233
11.3.1	<i>Zum Programm</i> .....	235
11.4	STACK.....	238
11.4.1	<i>Zum Programm</i> .....	242
<b>12</b>	<b>DATENSTRUKTUREN.....</b>	<b>246</b>
12.1	BINÄRER BAUM .....	246
12.1.1	<i>Zum Programm</i> .....	248
<b>13</b>	<b>ANHÄNGE.....</b>	<b>251</b>
13.1	ANHANG A: JAVA SYNTAX.....	251
13.1.1	<i>Schlüsselwörter</i> .....	251
13.1.2	<i>Einfache Datentypen</i> .....	252
13.1.3	<i>Operatoren</i> .....	252
13.2	ANHANG B: ÜBERSICHT KLASSENIBLIOTHEK.....	254
13.3	ANHANG C: KLASSENDIAGRAMME.....	258
13.4	ANHANG D: PRINZIPIEN GUTEN PROGRAMMIERENS .....	259
13.5	ANHANG E: ÜBERSICHT PROGRAMMIERSPRACHEN .....	266
<b>14</b>	<b>STICHWORTVERZEICHNIS .....</b>	<b>267</b>
14.1	STICHWORTVERZEICHNIS .....	267

**D**as vorliegende Skript ist im Rahmen der Vorlesungsveranstaltung „Einführung in die Programmierung“ an der Universität Zürich entstanden. Es richtet sich an programmierunerfahrene Studierende der Fachrichtung Wirtschaftsinformatik bzw. Informatik und bietet einen sanften Einstieg in die objektorientierte Programmierung mit Java. Von allem Anfang an steht die Objektorientierung und der Einsatz graphischer Benutzeroberflächen im Zentrum.

Philosophie des Skriptes ist es, die wesentlichsten Konzepte einer objektorientierten Programmiersprache - im besonderen von Java - anschaulich zu vermitteln. Aus diesem Grund werden Beispielprogramme eingesetzt, anhand deren die einzuführenden Konzepte sukzessive erläutert werden. Hierbei sind sämtliche Beispielprogramme Applications und basieren auf dem Java 1.1 Standard. Als Programmierungsumgebung wurde CodeWarrior von Metrowerks verwendet. Die Quellprogramme der Beispiele sind unter der Adresse [http://www.ifi.unizh.ch/study/course\\_material/eprog](http://www.ifi.unizh.ch/study/course_material/eprog) zu finden.

Das Skript deckt jedoch nicht die gesamte Palette von Java ab. Insbesondere werden innere Klassen und Threads nicht behandelt.

Zum Aufbau des Skriptes lässt sich folgendes erwähnen:

- ◆ Die einzelnen Kapitel sind im wesentlichen sequentiell zu lesen, da sie aufeinander aufbauen.
- ◆ Fast jedes Kapitel basiert auf einem Programmbeispiel, welches teilweise in mehreren erweiterten Versionen vorliegt. Zwecks einer einfacheren Präsentierbarkeit bilden alle Klassen eine einzige Datei. Der Abschnitt „Zum Programm“ erläutert jeweils das Beispielprogramm und führt die neu zu behandelnden Konzepte ein. Syntaxspezifische Fragen werden dann im Abschnitt „Syntax“ nochmals aufgegriffen und eingehender beleuchtet.
- ◆ Es wurden grundsätzlich zwei Schriftarten verwendet. Der Typ „Courier New“ kennzeichnet Programme, Programmauszüge, Elemente aus der Java Klassenbibliothek oder Java-Schlüsselwörter. Ansonsten wird die Schriftart „Times New Roman“ eingesetzt. Findet man diese *kursiv* vor, dann dokumentiert sie weiterführende Erläuterungen oder aufmunternde Intermezzi.
- ◆ Am Ende des Skriptes finden sich diverse Anhänge und auch ein Stichwortverzeichnis.

Da ein Lehrgang trotz redlicher Bemühungen nie perfekt sein kann, werden Anregungen, Korrekturen und konstruktive Kritik gerne unter der E-Mail-Adresse [schauer@ifi.unizh.ch](mailto:schauer@ifi.unizh.ch) entgegengenommen.

Zürich, im November 1998

# 1 Objektorientierung Programm Java

In diesem Kapitel werden Grundlagen vermittelt. Zuerst wird an zahlreichen, anschaulichen Beispielen das Konzept der Objektorientierung vorgestellt. Dann soll vermittelt werden, was ein Programm überhaupt ist und schlussendlich gehören der Programmiersprache Java noch ein paar Worte.

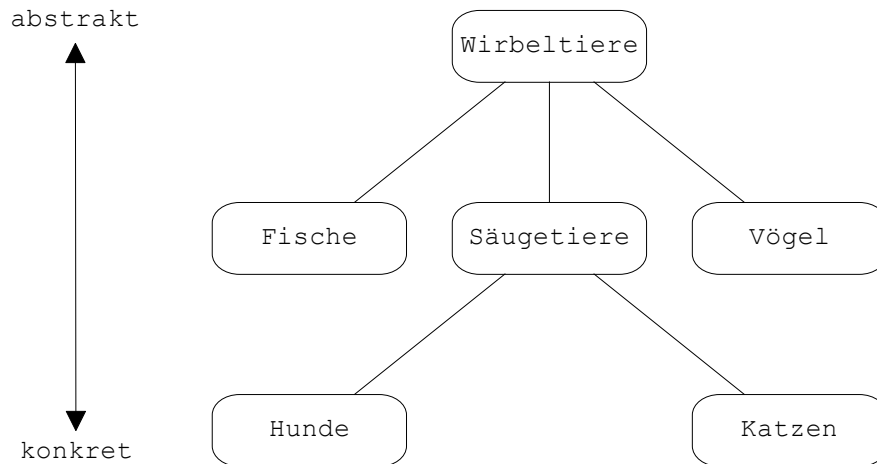
## 1.1 Objektorientierung

Die Programmiersprache **Java** ist unweigerlich mit dem Begriff der **Objektorientierung** verbunden: Java ist eine **objektorientierte Programmiersprache**. Hinter diesem scheinbar mystisch klingenden Wort steht aber nicht viel mehr als ein Konzept, das uns allen bereits aus dem Alltag vertraut ist.

### 1.1.1 Klassifizierung

In der Biologie ist die **Klassifikation** der Tierarten oder auch der Pflanzenarten eine bekannte Vorgehensweise. Abbildung 1-1 zeigt einen Ausschnitt eines Klassifikationsbaumes aus dem Tierreich.

Wie aus der Abbildung zu ersehen ist, unterscheidet man mehrere Gruppen von Tieren voneinander. So unterteilt man beispielsweise die Wirbeltiere in die Fische, Vögel und Säugetiere. Diese Trennung erfolgt aufgrund unterschiedlicher **Kriterien** wie zum Beispiel Regelung der Körpertemperatur, Art der Atmung, etc.



**Abbildung 1-1: Ausschnitt aus dem Tierreich**

Nun kann man aber auch in der Gruppe der Säugetiere Kriterien ausmachen, welche die Säugetiere ihrerseits in Untergruppen zerlegen; man gelangt zu den Hunden und Katzen. Durch den Vorgang der Klassifizierung entsteht ein **hierarchisches System**.

Entscheidend in einem solchen hierarchischen System ist die Tatsache, dass eine Untergruppe sämtlichen Kriterien ihrer Obergruppen genügt und somit **ein Teil deren** ist. Ein Hund ist also ein Säugetier. Als Säugetier gehört er zu den Wirbeltieren. Die Umkehrung obiger Aussage gilt jedoch nicht: ein Säugetier ist nicht zwingend ein Hund.

Da man mit fortschreitendem Herabsteigen im Klassifikationsbaum immer mehr Kriterien erfüllt, nehmen die Details und damit der Grad der **Konkretisierung** zu. Im Gegensatz dazu ist man auf der obersten Hierarchieebene äusserst **abstrakt**.

### 1.1.2 Klassen

In der Biologie bezeichnet man eine Gruppe in Abhängigkeit von der Hierarchiestufe, auf welcher sie sich im Klassifikationsbaum befindet; man stösst dort auf die Begriffe Reich, Stamm, Klasse, Ordnung, Familie, Gattung und Art.

In der objektorientierten Programmierung hingegen verwendet man für eine „Gruppe“ den Begriff **Klasse** (engl. **class**). Damit man die hierarchische Beziehung verschiedener Klassen untereinander ausdrücken kann, verwendet man die Begriffe **Oberklasse** (engl. **superclass**) und **Unterklasse** (engl. **subclass**). Hierbei wäre auf das Tierbeispiel bezogen z.B. „Säugetiere“ eine Oberklasse der „Katzen“ aber eine Unterklasse der „Wirbeltiere“.



### 1.1.3 Objekte

Wo ist nun aber in Abbildung 1-1 Bello, der Hund der Nachbarin zu finden?

Der in Abbildung 1-1 ersichtliche Klassifikationsbaum bestimmt sämtliche Kriterien, denen beispielsweise Bello genügen muss, damit wir ihn als Hund betrachten. Er beschreibt somit lediglich den **Typ** eines real existierenden Lebewesens, also das, was beispielsweise allen real existierenden Hunden gemeinsam ist, ohne dabei eine Aussage über irgendeinen bestimmten Hund, zum Beispiel Bello, zu machen.

Bello ist wohl ein Hund wie viele andere Hunde auch. Vor allem ist er aber ein real existierendes Lebewesen, ein **Individuum**, das es nur einmal gibt. Er unterscheidet sich beispielsweise vom rassengleichen Fido dadurch, dass sein Fell viel dunkler und glatter ist. Das macht ihn einzigartig. Da Bello also ein Individuum ist, findet man ihn in Abbildung 1-1 nicht.

Wofür wir weiter oben umgangssprachlich die Begriffe „Typ“ und „Individuum“ verwendet haben, stehen in der objektorientierten Programmierung die Begriffe **Klasse** (engl. **class**) und **Objekt** (engl. **object**).

Der Begriff **Instanz** (engl. **instance**) steht synonym für den Begriff Objekt. Dieser Ausdruck rührt von der Idee her, dass ein Objekt eine Instanz einer Klasse ist.

### 1.1.4 Attribute

In der objektorientierten Welt kann man mittels sogenannter **Attribute** (engl. **attributes**) die Eigenschaften einer Klasse definieren. Die Klasse Hund kann beispielsweise die Attribute Rasse, Geschlecht, Alter und Gewicht haben. Unser Bello hätte dann die Werte: Labrador, männlich, 4 Jahre und 26 kg.

An obigem Beispiel wird ein weiterer Unterschied zwischen einer Klasse und deren Instanzen deutlich: die Klasse definiert lediglich die Attribute, ohne Werte vorzugeben. Erst die Objekte der Klasse weisen dann für jedes Attribut einen Wert auf, der für jedes Objekt individuell sein kann.

Die Attributwerte eines Objektes können sich im Verlaufe der Zeit ändern; in einem Jahr ist Bello nämlich 5 Jahre alt und hat vielleicht auch ein Kilogramm zugenommen. So beschreiben die momentanen Attributwerte eines Objektes dessen **Zustand zu diesem Zeitpunkt**. Sobald sich aber die Attributwerte einer Instanz ändern, erfolgt ein Zustandswechsel.

### 1.1.5 Botschaften

*Bello ist bei Lisa Gut zu Hause. Er ist ihr ein treuer und aufmerksamer Begleiter. Lisa hat ihm eine Reihe von Anweisungen beigebracht, die er befolgt. So muss Bello beispielsweise dem Befehl „Sitz!“ gehorchen oder er muss auch aufgrund der Anweisung „Bring die Pantoffeln!“ Lisas Pantoffeln bringen.*

Diese Befehle oder Anweisungen, welche Lisa Bello gibt, werden in der objektorientierten Programmierung als **Botschaften** (engl. **messages**) bezeichnet. Dort kann man den Objekten Botschaften schicken. Wenn nun Lisa zu Bello sagt: „Sitz!“, dann drückt man dies in der objektorientierten Programmierung folgendermassen aus:

```
Bello.sitz();
```

Hierbei steht in obiger Notationsform vor dem Punkt das empfangende Objekt, also Bello, und nach dem Punkt die Botschaft.

Dass Bello die Pantoffeln bringen soll, wird dann auf folgende Weise formuliert:

```
Bello.bring(Pantoffeln);
```

Im Unterschied zur ersten Botschaft steht bei dieser in den runden Klammern nun etwas, nämlich die Pantoffeln. Wenn nun Lisa mit Bello spielt und ihm dabei einen Ball wirft, kann sie ihm die Botschaft

```
Bello.bring(Ball);
```

schicken.

Wie wir sehen, kann also das, was in den runden Klammern einer Botschaft steht, variieren. Man bezeichnet diesen Wert, welcher als Teil einer Botschaft versandt wird, als **Parameter**. Eine Botschaft muss aber nicht zwingenderweise Parameter haben. In einem solchen Fall werden die runden Klammern wie bei der Botschaft `sitz()` einfach leer gelassen.

Lisa gibt Bello immer sein Lieblingsfutter zu essen.

```
Bello.friss(Bello.Lieblingsfutter);
```

Als Parameter wird der Wert von Bellos Attribut `Lieblingsfutter` übergeben. In der objektorientierten Programmierung kann man auf die Attributwerte eines Objektes zugreifen, indem man dem Attributnamen, durch einen Punkt getrennt, den Objektnamen voranstellt.

Wenn nun Bello aufgrund des vielen guten Futters zunimmt, verändert sich der Wert seines Attributs `Gewicht`. Botschaften können somit einen Zustandswechsel bei der empfangenden Instanz auslösen. Sie können den Empfänger aber auch zum Aussenden einer weiteren Botschaft auffordern oder diesen lediglich informieren.

Auf Programmebene bewirkt eine Botschaft den **Aufruf einer Methode** (siehe auch Abschnitt 1.2.4). Methoden sind, wie auch die Attribute, in den Klassen deklariert.

### 1.1.6 Vererbung

Durch die Verwendung von **Oberklassen** (engl. **superclass**) und **Unterklassen** (engl. **subclass**) kann man in der objektorientierten Programmierung **hierarchische** Beziehungen der Klassen bzw. Objekte untereinander ausdrücken. Was bedeutet das nun aber, wenn eine Klasse Oberklasse einer anderen Klasse ist?

Wie wir bereits wissen, sind in einer Klasse all ihrer Attribute und Methoden deklariert. Hat eine Klasse nun eine Unterklasse, so sind all ihre Attribute und Methoden automatisch, d.h. ohne dass sie dort explizit aufgeführt werden, in der Unterklasse vorhanden. Die Oberklasse **vererbt** sie gewissermassen an ihre Unterklassen.

In Abschnitt 1.1.4 haben wir für die Klasse Hund folgende Attribute definiert: Rasse, Geschlecht, Alter und Gewicht. Wenn wir nun die Klasse Säugetier als Oberklasse wählen, dann können wir doch die Attribute Geschlecht, Alter und Gewicht bereits für die Klasse Säugetier festlegen, da jedes Säugetier, ob Elefant oder Hund, für diese Attribute Werte aufweist. In der Unterklasse Hund müssen wir dann einzig das für Hunde spezifische Attribut Rasse deklarieren. Die Attribute Geschlecht, Alter und Gewicht erbt die Unterklasse Hund von ihrer Oberklasse Säugetier.

Die **Vererbung** (engl. **inheritance**) ist eine fundamentales Prinzip in der objektorientierten Programmierung.

## 1.2 Programm

*Mutter spricht des Morgens zu Rotkäppchen:*

*„Rotkäppchen, heute hat Deine Grossmutter Geburtstag. Back ihr doch ihren Lieblingskuchen, nimm eine Flasche vom alten guten Wein aus dem Keller, leg alles in einen Korb und geh sie besuchen. Aber gib Acht, dass Du Dich nicht versäumst, und ja nicht vom Weg abkommst. Wenn Grossmutters Haustür offen ist, dann klopfst Du einmal und trittst danach artig ein, wenn nicht, musst Du ihr halt so lange rufen, bis sie Dir aufschliessen kommt.“*

In einem objektorientierten Programm würden die ersten paar Anweisungen, welche Mutter Rotkäppchen gibt, etwa folgendermassen aussehen:

```
Rotkäppchen.backKuchen(Grossmutter.Lieblingskuchen);  
Rotkäppchen.hole(Weinflasche);  
Rotkäppchen.legIn(Korb, Kuchen);  
Rotkäppchen.legIn(Korb, Weinflasche);  
Rotkäppchen.besuche(Grossmutter);
```

So wie im Beispiel von Rotkäppchen Mutters Anweisungen einen Text bilden, ist auch ein **objektorientiertes Programm** nichts anderes als ein Text. Ein solcher Text besteht aus der Deklaration der Klassen mitsamt deren Attributen und Methoden und schlussendlich den Objekten selber, die dann untereinander mittels Botschaften (Methodenaufruf) kommunizieren.

Da es sich selbst bei den Botschaften um eigentliche Arbeitsanweisungen handelt und ein objektorientiertes Programm schlussendlich auf einem Computer ausgeführt wird, ist ein **Programm** ein Text, der aus einer Menge von **Anweisungen** (engl. **statement**) an den Computer besteht.

Eine Sprache, in der ein Programm geschrieben ist, eine Sprache also, die eine formale, textuelle Beschreibung eines Arbeitsablaufes oder Problemlösungsverfahrens erlaubt, nennt man **Programmiersprache**. Sie muss sowohl für den Computer als auch für den Menschen verständlich sein.

In Analogie zu den verschiedenen natürlichen Sprachen, wie Deutsch, Englisch oder Französisch, gibt es auch unterschiedliche Programmiersprachen. Neben **Java** seien an dieser Stelle auch die Programmiersprachen **C++**, **C**, **Pascal** Programmiersprache und **Logo** Programmiersprache erwähnt<sup>1</sup>. Dies ist aber nur ein kleiner Ausschnitt aus allen vorhandenen Programmiersprachen. Die Vielfalt an Programmiersprachen ist aber nicht wie bei den natürlichen Sprachen auf geographische Ursachen zurückzuführen, sondern ein Zeugnis der Entwicklungsgeschichte der Informatik.

### 1.2.1 Sequenz

Mutter sagt zu Rotkäppchen: *“Back ihr doch ihren Lieblingskuchen, nimm eine Flasche vom alten guten Wein aus dem Keller, leg alles in einen Korb und geh sie besuchen.”* In dieser Aussage ist eine klare Reihenfolge der einzelnen Tätigkeiten enthalten. Die einzelnen Aktionen erfolgen **sequentiell**, eine nach der anderen.

---

<sup>1</sup> In Anhang E findet sich ein Stammbaum der Programmiersprachen.

```
Rotkäppchen.backKuchen(Grossmutter.Lieblingskuchen);  
Rotkäppchen.hole(Weinflasche);  
Rotkäppchen.legIn(Korb, Kuchen);  
Rotkäppchen.legIn(Korb, Weinflasche);  
Rotkäppchen.besuche(Grossmutter);
```

Die Sequenz von Arbeitsanweisungen kann auch in einem Programm wiedergegeben werden. Die obigen Methodenaufrufe erfolgen in der Reihenfolge ihres Auftretens, welche sich ergibt, wenn man das Programm von links nach rechts und von oben nach unten liest.

Ein Computer führt die Anweisungen in einem Programm also sequentiell **von links nach rechts** und **von oben nach unten** aus.

### 1.2.2 Selektion

Es kann auch vorkommen, dass nicht alle Anweisungen in einem Programm auch wirklich ausgeführt werden, sondern dass man die Auswahl zwischen zwei oder mehreren alternativen Aktionen hat. Eine solche **Selektion** ist meist mit einer Bedingung verknüpft.

Mutter weist Rotkäppchen an: *„Wenn Grossmutters Haustür offen ist, dann klopfst Du einmal und trittst danach artig ein, wenn nicht, musst Du ihr halt so lange rufen, bis sie Dir aufschliessen kommt.“*

```
falls (Haustür.istOffen())  
    Rotkäppchen.klopfeUndTreteEin();  
andernfalls Rotkäppchen.rufe(Grossmutter);
```

Wenn die in Klammern stehende **Bedingung** `Haustür.istOffen()` wahr ist, die Haustür also offen ist, erfolgt der Methodenaufwurf `klopfeUndTreteEin()`. Falls die Bedingung aber falsch ist, wird die Botschaft `rufe(Grossmutter)` versandt.

Eine mit einer Bedingung kombinierte Selektion wird als **bedingte Anweisung** (engl. **conditional statement**) bezeichnet.

### 1.2.3 Iteration

Mutter erklärt Rotkäppchen auch: *„...musst Du ihr halt so lange rufen, bis sie Dir aufschliessen kommt.“*

```
solange (Türe.istGeschlossen())  
    Rotkäppchen.rufe(Grossmutter);
```

Die Methode `Rotkäppchen.rufe(Grossmutter)` wird mehrmals ausgeführt. Sie wird nämlich so lange erneut aufgerufen, bis die Bedingung `Türe.istGeschlossen()` nicht mehr wahr ist, weil die Grossmutter die Türe aufmachen kommt.

Das mehrfache, unmittelbar aufeinanderfolgende Ausführen von Anweisungen wird als **Iteration** bezeichnet. Eine Iteration kann theoretisch endlos fortlaufen, was als sogenannte Endlosschleife bekannt ist und das explizite Anhalten des Computers durch den Benutzer erfordert.

In einem Programm ist eine Iteration immer mit einer **Bedingung** verbunden. Solange die Bedingung wahr ist, wird die Anweisung wiederholt.

### 1.2.4 Abstraktion

Manchmal ist es zweckmässig, mehrere Anweisungen, die gemeinsam einen ganzen Arbeitsablauf ausmachen, zusammenzufassen und mit einem Namen zu bezeichnen.

Dies geschah beispielsweise, als Mutter zu Rotkäppchen sagte: *“Back ihr doch ihren Lieblingsskuchen.”* Das Backen von Grossmutter's Lieblingsskuchen umfasst mehrere Teilanweisungen. Die Sequenz *“Back ihr doch ihren Lieblingsskuchen, nimm eine Flasche vom alten guten Wein aus dem Keller, leg alles in einen Korb und geh sie besuchen.”* würde aber enorm wachsen und komplexer werden, wenn die Stelle „back ihr doch ihren Lieblingsskuchen“ durch sämtliche Aktionen des Kuchenbackens ersetzt würde: *„150 g weiche Butter rühren, bis sich Spitzchen bilden, 1 Ei, 150 g Zucker und 1 Prise Salz zugeben und rühren bis die Masse hell ist, eine  $\frac{1}{2}$  geriebene Zitronenschale, ein  $\frac{1}{2}$  KL Zimt, eine Msp Nelkenpulver, 200 g gemahlene Nüsse beifügen und mischen, 200 g Mehl dazusieben, verrühren und  $\frac{2}{3}$  des Teiges auf dem Boden der Form ausstreichen, am Rand etwas erhöht, 200 g Johannisbeer- oder Himbeerkonfitüre auf den Teig verteilen, 3-4 EL Mehl dem restlichen Teig beifügen, auswallen, schmale Streifen schneiden oder Figuren ausstechen, Teigstreifen gitterartig auf die Füllung legen oder Figuren auf die Füllung geben, ein Ei verklopfen und damit die Torte bestreichen, die Torte in der unteren Ofenhälfte bei 180° C 30 bis 35 min backen, nimm eine Flasche vom alten guten Wein aus dem Keller, leg alles in einen Korb und geh sie besuchen.“*

Für uns Menschen wird es viel einfacher, wenn wir Tätigkeiten in eigenständige, in sich geschlossene Arbeitsabläufe zerlegen. Wir sehen von den Details ab, um den Überblick zu wahren. Dieses Vorgehen ist unter dem Begriff der **Abstraktion** bekannt.

**Methoden** sind in Java ein Mittel zur Abstraktion. Unter ihrem Namen vereinen sie mehrere Anweisungen zu einem in sich geschlossenen Ganzen. So wie im obigen Beispiel durch die Anweisung *“Back ihr doch ihren Lieblingsskuchen”* ein ganzer Arbeitsablauf, nämlich der des Kuchenbackens, ausgeführt wird, bewirkt im Programm der Methodenaufruf

```
Rotkäppchen.backKuchen(Grossmutter.Lieblingsskuchen);
```

dass sämtliche Anweisungen, die der Methode zugeordnet sind, realisiert werden. Man hat also die Möglichkeit, durch ein simples Setzen des Namens einer Methode in einem Programm, also mittels eines **Methodenaufrufs**, sämtliche in der Methode gekapselten

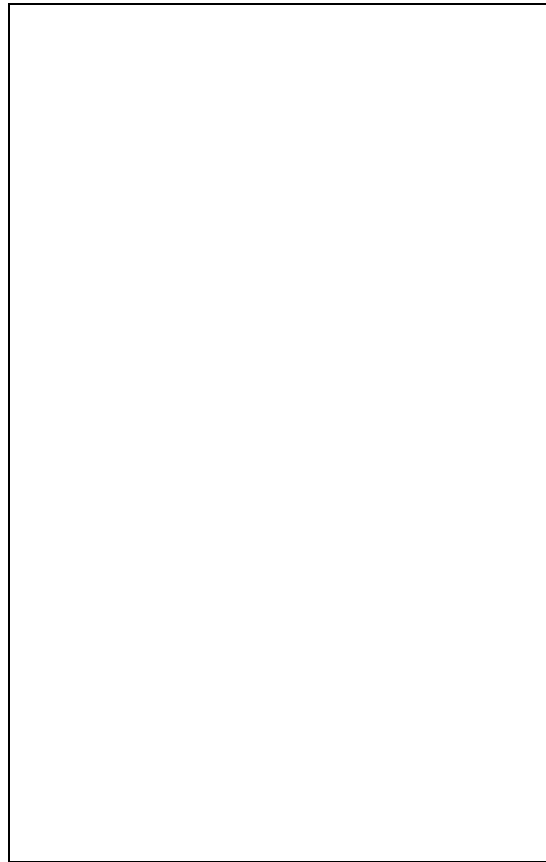
Anweisungen ausführen zu lassen. Ist eine Methode einmal deklariert, ist ihr Aufruf ein elegantes Umgehen des Niederschreibens sämtlicher in der Methode enthaltenen Anweisungen. Man kann sogar noch einen Schritt weitergehen: eigentlich braucht der Aufrufer einer Methode gar nicht zu wissen, welche Anweisungen in der Methode stehen, es muss ihm lediglich bekannt sein, was die Methode macht und unter welchen Bedingungen er sie aufrufen darf.

### 1.2.5 Benutzerinteraktion

Es gibt Programme (sog. Batch-Programme), die nach ihrem Start sämtliche Anweisungen ausführen und nach der letzten Anweisung enden, ohne dass der Benutzer auf irgendeine Weise Einfluss auf den Programmverlauf nimmt (z.B. das Drucken der Passagierliste eines Fluges oder die monatliche Gehaltsabrechnung einer Firma). Dann wiederum gibt es auch Programme, bei denen zur **Laufzeit**, also während des Ausführens der Anweisungen, eine **Interaktion** mit dem Benutzer stattfindet (z.B. Online Banking oder Surfen im Internet).

Dadurch, dass Java **graphische Benutzeroberflächen** (engl. **graphical user interface, GUI**) zur Verfügung stellt, wird eine hohe Benutzerinteraktion möglich. Beispielsweise kann der Benutzer eine Auswahl aus einem Menü treffen, in Textfelder Eingaben machen oder durch Knopfdruck Vorgänge starten. Als Eingabemedium stehen zumeist Tastatur und Maus zur Verfügung. Abbildung 1-2 zeigt eine solche graphische Benutzeroberfläche.

Voraussetzung einer Benutzer-Interaktion ist selbstverständlich, dass Benutzereingaben vom Programm erkannt und verarbeitet werden können.



**Abbildung 1-2: Graphische Benutzeroberfläche**

## 1.3 Java

**Java** ist eine **objektorientierte Programmiersprache**, die 1995 von der Firma Sun Microsystems in Kalifornien entwickelt wurde. Seinen Namen verdankt Java den Entwicklern, sie benannten ihr Produkt nämlich nach ihrer Lieblingskaffeesorte.

### 1.3.1 Klassenbibliothek

Damit man das Rad nicht nochmals neu erfinden muss, verfügt Java über bereits vordefinierte Klassen mitsamt Attributen und Methoden. So gibt es beispielsweise mathematische Methoden zur Berechnung der Quadratwurzel oder zur Erzeugung von Zufallszahlen, aber auch Klassen, mit deren Hilfe man graphische Benutzeroberflächen generieren kann. Die Gesamtheit all dieser Klassen bildet die **Klassenbibliothek**.



Für einen Java-Programmierer ist es somit unerlässlich, sich mit der Klassenbibliothek auseinanderzusetzen. Natürlich deklariert man in einem Java-Programm immer auch noch seine „eigenen“ Klassen, jedoch wird man zu einem grossen Teil Klassen und Methoden aus der Klassenbibliothek verwenden.

Die Klassenbibliothek ist in sogenannte **Pakete** (engl. **packages**) unterteilt, wobei ein solches Paket jeweils Klassen eines bestimmten Themenbereichs umfasst. So ist beispielsweise `math` der Name des Pakets, welches alle mathematischen Klassen beinhaltet, und `awt` (von abstract window toolkit) bezeichnet das Paket, dessen Klassen für die Programmierung graphischer Benutzeroberflächen verwendet werden.

### 1.3.2 Portabilität

Einer der grossen Vorzüge von Java ist die **Betriebssystemunabhängigkeit**. Dies hat zur Folge, dass ein Java-Programm auf jedem Computer – ob Macintosh oder Windows-PC – laufen kann, weshalb man Java auch als **portabel** bezeichnet.

Wie wird nun aber diese Portabilität erreicht?

In Abschnitt 1.2 wurde erwähnt, dass eine Programmiersprache sowohl für den Menschen als auch für den Computer verständlich ist. Diese Aussage ist zwar richtig, bedarf aber einer Erläuterung.

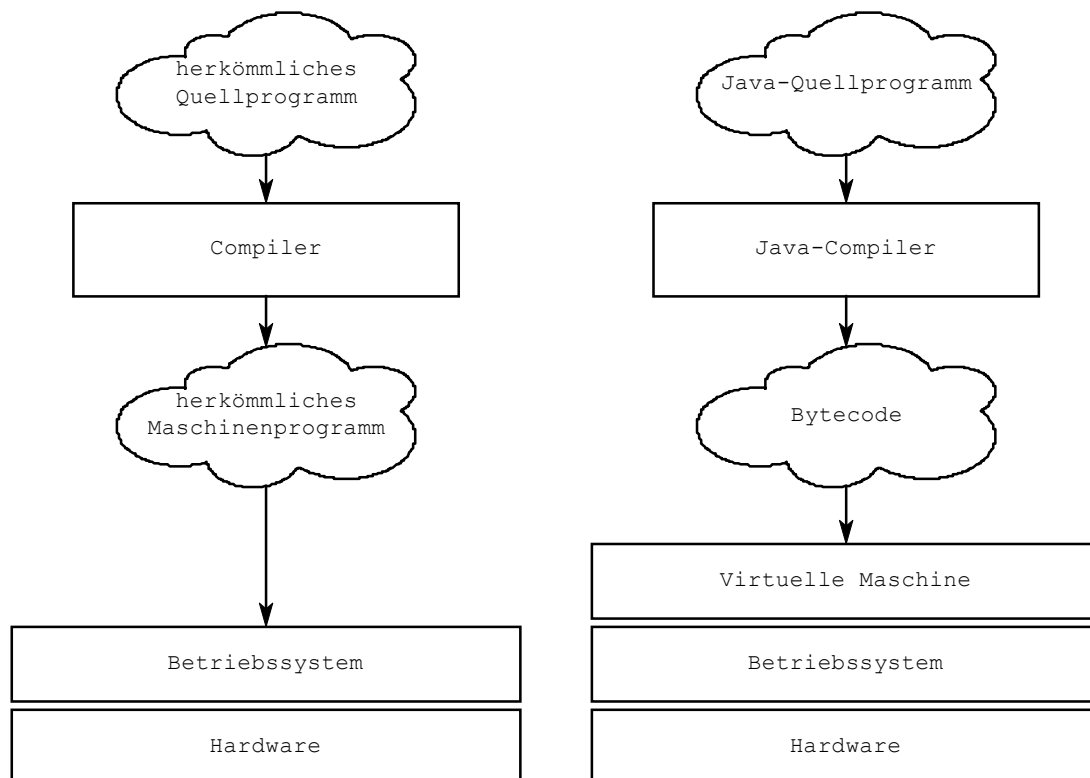
Die Sprache des Rechners (Hardware) ist eine **binäre** Sprache, eine Sprache also, die nur zwei verschiedene Zeichen unterscheidet. Damit der Rechner somit unser Programm versteht, bedarf es einer Übersetzung in seine Sprache, der sogenannten **Maschinensprache**.

Es gibt nun spezielle Übersetzungsprogramme, also wiederum Software, welche diese Aufgabe übernehmen. Sie werden als **Compiler** bezeichnet und bilden eine Vermittlerschicht zwischen dem zu übersetzenden Programm und dem Rechner. In Wirklichkeit aber kommuniziert der Compiler nicht direkt mit dem Rechner, dazu benötigt er ein weiteres „Vermittlerprogramm“, das Betriebssystem. Das **Betriebssystem** arbeitet sehr eng mit dem Rechner zusammen. Beispielsweise ermöglicht es die Kommunikation zwischen Anwendungsprogrammen (z.B. Winword, etc.) und Rechner. Bildlich gesprochen, kann man sich das alles wie eine Schichtentorte vorstellen: zuunterst ist der Rechner, also die sogenannte Hardware, auf welchem dann mehrere Schichten Software liegen. Bedeutend hierbei ist, dass jede Schicht eine Art Vermittler oder Übersetzer zwischen der direkt darüber und der direkt darunterliegenden Schicht ist.

Bei den herkömmlichen Programmiersprachen übersetzt der Compiler das in der Programmiersprache verfasste Programm, genannt **Quellprogramm**, in die jeweilige Maschinensprache, indem er ein in Binärcode (Maschinensprache) verfasstes **Maschinenprogramm** erzeugt. Das Maschinenprogramm kann vom jeweiligen Rechner direkt ausgeführt werden. Da es aber betriebssystemspezifisch ist, kann es nicht auf einem

Rechner anderen Typs laufen. Dies hat zur Folge, dass man für jeden Rechnertyp den dazugehörigen Compiler benötigt.

Bei Java hingegen erzeugt der Compiler einen sogenannten **Bytecode**, welcher betriebssystemunabhängig ist. Dieser kann jedoch aufgrund seiner Betriebssystemunabhängigkeit nicht direkt von einem Rechner ausgeführt werden, sondern wird von der **virtuellen Maschine** von Java (engl. **Java virtual machine**) interpretiert<sup>1</sup>. Eine virtuelle Maschine ist eine Software, welche die Funktionalität eines echten Rechners softwaremässig simuliert. Der Bytecode bildet also das Maschinenprogramm für die virtuelle Maschine und wird von dieser ausgeführt. Abbildung 1-3 veranschaulicht dies.



**Abbildung 1-3: Herkömmliche Programmiersprachen versus Java**

---

<sup>1</sup> Bei der virtuellen Maschine von Java handelt es sich um einen Interpreter. Obwohl ein Interpreter auch ein Übersetzer ist, speichert er im Gegensatz zum Compiler „seine Übersetzung“ nicht, sondern führt sie gleich während dem Übersetzen aus. Dies hat zur Folge, dass ein Interpreter bei jedem Programmaufruf das Programm neu übersetzen muss. Hingegen kann das von einem Compiler einmal übersetzte Maschinenprogramm direkt ausgeführt werden.

So wie man bei den herkömmlichen Programmiersprachen für jeden Rechnertyp den dazugehörigen Compiler benötigt, braucht es bei Java für jeden Rechnertyp die dazugehörige virtuelle Maschine. Was hat man nun gewonnen?

Vorteilhaft an der Java-Architektur ist, dass es sich bei dem durch den Java-Compiler generierten Bytecode um ein ausführbares, betriebssystemunabhängiges Programm handelt. Im Gegensatz dazu ist bei herkömmlichen Programmiersprachen das Maschinenprogramm zwar auch ausführbar, aber nicht betriebssystemunabhängig. Diese Tatsache kommt vor allem im Zusammenhang mit dem Austausch von Programmen zwischen Rechnern unterschiedlichen Typs sowie beim Versenden von Programmen im Internet zum Tragen.

Bei einem herkömmlichen Programm ist man gezwungen, das Quellprogramm zu übertragen, da allein dieses betriebssystemunabhängig ist. Bevor man aber dieses ausführen kann, muss es zuerst noch auf dem Zielrechner kompiliert werden. Java hingegen erlaubt es, direkt den Bytecode zu übermitteln – zum Beispiel über Internet –, welcher unmittelbar, ohne neuerliche Kompilation, auf dem Zielrechner ausgeführt werden kann.

### 1.3.3 Internet

Aufgrund seiner zuvor erläuterten Portabilität ist Java für das **Internet** äusserst attraktiv.

Wenn man heutzutage irgendeine WWW-Seite besucht, findet man häufig graphische Animationen vor. Solche sind in Java realisiert und werden als **applets** bezeichnet. Das Wort applet kommt von „little application“ und ist sinngemäss als kleines Programm zu übersetzen. Applets sind Java-Programme, welche in einem WWW-Browser laufen.

Von den applets unterscheiden sich die **Applications**. Hierbei handelt es sich ebenfalls um Java-Programme, welche aber auf dem eigenen Rechner laufen.

Da eine Application durch den Benutzer gestartet wird und die Kontrolle über den Programmverlauf voll und ganz in Javas Hand liegt, handelt es sich bei den Applications um eigenständig Programme. Bei den applets hingegen hat die Ausführungskontrolle der WWW-Browser, weswegen man auch von „nicht-eigenständigen Programmen“ spricht.

## 1.4 Zusammenfassung

Im ersten Abschnitt dieses Kapitels wurden wir mit der **Objektorientierung**, welche auf dem Prinzip der Klassifizierung beruht, vertraut gemacht. Die wichtigsten Begriffe hierzu sind:

- ◆ **Klasse** (engl. **class**): Eine Klasse beschreibt, was allen Objekten einer Klasse gemeinsam ist, indem sie deren Attribute und Methoden deklariert.
- ◆ **Objekt/Instanz** (eng. **object/instance**): Ein Objekt kann wohl aufgrund seines „Typs“ einer Klasse zugeordnet werden, ist aber immer ein Individuum. Es weist eigenständige Werte für die Attribute der Klasse auf, wodurch auch sein Zustand definiert wird.
- ◆ **Botschaft** (engl. **message**): Eine an ein Objekt geschickte Botschaft löst den Aufruf einer in der Klasse deklarierten Methode aus. Botschaften ermöglichen die Kommunikation unter den Objekten. Sie erlauben auch, Werte mitzuschicken, welche als Parameter bezeichnet werden.
- ◆ **Vererbung** (engl. **inheritance**): Die Vererbung erlaubt den Aufbau eines hierarchischen Systems unterhalb von Klassen bzw. Objekten. Hierbei vererbt eine Oberklasse (engl. superclass) sämtliche Attribute und Methoden an ihre Unterklassen (engl. subclass). Aufgrund der Vererbung ist eine Instanz einer Unterklasse auch eine Instanz einer Oberklasse; die Umkehrung gilt jedoch nicht.

Dann wurde der Begriff Programm eingehender erläutert: ein **Programm** ist ein Text, welcher aus einer Menge von Arbeitsanweisungen an den Computer besteht. Hierbei erlauben **Sequenz**, **Selektion**, **Iteration** und **Abstraktion** die unterschiedliche Gruppierung der Arbeitsanweisungen und bestimmen dadurch, wann und ob sie überhaupt ausgeführt werden. Programme kann man auch anhand der Unterstützung einer **Benutzerinteraktion** charakterisieren.

Zum Abschluss des Kapitels wurde die objektorientierte Programmiersprache **Java** noch eingehender vorgestellt. Sie verfügt über eine grosse **Klassenbibliothek**, welche eine Vielzahl vordefinierter Klassen mitsamt Attributen und Methoden umfasst. Hervorzuhalten, gerade hinsichtlich dem Internet, ist auch die **Betriebssystemunabhängigkeit**, welche dank dem Konzept der **virtuellen Maschine** realisiert wird.

# 2

## Programmaufbau Vererbung, Redefinition Instanz-/Klassenvariablen

Die nun bekannten Begriffe der Objektorientierung werden in diesem Kapitel in Zusammenhang mit der Programmierung gebracht. Für die Darlegung wird ein erstes einfaches Programm verwendet, welches in vier Schritten sukzessive in seiner Funktionalität erweitert wird. Sämtliche Programmversionen sind mittels graphischer Benutzeroberflächen interaktiv ausgestaltet.

### 2.1 Programmaufbau, Attribute und Methoden

Endlich ist es soweit: untenstehend befindet sich das erste, vollständige Java-Programm „Business Cards - Version 1“. Es handelt sich hierbei um eine Application, welche dank einer graphischen Benutzeroberfläche Daten in Interaktion mit dem Benutzer erfasst und sie danach in geordneter, vordefinierter Form ausgibt.

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

public class UserFrame extends Frame implements ActionListener {
    private TextField firstName, familyName, street, zipCode, city;

    private void place(Component comp,int x,int y,int width,int height) {
        comp.setBounds(x, y, width, height);
        add(comp);
    }

    public UserFrame() {
        Button button;
        setTitle("Business Cards");
        setLayout(null);
        setSize(200,300);
        setResizable(false);
        place(new Label("FirstName"),30,40,140,20);
        place(firstName=new TextField(),30,60,140,20);
    }
}
```

```
        place(new Label("FamilyName"), 30, 90, 140, 20);
        place(familyName=new TextField(), 30, 110, 140, 20);
        place(new Label("Street"), 30, 140, 140, 20);
        place(street=new TextField(), 30, 160, 140, 20);
        place(new Label("ZipCode"), 30, 190, 60, 20);
        place(zipCode=new TextField(), 30, 210, 60, 20);
        place(new Label("City"), 100, 190, 70, 20);
        place(city=new TextField(), 100, 210, 70, 20);
        place(button=new Button("PRINT"), 70, 250, 60, 20);
        button.addActionListener(this);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent event){
        System.out.println(firstName.getText()+" "+familyName.getText());
        System.out.println(street.getText());
        System.out.println(zipCode.getText()+" "+city.getText());
        System.out.println();
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 2.1.1 Zum Programm

In Abschnitt 1.2 wurde erklärt, dass ein Programm ein Text ist, welcher aus Arbeitsanweisungen an den Computer besteht. Wo sind nun aber diese Anweisungen zu finden?

#### 2.1.1.1 Anweisungen, import statement, geschweifte Klammern

In Java wird jede **Anweisung** (engl. **statement**) mit einem **Semikolon** beendet.

Betrachten wir die ersten zwei Anweisungen:

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

Das **Schlüsselwort** **import** besagt, dass etwas aus der Klassenbibliothek importiert wird. Dieses „etwas“ kann eine einzelne Klasse oder aber auch ein ganzes package sein. Ähnlich einer WWW-Adresse trennt der Punkt in obiger Notationsform einzelne „Ebenen“ voneinander: ganz links steht immer `java` und rechts darauf folgend, jeweils durch einen Punkt getrennt, das package und die Klasse – der Fokus wird gewissermassen immer

enger. Steht ein Stern, bedeutet dies, das alles aus dem jeweiligen Bereich importiert werden soll. In unserem Fall werden die packages `awt` und `event` importiert<sup>1</sup>. Dank der `import` Anweisung kann man nun sämtliche Klassen dieser zwei packages mitsamt deren Attributen und Methoden verwenden.

```
public class UserFrame extends Frame implements ActionListener {
```

Die nächste Zeile im Programm endet nun nicht mit einem Semikolon, sondern mit einer öffnenden geschweiften Klammer. Ein Paar **geschweifte Klammern** – also eine öffnende und eine schliessende – wird in Java dazu verwendet, mehrere Anweisungen zusammenzufassen. In obigen Fall handelt es sich um eine Klasse, wie das Schlüsselwort **class** besagt<sup>2</sup>. `UserFrame` bezeichnet den Namen dieser Klasse<sup>3</sup>.

Wo endet nun aber die Klasse `UserFrame`?

Bereits am Ende der Zeile

```
private void place(Component comp, int x, int y, int width, int height){
```

steht wiederum eine öffnende geschweifte Klammer. Man hat also in Java die Möglichkeit, geschweifte Klammerpaare ineinander zu verschachteln. Um nun das Ende, also die schliessende geschweifte Klammer, der Klasse `UserFrame` zu finden, muss man die paarweise öffnenden/schliessenden geschweiften Klammern durchgehen, bis man zur schliessenden Klammer von `UserFrame` gelangt. Hierbei gilt natürlich wie in der Mathematik, dass es zu einer öffnenden Klammer immer auch eine schliessende Klammer und somit gleich viele öffnende wie schliessende geschweifte Klammern gibt.

Die eingerückte Schreibweise des Programms erleichtert uns das Finden der schliessenden Klammer von `UserFrame` sie befindet sich unmittelbar vor der Zeile

```
public class TestProg {
```

Darauf folgt die Klasse `TestProg`, welche in der letzten Zeile des Programms endet.

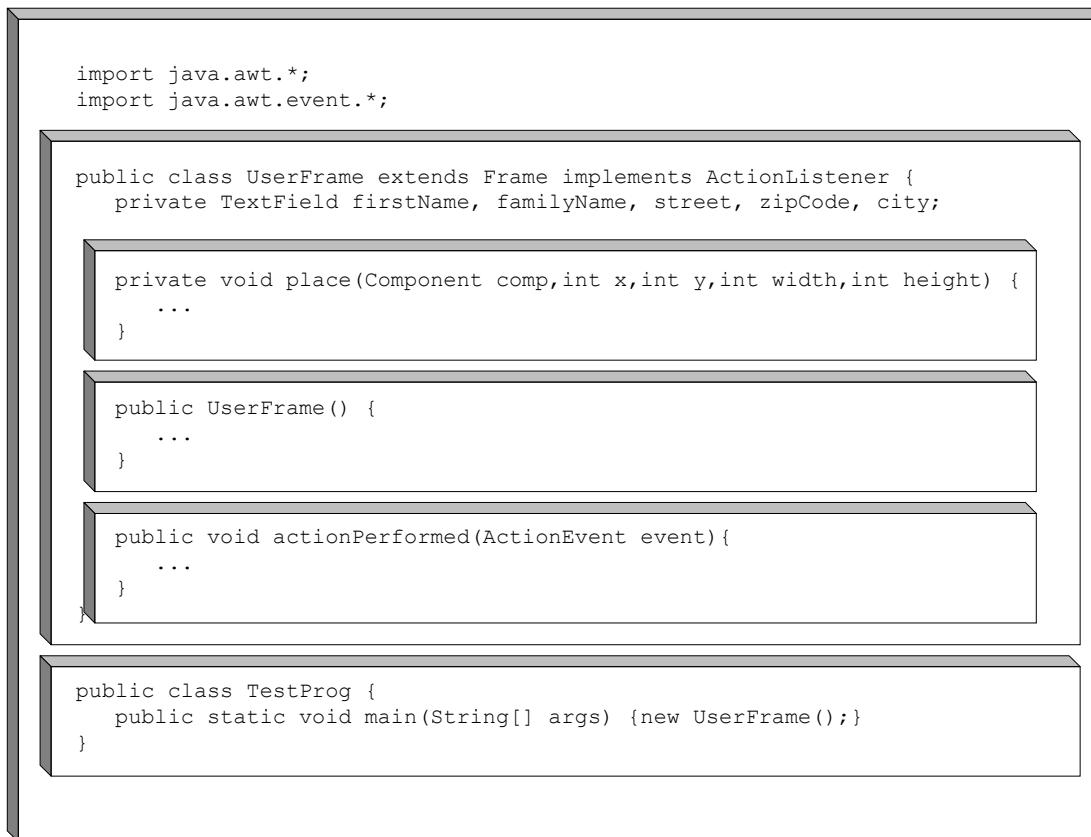
Unser Programm besteht somit im wesentlichen aus den zwei Klassen `UserFrame` und `TestProg` sowie zwei `import` Anweisungen. Abbildung 2-1 zeigt die ineinanderverschachtelte Struktur des Programms, welche durch die eingerückte Schreibweise verdeutlicht wird.

---

<sup>1</sup> Bei `event` handelt es sich um ein package, welches sich innerhalb des package `awt` befindet.

<sup>2</sup> Das Schlüsselwort `public` definiert den Gültigkeitsbereich (siehe Abschnitt 6.2.1.3 und 6.2.2.1) der Klasse `UserFrame`. Im Unterschied zum Schlüsselwort `private`, welches einen minimalen Gültigkeitsbereich festlegt, ist der durch das Schlüsselwort `public` definierte Gültigkeitsbereich unbeschränkt.

<sup>3</sup> Im Skript werden für die Namen von Klassen konsequent grosse Anfangsbuchstaben verwendet.



**Abbildung 2-1: Ineinanderverschachtelte Programmstruktur**

Bevor wir nun genauer betrachten, was innerhalb der zwei Klassen steht, probieren wir das Programm zunächst einmal aus.

***Siehe auch:*** 2.1.2.2, 2.1.2.3, 2.1.2.4

### 2.1.1.2 *TextField, Label, Button und Frame*

Abbildung 2-2 zeigt uns die graphische Benutzeroberfläche, so wie sie sich dem Benutzer beim Starten des Programms präsentiert.



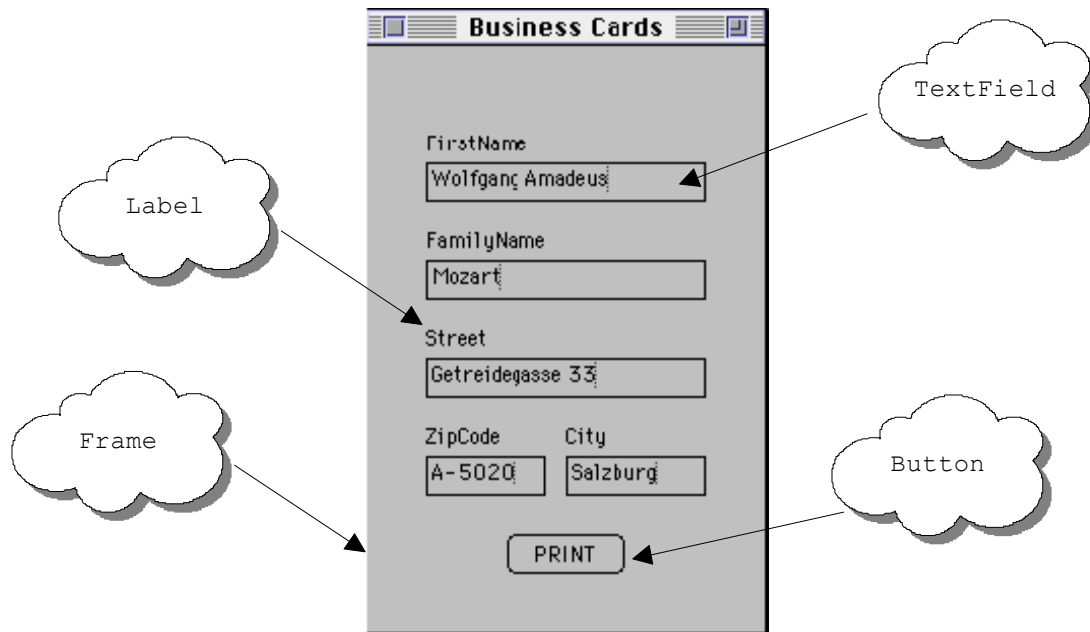
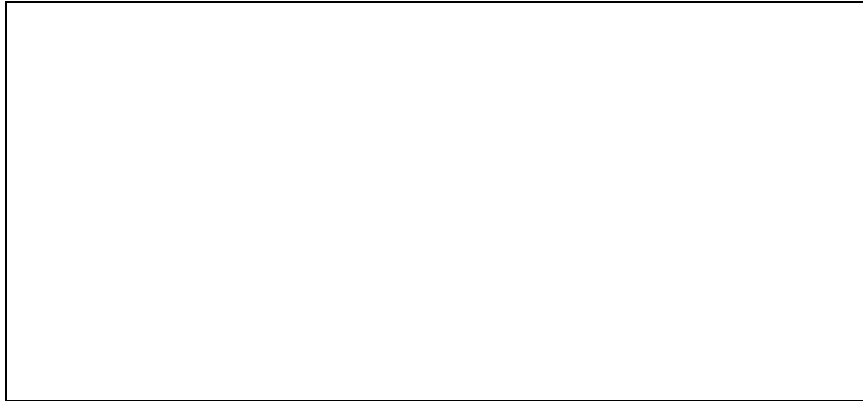


Abbildung 2-2: Business Cards- Version 1, UserFrame

Der Benutzer kann durch die Eingabe in die Textfelder (`TextField`) die Namen von Personen und deren Adresse erfassen. Hierbei stehen ihm die üblichen Editieroptionen mittels Tastatur und Maus zur Verfügung. Mit der Tabulatortaste gelangt man von einem `TextField` zum nächsten. Durch Aktivieren des Knopfes (`Button`) `PRINT` wird die Anschrift der Person generiert und auf der **Konsole** (engl. **console**) ausgedruckt (siehe Abbildung 2-3). Natürlich kann man auch weitere Personen erfassen und ausgeben. Hierzu muss man lediglich die noch vorhandenen Angaben der vorherigen Person mit den Angaben der neu zu erfassenden Person überschreiben und erneut den `PRINT`-Button drücken.



**Abbildung 2-3: Business Cards - Version 1, Konsole**

Abbildung 2-2 zeigt die Verwendung von Bausteinen graphischer Benutzeroberflächen. Solche Bausteine sind beispielsweise `TextField`, `Label`, `Button` und `Frame`. Ein **TextField** ist ein einzeliges Eingabefeld. Ein **Label** dient zur Beschriftung anderer Bausteine. Ein **Button** ermöglicht dem Benutzer, durch seine Aktivierung einen Prozess in Gang zu setzen. Als **Frame** schlussendlich bezeichnet man den fensterartigen Rahmen, welcher die zuvor aufgeführten Bausteine enthält und auch die übliche Funktionalität eines Fensters wie Titelleiste, Rahmen, Symbole für die Manipulation der Fenstergrösse etc. aufweist. Alle diese Bausteine sind bereits als vordefinierte Klassen im package `awt` der Java Klassenbibliothek enthalten und werden als sogenannte **Components** bezeichnet. Da es sich bei einem `Frame` um eine `Component` handelt, die selber wiederum andere `Components` enthalten kann, nennt man ihn auch **Container**.

Abbildung 2-2 verdeutlicht ebenfalls, dass man im Prinzip zwei **unterschiedliche Sichten** auf ein Programm voneinander unterscheiden kann. Zum einen gibt es die Sicht des Programmierers, also die Ebene des Quellprogramms, und zum anderen die Sicht des Benutzers und damit die Ebene der graphischen Benutzeroberfläche, welche durch Ausführen des Programms entsteht.

### *2.1.1.3 Attribute und Methoden*

Wenden wir uns nun wieder unserem Beispielprogramm zu und nehmen die Klasse `UserFrame` genauer unter die Lupe.

In der Zeile

```
private TextField firstName, familyName, street, zipCode, city;
```

wird deklariert, welches die **Attribute** der Klasse `UserFrame` sind. `UserFrame` hat die `TextFields` `firstName`, `familyName`, `street`, `zipCode` und `city` als **Attribute**.

Man kann also aus obiger Attributdeklaration die Namen (engl. **identifier**)<sup>1</sup> der jeweiligen Objekte, nämlich `firstName`, `familyName`, `street`, `zipCode` und `city`, und ihre Klasse, also `TextField`, entnehmen. Es ist an dieser Stelle noch zu erwähnen, dass es sich bei der Klasse `UserFrame` um einen speziell für unsere Zwecke massgeschneiderten Frame handelt. Hierauf wird aber noch ausführlich in Abschnitt 2.3 eingegangen.

Innerhalb der Klasse `UserFrame` findet man **Methodendeklarationen**. Eine Methodendeklaration erkennt man an einem **runden Klammerpaar** und dem nachfolgenden **geschweiften Klammerpaar**. In den runden, öffnenden und schliessenden Klammern stehen die **Parameter**. Falls die Methode keine Parameter hat, ist das Klammerpaar leer. Ansonsten werden mehrere Parameter mittels **Kommata** getrennt. Unmittelbar vor den runden Klammern ist der Name (engl. **identifier**)<sup>2</sup> der Methode zu finden. Die Klasse `UserFrame` stellt also die Methoden `place()`, `UserFrame()` und `actionPerformed()`<sup>3</sup> zur Verfügung. Hierbei ist `UserFrame()` die einzige Methode, welche keine Parameter hat.

Wenden wir uns nun der Methode `place()` zu:

```
private void place(Component comp, int x, int y, int width, int height){
    comp.setBounds(x, y, width, height);
    add(comp);
}
```

Wie bereits aus Abschnitt 1.2.4 hervorgeht, befinden sich innerhalb der geschweiften Klammern sämtliche Anweisungen der Methode `place()`. Das geschweifte Klammerpaar mitsamt Inhalt bezeichnet man als **Rumpf** (engl. **body**) der Methode. Die ersten zwei Zeilen hingegen, also alles bis zur öffnenden geschweiften Klammer, bezeichnet man als **Kopf** der Methode, wofür aber auch synonym die Begriffe **Methodenschnittstelle** oder **Signatur** verwendet werden.

Die Methode `place()` platziert die Komponenten innerhalb eines `UserFrame`. Dazu verwendet sie Methoden aus der Klassenbibliothek, genauer aus dem package `awt`. Sie hat fünf Parameter. Beim ersten handelt es sich um diejenige Komponente, welchen man plazieren möchte. Die Parameter `x` und `y` geben die Koordinaten der linken oberen Ecke dieser Komponente an und die Parameter `width` und `height` enthalten die Breite bzw. die Höhe der Komponente. Es ist hierbei zu beachten, dass sowohl `x` und `y` als auch `width` und `height` in **Rasterpunkten** (engl. **pixel**, Kurzwort für: picture element)

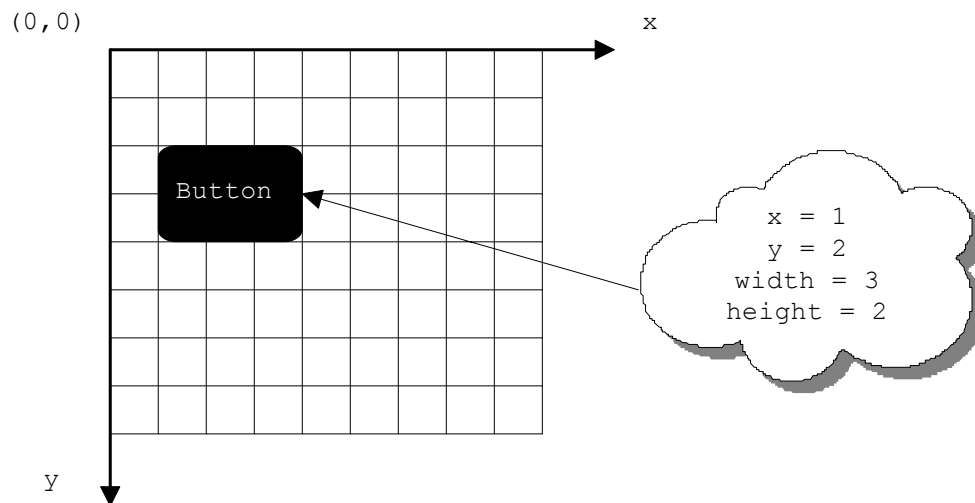
---

<sup>1</sup> Im Skript werden für die Namen von Attributen konsequent kleine Anfangsbuchstaben verwendet.

<sup>2</sup> Im Skript werden für die Namen von Methoden - wie auch für die Namen von Attributen - konsequent kleine Anfangsbuchstaben gesetzt.

<sup>3</sup> Um zu verdeutlichen, dass es sich um eine Methode handelt, werden nach dem Methodennamen die runden Klammern gesetzt.

gemessen werden<sup>1</sup>. Des weiteren werden im Unterschied zu den kartesischen Koordinaten die x-/y-Achsenabschnitte immer von der linken oberen Ecke aus gemessen. Abbildung 2-4 veranschaulicht dies.



**Abbildung 2-4: Platzierung eines Buttons innerhalb eines Rasters**

Gehen wir nun zur Methode `UserFrame()` über. In ihr erfolgt für jede Komponente ein **Aufruf** der Methode `place()`:

---

<sup>1</sup> Standardmässig beträgt die Auflösung bei einem 17“-Bildschirm 1024 x 768 pixels, bei einem 15“-Bildschirm 800 x 600 pixels und bei einem 20“-Bildschirm 1280 x 1024 pixels.

```
public UserFrame() {  
    Button button;  
    setTitle("Business Cards");  
    setLayout(null);  
    setSize(200,300);  
    setResizable(false);  
    place(new Label("FirstName"),30,40,140,20);  
    place(firstName=new TextField(),30,60,140,20);  
    place(new Label("FamilyName"),30,90,140,20);  
    place(familyName=new TextField(),30,110,140,20);  
    place(new Label("Street"),30,140,140,20);  
    place(street=new TextField(),30,160,140,20);  
    place(new Label("ZipCode"),30,190,60,20);  
    place(zipCode=new TextField(),30,210,60,20);  
    place(new Label("City"),100,190,70,20);  
    place(city=new TextField(),100,210,70,20);  
    place(button=new Button("PRINT"),70,250,60,20);  
    button.addActionListener(this);  
    setVisible(true);  
}
```

Während bei der **Methodendeklaration** die Namen und die Art ihrer Parameter festgelegt werden, stehen beim **Methodenaufruf** die tatsächlichen, aktuellen Werte der Parameter.

```
place(new Label("FirstName"),30,40,140,20);
```

Obiger Aufruf bewirkt, dass ein Objekt der Klasse `Label` mit der Beschriftung<sup>1</sup> „FirstName“ und einer Breite von 140 pixels bzw. einer Höhe von 20 pixels an der Stelle (30, 40) in einem Objekt der Klasse `UserFrame` platziert wird. Die Platzierung erfolgt analog zu der in Abbildung 2-4 illustrierten Vorgehensweise. Das Resultat ist in Abbildung 2-2 ersichtlich.

```
place(firstName=new TextField(),30,60,140,20);
```

Im Unterschied zur vorhergehenden Anweisung geben wir nun dem zu positionierenden Objekt den Namen `firstName`. Wir bezeichnen dieses Objekt, damit wir es später ansprechen und ihm Botschaften schicken können. Neben dem Objekt `firstName` gibt es auch noch andere Instanzen, die wir benennen, die wären `familyName`, `street`, `zipCode` und `city`. Es lässt sich nun unschwer erkennen, dass es sich bei all diesen Instanzen um die Attribute der Klasse handelt. Objekte, wie das weiter oben platzierte Label mit der Beschriftung „FirstName“, die **namenlos** sind, können später nicht mehr angesprochen werden.

Sämtliche Aufrufe von `place()` generieren nun der Reihe nach sämtliche Labels, TextFields und einen Button, bis schlussendlich die in Abbildung 2-2 ersichtliche graphische Benutzeroberfläche entsteht.

---

<sup>1</sup> Die Beschriftung ist der Text, welcher das Objekt der Klasse `Label` in der graphischen Benutzeroberfläche anzeigt.

Neben den in der Attributdeklaration aufgeführten Objekten gibt es aber noch ein weiteres Objekt, welches wir benennen. Es heisst `button` und ist eine Instanz der Klasse `Button`.

```
Button button;
```

Da die Instanz `button` innerhalb der Methode `UserFrame()` deklariert wurde und nicht innerhalb der Klasse `UserFrame`, handelt es sich nicht um ein Attribut der Klasse `UserFrame`. Dies hat zur Folge, dass es nur innerhalb der Methode selbst „bekannt“ ist und auch nur dort angesprochen werden kann.

### Der Aufruf

```
setTitle("Business Cards");
```

bewirkt, dass in der Titelleiste eines Objekts der Klasse `UserFrame` der Text „Business Cards“ erscheint. Hierbei handelt es sich im Gegensatz zu der Methode `place()`, welche durch den Programmierer neu deklariert wurde, um eine in der Klassenbibliothek bereits vorhandenen Methode.

Damit wir sämtliche Labels, TextFields und den einen Button mittels `place()` absolut positionieren können, also konkrete x-/ und y-Werte angeben dürfen, ist der Aufruf

```
setLayout(null);
```

notwendig. Es handelt sich bei `setLayout()` wiederum um eine im package `awt` deklarierte Methode.

```
setSize(200,300);
```

setzt die Breite einer Instanz der Klasse `UserFrame` auf 200 pixels und die Höhe auf 300 pixels.

```
setResizable(false);
```

macht das Ändern der Grösse eines Objektes der Klasse `UserFrame` durch Ziehen mit der Maus an einer Ecke des Fensters unmöglich.

```
setVisible(true);
```

macht ein Objekt der Klasse `UserFrame` überhaupt erst sichtbar.

Es lässt sich nun fragen, wer der Empfänger der zuvor behandelten Botschaften ist. Wenn nicht explizit ein Empfänger angegeben ist, dann wird die Botschaft an eine Instanz jener Klasse gesandt, in welcher der Aufruf erfolgt. Auf diese Problematik wird noch in Abschnitt 6.1.1.3 und 6.1.2.3 weiter eingegangen. In unserem Beispiel ist auf alle Fälle, wenn keine explizite Angabe gemacht wird, die in Abbildung 2-2 ersichtliche Instanz des `UserFrame` der Empfänger der Botschaften.

In unserem Beispielprogramm werden hauptsächlich Methoden aus der Klassenbibliothek verwendet. Neu deklariert wurden durch den Programmierer lediglich die Klassen

UserFrame und TestProg, die Attribute firstName, familyName, street, zipCode und city und die Methoden place(), UserFrame() und actionPerformed(). Eine Methode aus der Klassenbibliothek erkennt man daran, dass sie nicht im Programm deklariert wurde. Bei Unsicherheit hilft ansonsten auch ein Blick in Anhang B, wo sämtliche in diesem Skript verwendete Elemente aus der Java-Klassenbibliothek aufgelistet werden.

***Siehe auch:** 2.1.2.1, 2.1.2.5, Anhang B*

### 2.1.1.4 Event Handling

Graphische Benutzeroberflächen erlauben eine Interaktion mit dem Benutzer. So kann in unserem Beispielprogramm der Benutzer seine erfassten Daten durch Aktivieren des Buttons PRINT auf der Konsole ausdrucken lassen. Wenn der Benutzer den Button nicht betätigt, passiert nichts. Diese Art der Interaktion wird in Java durch das sogenannte **Event Handling** möglich. Auf das Event Handling wird noch in den Abschnitten 5.1.1.1 und 5.1.2.1 eingehend eingegangen. Im Programm haben die Anweisung `button.addActionListener()`, die Methode `actionPerformed()` und das Schlüsselwort `implements ActionListener` im Kopf der Klasse UserFrame mit diesem Event Handling zu tun.

Die Anweisung

```
button.addActionListener(this);
```

bewirkt, dass jedesmal, wenn der Button PRINT aktiviert wird, die Methode `actionPerformed()` aufgerufen wird.

### 2.1.1.5 String Concatenation

```
public void actionPerformed(ActionEvent event){
    System.out.println(firstName.getText()+" "+familyName.getText());
    System.out.println(street.getText());
    System.out.println(zipCode.getText()+" "+city.getText());
    System.out.println();
}
```

Das Versenden der Botschaft `println()` an `System.out` hat zur Folge, dass der als Parameter übergebene Text auf der Konsole ausgedruckt wird und danach ein Zeilenumbruch stattfindet. Hierbei bezeichnet `System.out` das momentane Ausgabemedium, also die Konsole (Java Console). Wird kein Parameter übergeben, so wird eine Leerzeile gedruckt. Das letzte `println()` hat also zur Folge, dass beim Ausdruck mehrerer Personen deren Adressen durch einen Abstand voneinander getrennt sind.

Schauen wir uns genauer den Parameter des ersten Methodenaufrufs an:

```
System.out.println(firstName.getText()+" "+familyName.getText());
```

Wie man sehen kann, stehen innerhalb der runden Klammern wiederum zwei Methodenaufrufe. Beim ersten handelt es sich um die Botschaft `getText()`, welche an das `TextField firstName` geschickt wird. Aufgrund dieser Botschaft liefert `firstName` den durch den Benutzer eingegebenen Text zurück. Dieselbe Botschaft wird auch an das `TextField familyName` gesandt. Zwischen den beiden Aufrufen von `getText()` ist zweimal der **‘+’-Operator** und ein Leerschlag `" "` zu finden. Der erste **‘+’-Operator** fügt den von `firstName` erhaltenen Text mit dem Leerschlag zusammen, welches dann wiederum als Ganzes durch den zweiten **‘+’-Operator** mit dem durch von `familyName` gelieferten Text vereint wird. Wir erhalten den in der ersten Zeile von Abbildung 2-3 sichtbaren Text, nämlich den Vornamen und Nachnamen der erfassten Person, getrennt durch einen Leerschlag. Mit dem **‘+’-Operator** kann man also Texte aneinanderfügen. Texte werden als **String** und das Zusammenfügen von Texten als **Concatenation** bezeichnet. In analoger Weise wird nun die Adresse ausgedruckt, worauf die in Abbildung 2-3 ersichtliche Ausgabe entsteht.

Wie uns die Methode `getText()` verdeutlicht, kann eine an ein Objekt gerichtete Botschaft dieses auch dazu veranlassen, einen Wert zurückzuliefern. Dies erlaubt es uns, der Methode `println()` das Ergebnis des Aufrufes `firstName.getText()` als Parameter zu übergeben. Damit `println()` wirklich einen Text als Parameter erhält, muss somit zuerst die Anweisung `getText()` ausgeführt werden. Ein Einblick in die Ausführungssequenz der Aufrufe zeigt, dass zuerst der Aufruf `firstName.getText()` und danach der Aufruf `familyName.getText()` erfolgt. Darauf werden die beiden Texte mittels des **‘+’-Operator** konkateniert und schlussendlich als Parameter der Methode `println()` übergeben, welche dann ausgeführt wird.

### 2.1.1.6 Die `main()` Methode

Abschliessend wollen wir uns noch kurz der Klasse `TestProg` widmen:

```
public class TestProg {  
    public static void main(String[] args) {new UserFrame();}  
}
```

In dieser Klasse gibt es nur eine Methode. Diese sogenannte **main() Methode** hat eine besondere Stellung in einer Java-Application. Es handelt sich nämlich bei ihr um die erste Anweisung, die beim Interpretieren ausgeführt wird, sie bildet quasi den **Einstiegspunkt** für die virtuelle Maschine ins Programm. In einer Application darf – im Unterschied zu einem applet – die `main()` Methode nie fehlen! Am besten kapselt man die `main()` Methode in einer eigenen Klasse, so wie das in diesem Beispielprogramm gemacht wurde.

Sehen wir uns zum Schluss noch in groben Zügen an, in welcher Reihenfolge die Anweisungen im Programm ausgeführt werden:



Der Einstieg erfolgt in der `main()` Methode der Klasse `TestProg`. Dort wird die in Abbildung 2-2 ersichtliche Instanz der Klasse `UserFrame` erzeugt und die Methode `UserFrame()` aufgerufen. Nun wird der Körper der Methode `UserFrame()` ausgeführt, wir erhalten das in der Abbildung sichtbare Objekt der Klasse `UserFrame`. Dank der Anweisung `button.addActionListener()` und der Methode `actionPerformed()` kann nun auf die Aktivierung des Buttons PRINT durch den Benutzer reagiert werden. Das Programm interagiert so lange mit dem Benutzer, bis dieser es willentlich beendet.

*Siehe auch: 2.1.2.6*

## 2.1.2 Syntax

In diesem Abschnitt sollen die in den Beispielprogrammen vorkommenden, syntaktischen Regeln von Java zusammengefasst und schematisch dargelegt werden.

### 2.1.2.1 Identifier

Ein **Identifier** ist ein Name, um ein „Ding“ in einem Programm zu benennen, damit später mit diesem Identifier auf das „Ding“ bezuggenommen werden kann. So werden beispielsweise Klassen, Attribute und Methoden mittels einem Identifier bezeichnet. Die Wahl eines Identifier steht dem Programmierer frei, sofern er sich an folgende Regeln hält:

Identifier
Erstes Zeichen: Buchstabe oder <code>'_'</code> oder <code>'\$'</code> Folgezeichen: Buchstaben, Zahlen, <code>'_'</code> oder <code>'\$'</code>  Keine Begrenzung der Länge

Beispiele: `UserFrame`  
`firstName`  
`place`

Da Gross- und Kleinbuchstaben unterschieden werden, ist `UserFrame` nicht identisch mit `userframe`.

Es sollte darauf geachtet werden, dass man für Identifier **aussagekräftige** Namen verwendet<sup>1</sup>.

### 2.1.2.2 Schlüsselwörter

Es gibt in Java **Schlüsselwörter** (engl. **keywords**), die bereits eine feste Bedeutung haben und daher reserviert sind. Sie dürfen deshalb nicht für die Benennung eigener Klassen, Attribute oder Methoden verwendet werden.

Bis zu diesem Zeitpunkt haben wir die Schlüsselwörter `import`, `package` und `class` kennengelernt. In Anhang A ist eine vollständige Übersicht aller Schlüsselwörter vorhanden.

### 2.1.2.3 Anweisung

In Java endet jede **Anweisung** (engl. **statement**) mit einem **Semikolon**.

Statement
<code>statement;</code>

Beispiele: `i = 0;`  
`x = x+1;`  
`add(comp);`

Mehrere Anweisungen können auch mittels **geschweiffter Klammern** zusammengefasst werden, wodurch wiederum eine Anweisung entsteht. Die schliessende geschweifte Klammer wird nicht mit einem Semikolon abgeschlossen, wohl aber die darin enthaltenen Anweisungen.

### 2.1.2.4 import statement

Um auf Klassen aus der Klassenbibliothek zugreifen zu können, muss man diese importieren. Nach dem Schlüsselwort **import** gibt man den Pfad der zu importierenden Einheit an. Der Punkt innerhalb einer Pfadangabe trennt ähnlich einer WWW-Adresse die einzelnen Ebenen voneinander.

---

<sup>1</sup> Siehe Anhang D: Prinzipien guten Programmierens.

ImportStatement
<pre>import java.PackageName.ClassName;</pre>

Beispiele: 

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

Wie die Beispiele illustrieren, kann man einen Stern ‘\*’ dazu verwenden, alles einer entsprechenden Ebene zu importieren. Es kann auch vorkommen, dass ein package innerhalb eines anderen package enthalten ist, was im letzten Beispiel der Fall ist.

Das package `java.lang` ist das einzige package, welches automatisch, ohne explizite Angabe importiert wird.

### 2.1.2.5 Klassendeklaration

Eine **Klasse** (engl. **class**) deklariert innerhalb der geschweiften Klammern ihre Attribute und Methoden. Das Schlüsselwort **class** ist zwingend.

ClassDeclaration
<pre>class Identifier {     AttributeDeclarations     MethodDeclarations }</pre>

Beispiel: 

```
siehe Klasse UserFrame im Beispielpogramm
„Business Cards – Version 1“
```

Die Deklaration ihrer Attribute bzw. Methoden ist notwendig, damit man im Programmverlauf auf die Attribute zugreifen bzw. die Methoden aufrufen kann.

### 2.1.2.6 Application

Rudimentärer, syntaktischer Aufbau einer **Application**:

### Application

```
ImportStatements
ClassDeclarations
public class TestProg {
    public static void main(String[] args) {
        Statements
    }
}
```

Beispiel: siehe Beispielprogramm „Business Cards - Version 1“

## 2.2 Checkbox

Das im vorangehenden Abschnitt vorgestellte Programm wurde in seiner Funktionalität um die Auswahl einer Anrede erweitert. Das Programm „Business Cards - Version 2“ sieht nun folgendermassen aus:

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

public class UserFrame extends Frame implements ActionListener {
    private TextField firstName, familyName, street, zipCode, city;
    private CheckboxGroup title;

    private void place(Component comp,int x,int y,int width,int height) {
        comp.setBounds(x, y, width, height);
        add(comp);
    }

    public UserFrame() {
        Button button;
        setTitle("Business Cards");
        setLayout(null);
        setSize(200,300);
        setResizable(false);
        title = new CheckboxGroup();
        place(new Checkbox("Mr",true,title),30,30,40,20);
        place(new Checkbox("Mrs",false,title),80,30,40,20);
        place(new Checkbox("Ms",false,title),130,30,40,20);
        place(new Label("FirstName"),30,50,140,20);
        place(firstName=new TextField(),30,70,140,20);
        place(new Label("FamilyName"),30,100,140,20);
    }
}
```

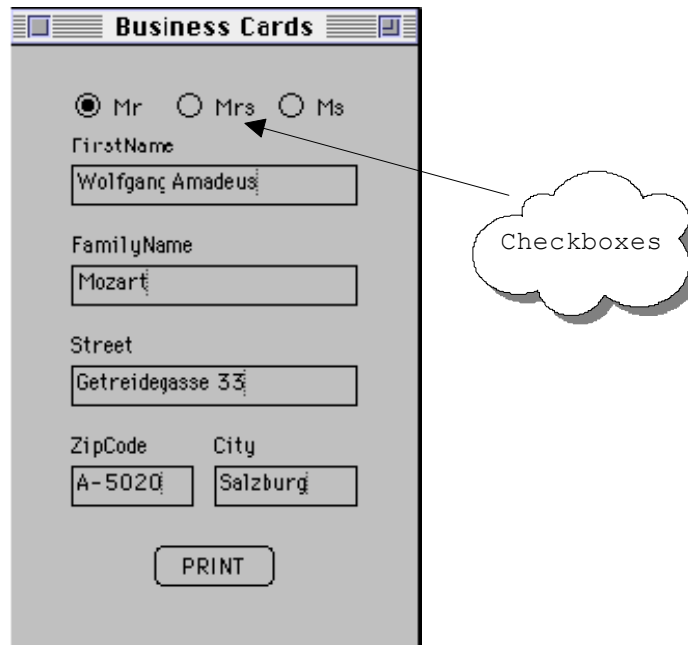
```
        place(familyName=new TextField(),30,120,140,20);
        place(new Label("Street"),30,150,140,20);
        place(street=new TextField(),30,170,140,20);
        place(new Label("ZipCode"),30,200,60,20);
        place(zipCode=new TextField(),30,220,60,20);
        place(new Label("City"),100,200,70,20);
        place(city=new TextField(),100,220,70,20);
        place(button=new Button("PRINT"),70,260,60,20);
        button.addActionListener(this);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent event){
        System.out.println(title.getSelectedCheckbox().getLabel()+" "+
                           firstName.getText()+" "+familyName.getText());
        System.out.println(street.getText());
        System.out.println(zipCode.getText()+" "+city.getText());
        System.out.println();
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 2.2.1 Zum Programm

Wie aus Abbildung 2-5 hervorgeht, unterscheidet sich die neue Version des Programmes nur minimal von der vorhergehenden Version. Sämtliche neue Anweisungen sind im Programmtext **fett** gedruckt.



**Abbildung 2-5: Business Cards - Version 2, UserFrame**

### 2.2.1.1 Checkbox, radio buttons und CheckboxGroup

Mit Hilfe von Checkboxes ist es nun möglich, beim Erfassen einer Person auch die Anredeform auszuwählen. Bei einer **Checkbox** handelt es sich um eine beschriftete Komponente, die entweder selektiert oder nicht selektiert ist; man kann also eine binäre Auswahl<sup>1</sup> treffen. Werden mehrere Checkboxes zusammen einer sogenannten **CheckboxGroup** zugeordnet, so kann jeweils immer nur eine dieser Checkboxes gleichzeitig selektiert sein. In Abbildung 2-5 sind die drei Checkboxes mit der Beschriftung Mr, Mrs und Ms sichtbar, welche zusammen einer CheckboxGroup zugeordnet sind. Übrigens unterscheiden sich Checkboxes, welche einer CheckboxGroup angehören, optisch von solchen, die keiner CheckboxGroup zugeordnet sind. Erstere werden auch aufgrund ihres gegenseitigen Ausschlusses umgangssprachlich als **radio buttons** bezeichnet.

In der Attributdeklaration finden wir nun das Objekt `title` der Klasse `CheckboxGroup` vor. In der Methode `UserFrame()` sind folgende Zeilen neu und interessant:

---

<sup>1</sup> Im Sinne von: wahr oder nicht wahr, selektiert oder nicht selektiert.

```
title = new CheckboxGroup();  
place(new Checkbox("Mr", true, title), 30, 30, 40, 20);  
place(new Checkbox("Mrs", false, title), 80, 30, 40, 20);  
place(new Checkbox("Ms", false, title), 130, 30, 40, 20);
```

Die erste Anweisung generiert eine Instanz der Klasse `CheckboxGroup` und bezeichnet diese mit `title`, damit wir nun fortan diese Instanz ansprechen können. In den folgenden drei Zeilen werden mit der uns bereits vertrauten Methode `place()` die in Abbildung 2-5 ersichtlichen Checkboxes platziert. Beim ersten Parameter der Methode `place()` handelt es sich ja um die zu positionierende Komponente. Schauen wir uns den ersten Parameter des ersten Aufrufs noch ein wenig genauer an:

```
new Checkbox("Mr", true, title)
```

Es handelt sich hier erneut um einen Methodenaufruf. Dieser liefert ein Objekt der Klasse `Checkbox` zurück, welches die Beschriftung (engl. label) "Mr" hat, zu Beginn des Programms selektiert ist und der `CheckboxGroup` `title` zugeordnet ist. Würde der zweite Parameter in obigem Aufruf `false` lauten, wäre die `Checkbox` nicht selektiert.

Der letzte Unterschied schlussendlich im Vergleich zur Version 1 ist in der Methode `actionPerformed()` zu finden, nämlich im ersten Aufruf von `println()`. Dort wird nun neu der Text

```
title.getSelectedCheckbox().getLabel()
```

durch ein Leerzeichen getrennt vor den Namen der Person gestellt.

Der `CheckboxGroup` `title` wird die Botschaft `getSelectedCheckbox()` gesandt. Daraufhin erhält man nun diejenige Instanz der Klasse `Checkbox`, die zur Zeit selektiert ist. Dieser selektierten Instanz der Klasse `Checkbox` wird nun die Botschaft `getLabel()` geschickt, worauf diese ihre Beschriftung zurückgibt. Die obigen zwei ineinandergeschachtelten Methodenaufrufe müssen also – wie immer – von links nach rechts gelesen werden.

## 2.3 Vererbung, Konstruktor und Redefinition

Erneut wurde das Programm „Business Cards - Version 2“ zu einer Version 3 erweitert. Hierbei sind die Neuerungen von Version 2 zu Version wiederum 3 **fett** hervorgehoben.

```
import java.awt.*;  
import java.awt.event.*;  
  
public class Card extends Frame {  
    private String title, firstName, familyName, street, zipCode, city;
```

```
public Card(String title, String firstName, String familyName,
            String street, String zipCode, String city) {
    this.title = title;
    this.firstName = firstName;
    this.familyName = familyName;
    this.street = street;
    this.zipCode = zipCode;
    this.city = city;
    setTitle(firstName);
    setSize(240,160);
    setResizable(false);
    setVisible(true);
}

public void paint(Graphics g) {
    g.setFont(new Font("Helvetica",Font.PLAIN,12));
    g.drawString(title+" "+firstName+" "+familyName,60,60);
    g.drawString(street,60,75);
    g.setFont(new Font("Helvetica",Font.BOLD,12));
    g.drawString(zipCode+" "+city,60,90);
}
}

public class UserFrame extends Frame implements ActionListener{
    private TextField firstName, familyName, street, zipCode, city;
    private CheckboxGroup title;

    private void place(Component comp,int x,int y,int width,int height) {
        comp.setBounds(x, y, width, height);
        add(comp);
    }

    public UserFrame() {
        Button button;
        setTitle("Business Cards");
        setLayout(null);
        setSize(200,300);
        setResizable(false);
        title = new CheckboxGroup();
        place(new Checkbox("Mr",true,title),30,30,40,20);
        place(new Checkbox("Mrs",false,title),80,30,40,20);
        place(new Checkbox("Ms",false,title),130,30,40,20);
        place(new Label("FirstName"),30,50,140,20);
        place(firstName=new TextField(),30,70,140,20);
        place(new Label("FamilyName"),30,100,140,20);
        place(familyName=new TextField(),30,120,140,20);
        place(new Label("Street"),30,150,140,20);
        place(street=new TextField(),30,170,140,20);
        place(new Label("ZipCode"),30,200,60,20);
        place(zipCode=new TextField(),30,220,60,20);
        place(new Label("City"),100,200,70,20);
        place(city=new TextField(),100,220,70,20);
        place(button=new Button("CARD"),70,260,60,20);
        button.addActionListener(this);
        setVisible(true);
    }
}
```



```
public void actionPerformed(ActionEvent event) {  
    Card card = new Card(title.getSelectedCheckbox().getLabel(),  
                          firstName.getText(), familyName.getText(),  
                          street.getText(), zipCode.getText(),  
                          city.getText());  
}  
}  
  
public class TestProg {  
    public static void main(String[] args) {new UserFrame();}  
}
```

### 2.3.1 Zum Programm

Diese Version unterscheidet sich von der vorherigen dadurch, dass anstelle einer Adressliste mit jedem Druck auf den CARD-Button eine Visitenkarte (Abbildung 2-6) generiert wird.

#### Abbildung 2-6: Business Cards - Version 3, Card

Die wesentlichste Veränderung von Version 2 zu Version 3 ist sicherlich, dass es eine zusätzliche, neue Klasse namens `Card` gibt. Eine Instanz dieser Klasse ist in Abbildung 2-6 ersichtlich. Dieser screen shot macht auch deutlich, dass `Card` wieder – wie auch schon `UserFrame` – ein fensterartiger Container ist. In Abschnitt 2.1.1.3 wurde erwähnt, dass es sich bei `UserFrame` um einen speziell für unsere Zwecke massgeschneiderten Frame handelt. Dasselbe gilt auch für `Card`. Was heisst das nun aber genau, dass `UserFrame` bzw. `Card` ein spezieller Frame ist?

### 2.3.1.1 Vererbung

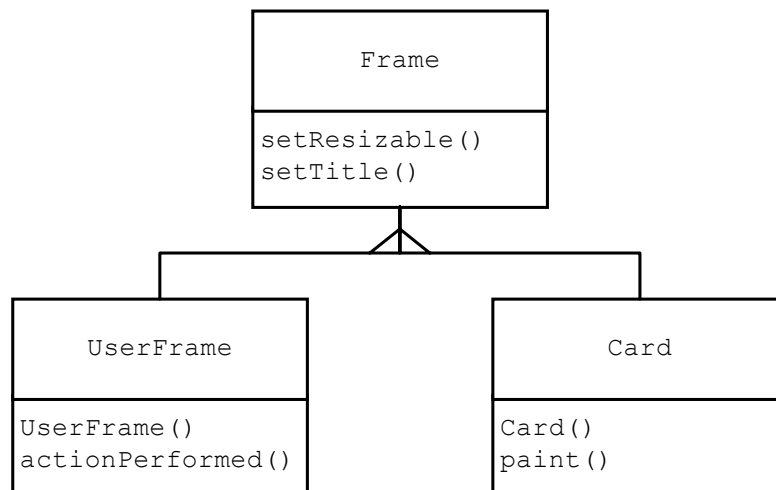
Die Antwort auf obige Frage bringt uns zu einem bereits in Abschnitt 1.1.6 angesprochenen, fundamentalen Prinzip der Objektorientierung: die **Vererbung** (engl. **inheritance**). So wie eine Katze als auch ein Hund ein Säugetier ist, handelt es sich bei `UserFrame` und bei `Card` um einen `Frame`. `UserFrame` und `Card` sind beide Unterklassen von der in der Klassenbibliothek deklarierten Klasse `Frame`. Hierbei erben sie – wie bereits in Abschnitt 1.1.6 erläutert – sämtliche Attribute und Methoden ihrer Oberklasse `Frame`.

```
public class Card extends Frame
```

Das Schlüsselwort **extends** besagt, dass `Card` eine Unterklasse von `Frame` ist.

In Java darf eine Oberklasse zwar mehrere Unterklassen, eine Unterklasse aber nur eine einzige direkte Oberklasse haben. Dies wird als **einfache Vererbung** bezeichnet. Im Gegensatz dazu erlauben andere Programmiersprachen, beispielsweise C++, auch sogenannte **Mehrfachvererbungen** (engl. **multiple inheritance**).

Das **Klassendiagramm**<sup>1</sup> in Abbildung 2-7 illustriert, dass `UserFrame` und `Card` die Methoden `setResizable()` und `setTitle()` von `Frame` erben.



**Abbildung 2-7: Klassenhierarchie von `Frame`, `UserFrame` und `Card`**

---

<sup>1</sup> Die Notationsform von Klassendiagrammen wird in Anhang C erläutert.

Aus Abbildung 2-7 kann man entnehmen, dass einige der im Programm vorkommenden Klassenbibliotheksmethoden wie beispielsweise `setLayout()`, `setSize()`, `setVisible()` nicht in der Klasse `Frame` deklariert sind. Bei den soeben erwähnten Methoden handelt es sich aber dennoch um Methoden, die an `UserFrame` und `Card` vererbt werden. Wie dies möglich ist, zeigt Abbildung 2-8.

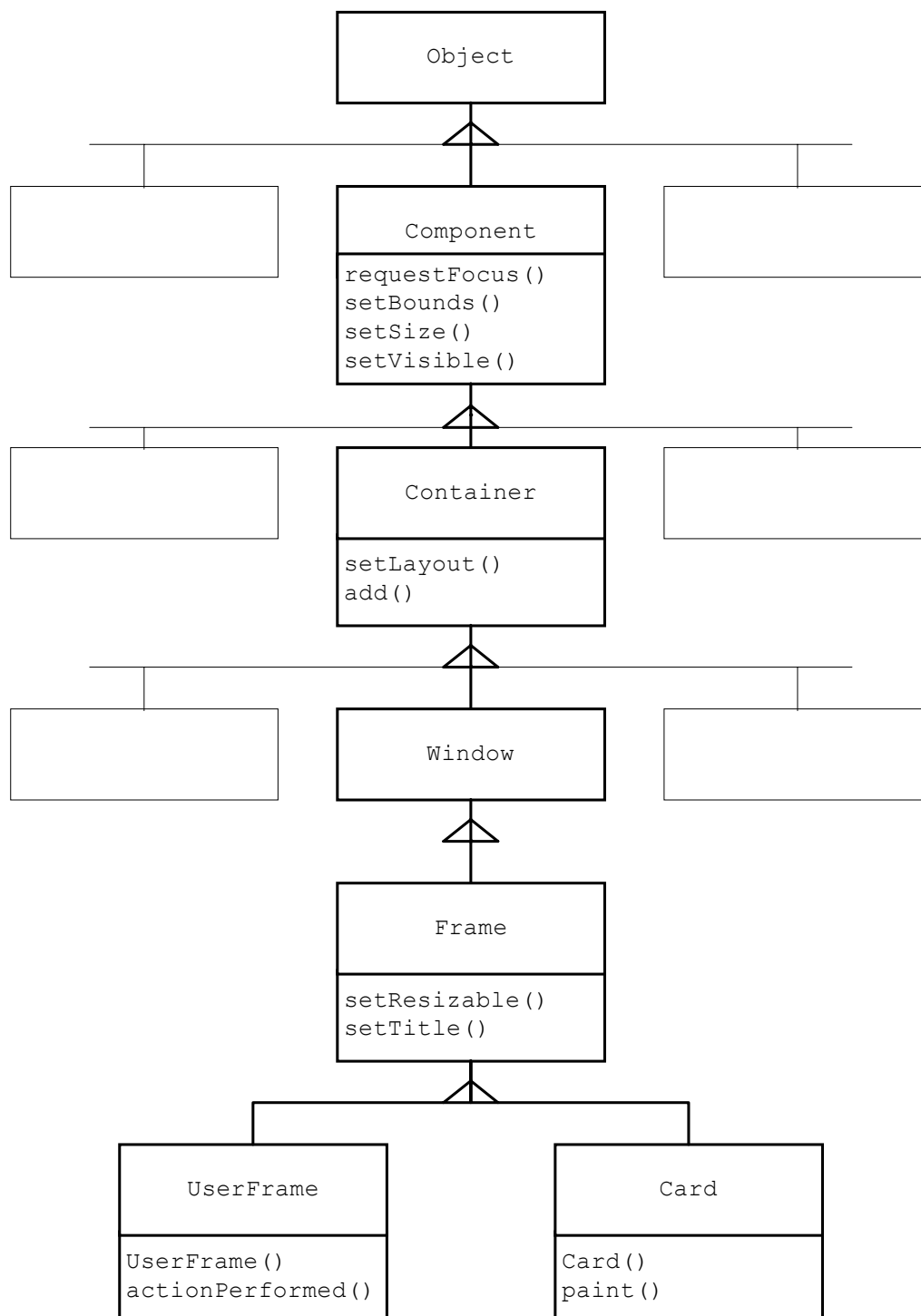
Abbildung 2-8 verdeutlicht, dass `Frame` wiederum eine Unterklasse ist, nämlich eine direkte Unterklasse von `Window` und eine indirekte von `Container`, `Component` und `Object`. Als Unterklasse erbt `Frame` sämtliche Attribute und Methoden von ihren Oberklassen, die sie dann wiederum an ihre Unterklassen weitervererbt: „Eererbtes“ kann also „weitervererbt“ werden.

In Abschnitt 2.1.1 wurde ein `Frame` auch als `Container` bzw. als `Component` bezeichnet. Dies sollte nun aufgrund der in der Abbildung veranschaulichten Vererbungshierarchie klar sein. Aufgrund der Vererbung ist nämlich eine Instanz immer auch eine Instanz einer Oberklasse. Eine Instanz eines `Frame` ist also beispielsweise eine Instanz von `Window`, aber auch eine Instanz von `Container`, `Component` und `Object`. Die umgekehrte Aussage gilt aber nicht: eine Instanz einer Oberklasse ist nicht auch automatisch Instanz deren Unterklassen.

Die Klasse **`Object`**<sup>1</sup> ist die Wurzel der Klassenbibliothekshierarchie. Sie bildet die oberste Hierarchiestufe, von welcher schlussendlich alle Klassen erben. Falls eine durch den Programmierer deklarierte Klasse wie beispielsweise `TestProg` keine Vererbungsangabe macht, ist sie ohne explizite Erwähnung eine direkte Unterklasse von `Object`.

---

<sup>1</sup> Die Namensgebung ist leider sehr unglücklich. `Object` hat nichts mit einer Instanz einer Klasse, also einem Objekt, gemeinsam: `Object` ist eine Klasse.



**Abbildung 2-8: Einbettung von Frame in die Klassenbibliothek**

**Siehe auch:** 2.3.2.1

### 2.3.1.2 Konstruktor

*Johny Gernimwind ist ein kleinstädtischer Theaterregisseur. Seit Kindheit an, nährt er eine hingebungsvolle Liebe für Märchen in seiner Brust. Das Rotkäppchen, welches seine Grossmutter dem bleichen, schüchternen Jungen, der er war, so lebhaft schildern konnte, hatte es ihm besonders angetan. War es nicht Sinnbild der Magie des Bösen, vermischt mit der apokalyptischen Eigendynamik des Guten? Als der konservative Fabrikbesitzer der Kleinstadt nun den provokativen, aber sehr beliebten Regisseur für eine Aufführung eines durch den Fabrikanten finanziell unterstützten Puppentheaters anging – man darf den guten Mann nun aber nicht verdächtigen, sein Wohlwollen gegenüber seinen Arbeiterkindern hätte in irgendeiner Weise etwas mit seiner bevorstehenden Bürgermeisterkandidatur zu tun – war Johny Gernimwind offensichtlich dazu bereit, die Verlockung, auf ein junges, noch formbares Publikum Einfluss zu nehmen, war zu gross. Keine Frage, er entschied sich für das Rotkäppchen.*

*Die Puppen wurden vom ortsansässigen Tischler gefertigt, welcher auch schon die wunderschönen Krippenfiguren für die Kirche des Städtchens geschnitzt hatte. Der Tischler, der nicht nur sein Handwerk verstand, sondern auch ein durchaus tüchtiger Geschäftsmann war, erstellte zuerst für jede Marionette, also für Mutter, Grossmutter, Rotkäppchen, den Jäger und den Wolf, eine Vorlage, mit dieser es ihm dann ein Leichtes sein sollte, mehrere Rotkäppchen oder mehrere Wölfe zu dreheln. Die Wünsche von Johny Gernimwind waren unergründlich, vielleicht wollte er neben einem lächelnden Rotkäppchen auch ein erschreckt weinendes haben. Des weiteren wäre es dem Tischler so auch unschwer möglich, einer plötzlichen Rotkäppchennachfrage weihnachtserhitzter Eltern nachzukommen.*

So wie unser Tischler eine Vorlage für die Erstellung seiner Marionetten einsetzt, braucht man in einem objektorientierten Programm eine „Vorlage“, anhand deren man die Objekte einer Klasse generiert. Bei dieser „Vorlage“ handelt es sich um eine Methode, welche als **Konstruktor** (engl. **constructor**) bezeichnet wird. Ein Konstruktor hat immer denselben Identifier<sup>1</sup> wie die Klasse selbst.

Im Beispielprogramm hat die Klasse `Card` die Attribute `title`, `firstName`, `familyName`, `street`, `zipCode` und `city`. Die Attributwerte der Instanzen der Klasse `Card` sind Texte wie beispielsweise „Mr“, „Wolfgang Amadeus“, „Mozart“, etc. `Card()` ist die Konstruktormethode:

---

<sup>1</sup> Im Unterschied zu den übrigen Methoden wird im Skript also der Name eines Konstruktors mit einem grossen Anfangsbuchstaben geschrieben.

```
public Card(String title, String firstName, String familyName,
            String street, String zipCode, String city) {
    this.title = title;
    this.firstName = firstName;
    this.familyName = familyName;
    this.street = street;
    this.zipCode = zipCode;
    this.city = city;
    setTitle(firstName);
    setSize(240,160);
    setResizable(false);
    setVisible(true);
}
```

Wie man aus der Methodenschnittstelle entnehmen kann, hat `Card()` die sechs Parameter `title`, `firstName`, `familyName`, `street`, `zipCode` und `city`, bei welchen es sich wiederum, wie auch bei den Attributen der Klasse `Card`, um Texte handelt. Dass diese Parameter genau gleich heissen, wie die Attribute der Klasse, hat auf den Programmablauf keinen Einfluss.

Um die Attribute von den gleichlautenden Parametern zu unterscheiden, wird das Schlüsselwort `this` verwendet. Das Schlüsselwort **this** bezeichnet dasjenige Objekt, welches der Empfänger der momentan ausgeführten Methode ist. Im Beispiel referenziert `this` somit diejenige Instanz der Klasse `Card`, die durch den Konstruktor gerade generiert wird.

Betrachten wir die Anweisung

```
this.title = title;
```

Der **'='-Operator** bewirkt eine sogenannte **Zuweisung** (engl. **assignment**): das Attribut `title` des durch `this` referenzierten Objektes erhält den Wert des Parameters `title`; ein allfälliger früherer Wert dieses Attributes wird dabei überschrieben. Die Wertübergabe findet somit **von rechts nach links** statt.

Der Aufruf des Konstruktors `Card()` erfolgt in der Methode `actionPerformed()`:

```
new Card(title.getSelectedCheckbox().getLabel(), firstName.getText(),
        familyName.getText(), street.getText(), zipCode.getText(),
        city.getText());
```

Dieses Beispiel zeigt, wie ein Konstruktor aufgerufen wird: nach dem Schlüsselwort **new** folgt der Konstrukturname mitsamt Parameterliste. Die Parameter enthalten in diesem Beispiel die Texte, welche der Benutzer in die Textfelder der graphischen Benutzeroberfläche eingegeben hat.

Da der Aufruf des Konstruktors in der Methode `actionPerformed()` passiert, wird jedesmal, wenn der Benutzer den Button `PRINT` aktiviert, ein neues Objekt der Klasse `Card` kreiert. Die Objekte der Klasse `Card` werden per Mausklick, dynamisch zur Laufzeit des Programms, erzeugt. Jede Visitenkarte ist eine eigenständige Instanz der Klasse `Card`.

Ein Konstruktor dient dazu, eine Instanz einer Klasse zu erzeugen, indem Speicherplatz für das zu erzeugende Objekt alloziert wird und allfällige Attributwerte initialisiert werden, also einen Anfangswert erhalten. Erst nachdem ein Objekt durch einen Konstruktor erschaffen wurde, existiert es und kann mit anderen Objekten kommunizieren.

**Siehe auch:** 2.3.2.1

### 2.3.1.3 Graphics und Fonts

In den vorangegangenen Programmversionen haben wir die Adresse einer Person jeweils auf die Konsole ausgedruckt. Nun soll sie auf einer Instanz der Klasse `Card` erscheinen, so dass eine Visitenkarte entsteht. Damit wir aber auf ein solches Objekt der Klasse `Card` einen Text schreiben können, braucht der Interpreter Informationen über das Aussehen<sup>1</sup> und die Ausmasse unseres Objektes. Solche graphische Informationen sind für jede Komponente in ihrem dazugehörigen **Graphics** Objekt enthalten. Eine Instanz der in der Klassenbibliothek vordefinierten Klasse `Graphics` liefert den graphischen Kontext einer Komponente.

Die in der Klasse `Card` deklarierte Methode `paint()` erhält als Parameter das zu einer Instanz der Klasse `Card` gehörende `Graphics` Objekt `g`:

```
public void paint(Graphics g) {
    g.setFont(new Font("Helvetica", Font.PLAIN, 12));
    g.drawString(title+" "+firstName+" "+familyName, 60, 60);
    g.drawString(street, 60, 75);
    g.setFont(new Font("Helvetica", Font.BOLD, 12));
    g.drawString(zipCode+" "+city, 60, 90);
}
```

Dem `Graphics` Objekt `g` werden abwechselungsweise die Botschaften `setFont()` und `drawString()` geschickt, welche insgesamt den in Abbildung 2-6 ersichtlichen Ausdruck realisieren. Die Methoden `drawString()` wie auch `setFont()` sind bereits in der Klasse `Graphics` enthalten. Erstere erlaubt das Ausdrucken eines Strings, letztere das Festlegen der zu verwendenden Schriftart.

Der erste Parameter der Methode `drawString()` ist der zu „zeichnende“ Text, die letzten zwei Parameter bestimmen die Positionierung des Strings innerhalb der Komponente, indem die x- bzw. y-Koordinaten der linken, unteren Ecke angegeben werden.

---

<sup>1</sup> Es sei an dieser Stelle daran erinnert, dass die Darstellung graphischer Benutzeroberflächen von Plattform zu Plattform leicht variieren kann: eine Instanz eines Frame hat beispielsweise unter Windows 95 - im Gegensatz zu Windows 3.1 oder Mac - einen Schliess-Button.

Die Methode `setFont()` verlangt als Parameter eine Instanz der Klasse **Font**, welche in der Klassenbibliothek enthalten ist.

Der Konstruktor der Klasse `Font` verlangt drei Parameter. Beim ersten handelt es sich um den Namen der Schriftart. Welche Schriftarten vom System unterstützt werden, ist natürlich plattformspezifisch<sup>1</sup>. Der zweite Parameter gibt den zu verwendenden Stil, also `PLAIN`, `BOLD`, oder `ITALIC` an. Diese Stiltypen sind als Attribute der Klasse `Font` deklariert. Der letzte Parameter bestimmt die Grösse in Punkten.

Sowohl die Klasse `Graphics` als auch `Font` sind Teil des package `awt`.

### 2.3.1.4 *paint() und Redefinition*

Abschliessend wollen wir uns noch eingehender die **paint()** Methode betrachten. Wie aus dem Programm hervorgeht, wird diese Methode nur deklariert, aber nirgends im Programm explizit aufgerufen. Abbildung 2-6 macht jedoch offenkundig, dass die Methode sicherlich für jedes Objekt der Klasse `Card` einmal ausgeführt worden sein muss, ansonsten wäre keine Adresse auf der Visitenkarte vorhanden. Der Aufruf der Methode `paint()` ist nicht in der Kontrolle des Programmierers, sondern in der des Systems. Dieses ruft die Methode auf, wenn eine `Component` das erste mal gezeichnet wird oder erneut gezeichnet werden muss. Letzteres ist beispielsweise der Fall, wenn der Benutzer die Grösse einer `Component` verändert oder wenn eine `Component` vorübergehend nicht sichtbar war, weil sie von einer anderen Komponente verdeckt wurde.

Wie man auch aus der Klassenbibliothek entnehmen kann, ist die `paint()` Methode bereits in der Klasse `Container` deklariert. Die neuerliche Vereinbarung dieser Methode in der Klasse `Card` wird als Redefinition bezeichnet. Unter **Redefinition** versteht man das Erweitern oder Neudefinieren einer ererbten Methode, indem man deren Methodenschnittstelle zwar beibehält, jedoch deren Rumpf erweitert bzw. neu definiert. Das Überschreiben des Rumpfes der ererbten Methode ist als **method overriding** bekannt.

Nebst der Eigenschaft, dass die Methode `paint()` immer dann durch das System aufgerufen wird, wenn eine `Component` aufdatiert werden muss, hat sie noch einen weiteren Vorteil: sie liefert ohne Aufruf eines Konstruktors das benötigte Objekt der Klasse `Graphics`.

---

<sup>1</sup> Der Aufruf `Toolkit.getDefaultToolkit().getFontList()` liefert eine Liste derjenigen Schriften, welche standardmässig von jedem System unterstützt werden sollten.



## 2.3.2 Syntax

### 2.3.2.1 Vererbung

Dass eine Klasse **Unterklasse** (engl. **subclass**) einer anderen Klasse ist, deklariert man mit dem Schlüsselwort **extends**. Hierbei folgt nach **extends** der Name der jeweiligen **Oberklasse** (engl. **superclass**).

SubclassDeclaration
<pre>class Identifier extends SuperclassIdentifier{     AttributeDeclarations     MethodDeclarations }</pre>

Beispiel: Klasse UserFrame im Beispielprogramm  
„Business Cards - Version 3“

Java erlaubt nur eine **einfache Vererbung**: eine Unterklasse darf nur eine direkte Oberklasse haben.

### 2.3.2.2 Konstruktor

Der Name eines **Konstruktors** (engl. **constructor**) ist immer derselbe wie derjenige der Klasse, in welcher er deklariert ist. Das Schlüsselwort **public**<sup>1</sup> ist zwingend.

ConstructorDeclaration
<pre>public ClassIdentifier() {     Statements }</pre>

Beispiele: Card() und UserFrame() im Beispielprogramm  
„Business Cards - Version 3“

Ein Konstruktor wird mit dem Schlüsselwort **new** aufgerufen.

---

<sup>1</sup> Auf das Schlüsselwort **public** wird in Abschnitt 6.2.1.3 und 6.2.2.1 eingegangen.

<b>ConstructorCall</b>
<b>new</b> ClassIdentifier();

Beispiel:    `new UserFrame();`

## 2.4 Instanzvariablen und Klassenvariablen

Hier nun die vierte und letzte Version der Programmreihe „Business Cards“. Die Änderungen sind wiederum **fett** markiert.

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

public class Card extends Frame {
    private String title, firstName, familyName, street, zipCode, city;
    private static int cardCount = 0;

    public Card(String title, String firstName, String familyName,
                String street, String zipCode, String city) {
        this.title = title;
        this.firstName = firstName;
        this.familyName = familyName;
        this.street = street;
        this.zipCode = zipCode;
        this.city = city;
        setTitle(firstName);
        setBounds(200+20*cardCount, 0+20*cardCount, 240, 160);
        cardCount++;
        setResizable(false);
        setVisible(true);
    }

    public void paint(Graphics g) {
        g.setFont(new Font("Helvetica", Font.PLAIN, 12));
        g.drawString(title+" "+firstName+" "+familyName, 60, 60);
        g.drawString(street, 60, 75);
        g.setFont(new Font("Helvetica", Font.BOLD, 12));
        g.drawString(zipCode+" "+city, 60, 90);
    }
}

public class UserFrame extends Frame implements ActionListener {
    private TextField firstName, familyName, street, zipCode, city;
    private CheckboxGroup title;
```

```
private void place(Component comp,int x,int y,int width,int height) {
    comp.setBounds(x, y, width, height);
    add(comp);
}

public UserFrame() {
    Button button;
    setTitle("Business Cards");
    setLayout(null);
    setSize(200,300);
    setResizable(false);
    title = new CheckboxGroup();
    place(new Checkbox("Mr",true,title),30,30,40,20);
    place(new Checkbox("Mrs",false,title),80,30,40,20);
    place(new Checkbox("Ms",false,title),130,30,40,20);
    place(new Label("FirstName"),30,50,140,20);
    place(firstName=new TextField(),30,70,140,20);
    place(new Label("FamilyName"),30,100,140,20);
    place(familyName=new TextField(),30,120,140,20);
    place(new Label("Street"),30,150,140,20);
    place(street=new TextField(),30,170,140,20);
    place(new Label("ZipCode"),30,200,60,20);
    place(zipCode=new TextField(),30,220,60,20);
    place(new Label("City"),100,200,70,20);
    place(city=new TextField(),100,220,70,20);
    place(button=new Button("PRINT"),30,260,60,20);
    button.addActionListener(this);
    place(button=new Button("CARD"),110,260,60,20);
    button.addActionListener(this);
    setVisible(true);
}

public void actionPerformed(ActionEvent event){
    if (event.getActionCommand().equals("PRINT")) {
        System.out.println(title.getSelectedCheckbox().getLabel()+" "+
            firstName.getText()+" "+
            familyName.getText());
        System.out.println(street.getText());
        System.out.println(zipCode.getText()+" "+city.getText());
        System.out.println();
    } else if (event.getActionCommand().equals("CARD")) {
        Card card = new Card(title.getSelectedCheckbox().getLabel(),
            firstName.getText(), familyName.getText(),
            street.getText(), zipCode.getText(),
            city.getText());
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 2.4.1 Zum Programm

Wenn man das Programm ausführt, zeigt sich, dass man neu die Möglichkeit hat, zwischen zwei Alternativen auszuwählen: man kann entweder durch Aktivieren des Buttons PRINT die Adresse auf der Konsole ausdrucken lassen oder mittels CARD eine Visitenkarte generieren. In der vierten Version sind gewissermassen die beiden vorangehenden Versionen vereint. Eine weitere Erneuerung ergibt sich auch in der Darstellung mehrerer Visitenkarten: sie werden leicht versetzt, in einer nach rechts unten verlaufenden Diagonale ausgegeben.

#### 2.4.1.1 Selektion

Die **Selektion** (siehe Abschnitt 1.2.2), entweder die Adresse auf der Konsole auszudrucken oder eine Visitenkarte zu erzeugen, erfolgt durch ein **if statement** in der Methode `actionPerformed()`:

```
if (event.getActionCommand().equals("PRINT")) {  
    .  
    .  
    .  
} else if (event.getActionCommand().equals("CARD")) {  
    .  
    .  
    .  
}
```

In der Methode `actionPerformed()` wird zuerst mittels

```
if (event.getActionCommand().equals("PRINT"))
```

die Bedingung überprüft, ob der PRINT-Button aktiviert wurde. Falls dies zutrifft, werden die darauffolgenden, im geschweiften Klammerpaar stehenden Anweisungen ausgeführt. Ansonsten werden diese übersprungen und die nach dem Schlüsselwort **else** in geschweiften Klammern stehenden statements ausgeführt.

```
if (event.getActionCommand().equals("CARD"))
```

Nach dem Schlüsselwort **else** wird erneut eine Bedingung, nämlich ob der Button CARD aktiviert wurde, überprüft. Trifft dies zu, werden die folgenden, in geschweiften Klammern stehenden Anweisungen ausgeführt. Andernfalls passiert gar nichts.

Die in den beiden geschweiften Klammerpaaren stehenden Anweisungen sind nicht unbekannt. Die ersten realisieren den bereits aus Programmversion zwei bekannten Ausdruck der Adresse auf die Konsole, die zweiten stimmen mit dem in Programmversion drei eingesetzten Aufruf des Konstruktors `Card()` überein.

Damit überhaupt gewährleistet ist, dass jedesmal, wenn der Button PRINT oder der Button CARD aktiviert wird, auch die Methode `actionPerformed()` aufgerufen wird, dürfen natürlich die folgenden Zeilen im Programm nicht fehlen:

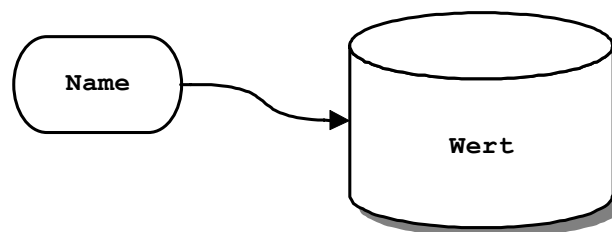
```
place(button=new Button("PRINT"), 30, 260, 60, 20);  
button.addActionListener(this);  
place(button=new Button("CARD"), 110, 260, 60, 20);  
button.addActionListener(this);
```

### 2.4.1.2 Instanzvariablen und Klassenvariablen

Bis anhin haben wir die Attribute einer Klasse nicht weiter unterschieden. Nun ist es aber an der Zeit, dies zu tun: man kann die Attribute in die sogenannten **Instanzvariablen** und **Klassenvariablen** unterteilen. Erstere sind dynamisch jedem einzelnen Objekt zugeordnet, letztere existieren statisch für die gesamte Klasse, d.h. ein einziges Mal.

Bevor wir nun weiter auf diese Thematik eingehen, soll kurz auf den Begriff Variable eingegangen werden.

Wie aus der Mathematik bekannt, ist eine **Variable** ein Datenbehälter mit einem Namen. Der Wert einer Variablen, also der Inhalt dieses „Datenbehälters“, kann durch eine Zuweisung verändert werden. Abbildung 2-9 soll dies veranschaulichen.



**Abbildung 2-9: Anschauliche Darstellung einer Variablen**

Auf Ebene der virtuellen Maschine handelt es sich beim Namen um eine Adresse des Speicherbereichs und beim Wert um den an dieser Adresse vorgefundenen Inhalt des Speichers.

Im Gegensatz zu den vorangehenden Programmversionen, bei welchen alle Attribute **Instanzvariablen** waren, kommt in diesem Programm zum ersten Mal eine **Klassenvariable** vor. In der Klasse `Card` wird die Klassenvariable `cardCount` deklariert, welche als Wert eine ganze Zahl enthalten kann und mit der Zahl Null initialisiert wird:

```
private static int cardCount = 0;
```

Das Schlüsselwort **static** macht klar, dass es sich um eine Klassenvariable handelt. Im Gegensatz dazu sind sämtliche andere Attribute der Klasse `Card` und auch der Klasse `UserFrame` Instanzvariablen, da sie kein `static` vorangestellt haben.

Schauen wir uns zuerst einmal an, wofür wir die Klassenvariable `cardCount` überhaupt benötigen. Sie wird allein in den zwei Zeilen

```
setBounds(200+20*cardCount, 0+20*cardCount, 240, 160);  
cardCount++;
```

verwendet. `setBounds()` ist eine Methode der Klasse `Component`. Ähnlich wie die Methode `setSize()` ermöglicht sie das Festsetzen der Breite und Höhe (letzte zwei Parameter) einer `Component`. Im Unterschied zu dieser kann man aber mittels `setBounds()` die `Component` zusätzlich noch in ihrer Lage verschieben, indem man die Koordinaten der linken oberen Ecke (erste zwei Parameter) angeben kann<sup>1</sup>. Die Anweisung `cardCount++` bewirkt, dass der Wert der Klassenvariablen `cardCount` um eins erhöht wird.

Wenn das erste Mal der Konstruktor `Card()` aufgerufen wird, hat `cardCount` den Wert Null: die Klassenvariable wurde ja bei ihrer Deklaration gerade damit initialisiert. Die erste Instanz der Klasse `Card` wird also an der Stelle  $x = 200 + 20 \cdot 0 = 200$  und  $y = 0 + 20 \cdot 0 = 0$  positioniert. Da nach dem Aufruf von `setBounds()` der Wert von `cardCount` um eins erhöht wird, wird die als zweite generierte Instanz an der Stelle  $x = 200 + 20 \cdot 1 = 220$  und  $y = 0 + 20 \cdot 1 = 20$  platziert. Weil `cardCount` mit jedem Aufruf des Konstruktors um eins erhöht wird und somit die x- bzw. y-Koordinaten jeweils um 20 anwachsen, erscheinen die Visitenkarten auf dem Bildschirm leicht versetzt.

Die Variable `cardCount` ist also ein Zähler, welcher die Anzahl der Instanzen der Klasse `Card` festhält. Was bedeutet dies nun, dass `cardCount` eine Klassenvariable ist?

Die Klasse `Card` hat die Instanzvariablen `title`, `firstName`, `familyName`, `street`, `zipCode` und `city`, welche alle als Wert einen Text haben. Es ist klar, dass die Werte dieser Variablen von Instanz zu Instanz unterschiedlich ausfallen können, da man ja mehrere Personen erfassen und deren Visitenkarten ausgeben kann: so gibt es beispielsweise den Wolfgang Amadeus Mozart, aber auch seine Frau Konstanze Mozart. Bei diesen Variablen handelt es sich um Attribute, die gewissermassen eine einzelne Instanz beschreiben und somit mit einer Instanz assoziiert sind, weshalb man sie als Instanzvariablen bezeichnet.

---

<sup>1</sup> Die Methode `setBounds()` wird neben der Methode `add()`, welche eine Methode der Klasse `Container` ist, auch in der Methode `place()` verwendet. Dort wird zuerst eine `Component` mittels `setBounds()` relativ positioniert und danach mittels `add()` dem `Container`, also einer Instanz der Klasse `UserFrame`, hinzugefügt.

Im Gegensatz dazu ist der Wert der Variable `cardCount` nicht von Instanz zu Instanz unterschiedlich, er beschreibt auch in keiner Weise eine einzelne Instanz, hat also im Prinzip gar nichts mit den jeweiligen Instanzen der Klasse `Card` zu tun. Die Variable `cardCount` ist ein Attribut der Klasse `Card`, welches angibt, wieviele Instanzen die Klasse momentan hat. Klassenvariablen sind also mit der Klasse selbst assoziiert.

Für jedes Objekt muss im Speicher Platz für die Instanzvariablen bzw. deren Werte angelegt werden – die Werte der Instanzvariablen können nämlich von Instanz zu Instanz variieren<sup>1</sup>. Im Gegensatz dazu braucht man für Klassenvariablen nur einmal Platz im Speicher anzulegen, also nicht pro Instanz, da die Klassenvariable ja für die Klasse selbst und nicht für ein einzelnes Objekt steht.

## 2.5 Zusammenfassung

In diesem Kapitel wurde der grobe Aufbau einer **Application** aufgezeigt; neben `import` statements und Klassendeklarationen enthält sie immer auch eine `main()` Methode, welche für die virtuelle Maschine den Einstieg ins Programm markiert.

Eine **Klasse** ihrerseits umfasst Attribute und Methoden. Die Vererbungshierarchie von Klassen kann man mit dem Schlüsselwort `extends` ausdrücken. Für die Erzeugung von Objekten deklariert eine Klasse einen Konstruktor.

Attribute unterteilen sich in **Instanzvariablen** und **Klassenvariablen**. Erstere sind dynamisch jedem einzelnen Objekt zugeordnet, letztere existieren statisch für jede Klasse und nur ein einziges Mal. Die Vergabe von Identifiers erlaubt es, auf eine Instanzvariable bzw. Klassenvariable zuzugreifen.

Eine **Variable** ist eine Datenbehälter, auf den mittels eines Namens zugegriffen wird. Der Inhalt dieses Datenbehälters, also der Variablenwert, kann durch eine Zuweisung verändert werden.

Bei einer **Methodendeklaration** unterscheidet man die Methodenschnittstelle (Kopf, Signatur) vom Rumpf der Methode. Die Signatur gibt den Identifier der Methode an und listet in runden Klammern den Typ und die Namen ihrer Parameter auf. Der durch geschweifte Klammern markierte Methodenrumpf enthält die Anweisungen, welche bei einem **Methodenaufruf** ausgeführt werden. Beim Aufruf einer Methode wird ihr Name und der Empfänger gesetzt. Als Parameter werden konkrete Werte übergeben.

---

<sup>1</sup> Natürlich kann es auch vorkommen, dass bei Objekten einzelne Werte ihrer Instanzvariablen übereinstimmen, was bei Wolfgang und Konstanze der Fall ist, da beide denselben Nachnamen haben. Ausschlaggebend ist die Tatsache, dass es überhaupt unterschiedliche Werte geben kann. Der Name Mozart wird - obwohl zweimal gleichlautend - auch zweimal physisch gespeichert.

Eine ererbte Methode kann in einer Unterklasse auch **redefiniert** werden, indem man die Signatur der Methode zwar beibehält, jedoch ihren Rumpf ändert. Da man hierdurch den Methodenrumpf überschreibt, spricht man auch von `method overriding`.

Die im package `awt` deklarierte Klasse **Graphics** ermöglicht die graphische Manipulation von Komponenten. Die Methode `paint()` erhält eine Instanz dieser Klasse, welche in sich den graphischen Kontext einer Komponente birgt, als Parameter. Die in der Klasse `Component` deklarierte Methode `paint()` wird vom System automatisch immer dann aufgerufen, wenn die Darstellung einer Komponente auf dem Bildschirm aufdatiert werden muss.

Es wurden die folgenden Bausteine von graphischen Benutzeroberflächen eingeführt:

- ◆ `TextField`, `Label`, `Button` und `Frame`
- ◆ `Component` und `Container`
- ◆ `Checkbox` und `CheckboxGroup`



# 3 Einfache Datentypen Operatoren Programmstrukturen

In diesem Kapitel wird ein grosser Teil der Syntax von Java behandelt. Es werden die einfachen Datentypen eingeführt sowie elementare Programmstrukturen erläutert. Zusätzlich werden gewisse, im vorangehenden Kapitel besprochene Konzepte nochmals aufgegriffen, um hinsichtlich ihrer Syntax eingehender beleuchtet zu werden.

## 3.1 int, boolean und Programmstrukturen

Untenstehendes Programm „Multiple Choice“ ist wiederum eine Application, bei welcher ganze Zahlen in Interaktion mit dem Benutzer auf ihre Eigenschaften hin, wie Geradzahligkeit etc., überprüft werden. Die zu untersuchenden Zahlen werden mit einem Zufallszahlengenerator erzeugt.

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

public class Number {
    private int value;           //instance variable 1<=value<=99
    private Random rndm;         //random generator

    public Number() {            //constructor
        rndm = new Random();
    }

    public void nextValue() {     //change value randomly
        value = Math.abs(rndm.nextInt())%99+1;
    }

    public String toString() {   //return value as string
        return String.valueOf(value);
    }
}
```

### 3.1 int, boolean und Programmstrukturen

---

```
public boolean isOdd() {                //return true if value is odd
    return (value%2!=0);
}

public boolean hasSameDigits() {        //return true if digits are equal
    return (value/10==value%10);
}

public boolean isASquare() {            //return true if value is a square
    int i,j;
    for(i=1,j=1;i<value;i=i+j) {j=j+2;}
    return (value==i);
}
}

public class UserFrame extends Frame implements ActionListener {
    private Number number;
    private Label message;
    private Checkbox odd, sameDigits, square;

    private void place(Component comp,int x,int y,int width,int height) {
        comp.setBounds(x, y, width, height);
        add(comp);
    }

    public UserFrame() {                //constructor
        Button button;
        setTitle("Multiple Choice");
        setLayout(null);
        setSize(200,200);
        setResizable(false);
        place(message=new Label(),40,30,150,20);
        place(odd=new Checkbox("is odd"),40,60,120,20);
        place(sameDigits=new Checkbox("has same digits"),40,80,120,20);
        place(square=new Checkbox("is a square"),40,100,120,20);
        place(button=new Button("DONE"),30,170,40,20);
        button.addActionListener(this);
        place(button=new Button("NEXT"),80,170,40,20);
        button.addActionListener(this);
        place(button=new Button("CHECK"),130,170,40,20);
        button.addActionListener(this);
        setVisible(true);
        number = new Number();
        number.nextValue();
        message.setText("Check if "+number.toString());
    }

    public void actionPerformed(ActionEvent event){
        if (event.getActionCommand().equals("DONE")) {
            int errorCount=0;
            if (odd.getState()^number.isOdd()) errorCount++;
            if (sameDigits.getState()^number.hasSameDigits()) errorCount++;
            if (square.getState()^number.isASquare()) errorCount++;
            if (errorCount==0)
                message.setText("All answers are correct!");
            else if (errorCount==1)
                message.setText("There is one error!");
            else
                message.setText("There are more than one errors!");
        }
    }
}
```

```
        message.setText("There are "+errorCount+" errors!");
    } else if (event.getActionCommand().equals("NEXT")) {
        number.nextValue();
        message.setText("Check if "+number.toString());
        odd.setState(false);
        sameDigits.setState(false);
        square.setState(false);
    } else if (event.getActionCommand().equals("CHECK")) {
        message.setText(number.toString()+" has the properties");
        odd.setState(number.isOdd());
        sameDigits.setState(number.hasSameDigits());
        square.setState(number.isASquare());
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 3.1.1 Zum Programm

Bevor wir uns nun eingehend mit dem Programmtext beschäftigen, schauen wir uns zunächst einmal an, wie sich das Programm überhaupt dem Benutzer präsentiert:

**Abbildung 3-1: Multiple Choice, UserFrame**

Das Programm generiert zufällig eine ganze, positive Zahl, welche der Benutzer dann darauf hin überprüfen muss, ob sie ungerade ist (is odd), gleiche Ziffern hat (has same digits) oder das Quadrat einer ganzen Zahl ist (is a square). Im obigen Beispiel wird die Zahl 11 untersucht. Der Benutzer kann nun die zutreffenden Eigenschaften ankreuzen, indem er die entsprechenden Checkboxes selektiert. Wenn er den Button DONE aktiviert, wird angezeigt, ob seine Antwort korrekt ist (siehe Abbildung 3-2).

#### **Abbildung 3-2: Multiple Choice, UserFrame**

Wenn der Benutzer den Button NEXT selektiert, wird eine neue Zufallszahl generiert, sagen wir die Zahl 4. Falls man nun die Eigenschaften von 4 in Erfahrung bringen möchte, ohne selber nachzurechnen, genügt es, den CHECK-Button zu aktivieren, die Lösung wird, wie in Abbildung 3-3 ersichtlich, präsentiert.

### Abbildung 3-3: Multiple Choice, UserFrame

Wenden wir uns nun dem Programm zu, um einen ersten Überblick zu erhalten:

Von Interesse für uns sind noch die Klassen `Number` und `UserFrame`. Hierbei ist `UserFrame` wiederum diejenige Klasse, welche für die Erzeugung der graphischen Benutzeroberfläche zuständig ist. Die Klasse `Number` hingegen ist einerseits für die Erzeugung der Zufallszahlen verantwortlich, andererseits deklariert sie Methoden, mit deren Hilfe man die generierten Zahlen auf die besagten Eigenschaften hin überprüfen kann.

In der Klassenbibliothek von Java im package `util` gibt es bereits ein Klasse `Random`, mit der man Zufallszahlen erzeugen kann. Wie aus der Attributdeklaration

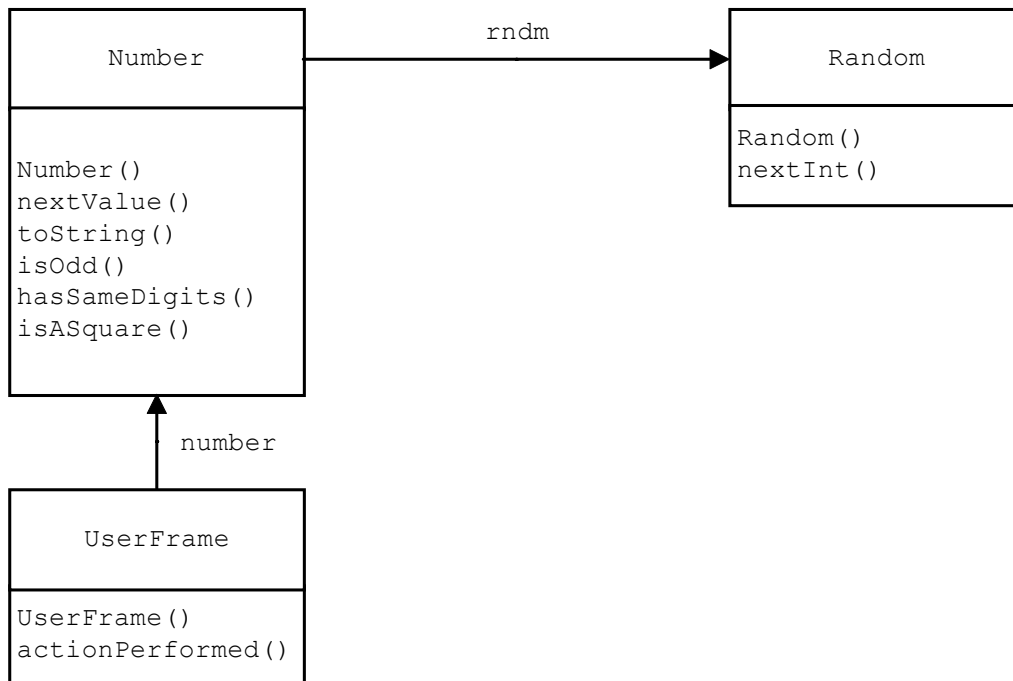
```
private Random rndm;
```

der Klasse `Number` hervorgeht, verfügt die Klasse `Number` über eine Instanz der Klasse `Random`. Die Klasse `Number` kann also Methoden der Klasse `Random` verwenden, indem sie an die Instanz `rndm` die entsprechenden Botschaften versendet. Grundlage für die Zusammenarbeit der beiden Klassen ist natürlich, dass das package `util` überhaupt importiert wird.

```
private Number number;
```

Auch zwischen Objekten der Klasse `Number` und `UserFrame` werden Botschaften ausgetauscht. `UserFrame` verfügt zu diesem Zweck über eine Instanz der Klasse `Number` namens `number`.

Abbildung 3-4 veranschaulicht mittels eines Klassendiagramms die Art und Weise der Zusammenarbeit und zeigt die wichtigsten Methoden.



**Abbildung 3-4: Zusammenarbeit der Klassen Number, Random und UserFrame**

#### 3.1.1.1 Kommentar

Was bei einem ersten Blick auf das Programm auffällt, sind die Texte, welchen das Zeichen `//` vorangeht. Dabei handelt es sich um **Kommentar**. Alles, beginnend mit dem Zeichen `//` bis zum Ende der Zeile wird vom Interpreter ignoriert und somit auch nicht ausgeführt. Der Kommentar ist allein für den Programmierer und den Leser des Programms bestimmt. Er enthält knappe, aber treffende Bemerkungen und Erläuterungen zum Programmverlauf. Das Verwenden von geeigneten Kommentaren leistet einen wesentlichen Beitrag zur Verständlichkeit eines Programms<sup>1</sup>.

**Siehe auch:** 3.1.2.1

---

<sup>1</sup> Siehe Anhang D: Prinzipien guten Programmierens.

### 3.1.1.2 Datentypen

In der Klasse `Number` werden die Instanzvariablen

```
private int value;  
private Random rndm;
```

deklariert. Bei der Instanzvariablen `rndm` handelt es sich um die bereits erwähnte Instanz der Klasse `Random`. Die Instanzvariable `value` hingegen ist vom Typ `int`.

Bis zu diesem Zeitpunkt haben wir das Konzept der Variablen kennengelernt. Wie in Abschnitt 2.4.1.2 erläutert, hat eine Variable einen Namen und beinhaltet einen Wert. Einer Variablen wird zusätzlich ein **Datentyp** zugeordnet, welcher einen Wertebereich definiert, in welchem die Werte einer Variablen dieses Typs liegen dürfen sowie die Operationen, die für diese Werte verwendet werden können.

Auf Ebene des Computers wird ein Datentyp dazu verwendet, Variableninhalte, also binäre Daten, richtig zu interpretieren.

In Java gibt es Datentypen für Zahlen, Zeichen und Wahrheitswerte. Bei den Strings und Klassen handelt es sich auch um Datentypen. Sie werden als sogenannte **komplexe Datentypen** (engl. **reference data types**) bezeichnet und unterscheiden sich in gewisser Weise von den sogenannten **einfachen Datentypen** (engl. **primitive data types**), worauf noch in Abschnitt 4.2 ausführlicher eingegangen wird.

In der Attributdeklaration wird der Datentyp der Variablen ihrem Bezeichner vorangestellt.

**Siehe auch:** 3.1.2.2, 3.1.2.3, 3.1.2.4, 3.1.2.5

### 3.1.1.3 Der Datentyp `int`

Wie wir bereits in Abschnitt 3.1.1.2 erfahren haben, ist die Instanzvariable `value` der Klasse `Number` vom Datentyp `int`.

Der Datentyp `int` umfasst die negativen und positiven **ganzen Zahlen** im Wertebereich von **-2147483648 bis 2147483647**. Er beinhaltet also nur eine Teilmenge der ganzen Zahlen. Dies rührt daher, dass in Java intern für die Darstellung einer Zahl vom Typ `int` **32 bit** verwendet werden. Mit 32 bit können wir  $2^{32} = 4294967296$  verschiedene Werte darstellen; so beispielsweise die positiven ganzen Zahlen von 0 bis 4294967295 oder, wenn man davon ausgeht, dass man ein bit für die Darstellung des Vorzeichens verwendet, die positiven und negativen ganzen Zahlen von -2147483648 bis 2147483647.

Nebst den arithmetischen Operationen und Vergleichsoperationen gibt es auch Methoden, die mit `int`-Werten Berechnungen ausführen.

**Siehe auch:** 3.1.2.8

### 3.1.1.4 Arithmetische Operatoren und Zuweisung

Dringen wir nun tiefer in die Klasse `Number` ein: In ihrem Konstruktor `Number()` wird eine Instanz der Klasse `Random` generiert.

#### Die Methode

```
public void nextValue() {  
    value = Math.abs(rndm.nextInt())%99+1;  
}
```

weist der Instanzvariablen `value` eine positive ganze Zahl zwischen 1 und 99 zu.

#### Im Ausdruck

```
Math.abs(rndm.nextInt())%99+1
```

findet man einerseits einen Methodenaufruf (`Math.abs(rnd.nextInt())`) vor, welcher als Parameter erneut einen Methodenaufruf (`rndm.nextInt()`) beinhaltet, und andererseits die **arithmetischen Operatoren** `%` und `+`. Wie in der Mathematik, sind auch arithmetische Operatoren mit **Prioritäten** verknüpft, es ist geregelt, welche Operation vor einer anderen ausgeführt werden muss. So hat beispielsweise der sogenannte **Modulo-Operator** (`%`) eine höhere Priorität als die **Addition** (`+`). Wenn wir die Prioritäten durch Setzen von Klammern verdeutlichen, erhalten wir folgenden Ausdruck:

```
((Math.abs(rndm.nextInt())%99)+1)
```

Als erstes wird der Methodenaufruf `Math.abs(rndm.nextInt())` ausgeführt. Da dieser als Parameter wiederum einen Methodenaufruf beinhaltet, wird schlussendlich der Methodenaufruf `rndm.nextInt()` als allererste Aktion gestartet. Dem Objekt `rndm` wird die Botschaft `nextInt()` gesendet, bei welcher es sich um eine in der Klasse `Random` deklarierte Methode handelt, welche zufällig eine ganze Zahl vom Typ `int` generiert. Diese `int`-Zahl wird nun als Parameter der Methode `abs()` übergeben. Die in der Klasse `Math` aufgeführte Methode, liefert den Absolutbetrag des ihr als Parameter übergebenen `int`-Wertes zurück. Die beiden Methodenaufrufe liefern also schlussendlich eine zufällig generierte, positive, ganze Zahl.

Der Modulo-Operator liefert den Rest, der bei einer Division zweier ganzzahligen Zahlen anfällt.

```
Math.abs(rndm.nextInt())%99
```

Wenn wir also beispielsweise `103 % 99` rechnen, erhalten wir 4, da 103 geteilt durch 99 eins Rest vier ergibt. Die Resultate von einer positiven, ganzen Zahl `% 99` bewegen sich also in den Schranken von 0 bis 98.



```
((Math.abs(rndm.nextInt())%99)+1)
```

Da man nun zu dieser positiven Zahl, welche zwischen 0 und 98 liegt, 1 addiert, wird der Instanzvariablen `value` in der Anweisung

```
value = Math.abs(rndm.nextInt())%99+1;
```

ein Wert zwischen 1 und 99 zugewiesen.

**Siehe auch:** 3.1.2.6, 3.1.2.10

### 3.1.1.5 Methoden

Wenn wir die Methoden `nextValue()` und `isOdd()` miteinander vergleichen, dann unterscheiden sie sich nicht nur allein durch ihren Namen und Körper voneinander, sondern auch durch das in der Methodenschnittstelle deklarierte Wort `void` bzw. `boolean`:

```
public void nextValue() {                                public boolean isOdd() {
    value = Math.abs(rndm.nextInt())%99+1;                return (value%2!=0);
}                                                         }
```

Wie bereits in Abschnitt 2.1.1.5 angedeutet wurde, können Methoden einen Wert zurückliefern. Der Datentyp dieses zurückgelieferten Wertes steht im Kopf der Methode unmittelbar vor ihrem Namen. Wird jedoch das Schlüsselwort **void** gesetzt, besagt dieses, dass die Methode keinen Wert zurückliefert. Methoden, welche einen Wert ausgeben, werden gelegentlich als Funktionen bezeichnet und werden von den sogenannten Prozeduren abgegrenzt<sup>1</sup>.

Sicherlich ist auch das im Körper der Methode `isOdd()` verwendete Schlüsselwort **return** auffallend. Dieses ist für eine Methode mit Rückgabewert obligatorisch und bestimmt den Wert, welcher zurückgeliefert wird.

**Siehe auch:** 3.1.2.7, 3.1.2.8

### 3.1.1.6 Der Datentyp boolean

Wie man aus der Methodenschnittstelle entnehmen kann, liefert die Methode `isOdd()` einen Wert zurück, der vom Typ `boolean` ist.

---

<sup>1</sup> Es ist an dieser Stelle jedoch darauf hinzuweisen, dass diese Begriffe vor allem in herkömmlichen Programmiersprachen wie z.B. C verwendet werden. In Java spricht man mehrheitlich von Methoden, die entweder `void` oder sogenannt „ge-typed“ sind.

In Abschnitt 1.2.2 wurden bereits Bedingungen erwähnt, welche entweder wahr oder nicht wahr sind und so gewissermassen einen Wahrheitswert widerspiegeln. Der Datentyp **boolean** ist nun derjenige Datentyp, welcher zur Repräsentation von Wahrheitswerten verwendet wird. Er ist nach dem englischen Mathematiker George Boole (1815-1864) benannt. Der Wertebereich von `boolean` umfasst nur zwei Werte, nämlich **true** (wahr) und **false** (falsch).

Es werden ausschliesslich logische Operationen auf Werten vom Typ `boolean` ausgeführt.

***Siehe auch:** 3.1.2.11, 3.1.2.13*

### 3.1.1.7 Vergleichsoperatoren

Betrachten wir die Methode `isOdd()`:

```
public boolean isOdd() {  
    return (value%2!=0);  
}
```

Das `return` statement bewirkt, dass der Wert des Ausdrucks

`(value%2!=0)`

zurückgegeben und die Methode verlassen wird. Der Wert des Ausdrucks muss gemäss Signatur vom Typ `boolean` sein.

Im Ausdruck findet man zum einen den Modulo-Operator und zum anderen einen **Vergleichsoperator** `'!='` vor, welcher seine beiden Operanden auf **Nicht-Identität** überprüft. Die `'!='`-Operation evaluiert also entweder zu `true` oder `false` und somit zu einem Booleschen Wert. Wie man aus der Übersichtstabelle der Operatoren in Anhang A entnehmen kann, hat der Modulo-Operator eine höhere Priorität, womit der Ausdruck

`value%2`

zuerst ausgewertet wird. Wenn man `value` - hierbei handelt es sich ja um eine positive, ganze Zahl zwischen 1 und 99 - `% 2` berechnet, erhält man 1, falls `value` ungerade ist, und 0, wenn `value` eine gerade Zahl ist. Das Resultat wird nun hinsichtlich seiner Identität mit Null verglichen: Falls `value` eine ungerade Zahl ist, ergibt der Ausdruck

`(value%2!=0)`

den Wert `true`, da 1 ungleich 0 ist, ansonsten handelt es sich um eine gerade Zahl und man erhält den Wert `false`, da 0 nicht ungleich 0 ist.

Weil nun also in der Anweisung

```
return (value%2!=0);
```

der Ausdruck `value%2!=0` den Rückgabewert der Methode bestimmt, untersucht die Methode `isOdd()` die Instanzvariable `value` auf Ungeradzahligkeit.

In der Methode

```
public boolean hasSameDigits() {  
    return (value/10==value%10);  
}
```

wird nun ein weiterer Vergleichsoperator eingesetzt, dieses Mal wird auf **Identität** geprüft.

Da sich die Werte der Instanzvariable `value` in den Grenzen zwischen 1 und 99 bewegen, ergibt der Ausdruck `value/10` den Zehner und `value%10` den Einer der Zahl `value`. Diese zwei Ziffern werden nun mittels dem '**==**'-Operator auf ihre Gleichheit überprüft, das Boolesche Ergebnis (`true` oder `false`) der Operation ist der Rückgabewert der Methode `hasSameDigits()`.

**Siehe auch:** 3.1.2.12, Anhang A

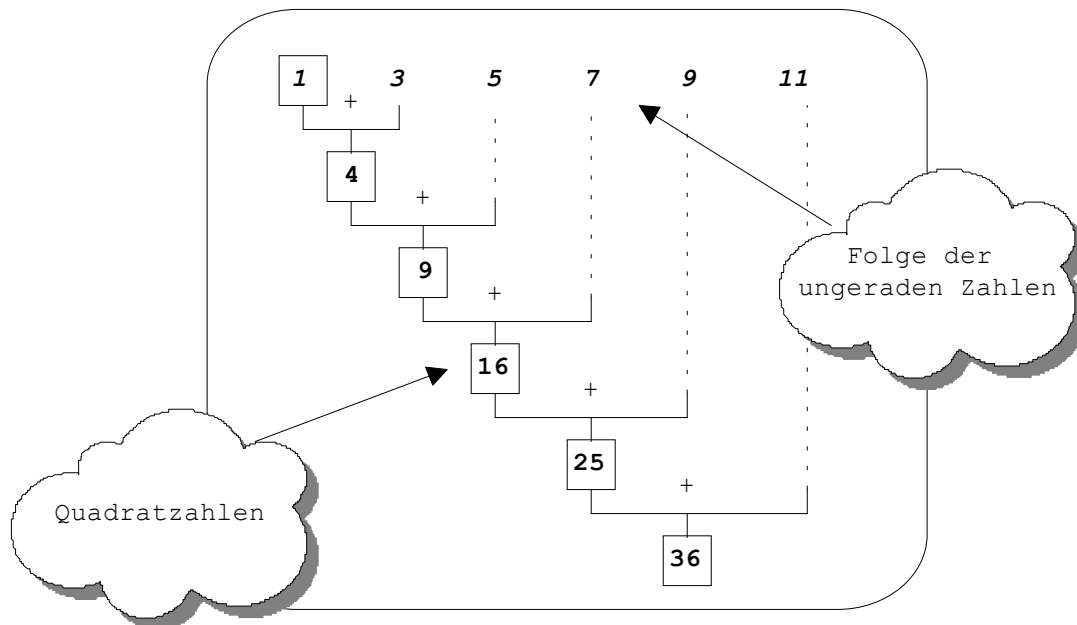
### 3.1.1.8 for Schleife

In der Methode

```
public boolean isASquare() {  
    int i,j;  
    for(i=1,j=1;i<value;i=i+j) {j=j+2;}  
    return (value==i);  
}
```

wird mit Hilfe einer **Iteration** (siehe Abschnitt 1.2.3) berechnet, ob `value` das Quadrat einer ganzen Zahl ist. In Java gibt es mehrere Programmstrukturen, sogenannte **Schleifen** (engl. **loop**), welche eine Iteration realisieren. Man unterscheidet die `while`, `for` und `do` Schleife. Ihnen allen ist gemeinsam, dass die Iteration an eine Boolesche Bedingung geknüpft ist.

Wie Abbildung 3-5 zeigt, kann man die ganzzahligen Quadratzahlen ermitteln, indem man die Reihe der ungeraden Zahlen berechnet:



**Abbildung 3-5: Berechnung ganzzahliger Quadratzahlen**

Die soeben demonstrierte Berechnung erfolgt in einer **for Schleife** in der Zeile

```
for (i=1, j=1; i<value; i=i+j) {j=j+2;}
```

Der nach dem Schlüsselwort **for** in runden Klammern stehende Ausdruck

```
(i=1, j=1; i<value; i=i+j)
```

enthält Steuerungsinformationen und wird als Schleifenkopf bezeichnet. Die in den darauffolgenden geschweiften Klammern stehende Anweisung

```
{j=j+2;}
```

bildet den Schleifenrumpf und wird in Abhängigkeit von einer Booleschen Bedingung wiederholt. Diese Bedingung findet sich in der Mitte des durch Semikolons getrennten dreiteiligen Schleifenkopfes. Solange der Ausdruck `i<value` den Wert `true` ergibt, wird die in geschweiften Klammern stehende Anweisung `j=j+2` wiederholt ausgeführt<sup>1</sup>. Aus diesem Grund spricht man auch von einer **Laufbedingung**. Jedes Mal, nachdem der Interpreter also den Schleifenrumpf ausgeführt hat, springt er zum Schleifenkopf zurück, um die Laufbedingung zu überprüfen. Ergibt sie `true`, erfolgt eine neuer

---

<sup>1</sup> Selbstverständlich können in diesen geschweiften Klammern auch mehrere Anweisungen stehen.

Iterationsdurchlauf, ansonsten wird die in geschweiften Klammern stehende Anweisung `j=j+2` übersprungen und das nächste statement im Programm, also

```
return (value==i);
```

ausgeführt. Da die Boolesche Bedingung einer for Schleife immer vor der Ausführung der zu iterierenden Anweisungen überprüft wird, kann es vorkommen, dass der Schleifenrumpf gar nie ausgeführt wird.

Abgesehen davon, dass eine for Schleife ein Mittel zur Iteration ist, ermöglicht sie aufgrund ihrer Syntax die Verwendung von **Schleifenvariablen**. Solche Schleifenvariablen werden innerhalb des Schleifenrumpfes einer for Schleife verwendet.

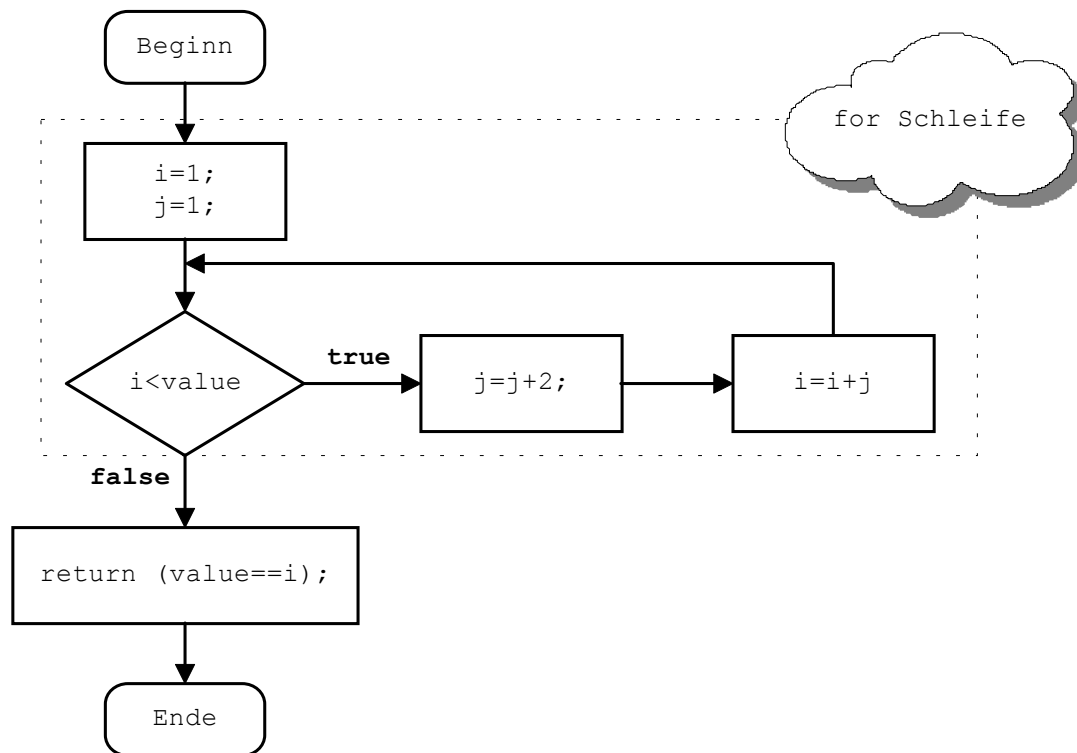
Im ersten Teil des Schleifenkopfes werden die Schleifenvariablen `i` und `j` deklariert und sogleich mit dem Wert Eins initialisiert. Da Schleifenvariablen innerhalb der for Schleife deklariert werden, sind sie auch nur innerhalb der for Schleife ansprechbar, weshalb sie als **lokale Variablen** bezeichnet werden. Es fragt sich nun, wann die Initialisierung der Schleifenvariablen passiert.

Die Initialisierung der Schleifenvariablen erfolgt nur einmal, als allererste Anweisung einer for Schleife, bevor die Laufbedingung überprüft wird.

Der letzte Abschnitt des Schleifenkopfes erlaubt nun, die Werte von Schleifenvariablen zu verändern. Hierbei wird diese Änderung **nach jedem Iterationsdurchgang**, und zwar vor der Überprüfung des Booleschen Ausdrucks `i<value`, durchgeführt. Jedes Mal nachdem also die in geschweiften Klammern stehende Anweisung `j=j+2` ausgeführt wurde, erhält `i` den Wert `i+j`. Sollte der Schleifenrumpf nie aufgerufen werden<sup>1</sup>, erfolgt die Zuweisung `i=i+j` ebenfalls nie. Es ist für das Verständnis der Funktionsweise einer for Schleife hilfreich, sich die Veränderung der Schleifenvariablen als letztes statement innerhalb des Schleifenrumpfes vorzustellen. Das in Abbildung 3-6 ersichtliche Flussdiagramm illustriert die Anweisungssequenz der Methode `isASquare()` und der for Schleife.

---

<sup>1</sup> Sofern der Boolesche Ausdruck `i<value` von Anfang an `false` ergibt.



**Abbildung 3-6: Flussdiagramm für for Schleife der Methode isASquare()**

Untenstehende Tabelle zeigt fünf Iterationsdurchläufe durch die for Schleife mit zugehörigen Werten von *i* und *j*. Hierbei sind jeweils die Werte vor Beginn des Schleifendurchlaufs angegeben.

<i>Durchlauf</i>	<i>i</i>	<i>j</i>
1	1	1
2	4	3
3	9	5
4	16	7
5	25	9

Wie man aus der Tabelle entnehmen kann, bilden die Werte von *j* die Folge der ungeraden Zahlen und die von *i* die erwünschten Quadratzahlen. Da *j* mit dem Wert 1 initialisiert

wird und mit jedem Durchgang um 2 erhöht wird, erhält man eine Folge von ungeraden Zahlen, bei 1 beginnend. Analog zu der in Abbildung 3-5 illustrierten Vorgehensweise bewirkt die Anweisung `i=i+j` am Ende jeder Iteration, dass die Teilsummen der Folge der ungeraden Zahlen und somit die Quadratzahlen gebildet werden.

Sobald die Iterationsbedingung `i<value` nicht mehr erfüllt ist, wird untersucht, ob `value` gleich der im vorangehenden Durchgang ermittelten Quadratzahl und somit eine Quadratzahl ist. Das Boolesche Ergebnis der Vergleichsoperation bildet den Rückgabewert der Methode `isASquare()`.

***Siehe auch:** 3.1.2.5, 3.1.2.14 bis 3.1.2.17*

### 3.1.1.9 Strings

Die Methode

```
public String toString() {  
    return String.valueOf(value);  
}
```

in der Klasse `Number` gibt nun als Rückgabewert einen **String** zurück. Wie wir bereits in Abschnitt 2.1.1.5 gesehen haben, ist ein String ein Text und somit auch ein Datentyp. Ein String gehört zu den komplexen Datentypen (hierauf wird in Abschnitt 4.1 ausführlicher eingegangen).

Die Klasse **String** ist in der Klassenbibliothek deklariert. Ihre Methode `valueOf()` konvertiert einen als Parameter erhaltenen `int`-Wert zu einem String und liefert diesen als Rückgabewert. In der Methode `toString()` wird der Klasse `String` die Botschaft `valueOf()` mit der Instanzvariablen `value` als Parameter geschickt<sup>1</sup>, worauf diese einen String mit dem Inhalt von `value` zurückgibt.

### 3.1.1.10 if statement

Gehen wir nun zur Klasse `UserFrame` über. In ihr, d.h. in der Methode `actionPerformed()`, finden wir mehrere ineinandergeschachtelte `if` statements. Betrachten wir zunächst einmal, wie die graphische Benutzeroberfläche im Konstruktor generiert wird.

Die Klasse `UserFrame` verfügt über die Instanzvariablen `number` der Klasse `Number`, `message` der Klasse `Label` und über die Checkboxes `odd`, `sameDigits` und

---

<sup>1</sup> Es sei darauf hingewiesen, dass hier einer Klasse und nicht einer Instanz eine Botschaft geschickt wird. Mehr zu diesem Thema findet sich in Abschnitt 6.1.

square. Im Konstruktor `UserFrame()` erfolgt nun die Platzierung der Komponenten in bekannter Weise. Hervorzuhalten ist die Zeile

```
place(message=new Label(), 40, 30, 150, 20);
```

da hier ein noch unbekannter Konstruktor der Klasse `Label` verwendet wird. Im Unterschied zu dem in Abschnitt 2.1 verwendeten Konstruktor generiert dieser ein `Label`, welches keine Beschriftung hat.

Es wird auch ein neuer Konstruktor zur Generierung der `Checkboxes` verwendet. Da eine Zahl z.B. ungerade sein kann und zugleich auch eine ganzzahlige Wurzel aufweisen kann, sollte der Benutzer auch mehrer `Checkboxes` selektieren können. Aus diesem Grund dürfen die `Checkboxes` keiner `CheckboxGroup` zugeordnet werden. Dem Konstruktor wird lediglich der Name der `Checkbox` als Parameter übergeben:

```
place(odd=new Checkbox("is odd"), 40, 60, 120, 20);
```

In den Zeilen

```
number = new Number();  
number.nextValue();  
message.setText("Check if "+number.toString());
```

wird zuerst die Instanzvariable `number` als Instanz der Klasse `Number` initialisiert. Mit dem Aufruf `number.nextValue()` wird nun der Instanzvariablen `value` der Instanz `number` eine positive Zufallszahl zwischen 1 und 99 zugewiesen. In der letzten Zeile schlussendlich erhält das `Label` `message` die Botschaft `setText()`. Diese bewirkt, dass der als Parameter übergebene String im `Label` `message` ausgegeben wird. Die an `number` versandte Botschaft `toString()` liefert den Wert der Instanzvariablen `value` als String zurück, welcher an den String „Check if“ angehängt wird. Wenn `value` beispielsweise den Wert 63 hätte, würde auf dem `Label` `message` also der Text „Check if 63“ zu sehen sein.

Wie man ebenfalls aus dem Konstruktor `UserFrame()` entnehmen kann, gibt es auf der graphischen Benutzeroberfläche die drei Buttons `DONE`, `NEXT` und `CHECK`. Interessant sind hierbei die Zeilen

```
place(button=new Button("DONE"), 30, 170, 40, 20);  
button.addActionListener(this);  
place(button=new Button("NEXT"), 80, 170, 40, 20);  
button.addActionListener(this);  
place(button=new Button("CHECK"), 130, 170, 40, 20);  
button.addActionListener(this);
```

Zu Beginn wird im Konstruktor `UserFrame()` eine lokale Variable `button` vom Typ `Button` deklariert. Diese Variable wird nun dazu verwendet, die der Reihe nach erzeugten Instanzen dieser Klasse `Button` kurzfristig zu bezeichnen. Anfänglich referenziert `button` diejenige Instanz der Klasse `Button`, welche die Beschriftung „DONE“ hat. Ab der dritten Zeile jedoch benennt sie die Instanz der Klasse `Button` mit



der Beschriftung „NEXT“ und in den zwei letzten Zeilen schliesslich die Instanz mit der Beschriftung „CHECK“. Damit gewährleistet ist, dass jedes Mal auch die Methode `actionPerformed()` aufgerufen wird, wenn einer der drei Buttons aktiviert wird, dürfen die Zeilen

```
button.addActionListener(this);
```

nicht fehlen. Hierbei ist allerdings zu beachten, dass `button` jeweils in Abhängigkeit vom momentanen Standpunkt im Programm unterschiedliche Instanzen der Klasse `Button` bezeichnet.

In der Methode

```
public void actionPerformed(ActionEvent event){
    if (event.getActionCommand().equals("DONE")) {
        int errorCount=0;
        if (odd.getState() ^ number.isOdd()) errorCount++;
        if (sameDigits.getState() ^ number.hasSameDigits()) errorCount++;
        if (square.getState() ^ number.isASquare()) errorCount++;
        if (errorCount==0)
            message.setText("All answers are correct!");
        else if (errorCount==1)
            message.setText("There is one error!");
        else
            message.setText("There are "+errorCount+" errors!");
    } else if (event.getActionCommand().equals("NEXT")) {
        number.nextValue();
        message.setText("Check if "+number.toString());
        odd.setState(false);
        sameDigits.setState(false);
        square.setState(false);
    } else if (event.getActionCommand().equals("CHECK")) {
        message.setText(number.toString()+" has the properties");
        odd.setState(number.isOdd());
        sameDigits.setState(number.hasSameDigits());
        square.setState(number.isASquare());
    }
}
```

wird nun mit Hilfe von `if statements` untersucht, welcher Button ein Benutzer aktiviert hat und in Abhängigkeit davon reagiert. Dies kennen wir bereits aus Abschnitt 2.4.1.1. In Java bilden **if statements** einen **bedingten Ausdruck** und sind somit ein Mittel zur **Selektion** (siehe Abschnitt 1.2.2).

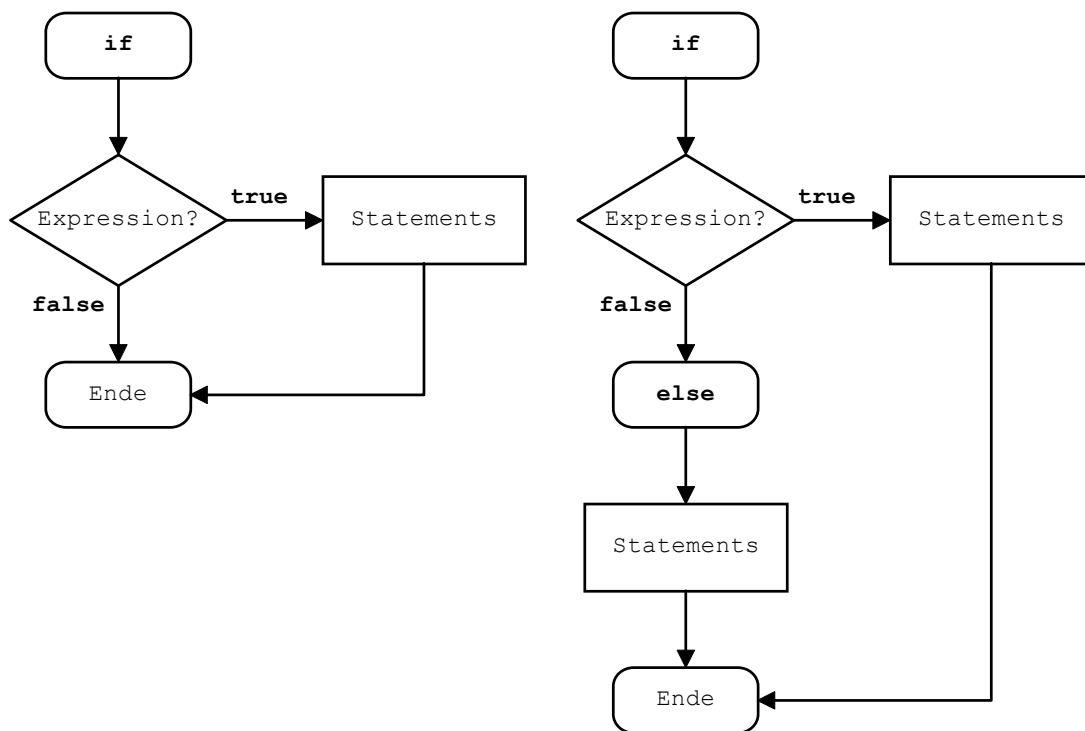
Nach dem Schlüsselwort `if` folgt in runden Klammern die Boolesche Bedingung

```
(event.getActionCommand().equals("DONE"))
```

Der obige Ausdruck liefert `true`, falls der Button `DONE` durch den Benutzer selektiert wurde, ansonsten `false`. Wenn nun die Bedingung `true` ergibt, werden die darauffolgenden Anweisungen

```
int errorCount=0;
if (odd.getState()^number.isOdd()) errorCount++;
if (sameDigits.getState()^number.hasSameDigits()) errorCount++;
if (square.getState()^number.isASquare()) errorCount++;
if (errorCount==0)
    message.setText("All answers are correct!");
else if (errorCount==1)
    message.setText("There is one error!");
else
    message.setText("There are "+errorCount+" errors!");
```

ausgeführt. Liefert die Bedingung aber false, werden die obigen Anweisungen übersprungen. Falls ein **else statement** folgt, wird mit der Interpretation bei diesem fortgesetzt, ansonsten die Methode verlassen. Abbildung 3-7 veranschaulicht dies anhand eines Flussdiagramms.



**Abbildung 3-7: if statement ohne (links) und mit else Anweisung (rechts)**

In obigem Programm hat es nun ein else statement. Nach diesem folgt aber wiederum ein if mit einem else statement.

```
else if (event.getActionCommand().equals("NEXT")) {
    number.nextValue();
    message.setText("Check if "+number.toString());
    odd.setState(false);
    sameDigits.setState(false);
    square.setState(false);
} else if (event.getActionCommand().equals("CHECK")) {
    message.setText(number.toString()+" has the properties");
    odd.setState(number.isOdd());
    sameDigits.setState(number.hasSameDigits());
    square.setState(number.isASquare());
}
```

Man hat also die Möglichkeit, mehrere if-/else Anweisungen ineinander zu verschachteln. Dabei kann die Verschachtelungstiefe beliebig tief sein.

Im obigen Programmauszug überprüft das erste if, ob der Button NEXT aktiviert wurde. Falls dies zutrifft, werden die Anweisungen

```
number.nextValue();
message.setText("Check if "+number.toString());
odd.setState(false);
sameDigits.setState(false);
square.setState(false);
```

ausgeführt, ansonsten wird zum zugehörigen else statement gewechselt:

```
else if (event.getActionCommand().equals("CHECK"))
```

Hier trifft man erneut auf ein if statement, dieses Mal aber ohne else statement. In der if Anweisung wird wieder abgefragt, ob der Button CHECK aktiviert wurde. Bei Zutreffen werden die Anweisungen

```
message.setText(number.toString()+" has the properties");
odd.setState(number.isOdd());
sameDigits.setState(number.hasSameDigits());
square.setState(number.isASquare());
```

aufgerufen, ansonsten wird die Methode `actionPerformed()` verlassen.

Die bedingten Anweisungen in der Methode `actionPerformed()` untersuchen also, welcher Button in der graphischen Benutzeroberfläche aktiviert wurde, um dann entsprechend zu reagieren. Wurde keiner der Buttons gedrückt, geschieht nichts.

***Siehe auch: 3.1.2.18***

#### *3.1.1.11 Logische Operatoren und arithmetischer Inkrement-/Dekrement-Operator*

Dringen wir nun tiefer in die Methode `actionPerformed()` ein. Wird durch den Benutzer der Button DONE aktiviert, so werden die Anweisungen

```
int errorCount=0;
if (odd.getState()^number.isOdd()) errorCount++;
if (sameDigits.getState()^number.hasSameDigits()) errorCount++;
if (square.getState()^number.isASquare()) errorCount++;
if (errorCount==0)
    message.setText("All answers are correct!");
else if (errorCount==1)
    message.setText("There is one error!");
else
    message.setText("There are "+errorCount+" errors!");
```

aufgerufen. Diese enthalten wiederum if-/else statements. Hierbei ist zu beachten, dass ein else immer zum letzten if statement gehört.

In der ersten Anweisung wird eine lokale Variable `errorCount` deklariert und sogleich mit Null initialisiert. Sie dient dem Festhalten der Fehler, die ein Benutzer macht und ist vom Datentyp `int`.

#### Die Zeile

```
if (odd.getState()^number.isOdd()) errorCount++;
```

enthält ein if statement ohne ein else statement. Falls die in Klammern stehende Bedingung erfüllt ist, wird die Anweisung `errorCount++` ausgeführt.

Die Anweisung `errorCount++` bewirkt, dass der Wert der Variablen `errorCount` um eins erhöht wird. Der arithmetische Operator ‘++’ wird als **Inkrement** (engl. **increment**) bezeichnet (eins dazuzählen). Natürlich gibt es auch einen arithmetischen **Dekrement**-Operator (engl. **decrement**) der Form ‘--’ (eins abzählen). Beide können als sogenannte **Postfix-Notation** (engl. **postfix expression**) der Variablen folgen (z.B. `errorCount++`) oder als **Präfix-Notation** (engl. **prefix expression**) der Variablen vorangestellt werden (z.B. `++errorCount`). Hierbei liegt der Unterschied der beiden Notationsformen in der Reihenfolge, in welcher das Inkrement/Dekrement und die Ausgabe eines Wertes erfolgt: bei einem Postfix-Ausdruck wird der Wert der Variablen vor der Erhöhung/Erniedrigung ausgegeben und erst anschliessend inkrementiert/dekrementiert, bei einem Präfix-Ausdruck wird der Variablenwert zuerst erhöht/erniedrigt und danach ausgegeben. Beiden Notationsformen ist aber gemeinsam, dass der Wert der Variablen schlussendlich erhöht/erniedrigt wird. Abbildung 3-8 illustriert anhand eines Programmfragments den Unterschied der beiden Notationsformen. Es sei an dieser Stelle vor den Tücken eines Postfix- bzw. Präfix-Inkrement/Dekrement gewarnt, denn dies ist eine häufige Fehlerquelle!

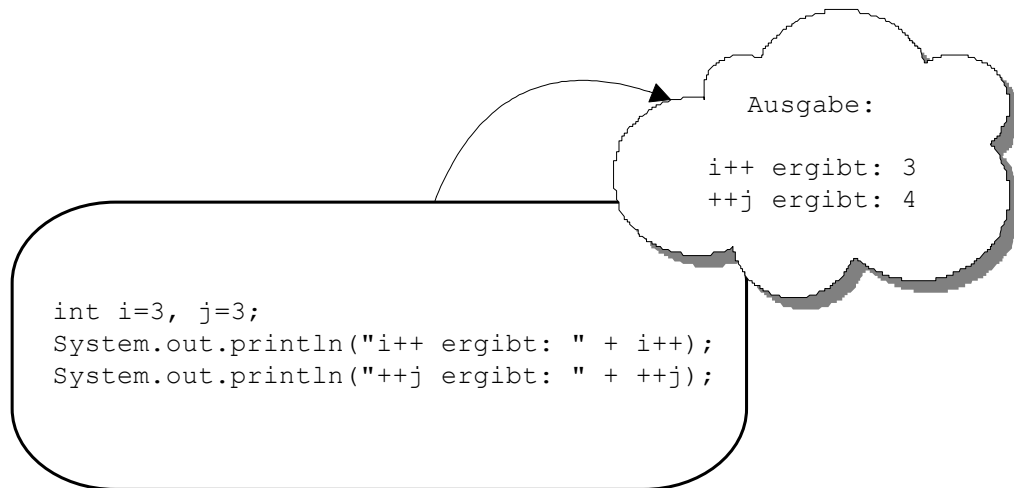


Abbildung 3-8: Präfix- versus Postfix-Notation

### Der Boolesche Ausdruck

```
odd.getState() ^ number.isOdd()
```

enthält den logischen Operator **exklusives Oder**, welcher die links und rechts stehenden Methodenaufrufe miteinander verknüpft. Die an die Checkbox `odd` versandte Botschaft `getState()` gibt an, ob die Checkbox selektiert ist. Sie liefert also einen Booleschen Rückgabewert, welcher `true` ist, falls die Checkbox markiert ist. Mit dem Aufruf `number.isOdd()` wird nun geprüft, ob die Instanzvariable `value` des Objekts `number` wirklich eine ungerade Zahl ist. Der Rückgabewert der in der Klasse `Number` deklarierten Methode `isOdd()` ist wiederum von Typ `boolean`. Die beiden Wahrheitswerte werden nun mit dem exklusiven Oder verknüpft, was wiederum zu einem Booleschen Wert führt. Gemäss der in Abschnitt 3.1.2.13 definierten Wahrheitstabelle des exklusiven Oders ergibt der Ausdruck

```
odd.getState() ^ number.isOdd()
```

immer dann `true`, wenn die Angabe des Benutzers nicht mit dem Resultat der Methode `isOdd()` übereinstimmt, also falsch ist. In diesem Fall wird auch die Variable `errorCount` um eins erhöht.

In analoger Weise werden nun die Zeilen

```
if (sameDigits.getState() ^ number.hasSameDigits()) errorCount++;
if (square.getState() ^ number.isASquare()) errorCount++;
```

ausgewertet.

Damit man in der graphischen Benutzeroberfläche ausgeben kann, wieviele Fehler der Benutzer gemacht hat, braucht es die Anweisungen

```
if (errorCount==0)
    message.setText("All answers are correct!");
else if (errorCount==1)
    message.setText("There is one error!");
else
    message.setText("There are "+errorCount+" errors!");
```

Hier findet man ein if-else-statement, das in seinem else-Teil wiederum ein if-else-statement aufweist. Zuerst wird geprüft, ob der Benutzer keine Fehler gemacht hat. Falls dem so ist, wird in der graphischen Benutzeroberfläche der String „All answers are correct!“ angezeigt. Wenn `errorCount` jedoch nicht gleich Null ist, wird untersucht, ob `errorCount` gleich eins ist. Bei Zutreffen wird der Text „There is one error!“ ausgegeben. Falls `errorCount` aber nicht gleich eins ist, muss er grösser gleich zwei sein, der String „There are x errors!“ wird angezeigt, wobei x die konkrete Anzahl der Fehler ist<sup>1</sup>.

Wählt der Benutzer den Button NEXT, kommen die folgenden Programmzeilen zum Tragen:

```
number.nextValue();
message.setText("Check if "+number.toString());
odd.setState(false);
sameDigits.setState(false);
square.setState(false);
```

Der Instanzvariablen `value` des Objekts `number` wird nun eine neue Zufallszahl zwischen 1 und 99 zugewiesen und der Benutzer wird erneut aufgefordert, die Eigenschaften der Zahl anzugeben. Mit Hilfe der Methode `setState()` kann man nun willentlich den Zustand einer Checkbox angeben: `false` setzt sie auf „nicht selektiert“.

Der Button CHECK schlussendlich bewirkt die Ausführung der folgenden Anweisungen:

```
message.setText(number.toString()+" has the properties");
odd.setState(number.isOdd());
sameDigits.setState(number.hasSameDigits());
square.setState(number.isASquare());
```

Nun werden die Checkboxes entsprechend dem Resultat der Methodenaufrufe `isOdd()`, `hasSameDigits()` und `isASquare()` gesetzt.

***Siehe auch:*** 3.1.2.10, 3.1.2.13

---

<sup>1</sup> Der '+'-Operator hat hier nichts mit der arithmetischen Addition zu tun; sind seine Operanden Strings, bewirkt er deren Concatenation (siehe Abschnitt 2.1.1.5 und 4.1.1.4). Es ist ebenfalls zu beachten, dass im Ausdruck "There are "+`errorCount` der Wert der Variablen `errorCount` automatisch zu einem String konvertiert wird, da einer der Operanden, nämlich "There are ", bereits ein String ist.

### 3.1.1.12 switch statement

In der Methode `actionPerformed()` könnte man die Zeilen

```
if (errorCount==0)
    message.setText("All answers are correct!");
else if (errorCount==1)
    message.setText("There is one error!");
else
    message.setText("There are "+errorCount+" errors!");
```

ohne Konsequenzen durch die folgenden ersetzen:

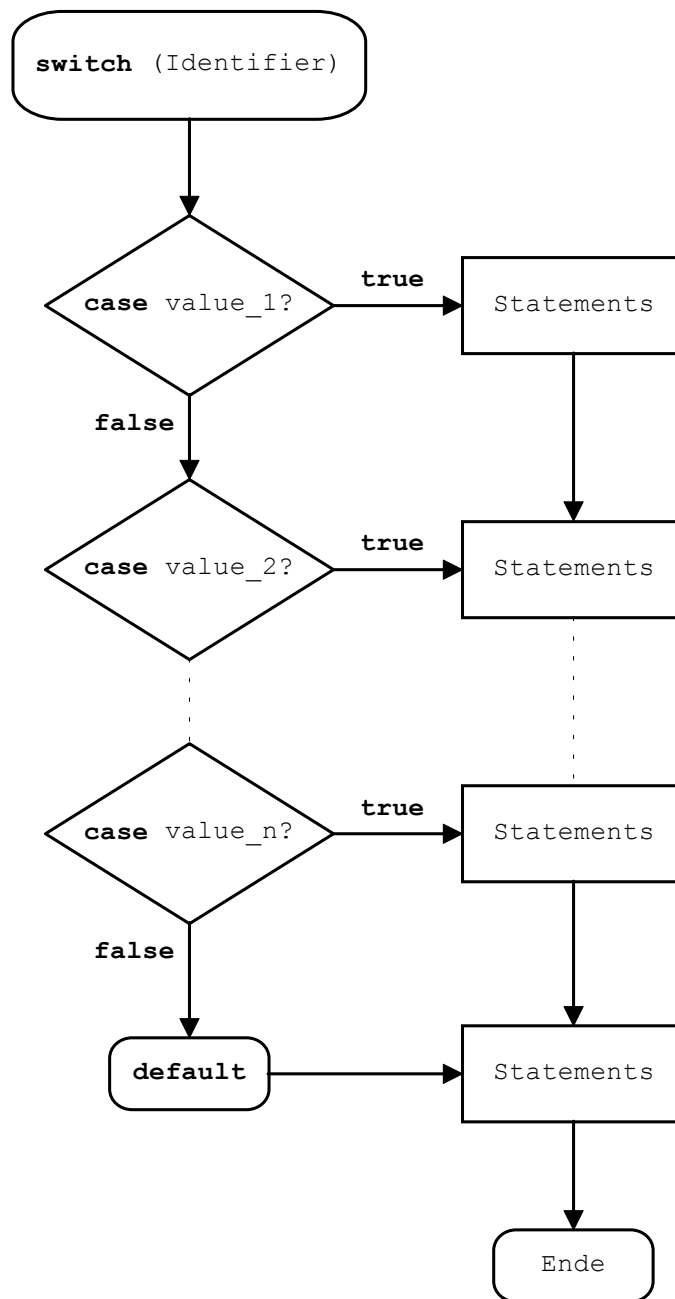
```
switch (errorCount) {
    case 0: message.setText("All answers are correct!"); break;
    case 1: message.setText("There is one error!"); break;
    default: message.setText("There are "+errorCount+" errors!");
}
```

Ein sogenanntes **switch statement** ermöglicht die Auswahl auszuführender Anweisungen in Abhängigkeit vom Wert eines Ausdrucks.

Im Beispiel wird der Wert der lokalen Variablen `errorCount` untersucht, er steht in runden Klammern nach dem Schlüsselwort **switch**. Weist er den Wert 0 auf, wird die Anweisung `message.setText("All answers are correct!")` ausgeführt. Ist er Eins, erfolgt die Ausgabe von „There is one error!“. Ansonsten wird der Text „There are x errors!“ angezeigt, wobei x die Anzahl der Fehler angibt.

Nach dem Schlüsselwort **case** steht also ein Wert, welcher `errorCount` annehmen kann. Nach dem Doppelpunkt folgen die auszuführenden Anweisungen, welche aufgerufen werden, falls `errorCount` den besagten Wert aufweist. Die nach dem Schlüsselwort **default** stehenden Anweisungen werden aufgerufen, falls keiner der durch die Schlüsselwörter `case` bezeichneten Werte zutrifft.

Da ein `switch statement` im Prinzip den Einstieg in eine Programmsequenz in Abhängigkeit von einem Wert realisiert, werden, sobald ein `case statement` zutrifft, sämtliche nachfolgenden Anweisungen ausgeführt (siehe Abbildung 3-9). Um dies zu verhindern, muss man das Schlüsselwort **break** verwenden: Es bewirkt, dass das `switch statement` verlassen wird (siehe Abbildung 3-10).



**Abbildung 3-9: switch statement ohne break**



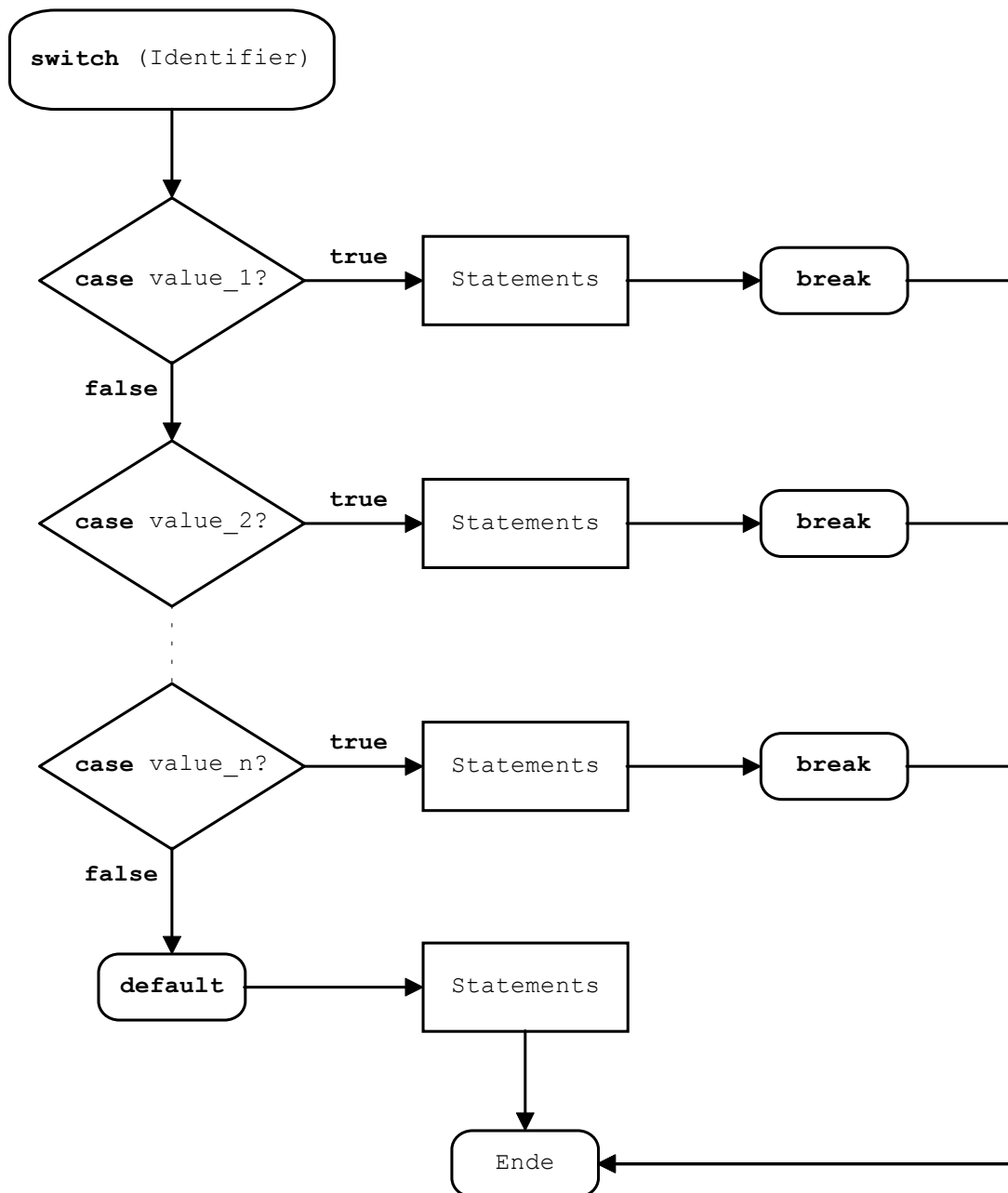


Abbildung 3-10: switch statement mit break

*Siehe auch: 3.1.2.19*

### 3.1.2 Syntax

#### 3.1.2.1 Kommentar

Man kann in Java zwei verschiedene Arten von **Kommentar** (engl. **comment**) verwenden.

Gebraucht man die Zeichenfolge '//', wird alles bis zum Ende der Zeile als Kommentar betrachtet.

<b>SingleLineComment</b>
//Kommentar bis zum Ende der Zeile

Man kann den als Kommentar zu betrachtenden Text aber auch unabhängig vom Zeilenende eingrenzen. Alles was zwischen der Zeichenfolge '/\*' und '\*/' steht, wird ignoriert. Hierbei ist noch darauf aufmerksam zu machen, dass man diesen Kommentartyp nicht wie die geschweiften Klammern bei Anweisungsblöcken ineinander verschachteln darf. Bei der Anordnung

```
/* ... /* ... */ ... */
```

würde schon das erste Zeichen '\*/' als das Ende des Kommentars interpretiert werden, eine syntaktische Fehlermeldung wäre die Folge.

<b>MultipleLineComment</b>
/*Kommentar, welcher unabhängig vom Zeilenende ist und sich auch über mehrere Zeilen erstrecken kann*/

Der Vollständigkeit halber sei noch auf ein dritter Kommentartyp der Form /\*\* ...\*/ verwiesen, welcher zur automatischen Erstellung einer Dokumentation eingesetzt werden kann. Er wird jedoch in diesem Skript nicht verwendet.

#### 3.1.2.2 Variablendeklaration

Damit man von einer **Variablen** Gebrauch machen darf, d.h. sie mit ihrem Namen im Programmverlauf ansprechen darf, muss sie an irgendeiner Stelle im Programm einmal **deklariert** werden.

VariableDeclaration
<pre>type Identifier;</pre>

Beispiele: `int value;`  
`Random rndm;`  
`Label message;`

Dem Namen der Variablen (Identifier) wird ihr Datentyp (type) vorangestellt.

Mehrere Variablen gleichen Typs können auch in folgender Kurzform deklariert werden:

VariableDeclaration
<pre>type Identifier_1, ..., Identifier_n;</pre>

Beispiel: `Checkbox odd, sameDigits, square;`

### 3.1.2.3 Gültigkeitsbereich einer Variablen

Jeder Variablen ist ein **Gültigkeitsbereich** (eng. **scope**) oder **Sichtbarkeitsbereich** zugeordnet, welcher bestimmt, von welchen Programmbereichen aus man auf den Wert der Variablen zugreifen kann. Ausserhalb ihres Gültigkeitsbereichs ist die Variable gar nicht sichtbar und kann somit gar nicht angesprochen werden.

### 3.1.2.4 Instanzvariablen und Klassenvariablen

**Instanz- und Klassenvariablen** werden auf Ebene der Klasse deklariert, befinden sich also direkt innerhalb der geschweiften Klammern einer Klasse.

InstanceVariableDeclaration
<pre>type Identifier;</pre>

Beispiele: `String title;`  
`TextField firstName;`

Klassenvariablen weisen in ihrer Deklaration zusätzlich noch das Schlüsselwort **static** auf. Hierdurch kann man sie von den Instanzvariablen unterscheiden.

<b>ClassVariableDeclaration</b>
<b>static</b> type Identifier;

Beispiel:    `static int cardCount;`

Der Gültigkeitsbereich von Instanz- und Klassenvariablen ist standardmässig das package, in welchem sich ihre Klasse befindet. Mittels sogenannter **visibility modifiers** (siehe Abschnitt 6.2.1.3 und 6.2.2.1) kann man jedoch deren Sichtbarkeitsbereich willentlich bestimmen.

#### 3.1.2.5 Lokale Variablen

**Lokale Variablen** werden innerhalb Methoden oder innerhalb statements deklariert.

<b>LocalVariableDeclaration</b>
type Identifier;

Beispiel:    `for(int i=1,int j=1;i<value;i=i+j)  
              {j=j+2;}`

Ihre Sichtbarkeit ist immer auf die Methode oder das statement, in welchem sie deklariert wurden, beschränkt.

#### 3.1.2.6 Zuweisung

Der **Zuweisungsoperator** '=' erlaubt es, einer Variablen einen Wert zuzuweisen. Die Zuweisung erfolgt jeweils **von rechts nach links**.

Assignment
------------

Identifier = Expression;
--------------------------

Beispiele: `j = j+2;`  
`value = Math.abs(rndm.nextInt())%99+1;`

Bei Expression kann es sich direkt um einen zum Wertebereich des Datentyps gehörenden Wert oder aber auch um mehrere, mittels Operatoren miteinander verknüpften Methodenaufrufe handeln, die als Gesamtheit einen Wert zurückliefern. Essentiell jedoch ist, dass Expression vom gleichen Typ wie Identifier ist.

Es ist auch möglich, einer Variablen bei ihrer Deklaration einen Anfangswert zuzuweisen und sie somit zu initialisieren.

Assignment
------------

<code>type Identifier = Expression;</code>
--

Beispiel: `int errorCount = 0;`

### 3.1.2.7 Methodendeklaration

Damit man **Methoden** überhaupt aufrufen kann, müssen sie irgendwann einmal **deklariert** werden. In ihrer Deklaration unterscheidet man den **Kopf** (bzw. Signatur oder Methodenschnittstelle) vom **Rumpf** einer Methode, welcher die auszuführenden statements enthält. Der Kopf einer Methode beschreibt ihre Kommunikationsschnittstelle.

MethodDeclaration
-------------------

<code>Identifier(ParameterList) {     Statements }</code>
---

Beispiel: `private void place(Component comp,int x,  
 int y,int width,  
 int height) {  
 comp.setBounds(x, y, width, height);  
}`

```
        add(comp);  
    }
```

Die Methodenschnittstelle deklariert in einem runden Klammerpaar die Namen und Datentypen ihrer **Parameter**. Mehrere Parameter werden durch Kommata voneinander getrennt. Hat die Methode keine Parameter, sind die runden Klammern leer. Der Gültigkeitsbereich von Parametern ist auf die Methode beschränkt - Parameter sind lokale Variablen.

ParameterList
type Identifier <sub>1</sub> , ..., type Identifier <sub>n</sub>

Beispiel:   Component comp, int x, int y, int width,  
              int height

Die Methodenschnittstelle besagt ebenfalls, ob eine Methode einen Rückgabewert liefert (typed method) oder nicht (void method).

Für eine **typed method** ist das Schlüsselwort **return** obligatorisch. Es liefert Expression als Rückgabewert der Methode zurück, weshalb Expression vom Datentyp type sein muss. Das return statement bewirkt ebenfalls, dass die Programmkontrolle an den Ort des Methodenaufrufs zurückgeht und somit die Methode verlassen wird. Es ist sicher zu stellen, dass die Methode in jedem Fall mit einem return statement verlassen wird. Gerade bei Selektionen, wo es möglich ist, eine Methode an verschiedenen Stellen zu verlassen, ist besonders darauf zu achten.

TypedMethodDeclaration
type Identifier(ParameterList) { . . <b>return</b> Expression; }

Beispiel:   boolean isOdd() {  
              return (value%2!=0);  
          }

Liefert eine Methode keinen Rückgabewert zurück, steht anstelle von type das Schlüsselwort **void**. Das return statement muss nicht verwendet werden.

**VoidMethodDeclaration**

```
void Identifier(ParameterList) {
    Statements
}
```

Beispiel: 

```
void nextValue() {
    value = Math.abs(rndm.nextInt())%99+1;
}
```

**3.1.2.8 Methodenaufruf**

Der **Aufruf** einer Methode bewirkt, dass sämtliche in ihrem Rumpf gekapselten Anweisungen ausgeführt werden. Als Parameter werden konkrete Werte übergeben. Hierbei müssen Anzahl, Reihenfolge und Typ der Werte mit der in der Methodenschnittstelle aufgeführten Parameterdeklaration übereinstimmen.

**MethodCall**

```
Receiver.Identifier(value_1, ..., value_n);
```

Beispiel: 

```
comp.setBounds(x, y, width, height);
```

Der Botschaftsempfänger (Receiver), in obigem Beispiel also `comp`, wird dem Aufruf vorangestellt.

**3.1.2.9 Der Datentyp int**

Nachfolgende Tabelle zeigt in schematischer Weise die Charakteristika des Datentyps **int**. Bei Default handelt es sich um den Wert, mit welchem eine Variable dieses Typs automatisch initialisiert wird.

Beschreibung	Wertebereich	Default	Grösse
positive und negative ganze Zahlen	von -2147483648 bis 2147483647	0	32 bit

Nebst dem Datentyp `int` werden in Java noch die Datentypen `byte`, `short` und `long` für die Repräsentation von ganzen Zahlen verwendet<sup>1</sup>. Sie unterscheiden sich voneinander durch die Bandbreite ihres Wertebereichs.

### 3.1.2.10 Arithmetische Operatoren

Bei den **arithmetischen Operatoren** gibt es sowohl **unäre** als auch **binäre** Operatoren. Binäre Operatoren verknüpfen die links und rechts vom Operator stehenden Operanden. Unäre Operatoren werden nur mit einem Operanden verwendet. In der folgenden Tabelle sind binäre und unäre arithmetische Operatoren aufgeführt. Die unären Operatoren sind speziell als solche bezeichnet.

<i>Prior.</i>	<i>Operator</i>	<i>Beschreibung</i>	<i>Assoz.</i>
1	<code>++</code>	unäres Postfix- oder Präfix-Inkrement	R
	<code>--</code>	unäres Postfix- oder Präfix-Dekrement	R
	<code>+</code> , <code>-</code>	unäres Plus, unäres Minus (Vorzeichen)	R
2	<code>*</code> , <code>/</code> , <code>%</code>	Multiplikation, Division, Modulo	L
3	<code>+</code> , <code>-</code>	Addition, Subtraktion	L

*Prior.* gibt die relative Priorität des Operators an, hält also fest, welche Operation vor einer anderen auszuführen ist. Haben zwei Operationen die gleiche Priorität, dann bestimmt deren Assoziativität (*Assoz.*), ob der Ausdruck von links nach rechts (L) oder von rechts nach links (R) zu evaluieren ist.

Durch das Setzen von **Klammern** kann man die in der Tabelle definierten Vorrangsregeln (Reihenfolge des Abarbeitens) überschreiben.

---

<sup>1</sup> In Anhang A befindet sich eine Zusammenstellung sämtlicher Datentypen.



### 3.1.2.11 Der Datentyp boolean

<i>Beschreibung</i>	<i>Wertebereich</i>	<i>Default</i>	<i>Grösse</i>
Wahrheitswerte	<b>true</b> und <b>false</b>	<b>false</b>	1 bit <sup>1</sup>

### 3.1.2.12 Vergleichsoperatoren

Das Resultat einer **Vergleichsoperation** ergibt immer einen **Booleschen Wert**. Die untenstehenden Vergleichsoperatoren sind binäre Operatoren.

<i>Prior.</i>	<i>Operator</i>	<i>Beschreibung</i>	<i>Assoz.</i>
1	<	kleiner als	L
	<=	kleiner gleich	L
	>	grösser als	L
	>=	grösser gleich	L
2	==	gleich (identisch)	L
	!=	ungleich (nicht-identisch)	L

### 3.1.2.13 Logische Operatoren

Java implementiert die folgenden logischen Operatoren. Im Gegensatz zu den '&'- bzw. '|'-Operatoren evaluieren die '&&'- bzw. '||'-Operatoren den rechten Operanden nicht mehr, falls das Resultat bereits bestimmt ist<sup>2</sup>.

---

<sup>1</sup> Da bei einem boolean nur zwischen zwei Zuständen unterschieden wird, würde eigentlich zur Repräsentation nur ein bit benötigt. Da der Speicherzugriff aber nicht bitweise erfolgen kann, wird ein boolean intern als int dargestellt.

<sup>2</sup> Falls bei einer Disjunktion der erste Operand den Wert true hat, muss das Ergebnis unabhängig vom Wert des zweiten Operanden ebenfalls true sein.

<i>Prior.</i>	<i>Operator</i>	<i>Beschreibung</i>	<i>Assoz.</i>
1	!	Negation	R
2	&	Konjunktion	L
3	^	exklusives Oder (XOR)	L
4		Disjunktion	L
5	&&	Konjunktion	L
6		Disjunktion	L

Die folgenden screen shots stammen aus einem Programm, welches logische Schaltungen simuliert.

Abbildung 3-11, Abbildung 3-12 und Abbildung 3-13 illustrieren die Wahrheitstabellen der logischen Operatoren **Konjunktion**, **Disjunktion** und **XOR**:



**Abbildung 3-11: Konjunktion**



**Abbildung 3-12: Disjunktion**



**Abbildung 3-13: Exklusives Oder (XOR)**



**Abbildung 3-14: NOR**

Gemäss dem **Gesetz von de Morgan** ist der in Abbildung 3-14 ersichtliche Ausdruck  $\neg(a \mid b)$  äquivalent zu dem in Abbildung 3-15 ersichtlichen Ausdruck  $(\neg a \& \neg b)$ . Beide Abbildungen illustrieren den logischen Operator NOR.



**Abbildung 3-15: NOR**



**Abbildung 3-16: Halbaddierer**

Abbildung 3-16 und Abbildung 3-17 zeigen den Einsatz von logischen Gattern für Addierschaltungen:





**Abbildung 3-17: Halbaddierer**

#### *3.1.2.14 Schleifen*

**Schleifen** ermöglichen die mehrfache Ausführung von Anweisungen. Sie sind **ein Mittel zur Iteration**.

Ihrer Struktur nach kann man bei Schleifen einen **Schleifenkopf** und einen **Schleifenrumpf** unterscheiden. Im Schleifenkopf sind Kontrollinformationen für die Anzahl der Durchläufe enthalten. Die mehrfach zu iterierenden, in geschweiften Klammern stehenden Anweisungen bilden den Schleifenrumpf.

### 3.1.2.15 while Schleife

Das Schlüsselwort **while** kennzeichnet eine **while Schleife**.

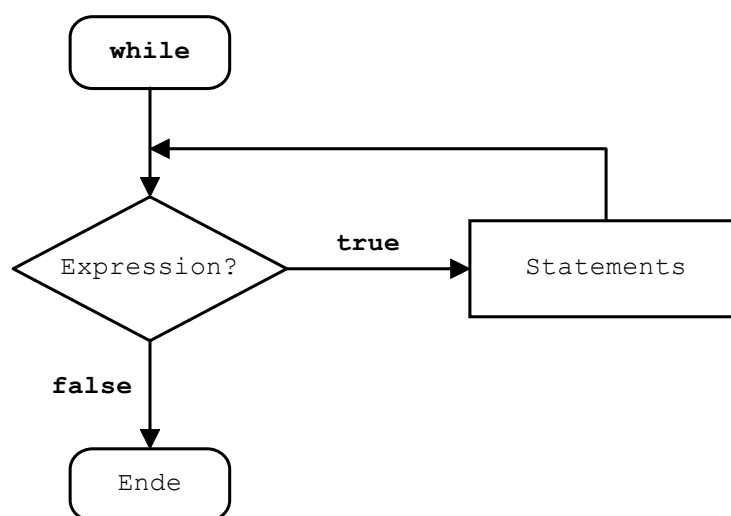
WhileStatement
<pre><b>while</b> (Expression) {     Statements }</pre>

Beispiel:

```
i = 1;  
j = 1;  
while (i<value) {  
    j = j+2;  
    i = i+j;  
}
```

Expression muss zu einem **Booleschen Ausdruck** evaluieren. Da dieser immer vor der Ausführung des Schleifenrumpfes ausgewertet wird, kann es vorkommen, dass der Rumpf einer while Schleife gar nie aufgerufen wird. Eine while Schleife ist daher **kopfgesteuert**. Ist im Rumpf nur ein statement aufgeführt, können die geschweiften Klammern weggelassen werden.

Das in Abbildung 3-18 ersichtliche Flussdiagramm zeigt die Anweisungssequenz einer while Schleife.



**Abbildung 3-18: Flussdiagramm für eine while Schleife**

### 3.1.2.16 do Schleife

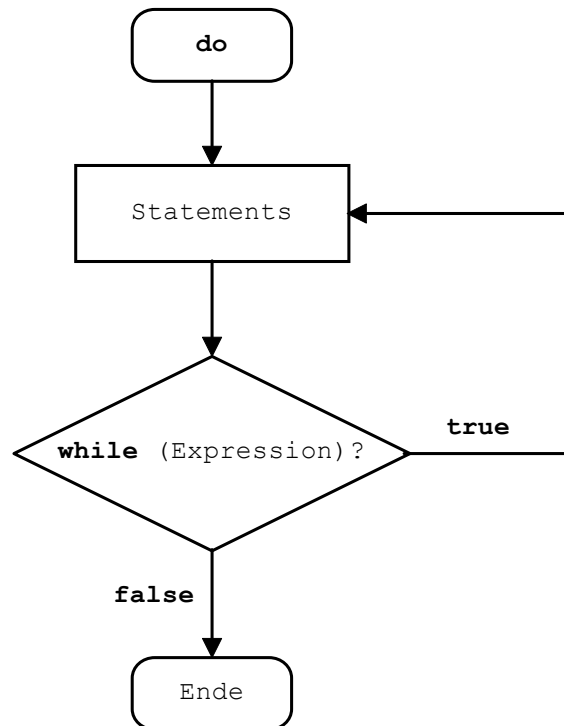
Die Schlüsselwörter **do** - **while** umrahmen den Rumpf einer **do Schleife**.

DoStatement
<pre>do {     Statements } while (Expression);</pre>

Beispiel:   do {  
            s = Character.toUpperCase  
                (Character.forDigit  
                (value%radix, radix))+s;  
            value=value/radix;  
        } while (value>0);

Bei Expression muss es sich wiederum um einen **Booleschen Ausdruck** handeln. Da dieser aber erst nach der Ausführung des Schleifenrumpfes ausgewertet wird, wird der Rumpf einer do Schleife mindestens einmal aufgerufen. Die do Schleife ist also – im Gegensatz zur while Schleife - **fussgesteuert**. Ist im Schleifenrumpf nur ein statement vorhanden, können die geschweiften Klammern weggelassen werden.

Abbildung 3-19 veranschaulicht die do Schleife mittels eines Flussdiagramms.



**Abbildung 3-19: Flussdiagramm für eine do Schleife**

#### 3.1.2.17 for Schleife

Eine **for Schleife** ist eine strukturierte Form einer while Schleife. Sie beinhaltet das Schlüsselwort **for**.

ForStatement
<pre><b>for</b> (Assignment; Expression; In-/Decrement) {     Statements }</pre>

Beispiel:    `for (i=1,j=1; i<value; i=i+j)`  
              `j = j+2;`

Der Teil Assignment des Schleifenkopfes bestimmt die Initialisierung einer oder mehrerer lokalen Schleifenvariablen. Sie erfolgt einmal, zu Beginn der Schleife. Wenn dieser Abschnitt leergelassen wird, erfolgt keine Initialisierung.

Expression beinhaltet einen **Booleschen Ausdruck**, welcher die Laufbedingung definiert. Gibt man keine Bedingung an, wird standardmässig der Boolesche Ausdruck `true` eingesetzt, was zu einer Endlosschleife führt.

Im Abschnitt In-/Decrement des Schleifenkopfes wird definiert, wie am Ende jedes Schleifendurchlaufs die im Initialisierungsteil festgelegten Schleifenvariablen aufdatiert werden müssen. Auch dieser kann weggelassen werden.

Wie auch bei einer `while` Schleife wird der Schleifenrumpf nicht zwingend durchlaufen, da die Laufbedingung vor dem ersten Durchgang überprüft wird. Somit ist also auch die `for` Schleife **kopfgesteuert**. Es können ebenfalls die geschweiften Klammern fehlen, wenn der Rumpf nur ein statement aufweist.

Abbildung 3-20 zeigt die Abarbeitung einer `for` Schleife.

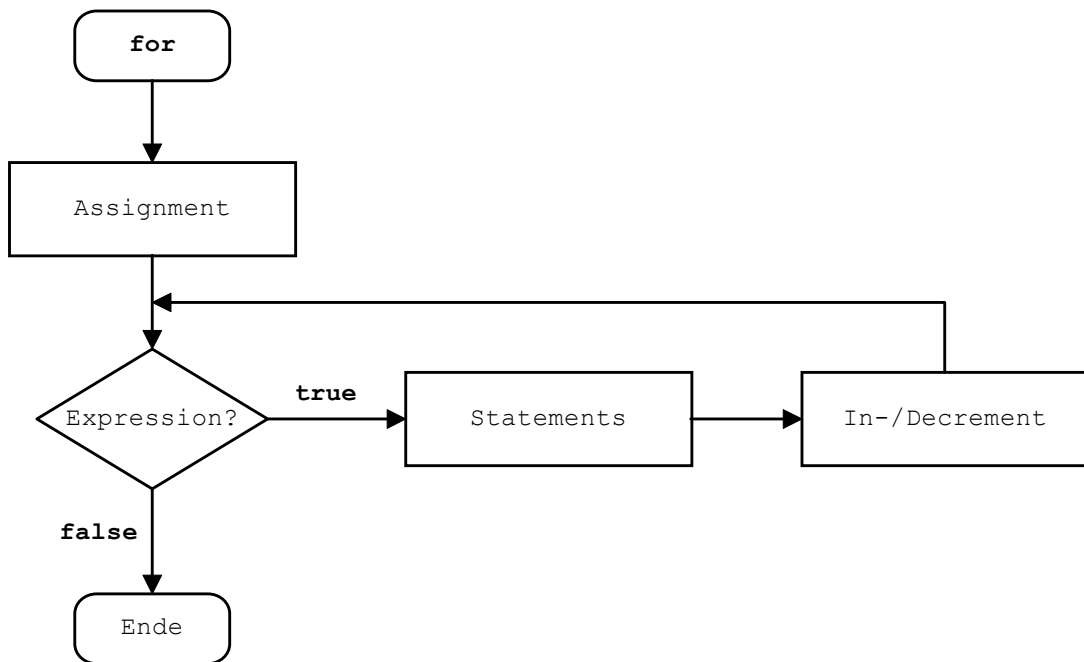


Abbildung 3-20: Flussdiagramm für eine `for` Schleife

### 3.1.2.18 if statement

Ein **if statement** ist eine **bedingte Anweisung** und realisiert als solche eine **Selektion**.

IfStatement
<pre>if (Expression) {     Statements }</pre>

Beispiel: 

```
if (odd.getState() ^ number.isOdd())  
    errorCount++;
```

Expression evaluiert zu einem Booleschen Wert. Nur wenn dieser `true` ergibt, werden die in geschweiften Klammern stehenden Anweisungen ausgeführt, ansonsten werden sie übersprungen.

Wenn man die Auswahl zwischen zwei Alternativen in Abhängigkeit von einem Booleschen Wert ausdrücken will, kann man ein **if statement** kombiniert mit einem **else statement** verwenden (siehe Abbildung 3-7).

IfElseStatement
<pre>if (Expression) {     Statements } else {     Statements }</pre>

Beispiel: 

```
if (errorCount==1)  
    message.setText("There is one error!");  
else  
    message.setText("There are "+errorCount+  
                    "errors!");
```

Nun werden, falls Expression zu `false` evaluiert, die Anweisungen nach dem **else statement** ausgeführt.

Sofern auf ein **if** bzw. **else statement** nur eine Anweisung folgt, können die geschweiften Klammern weggelassen werden.

Man hat auch die Möglichkeit, mehrere **if/else statements** ineinander zu **verschachteln**.

**IfElseStatement**

```
if (Expression) {  
    if (Expression) {  
        Statements  
    } else {  
        Statements  
    }  
}  
else {  
    Statements  
}
```

Beispiel:    Siehe Programmbeispiel „Multiple Choice“,  
             Methode actionPerformed()

Hierbei bezieht sich ein else statement immer auf ein unmittelbar davor stehendes if statement.

### 3.1.2.19 switch statement

Ein **switch statement** wird mit dem Schlüsselwort **switch** gekennzeichnet. Es erlaubt die Auswahl auszuführender Anweisungen in Abhängigkeit vom Wert eines Identifier. Hierbei darf dieser Identifier nur Werte vom Typ byte, char<sup>1</sup>, short oder int annehmen.

Der Wert, welcher ein Identifier annehmen kann, steht jeweils nach dem Schlüsselwort **case** und ist mit value\_i bezeichnet, die auszuführenden Anweisungen folgen nach dem Doppelpunkt. Hierbei dürfen auch mehrere Anweisungen vorkommen, wobei diese nicht in geschweifte Klammern eingeschlossen werden müssen.

Das Schlüsselwort **default** markiert jene Einsprungsstelle, an der fortgefahren wird, wenn keines der vorhergehenden case statements greift. Seine Verwendung ist optional.

Da ein switch statement im Prinzip den Einstieg in eine Programmsequenz in Abhängigkeit von einem Wert realisiert, werden, sobald ein case statement zutrifft, sämtliche nachfolgenden Anweisungen ausgeführt (siehe Abbildung 3-9). Um dies zu verhindern, muss man das Schlüsselwort **break** verwenden: Es bewirkt, dass das switch statement verlassen wird (siehe Abbildung 3-10).

---

<sup>1</sup> Siehe Abschnitt 3.2.

### SwitchStatement

```
switch (Identifier) {  
    case value_1: Statements; break;  
    case value_2: Statements; break;  
    .  
    .  
    default: Statements;  
}
```

```
Beispiel:  switch (errorCount) {  
            case 0:  message.setText("All answers are  
                        correct!"); break;  
            case 1:  message.setText("There is one  
                        error!"); break;  
            default: message.setText("There are "+  
                        errorCount+" errors!");  
        }
```

## 3.2 Unicode und char

Die Application „Unicode“ erlaubt die Darstellung des auf einem Rechner implementierten Zeichensatzes.

```
import java.awt.*;  
  
public class UserFrame extends Frame {  
    static final char fromChar = '\u0020';  
    static final char toChar = '\u00ff';  
  
    private void place(Component comp,int x,int y,int width,int height) {  
        comp.setBounds(x, y, width, height);  
        add(comp);  
    }  
  
    public UserFrame() {  
        TextArea textArea;  
        setTitle("Unicode");  
        setLayout(null);  
        setSize(250,350);  
        setResizable(false);  
        setVisible(true);  
        place(textArea = new TextArea(),30,30,190,290);  
        textArea.setBackground(Color.white);  
        textArea.setFont(new Font("Monaco",Font.PLAIN,12));  
    }  
}
```



```
        textArea.append(" ");
        for (int i=0;i<16;i++)
            textArea.append(Integer.toHexString(i));
        textArea.append("\n");
        char c=fromChar;
        while (c<toChar) {
            textArea.append("\n "+Integer.toHexString(c)+" ");
            for (int i=0;i<16;i++)
                textArea.append(String.valueOf(c++));
        }
    }
}

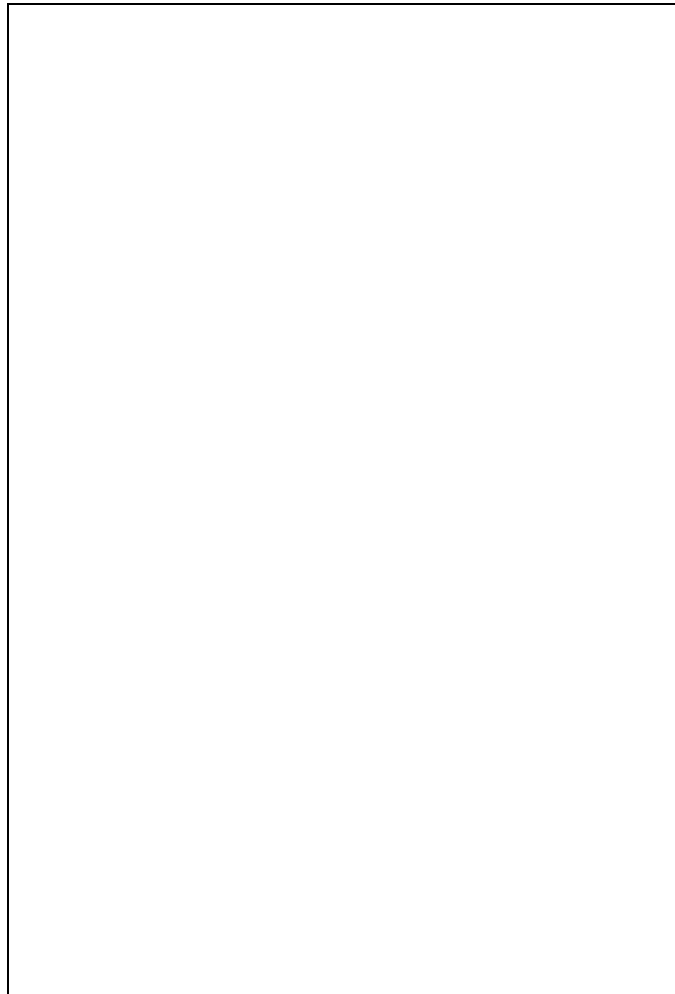
public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 3.2.1 Zum Programm

Für die Repräsentation von Zeichen wird der Datentyp `char` verwendet. Das obige Programm ermöglicht die Ausgabe von auf einem Rechner darstellbaren Zeichen.

#### 3.2.1.1 *TextArea*

Wie aus Abbildung 3-21 hervorgeht, wird für die Ausgabe der Zeichen eine sogenannte `TextArea` verwendet. Die Klasse **`TextArea`** ist eine Unterklasse von `Component` bzw. `TextComponent` und wird, wie auch ein `TextField`, zur Texteingabe verwendet. Im Unterschied zu diesem erlaubt sie aber eine mehrzeilige Eingabe.



**Abbildung 3-21: „Unicode“-Zeichensatz auf einem Macintosh**

In der Klasse `UserFrame` wird die Instanzvariable

```
TextArea textArea;
```

deklariert.

Die Anweisungen

```
place(textArea = new TextArea(), 30, 30, 190, 290);  
textArea.setBackground(Color.white);  
textArea.setFont(new Font("Monaco", Font.PLAIN, 12));  
textArea.append("      ");
```

generieren und platzieren eine Instanz der Klasse `TextArea`, setzen ihre Hintergrundfarbe auf weiss und bestimmen die Schriftart der anzuzeigenden Texte. Sämtliche verfügbaren

Farben sind in der Klasse `Color` im package `java.awt` deklariert. Die Methode `append()` fügt den als Parameter übergebenen String am Ende der `TextArea` ein.

### 3.2.1.2 Konstanten

Die Klasse `UserFrame` deklariert die beiden Variablen

```
static final char fromChar = '\u0020';  
static final char toChar = '\u00ff';
```

Sie sind vom Typ `char` (siehe Abschnitt 3.2.1.3) und werden neben der Typendeklaration zusätzlich noch mit den Schlüsselwörtern `static` und `final` bezeichnet.

Die Verwendung des Schlüsselwortes **`static`** zusammen mit dem Schlüsselwort **`final`** deklariert eine **Konstante**. Der Wert der Variablen `fromChar` und `toChar` kann also nicht durch eine Zuweisung geändert werden.

***Siehe auch:*** 3.2.2.1

### 3.2.1.3 Der Datentyp char

Die Konstanten

```
static final char fromChar = '\u0020';  
static final char toChar = '\u00ff';
```

sind vom Datentyp `char`.

Der Datentyp **`char`** repräsentiert einzelne Zeichen. Für seine Darstellung werden **16 bit** verwendet, weshalb man theoretisch  $2^{16} = 65536$  verschiedene Zeichen festlegen kann.

Java verwendet für die Codierung der Zeichen den sogenannten Unicode. Der **Unicode** ist ein Standard für eine 16-bit-Zeichenkodierung. Er wurde durch das Unicode Konsortium definiert und hat das Ziel, den Austausch und die Anzeige geschriebener Texte unterschiedlichster Sprachen zu unterstützen. Der momentane Standard definiert 38885 verschiedene Zeichen, wobei jedes Zeichen durch eine ganze, positive Zahl identifiziert wird, und die Numerierung bei Null beginnt.

Der Unicode beinhaltet den sogenannten **ASCII-Code**. ASCII steht für American Standard Code for Information Interchange. Dieser verwendet für die Zeichenkodierung 8 bit. Von den maximal möglichen 256 Zeichen sind aber lediglich die ersten 128 standardisiert. Weil es sich aber bei den Zeichen 0 bis 31 und 127 um Steuerungszeichen (z.B. Zeilenvorschub, Wagenrücklauf etc.) handelt, beinhaltet der Unicode nur die ASCII-Zeichen von 32 bis 126, welche dort konsequenterweise dieselben Positionen besetzen.

Der ASCII-Code hat sich im Vergleich zum Unicode als älterer Standard viel besser durchgesetzt; er wird praktisch von allen Plattformen unterstützt. Hingegen kann es vorkommen, dass einige Rechner heutzutage noch nicht das ganze Spektrum des Unicodes abdecken. So zeigt Abbildung 3-21 die „subjektive“ Interpretation eines Macintosh-Computers der Zeichen der Positionen 32 (hexadezimal 20) bis 255 (hexadezimal FF); die abgebildete Tabelle muss also nicht zwingenderweise mit der Unicode-Tabelle übereinstimmen, beinhaltet aber von der Position 32 bis 126 (hexadezimal 7E) den ASCII-Code.

Es ist aber zu erwarten, dass sich der Unicode in näherer Zukunft als Standard durchsetzt. Für den Programmierer ergibt sich momentan die Konsequenz, dass beim Einsatz von Sonderzeichen Vorsicht geboten ist, da deren Codierung möglicherweise auf einem anderen Rechner anders interpretiert werden könnte und somit die Portabilität nicht gewährleistet ist.

### In der Deklaration

```
static final char fromChar = '\u0020';  
static final char toChar = '\u00ff';
```

werden die Konstanten `fromChar` und `toChar` sogleich auch initialisiert: ein `char` wird in **einfachen Anführungszeichen** angegeben. Hierbei kann man entweder direkt das Zeichen oder dessen Unicode setzen. Der Unicode eines Zeichens wird als sogenannte **escape sequence** in der Form

```
\uxxxx
```

angegeben, wobei `xxxx` die hexadezimale Position des Zeichens im Unicode angibt.

Die beiden Konstanten bestimmen die untere und obere Schranke des auszugebenden Ausschnittes aus der Unicode-Tabelle.

Wie in Abbildung 3-21 ersichtlich ist, erfolgt die Ausgabe der Unicode-Tabelle in einer Art Matrix: die erste Spalte und erste Zeile bestimmen den Hexadezimalwert der Zeichen, in den restlichen Spalten bzw. Zeilen sind die Zeichen selber dargestellt. So hat beispielsweise der Kleinbuchstabe ‘a’ den Hexadezimalwert 61 (97 dezimal).

Der Aufbau dieser Matrix erfolgt in den Zeilen:

```
for (int i=0;i<16;i++)
    textArea.append(Integer.toHexString(i));
textArea.append("\n");
char c=fromChar;
while (c<toChar) {
    textArea.append("\n "+Integer.toHexString(c)+" ");
    for (int i=0;i<16;i++)
        textArea.append(String.valueOf(c++));
}
```

Die ersten zwei Anweisungen generieren die erste Zeile, welche die Hexadezimalziffern 0 bis f enthält. Die im package `java.lang` in der Klasse `Integer` deklarierte Methode `toHexString()` gibt einen String zurück, welcher den als Parameter übergebenen `int`-Wert als Hexadezimalzahl darstellt.

In der Anweisung

```
textArea.append("\n");
```

wird eine weitere **escape sequence** eingesetzt, welche bewirkt, dass ein Zeilenumbruch stattfindet. Im Ausdruck erfolgt hierdurch eine Leerzeile.

In den Folgezeilen wird eine `while` Schleife mit einer `for` Schleife verschachtelt<sup>1</sup>, um so den Rest der Matrix zu erzeugen:

```
char c=fromChar;
while (c<toChar) {
    textArea.append("\n "+Integer.toHexString(c)+" ");
    for (int i=0;i<16;i++)
        textArea.append(String.valueOf(c++));
}
```

In der ersten Anweisung wird die Schleifenvariable `c` der `while` Schleife mit `fromChar` initialisiert. Die `while` Schleife wird so lange durchlaufen, bis die obere Schranke des auszugebenden Bereiches `toChar` erreicht ist.

Die Anweisung

```
textArea.append("\n "+Integer.toHexString(c)+" ");
```

generiert die erste Spalte (d.h. bei jedem Durchgang der `while` Schleife das erste Element) einer Zeile. Die `for` Schleife erzeugt nun in einem Durchgang den Rest dieser Zeile:

```
for (int i=0;i<16;i++)
    textArea.append(String.valueOf(c++));
```

Der momentane Wert der Variablen `c` wird mit Hilfe der Methode `append()` ausgegeben. Da diese einen String als Parameter erfordert, wird `c` der Methode

---

<sup>1</sup> Hierbei befindet sich die `for` Schleife im Rumpf der `while` Schleife.

`valueOf()` übergeben. Erst nachdem die Ausgabe erfolgt ist, wird der Wert der Variablen `c` um eins erhöht (Postfix-Inkrement, siehe Abschnitt 3.1.1.11).

Da ein `char` den Unicode eines Zeichens und nicht das Zeichen selber speichert, wird er intern wie ein `int` dargestellt. Nach Aussen präsentiert er sich einfach anders: als Zeichen und nicht als Zahlenwert. Daher kann man mit ihm auch arithmetische Operationen und Vergleichsoperationen ausführen. Aufgrund ihrer ähnlichen Eigenschaften vereint man die Datentypen `char`, `byte`, `short`, `int` und `long` unter dem Begriff der **Integer-Datentypen**.

*Siehe auch: 3.2.2.2*

### 3.2.2 Syntax

#### 3.2.2.1 Konstanten

**Konstanten** werden mit den Schlüsselwörtern **`static final`** bezeichnet.

Constants
<b><code>static final</code></b> Identifier;

Beispiele: `static final char fromChar;`  
`static final char toChar;`

Das Schlüsselwort `final` bewirkt, dass der Wert einer Variablen nicht verändert werden kann. Das Schlüsselwort `static` wird für die Deklaration von lokalen Variablen weggelassen. Eine echte Konstante hingegen ist aber nur eine `final` attribuierte Klassenvariable, da eine solche wirklich nur einmal gespeichert wird.

#### 3.2.2.2 Der Datentyp `char`

Die Datentypen **`char`**, `byte`, `short`, `int` und `long` werden aufgrund ihrer Eigenschaften unter dem Begriff der **Integer-Datentypen** vereint. Hiervon heben sich die sogenannten Floating-Point-Datentypen ab, welche die Datentypen `float` und `double` zur Darstellung von Fließkommazahlen umfassen<sup>1</sup>.

---

<sup>1</sup> Siehe Übersicht sämtlicher Datentypen in Anhang A.

<i>Beschreibung</i>	<i>Wertebereich</i>	<i>Default</i>	<i>Grösse</i>
Unicode-Zeichen	von \u0000 bis \uFFFF	\u0000	16 bit

Folgende Werte kann man einer `char`-Variablen zuweisen:

```
char Identifier = 'a';
```

Das Zeichen selber, in einfachen Anführungszeichen eingeschlossen,

```
char Identifier = 97;
```

der dezimale Unicode-Wert, ohne Anführungszeichen,

```
char Identifier = '\u0061';
```

der hexadezimale Unicode-Wert, als escape sequence, in einfachen Anführungszeichen eingeschlossen.

Die **escape sequence** hat die Form

```
\uxxxx
```

wobei `xxxx` den hexadezimalen Unicode-Wert des Zeichens angibt.

## 3.3 Zusammenfassung

Dieses Kapitel befasst sich zu einem grossen Teil mit Datentypen. Ein **Datentyp** wird einer Variablen zugeordnet und beschreibt deren Wertebereich sowie die Operationen, welche auf diesen Werten ausgeführt werden dürfen. Der Datentyp einer Variablen wird in ihrer Deklaration festgelegt.

Man unterscheidet Methoden, welche einen Wert zurückliefern (**typed methods**) von solchen, die keinen Wert zurückliefern (**void methods**). Typed methods müssen in ihrer Deklaration den Datentyp des zurückgelieferten Wertes angeben. Solche Methoden müssen immer mit dem Schlüsselwort `return` verlassen werden.

Java definiert zur Repräsentation von ganzen Zahlen bzw. Zeichen die **Integer-Datentypen** `byte`, `short`, `int`, `long` und `char`. Zur Darstellung von Fließkommazahlen werden die **Floating-Point-Datentypen** `float` und `double` verwendet. Auf Werten beider Datentypgruppen dürfen arithmetische Operationen bzw. Vergleichsoperationen ausgeführt werden.

Für das Festhalten von Wahrheitswerten wird der Datentyp **boolean** verwendet. Boolesche Werte dürfen ausschliesslich in logischen Operationen eingesetzt werden.

Des weiteren wurden Programmstrukturen erläutert. **Schleifen**, ein Mittel zur Iteration, können mittels `while`, `for` und `do statements` realisiert werden. Eine Selektion kann durch den Einsatz von **if** oder **switch** statements erreicht werden.

Neu wurden auch die **Konstanten** eingeführt, welche mit den Schlüsselwörtern `static` und `final` deklariert werden.

Der bisher unbekannte Baustein einer graphischen Benutzeroberfläche **TextArea** wurde ebenfalls zum ersten Mal eingesetzt.



# 4 Strings

## Komplexe Datentypen

### Type Casting

Es werden nun die im vorhergehenden Kapitel eingeführten einfachen Datentypen den komplexen Datentypen gegenübergestellt. Zuerst wird aber noch die bereits des öfteren verwendete Klasse `String` eingehender vorgestellt. Die Darlegung basiert nicht auf einem Beispielprogramm.

## 4.1 Strings

Bisher haben wir schon mehrmals Strings verwendet. In Abschnitt 2.1.1.5 haben wir sie als Texte kennengelernt und haben auch erfahren, dass man sie mit dem '+'-Operator konkatenieren kann. In Abschnitt 3.1.1.9 wurde dann erläutert, dass ein String immer eine Instanz der Klasse `String` ist.

### 4.1.1 Die Klasse `String`

Die Klasse **`String`** ist im package `java.lang` deklariert. Sie umfasst neben diversen Konstruktoren auch verschiedene Methoden, mit deren Hilfe man `String`-Objekte bearbeiten kann.

#### 4.1.1.1 Methoden

Bereits in Abschnitt 2.4.1.1 sind wir auf die in der Klasse `String` enthaltene Methode

```
public boolean equals(Object anObject)
```

gestossen. Sie untersucht, ob die als Parameter übergebene Instanz `anObject` ein `String` und identisch mit dem die Botschaft empfangenden `String` ist. Das Boolesche Ergebnis dieses Vergleichs ist Rückgabewert dieser Methode.

Die Methode

```
public static String valueOf(int i)
```

haben wir in Abschnitt 3.1.1.9 kennengelernt und die Methode

```
public static String valueOf(char c)
```

in Abschnitt 3.2.1.3. Es gibt noch weitere solche Methoden, welche die Konvertierung eines anderen Datentyps in einen String erlauben.

Dann gibt es auch noch Methoden zur Stringmanipulation: beispielsweise kann man einen String in Teilstrings zerlegen, seine Länge ermitteln, sämtliche Grossbuchstaben durch Kleinbuchstaben - oder umgekehrt - ersetzen etc.

### 4.1.1.2 Erzeugung

Es ist möglich, eine Instanz der Klasse `String` ohne explizites Aufrufen eines Konstruktors zu generieren,

```
String name = "Rotkäppchen";
```

indem man einen Text, welcher in **doppelten Anführungszeichen** steht, einer Variablen vom Typ `String` zuweist.

Obige Anweisung bewirkt, dass eine Instanz der Klasse `String` angelegt wird, welche den Identifier `name` und den Text „Rotkäppchen“ enthält. Dasselbe Ergebnis kann auch durch den Aufruf eines in der Klasse `String` deklarierten Konstruktors erzielt werden, beispielsweise durch

```
String name = new String("Rotkäppchen");
```

### 4.1.1.3 Eigenschaften

`String`-Objekte sind **unveränderlich**. Wohl kann man der in Abschnitt 4.1.1.2 aufgeführten Variablen `name` den Wert „Wolf“ zuweisen.

```
name = "Wolf";
```

Hierbei wird aber implizit ein neues `String`-Objekt mit dem Text „Wolf“ generiert.

Des weiteren sind die Elemente eines Strings, also die einzelnen Zeichen, von links nach rechts durchnummeriert. Es ist aber zu beachten, dass die **Indizierung** bei Null beginnt und somit die Länge des Strings um eins grösser ist als der Index des letzten Elementes.

#### 4.1.1.4 String Concatenation

Wenn man den **‘+’-Operator** auf String-Objekte anwendet, wird als Resultat ein neues String-Objekt kreiert, welches durch Zusammenfügen seiner Operanden entsteht (siehe Abschnitt 2.1.1.5).

Voraussetzung einer solchen **String Concatenation** ist, dass mindestens einer der Operanden eine Instanz der Klasse `String` ist. Ist diese Bedingung erfüllt, erfolgt auch eine automatische Konvertierung allfälliger Operanden anderen Typs. Diese Konvertierung geschieht bei einfachen Datentypen (siehe Abschnitt 4.2.1.1) durch einen Type Cast (siehe Abschnitt 4.3), bei komplexen Datentypen (siehe Abschnitt 4.2.1.2) durch den Aufruf der Methode `toString()`<sup>1</sup>.

## 4.2 Einfache versus komplexe Datentypen

Wie bereits in Abschnitt 3.1.1.2 beschrieben wurde, ist der Datentyp eines Objekts die Klasse, von welcher es Instanz ist. Klassen, Interfaces<sup>2</sup> und Arrays<sup>3</sup> werden als **komplexe Datentypen** bezeichnet.

### 4.2.1 Wertsemantik versus Referenzsemantik

In Abschnitt 2.4.1.2 haben wir das Konzept der Variablen und in Abschnitt 3.1.1.2 dasjenige des Datentyps kennengelernt. Nun bedarf es weiterer Erläuterungen, um diese Konzepte in Zusammenhang mit den einfachen bzw. komplexen Datentypen zu bringen.

#### 4.2.1.1 Einfache Datentypen

Java kennt die in Abschnitt 3.1.2.9 und 3.2.2.2 vorgestellten **einfachen Datentypen** (engl. **primitive data types**) `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` und `double`. Wie Abbildung 4-1 veranschaulicht, enthält eine Variable eines einfachen Datentyps direkt den Variablenwert: der Identifier `fromChar` weist auf die Adresse im Speicher, welche den Variableninhalt enthält. Der an dieser Stelle vorgefundene Inhalt

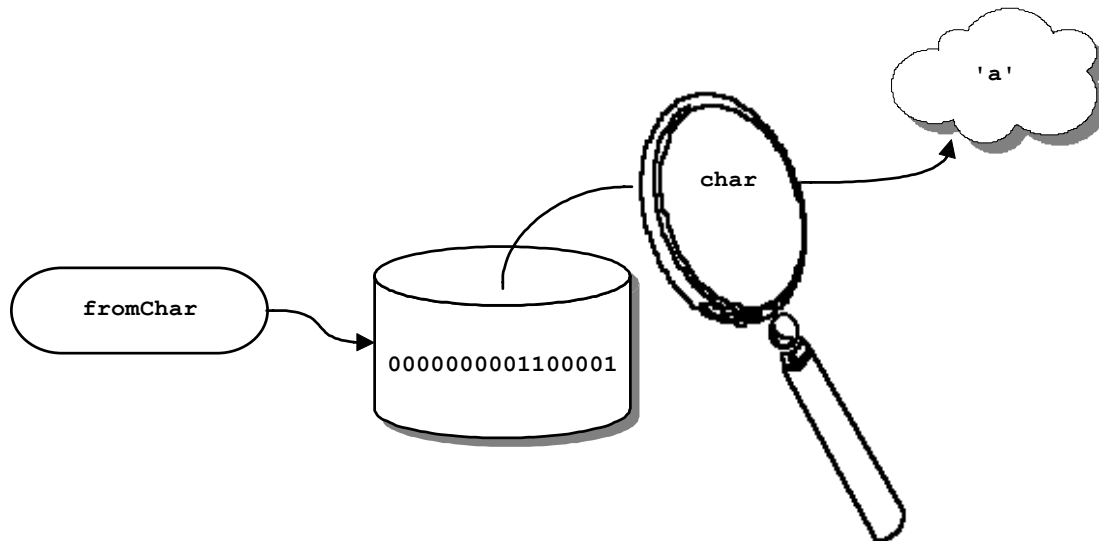
---

<sup>1</sup> Die Methode `toString()` ist bereits in der Klasse `Object` definiert und wird somit an sämtliche Unterklassen weitervererbt und eventuell redefiniert.

<sup>2</sup> Siehe Abschnitt 9.1.

<sup>3</sup> Siehe Abschnitte 7.3.1.1 und 7.3.2.1

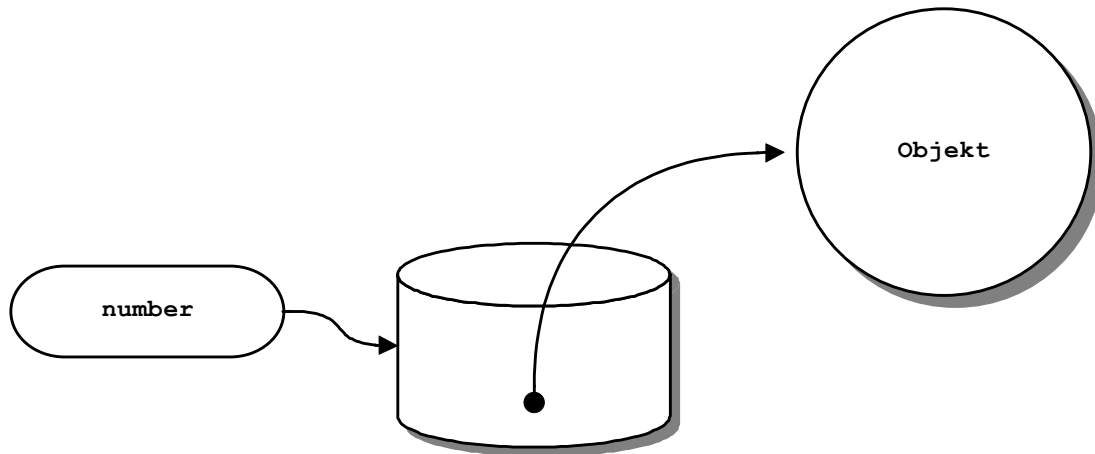
wird gemäss dem Datentyp `char` als Kleinbuchstabe `a` interpretiert. Diese Eigenschaft wird als **Wertsemantik** bezeichnet.



**Abbildung 4-1: Wertsemantik einfacher Datentypen**

### 4.2.1.2 Komplexe Datentypen

Eine Variable eines **komplexen Datentyps** (engl. **reference data types**) enthält nicht direkt ihren Wert, also das Objekt, sondern lediglich eine **Referenz** (Verweis, Zeiger; engl. **reference**, **pointer**) auf dieses (Abbildung 4-2), was als **Referenzsemantik** bezeichnet wird.



**Abbildung 4-2: Referenzsemantik komplexer Datentypen**

Der Default-Wert einer Variablen eines komplexen Datentyps ist der sogenannte **null-Wert**. Das Schlüsselwort `null` besagt, dass kein Verweis auf ein Objekt vorhanden ist und somit ein solches auch nicht existiert. Erst durch den Aufruf eines Konstruktors wird ein entsprechendes Objekt generiert und der Variablen die Objektreferenz zugewiesen.

Obwohl sich der Programmierer um die Erzeugung von Objekten kümmern muss, ist das Aufsammeln nicht mehr benötigter Objekte, die sogenannte **Garbage Collection**, dem System überlassen. Es findet sämtliche nicht mehr referenzierte Objekte und vernichtet diese.

#### 4.2.2 „Pass by Value“ versus „Pass by Reference“

##### 4.2.2.1 Einfache Datentypen

Wie in Abschnitt 4.2.1.1 erläutert wurde, steht hinter einer Variablen einfachen Datentyps immer direkt ein Wert. Aufgrund dieser Eigenschaft wird bei der Übergabe einer solchen Variablen, z.B. innerhalb eines Ausdrucks oder als Parameter, immer ein konkreter Wert übertragen, was als „**pass by value**“ bezeichnet wird.

Die Zeilen

```
int i = 3;
int j = i;
i = 2;      //j==3
```

illustrieren eine Variablenübergabe gemäss „pass by value“:

In der zweiten Anweisung wird der Wert der Variablen `i`, also die Zahl 3, der Variablen `j` zugewiesen. Da durch diese Transaktion `i` seinen Wert behält, kann man sich vorstellen, dass nicht der Originalwert von `i` sondern eine Kopie dieses Wertes an `j` übergeben wird. Wenn nun in der letzten Zeile `i` den Wert 2 erhält, hat dies keinen Einfluss auf die Variable `j`: sie hat immer noch den Wert 3.

### 4.2.2.2 Komplexe Datentypen

Im Gegensatz zu den einfachen Datentypen enthalten komplexe Datentypen eine Referenz auf ein Objekt (siehe Abschnitt 4.2.1.2). Somit wird bei einer Variablenübergabe eine Referenz und nicht ein Wert übergeben, was als „**pass by reference**“ bezeichnet wird.

Eigenschaften und Konsequenzen des „pass by reference“ demonstriert folgender Programmausschnitt:

```
Checkbox a, b;  
a = new Checkbox();  
b = a;  
a.setState(true);  
boolean state = b.getState();           //state hat den Wert true
```

In der zweiten Anweisung wird ein Objekt der Klasse `Checkbox` generiert und eine Referenz darauf der Variablen `a` zugewiesen. Darauf wird in der dritten Zeile der Inhalt der Variablen `a` der Variablen `b` zugewiesen. Da `a` aber nicht einen Wert sondern eine Referenz auf ein Objekt enthält, wird `b` somit auch ein Verweis auf die zuvor erzeugte Instanz der Klasse `Checkbox` zugewiesen. Weil nun beide Variablen auf dasselbe Objekt zeigen, hat die durch die Anweisung `a.setState(true)` ausgelöste Änderung auch einen Einfluss auf `b`, `state` hat nämlich den Wert `true`.

Möchte man hingegen tatsächlich einer Variablen eine Kopie eines Objektes zuweisen, muss man die sogenannte **clone()**-Methode verwenden, welche in der Klasse `Object` deklariert ist und in etlichen Unterklasse redefiniert wird.

### 4.2.3 Prüfen auf Gleichheit

Der **'=='-Operator** vergleicht Variableninhalte. Angewendet auf einfache Datentypen vergleicht er also deren Werte. Setzt man ihn hingegen bei komplexen Datentypen ein, werden Referenzen auf Gleichheit untersucht. In diesem Fall wird also überprüft, ob beide Variablen auf dasselbe Objekt referenzieren.

Will man aber zwei Objekte hinsichtlich ihrer Werte, also ihrer Daten vergleichen, muss man die Methode `equals()`<sup>1</sup> gebrauchen. Sie ist ebenfalls in der Klasse `Object` aufgeführt und wird in vielen Unterklassen entsprechend redefiniert.

## 4.3 Type Casting

Java ist eine **strenge getypte** Sprache: Zuweisungen sind nur zwischen Variablen gleichen oder zueinander verträglichen Typs möglich. Ansonsten ist eine explizite **Typenkonvertierung** (engl. **type cast**) erforderlich.

Eine Typenkonvertierung passiert, indem man den Zieldatentyp in runden Klammern dem zu konvertierenden Wert voranstellt:

```
float x = 7;  
int i = (int) x/2;           //i==3
```

Prinzipiell kann eine Variable eines einfachen Typs nie zu einem komplexen Datentyp konvertiert werden. Innerhalb dieser Gruppen gibt es aber Typen, welche eine Zuweisung mit oder ohne Typenkonvertierung erlauben.

Grundsätzlich gilt, dass eine konversionslose Zuweisung immer dann möglich ist, wenn dabei keine Information verloren geht.

***Siehe auch:** 4.3.4.1*

### 4.3.1 Einfache Datentypen

Untenstehende Tabelle illustriert für die einfachen Datentypen, ob sie typverträglich (☺) sind oder ob ein type cast (cast) notwendig ist. Falls aber ein Typ überhaupt nicht konvertierbar ist, wird dies mit '-' vermerkt.

---

<sup>1</sup> Sie wurde bereits im Beispielprogramm in Abschnitt 2.4.1.1 und 3.1.1.10 verwendet.

## 4.3 Type Casting

---

<i>von/zu</i>	<i>byte</i>	<i>short</i>	<i>int</i>	<i>long</i>	<i>char</i>	<i>float</i>	<i>double</i>	<i>boolean</i>
<i>byte</i>	☺	☺	☺	☺	cast	cast	cast	-
<i>short</i>	cast	☺	☺	☺	cast	cast	cast	-
<i>int</i>	cast	cast	☺	☺	cast	cast	cast	-
<i>long</i>	cast	cast	cast	☺	cast	cast	cast	-
<i>char</i>	cast	cast	☺	☺	☺	cast	cast	-
<i>float</i>	cast	cast	cast	cast	cast	☺	☺	-
<i>double</i>	cast	cast	cast	cast	cast	cast	☺	-
<i>boolean</i>	-	-	-	-	-	-	-	☺

### 4.3.1.1 Integer-Datentypen

Innerhalb des Integer-Datentyps dürfen Variablen der Typen `byte`, `short`, `int` und `long` sich selber oder einem Typ mit einem weiteren Spektrum konversionslos zugewiesen werden. Konvertiert man aber mittels eines `type cast` einen Typ eines weiteren Spektrums (z.B. `int`) zu einem mit einem engeren (z.B. `byte`), werden die „überflüssigen“ bit einfach abgeschnitten.

Der Datentyp `char` darf ohne Konvertierung sich selber und den Integers `int` und `long` zugewiesen werden. Für die Umwandlung eines `byte`, `short`, `int` oder `long` in einen `char` bedarf es aber eines `type casts`. Hierbei ist aber zu beachten, dass ein negativer `int`-Wert ein anderes Zeichen beschreibt als der Betrag seines Wertes und eine solche Zuweisung somit gar keinen Sinn macht.

### 4.3.1.2 Floating-Point-Datentypen

Für die Floating-Point-Datentypen gilt analog, dass ein Variablentyp sich selber und einem Typ eines weiteren Spektrums konversionslos zugewiesen werden darf. Eine Variable des Datentyps `float` kann also problemlos ohne `type cast` einer `double`-Variablen zugewiesen werden.

Eine Zuweisung eines Integer-Wertes zu einer Floating-Point-Variable - oder umgekehrt - erfordert jedoch zwingend einen `type cast`. Wird eine Floating-Point-Variable zu einem Integer konvertiert, wird zur nächst tieferen ganzen Zahl gerundet - die Nachkommastellen werden also abgeschnitten.



#### 4.3.1.3 Datentyp *boolean*

Der Datentyp `boolean` hingegen ist mit keinem anderen Datentyp verträglich und kann zu keinem anderen Datentyp - auch nicht umgekehrt - konvertiert werden.

### 4.3.2 Komplexe Datentypen

Grundsätzlich ist bei den komplexen Datentypen eine Zuweisung bzw. ein `type cast` nur innerhalb der Vererbungshierarchie möglich.

Eine konversionslose Zuweisung ist immer dann erlaubt, wenn man sich im Hierarchiebaum von unten nach oben bewegt, wenn man also etwas „Tieferes“ etwas „Höherem“ zuordnet. Der umgekehrte Weg erfordert einen `type cast`.

#### 4.3.2.1 Klassen

Aufgrund der Tatsache, dass eine Instanz einer Unterklasse nicht nur Instanz dieser sondern auch Instanz sämtlicher Oberklassen ist, ist es möglich, einer Variablen einer Oberklasse auch eine Instanz einer ihrer Unterklassen zuzuweisen:

```
Button b = new Button("OK");  
Component c = b;
```

Der umgekehrte Fall erfordert aber einen `type cast`: erst nachdem eine Instanz einer Oberklasse zu einer ihrer Unterklassen konvertiert wurde, kann sie einer Variablen dieser Unterklasse zugewiesen werden

```
b = (Button)c;
```

oder auch Botschaftsempfänger einer in ihrer Unterklassen deklarierten Methode sein:

```
((Button)c).getLabel();
```

Ist aber eine Klasse nicht Unterklasse einer anderen Klasse, ist weder eine Zuweisung noch ein `type cast` möglich.

#### 4.3.2.2 Interfaces

Da Interfaces<sup>1</sup> immer von Klassen implementiert werden, können grundsätzlich nur Instanzen von Klassen vom Typ eines Interface sein. Man darf also ohne weiteres einer

---

<sup>1</sup> Ausführlichere Angaben über Interfaces sind in Abschnitt 9.1 zu finden. Der vorliegende Abschnitt kann auch erst in Anschluss an Abschnitt 9.1 gelesen werden.

Variablen eines Interface-Typs eine Instanz einer Klasse zuweisen, falls diese oder eine ihrer Oberklassen das Interface implementiert.

Will man hingegen eine Instanz einer Klasse zu einem Interface-Typ konvertieren, benötigt man einen type cast. Dieser ist wiederum nur möglich, falls die Klasse wirklich das entsprechende Interface implementiert.

### 4.3.3 Implizite Konvertierung

Neben type cast gibt es in der Klassenbibliothek auch Methoden und Klassen, welche eine implizite Typenkonvertierung ermöglichen. Dies sogar manchmal in Fällen, wo gar kein type cast, also eine explizite Konvertierung, möglich wäre.

So haben wir beispielsweise bereits die in der Klasse String (siehe Abschnitt 4.1) deklarierten Methoden `valueOf()` kennengelernt, die den als Parameter erhaltenen Wert als String dargestellt zurückliefert.

Für jeden einfachen Datentyp gibt es in der Klassenbibliothek auch eine Klasse, welche diesen Datentyp repräsentiert. Die im package `java.lang` deklarierten Klassen `Byte`, `Short`, `Integer`, `Long`, `Character`, `Float`, `Double` und `Boolean` erlauben es, den Wert eines einfachen Datentyps in einem Objekt zu kapseln und ermöglichen dadurch eine **implizite Konversion** zwischen einfachen und komplexen Datentypen.

### 4.3.4 Syntax

#### 4.3.4.1 Type Cast

Bei einem **type cast** wird - in Klammern - der neue Datentyp dem zu konvertierenden Wert vorangestellt.

<b>TypeCast</b>
<code>(type) Expression;</code>

Beispiel:    `float x = 7;`  
             `int i = (int) x/2;                //i==3`

## 4.4 Zusammenfassung

Der Schwerpunkt dieses Kapitels liegt in der Einführung der komplexen Datentypen und deren Abgrenzung zu den einfachen Datentypen.

Eingangs wurde die schon des öfteren verwendete Klasse **String** eingehender betrachtet. Obwohl es sich bei einem String um einen komplexen Datentyp handelt, weist er trotzdem gewisse Eigenheiten auf, welche ihn von den übrigen komplexen Datentypen abheben. So kann man seine Erzeugung, Invariabilität, und Indizierung sowie die String Concatenation hervorhalten.

**Einfache Datentypen** unterscheiden sich von den **komplexen Datentypen** durch ihre Wertsemantik und der Tatsache, dass eine Zuweisung mittels „pass by value“ geschieht. Die komplexen Datentypen zeichnen sich durch ihre Referenzsemantik und der Zuweisung mittels „pass by reference“ aus. Überdies sei auch der für das Aufsammeln nicht mehr benötigter Objekte verantwortliche Garbage Collector erwähnt und der Unterschied zwischen einem Identitätsvergleich mit dem ‘==’-Operator und der Methode `equals()`.

Da Java eine streng getypte Sprache ist, kann eine konversionslose Zuweisung nicht immer durchgeführt werden. Manchmal ist für eine Zuweisung ein **type cast** erforderlich oder eine Zuweisung ist schon gar nicht möglich. Prinzipiell kann eine Variable eines einfachen Typs nie mit einem type cast zu einem komplexen Datentyp konvertiert werden. Die in der Klassenbibliothek deklarierten Klassen `Byte`, `Short`, `Integer`, `Long`, `Character`, `Float`, `Double` und `Boolean` erlauben aber in diesem Fall eine implizite Konvertierung.

# 5

## Event Handling ActionEvent ActionListener

Das vorliegende Kapitel erläutert das Event Handling. Die Ausführungen beziehen sich auf ein Beispielprogramm, welches das Event Handling am Beispiel von ActionEvents und einem ActionListener aufzeigt.

### 5.1 ActionEvent und ActionListener

Die folgende Application „Game“ implementiert ein Spiel, bei welchem der Benutzer gegen den Computer antritt. Hierbei hat man zu Beginn eine gewisse Anzahl von Elementen, von welchen man nun abwechselungsweise zieht. Man kann bei einem Zug höchstens bis zu drei Elemente aufs Mal nehmen, darf aber nicht passen. Wer das letzte Element wegnimmt, hat verloren.

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

public class UserFrame extends Frame implements ActionListener {
    private Label message;
    private TextField text;
    private Button turn, take1, take2, take3;

    private void place(Component comp,int x,int y,int width,int height) {
        comp.setBounds(x, y, width, height);
        add(comp);
    }

    public UserFrame() {
        setTitle("Game");
        setLayout(null);
        setSize(360,240);
        setResizable(false);
        place(message=new Label(),30, 50,300, 20);
        place(text=new TextField(),30,70,300, 20);
    }
}
```

```
place(turn=new Button(),140,140,80,20);
place(take1=new Button("TAKE ONE"),40,100,80,20);
place(take2=new Button("TAKE TWO"),140,100,80,20);
place(take3=new Button("TAKE THREE"),240,100,80,20);
turn.addActionListener(this);
take1.addActionListener(this);
take2.addActionListener(this);
take3.addActionListener(this);
text.setEchoChar('●');
setVisible(true);
newGame();
}

public void actionPerformed(ActionEvent event) {
    String command = event.getActionCommand();
    if (command.equals("START")) start();
    else if (command.equals("NEW GAME")) newGame();
    else if (command.equals("YOUR TURN")) myTurn();
    else if (command.equals("TAKE ONE")) userTakes(1);
    else if (command.equals("TAKE TWO")) userTakes(2);
    else if (command.equals("TAKE THREE")) userTakes(3);
}

public int numberOfTokens() {return text.getText().length();}

public void decrementTokens(int i) {
    text.setText(text.getText().substring(i));
}

public void setTakeButtonsVisible(boolean yes) {
    take1.setVisible(yes);
    take2.setVisible(yes);
    take3.setVisible(yes);
}

public void newGame() {
    message.setText("Enter tokens and START:");
    text.setText("");
    text.setEditable(true);
    text.requestFocus();
    turn.setLabel("START");
    setTakeButtonsVisible(false);
}

public void start() {
    switch (numberOfTokens()) {
        case 0: message.setText("Please try again to enter tokens
                                and START!");
                break;
        case 1: message.setText("Entering only one single token
                                is unfair!");
                break;
        default: text.setEditable(false);
                 myTurn();
    }
}

public void userTakes(int k) {
```

```
        if (k>numberOfTokens())
            message.setText("You can't take "+k+" tokens!");
        else {
            setTakeButtonsVisible(false);    //hide takeButtons
            decrementTokens(k);
            if (numberOfTokens()==0)          //no token left, computer wins
                computerWins();
            else {
                message.setText("You took "+k+", it's my turn!");
                turn.setLabel("YOUR TURN");
            }
        }
    }

    public void myTurn() {
        turn.setLabel("NEW GAME");
        int i = (numberOfTokens()-1)%4;
        //(numberOfTokens - i) is a multiple of 4 plus 1
        if (i==0) i=1;                        //user has a chance to win
        decrementTokens(i);
        if (numberOfTokens()==0)              //no token left, user wins
            userWins();
        else {
            message.setText("I took "+i+", it's your turn!");
            setTakeButtonsVisible(true);      //show takeButtons
        }
    }

    public void computerWins() {
        message.setText("Sorry! You lost because you took away
                        the last token");
    }

    public void userWins() {
        message.setText("Congratulations! I have to take the last
                        token, you win!");
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 5.1.1 Zum Programm

Beim Start der Application wird der Benutzer gebeten, die Anzahl der Elemente zu bestimmen, indem er durch Betätigen irgendeiner Taste die sogenannten tokens generiert. Wie auch bei einer Passworteingabe werden die effektiven Werte der aktivierten Tasten kaschiert: die tokens erscheinen im TextField als ausgefüllte Kreise (siehe Abbildung 5-1).

### **Abbildung 5-1: Game, UserFrame**

Der Computer macht den ersten Zug und gibt an, wieviele Elemente er wegnimmt. Nun ist der Benutzer an der Reihe. Mittels der Buttons TAKE ONE, TAKE TWO und TAKE THREE kann er ziehen oder mit dem Button NEW GAME zum Anfang des Spiels zurückkehren (siehe Abbildung 5-2).

### **Abbildung 5-2: Game, UserFrame**

Wie man in Abbildung 5-3 sehen kann, hat der Benutzer 3 tokens gezogen. Da der Computer nun das letzte Element nehmen muss, hat der Benutzer gewonnen (siehe Abbildung 5-4).



**Abbildung 5-3: Game, UserFrame**

**Abbildung 5-4: Game, UserFrame**

Das Programm besteht aus den zwei Klassen `UserFrame` und `TestProg`, wobei die Klasse `TestProg` wiederum den Einstiegspunkt ins Programm markiert.

Im Konstruktor `UserFrame()` erfolgt die Erstellung der graphischen Benutzeroberfläche. Er hat die Instanzvariablen `message` (`Label`), `text` (`TextField`) und die Buttons `turn`, `take1`, `take2` und `take3`. Das `TextField` `text` wird zur Darstellung der tokens verwendet, das `Label` `message` ermöglicht die Ausgabe eines Kommentars oberhalb des `TextFields`.

Die durch

```
text.setEchoChar('●');
```

an das `TextField` `text` versandte Botschaft `setEchoChar()` bewirkt, dass die Eingabe des Benutzers maskiert wird, das als Parameter übergebene Zeichen vom Typ `char` wird zur Verschlüsselung verwendet.

Interessant ist auch die Tatsache, dass der Button `turn` im Gegensatz zu den Buttons `take1`, `take2` und `take3` bei seiner Instanzierung nicht beschriftet wird<sup>1</sup>:

```
place(turn=new Button(),140,140,80,20);
place(take1=new Button("TAKE ONE"),40,100,80,20);
place(take2=new Button("TAKE TWO"),140,100,80,20);
place(take3=new Button("TAKE THREE"),240,100,80,20);
```

Die Methode

```
public int numberOfTokens() {return text.getText().length();}
```

liefert als Rückgabewert die Anzahl der im `TextField` vorhandenen Elemente zurück. Mit dem Aufruf `text.getText()` erhält man den im `TextField` `text` enthaltenen String. An diesen wird die Botschaft `length()` versandt, welche die Länge des Strings und somit die Anzahl der Elemente zurückgibt.

Das Ziehen der Elemente und somit das Reduzieren der tokens im `TextField` ermöglicht die Methode

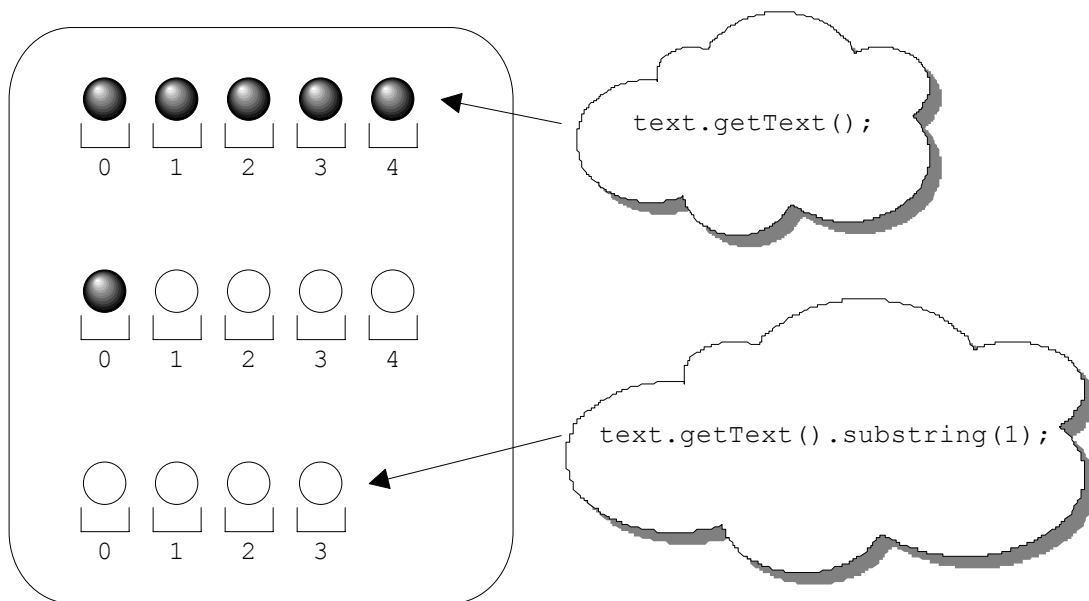
```
public void decrementTokens(int i) {
    text.setText(text.getText().substring(i));
}
```

Dem durch den Aufruf `text.getText()` erhaltenen String wird die Botschaft `substring()` gesandt. Sie erfordert als Parameter eine Indexangabe und liefert denjenigen Teilstring zurück, der aus den Elementen vom Beginn der deklarierten Indexposition aus bis zum Ende des Strings besteht. Dieser Teilstring wird nun mittels

---

<sup>1</sup> Der neu verwendete Konstruktor `Button()` verlangt keinen Parameter und generiert einen Button ohne Beschriftung.

`text.setText()` im TextField angezeigt. Da die Indizierung eines Strings bei Null und nicht bei eins beginnt, kann der Aufruf `substring(i)` als das Wegnehmen von `i` Elementen interpretiert werden. Obwohl es für den Benutzer so aussieht, als würden die tokens von hinten her weggenommen, wird der String tatsächlich von vorne her reduziert. Der falsche Eindruck entsteht dadurch, dass ein String in einem TextField links ausgerichtet ist:



**Abbildung 5-5: Funktionsweise der Methode substring()**

Am Ende des Konstruktors und auch jedes Mal, wenn der Benutzer den Button NEW GAME aktiviert, wird die Methode

```
public void newGame() {
    message.setText("Enter tokens and START:");
    text.setText("");
    text.setEditable(true);
    text.requestFocus();
    turn.setLabel("START");
    setTakeButtonsVisible(false);
}
```

aufgerufen. Sie ermöglicht dem Benutzer die Eingabe der tokens und fordert ihn zum Spielen auf (siehe Abbildung 5-1).

Die Botschaft `setEditable()` setzt das empfangende Objekt in Abhängigkeit vom Booleschen Parameter in den Editiermodus. Durch die Anweisung

`text.setEditable(true)` kann der Benutzer also Eingaben in das `TextField` machen.

Aufgrund der Botschaft `requestFocus()` erscheint im `TextField text` der Cursor. Man kann somit mit der Eingabe der tokens beginnen.

Durch die Anweisung

```
turn.setLabel("START");
```

erhält der Button `turn` die Beschriftung „START“.

Die Methode

```
public void start() {
    switch (numberOfTokens()) {
        case 0: message.setText("Please try again to enter tokens
                                and START!");
                break;
        case 1: message.setText("Entering only one single token
                                is unfair!");
                break;
        default: text.setEditable(false);
                 myTurn();
    }
}
```

die immer dann aufgerufen wird, wenn der Benutzer den START-Button aktiviert, überprüft, ob mindestens zwei tokens eingegeben wurden. Falls dies der Fall ist, wird der Editiermodus von `text` deaktiviert und die Methode `myTurn()` aufgerufen.

```
public void myTurn() {
    turn.setLabel("NEW GAME");
    int i = (numberOfTokens()-1)%4;
    //(numberOfTokens - i) is a multiple of 4 plus 1
    if (i==0) i=1; //user has a chance to win
    decrementTokens(i);
    if (numberOfTokens()==0) //no token left, user wins
        userWins();
    else {
        message.setText("I took "+i+", it's your turn!");
        setTakeButtonsVisible(true); //show takeButtons
    }
}
```

Diese Methode bildet nun gewissermassen das Herzstück des ganzen Programms; in ihr ist nämlich die Spielstrategie des Computers enthalten. In den Zeilen

```
int i = (numberOfTokens()-1)%4;
//(numberOfTokens - i) is a multiple of 4 plus 1
if (i==0) i=1;
```

wird berechnet, wieviele Elemente der Computer ziehen muss, damit seine Gewinnchancen möglichst hoch sind. Hierbei ist der Kommentar in diesen Zeilen eine

**Zusicherung** (engl. **assertion**): die Aussage ist an dieser Stelle im Programm immer gültig und widerspiegelt die Spielstrategie.

Überlegen wir uns nun anhand verschiedener Fälle, wie diese Strategie aussehen muss. Wenn 2, 3 oder 4 Elemente im TextField vorhanden sind, gewinnt derjenige, welcher den nächsten Zug macht. Falls nun aber 5 tokens vorrätig sind, hat derjenige, der an der Reihe ist, garantiert verloren, denn mit jedem möglichen Zug tritt der Fall ein, dass entweder 2, 3 oder 4 Elemente übrigbleiben, was für den Mitspieler bedeutet, dass dieser gewinnt (vorausgesetzt, er macht keinen Fehler). Sind nun aber 6 Elemente vorhanden, kann man den „Spiess umdrehen“: man nimmt nun nur 1 token weg, damit für den Gegenspieler die „unglückseligen“ 5 Elemente übrigbleiben, und man ist siegessicher. Dies gilt auch noch für 7 und 8 Elemente. Bei 9 Elementen aber trifft der selbe Fall ein wie vorher bei 5: derjenige, der Ziehen muss, hat verloren, denn mit jedem Zug, den er macht, bleiben für den Gegenspieler die siegessicheren 8, 7 oder 6 Elemente übrig. Das Ganze wiederholt sich immer wieder mit einer Periodizität von vier<sup>1</sup>. Die untenstehende Tabelle soll dies veranschaulichen. Hierbei gibt Chance die Gewinnwahrscheinlichkeit für denjenigen, welcher den nächsten Zug macht, an.

Vorrat	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Zug	?	1	2	3	?	1	2	3	?	1	2	3	?	1
Chance	☹	☺	☺	☺	☹	☺	☺	☺	☹	☺	☺	☺	☹	☺

Wie aus obiger Tabelle hervorgeht, muss der Computer also versuchen, seinen Zug so zu wählen, dass 1, 5, 9, 13, etc. Elemente übrig bleiben. Er muss also versuchen, so viele Elemente wegzunehmen, dass er „ein Vielfaches von vier plus eins“<sup>2</sup> tokens dem Benutzer hinterlässt.

Die Antwort auf obige Frage, wieviel Elemente man wegnehmen muss, bildet im Prinzip die Differenz auf „ein Vielfaches von vier plus eins“:

```
int i = (numberOfTokens()-1)%4;
```

Falls nun aber der Fall eintritt, dass, wenn der Computer ziehen will, bereits „ein Vielfaches von vier plus eins“ vorliegt, dann erhält man nach obiger Formel für  $i$  den Wert 0. In einer solchen Situation ist der Benutzer derjenige, der gewinnt. Vorausgesetzt

<sup>1</sup> Die Periodizität hängt von der maximalen Anzahl der Elemente, die man ziehen darf, ab. Dürfte man auch vier tokens wegnehmen, wäre die Periodizität fünf.

<sup>2</sup> Die Zahl vier ergibt sich aus der Periodizität von vier. Dass man eins addieren muss, hat seinen Grund darin, dass man ein Element übriggelassen muss, um zu gewinnen.

*natürlich, er macht keine Fehler. Man bestimmt dann, dass der Computer 1 Element ziehen soll:*

```
if (i==0) i=1;
```

Wenden wir uns nun den restlichen Anweisungen der Methode zu:

Durch das erste statement

```
turn.setLabel("NEW GAME");
```

wird dem Button turn die Beschriftung „NEW GAME“ zugewiesen.

In den Zeilen

```
int i = (numberOfTokens()-1)%4;  
if (i==0) i=1;
```

wird, wie bereits ausführlich erläutert, die Anzahl der zu ziehenden Elemente festgelegt, welche dann aufgrund des Aufrufs

```
decrementTokens(i);
```

tatsächlich weggenommen werden.

Im Abschnitt

```
if (numberOfTokens()==0) //no token left, user wins  
    userWins();  
else {  
    message.setText("I took "+i+", it's your turn!");  
    setTakeButtonsVisible(true); //show takeButtons  
}
```

wird der weitere Spielverlauf festgelegt. Sind nach dem Zug des Computers keine Elemente mehr im TextField, hat der Benutzer gewonnen. Der Aufruf der Methode `userWins()` bewirkt, dass das Label `message` den Text „Congratulations! I have to take the last token, you win!“ anzeigt. Sind aber immer noch tokens vorhanden, gibt der Computer an, wieviele Elemente er gezogen hat. Die Buttons TAKE ONE, TAKE TWO und TAKE THREE werden nun sichtbar.

Wenn der Benutzer durch Aktivierung eines TAKE-Buttons Elemente wegnimmt, wird die Methode

```
public void userTakes(int k) {
    if (k>numberOfTokens())
        message.setText("You can't take "+k+" tokens!");
    else {
        setTakeButtonsVisible(false);           //hide takeButtons
        decrementTokens(k);
        if (numberOfTokens()==0)                 //no token left, computer wins
            computerWins();
        else {
            message.setText("You took "+k+", it's my turn!");
            turn.setLabel("YOUR TURN");
        }
    }
}
```

ausgeführt. In ihr wird sichergestellt, dass der Benutzer nicht mehr tokens ziehen kann, als noch vorhanden sind. Ansonsten wird der Zug realisiert. Sollten nach der Wegnahme keine Elemente mehr vorhanden sein, hat der Computer gewonnen. Es erfolgt die Ausgabe des Textes „Sorry! You lost because you took away the last token“. Ansonsten wird angezeigt, wieviele Element der Benutzer gezogen hat und dem Button `turn` die Beschriftung „YOUR TURN“ zugewiesen, damit der Benutzer den Computer nun zum nächsten Zug auffordern kann.

#### 5.1.1.1 Event Handling

*Johny Gernimwind liegt es sehr am Herzen, dass das Rotkäppchen ein interaktives Puppentheater wird. Jedes Kind im Publikum hat ein Glöckchen, dank dessen es das Rotkäppchen oder den Jäger warnen kann. Wenn der Jäger zum Beispiel zum Haus der Grossmutter kommt, müsste er gemäss dem Märchen die Grossmutter zuerst im Haus suchen. Erst dann findet er den schlafenden, vollgefressenen Wolf auf der Wiese hinter dem Haus. Sollte er nun aber das laute Schellen der Kinder vernehmen, geht er direkt hinters Haus, um die Grossmutter und das Rotkäppchen aus dem Bauch des bösen Wolfs zu befreien. Auch werden die Kinder wohl das Rotkäppchen vor der Grossmutter mit den grossen Ohren, Augen und Händen warnen. Hierauf soll das Rotkäppchen zu flüchten versuchen - natürlich ist der Wolf aber schneller. Sollten die Glöckchen nicht ertönen, nimmt das Märchen seinen üblichen Verlauf. Im Gegensatz zum Rotkäppchen und zum Jäger darf der Wolf nicht auf das Schellen der Kinder hören, weil sie ihn in die Irre führen würden.*

So wie beim Rotkäppchen eine Interaktion zwischen dem Publikum und den Puppen (Rotkäppchen und Jäger) stattfindet, kann in einem Java-Programm der Benutzer mit dem Programm interagieren. Wie bereits in Abschnitt 2.1.1.4 erwähnt, wird eine solche Interaktion durch das **Event Handling** möglich.

Im Puppentheater wird die Kommunikation zwischen den Kindern und den Marionetten durch die Glöckchen realisiert, in einem Java-Programm mit Maus und Tastatur.

In Analogie zur Erzeugung eines Tones durch das Schellen hat im Programm die Aktivierung einer Komponente die Erzeugung eines **Events** (Ereignis) zur Folge. Hierbei ist ein solcher Event wiederum ein Objekt, nämlich eine Instanz einer Eventklasse. Man unterscheidet in Abhängigkeit vom Typ der generierenden Komponente verschiedene, bereits vordefinierte Eventklassen. So erzeugen beispielsweise Buttons sogenannte **ActionEvents** und Checkboxes sogenannte **ItemEvents**. Ein Event-Objekt kapselt Informationen über den Auslöser, den Typ oder den Zeitpunkt des Ereignisses in sich. Es stellt auch Methoden zur Verfügung, die einen Zugriff auf seine Attributwerte erlauben.

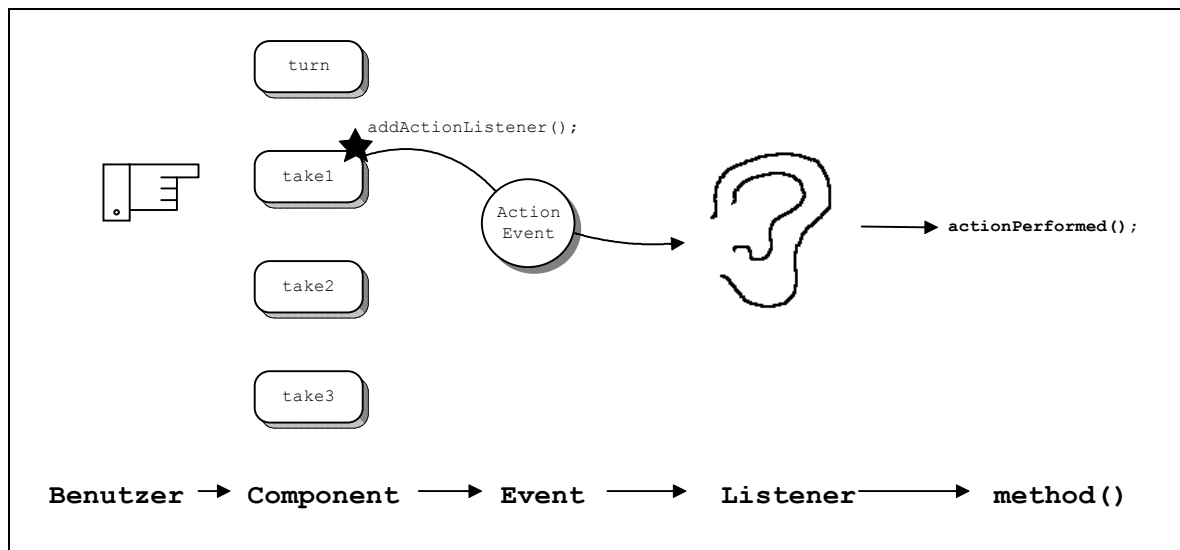
Im Puppentheater sind es der Jäger und das Rotkäppchen, welche das Schellen der Kinder hören. Wer „hört“ nun im Programm auf die von den Komponenten erzeugten Ereignis-Objekte? Dort sind es die sogenannten **Listeners**. Für jeden Event-Typ gibt es einen zugehörigen Listener, dessen Schnittstelle bereits in der Java-Klassenbibliothek deklariert ist. So hört beispielsweise ein sogenannter **ActionListener** auf Objekte der Klasse `ActionEvent`. Objekte anderen Typs aber ignoriert er. Was ist nun die Aufgabe eines Listener?

Ein Listener deklariert eine **bestimmte Methode**, welche jedes Mal, wenn ein Ereignis eintritt, aufgerufen wird. So wie im Puppentheater festgelegt ist, wie z.B. der Jäger aufgrund des Schellens reagieren muss, bestimmt diese Methode den weiteren Programmverlauf. Im Fall des `ActionListener` wird die Methode **`actionPerformed()`** aufgerufen.

Im Puppentheater wurde durch den Regisseur Johnny Gernimwind bestimmt, dass nur das Rotkäppchen und der Jäger auf das Schellen der Kinder achten sollen, nicht aber der Wolf. In einem Java-Programm kann man angeben, auf welche Component-Events ein Listener „hören“ soll, indem man ihn bei einer Komponente **registriert**. Ein Listener empfängt somit nur Ereignis-Objekte derjenigen Komponenten, bei welchen er registriert wurde. So kann man beispielsweise mit der Botschaft **`addActionListener()`** einen `ActionListener` bei einem Button registrieren.

Abbildung 5-6 demonstriert den Mechanismus des Event Handling anhand des Beispielprogramms. Gezeigt wird der Ablauf, falls der Benutzer den Button `take1` aktiviert.





**Abbildung 5-6: Illustration des Event Handling am Programmbeispiel**

Listeners werden durch **Interfaces**<sup>1</sup> realisiert. Ein Interface ist, wie auch eine Klasse, ein Konzept der Objektorientierung. Da es sogenannte abstrakte Methoden<sup>2</sup> enthält, wird es zur Definition von Methodenschnittstellen eingesetzt.

Bei einer **abstrakten Methode** ist lediglich der Methodenkopf deklariert. Ihr Rumpf ist leer. Die Deklaration der gesamten Methode, also der Methodenkopf einschliesslich ihrem Rumpf, was als **Implementierung** bezeichnet wird, erfolgt in einer anderen Klasse.

Eine **Klasse**, welche ein **Interface** sogenannten **implementiert**, muss für sämtliche im Interface deklarierten abstrakten Methoden einen Methodenrumpf aufweisen. Hierbei kann es mehrere solche Klassen geben, welche jeweils unterschiedliche Implementierungen für die abstrakten Methoden deklarieren.

Da abstrakte Methoden einen leeren Rumpf aufweisen, sind es nicht die Listener-Interfaces, welche bei einer ereignisgenerierenden Komponente registriert werden, sondern die das Interface implementierenden Klassen. Aufgrund der Tatsache, dass eine solche Klasse immer Instanz des implementierenden Interface ist, bezeichnet man sie auch als Listener.

Die Deklaration eines Listener in der Klassenbibliothek als Interface hat den Vorteil, dass zwar bestimmt ist, welche Methode aufgrund eines eintreffenden Ereignisses aufgerufen

<sup>1</sup> Auf Interfaces wird in Abschnitt 9.1 tiefer eingegangen.

<sup>2</sup> Abstrakte Methoden werden in Abschnitt 8.1 behandelt.

werden muss, die eigentliche Ereignishandhabung aber von Programm zu Programm massgeschneidert werden kann.

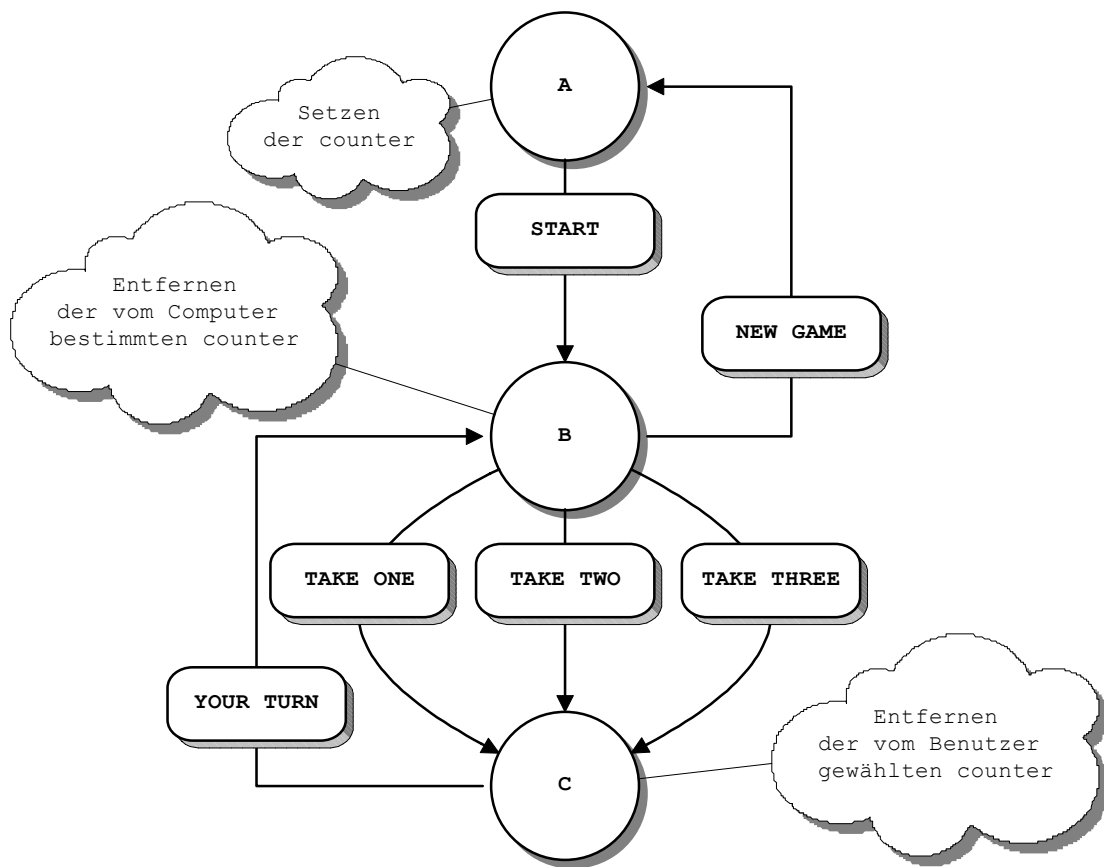
Schauen wir uns nun alles am Beispielprogramm an:

Damit wir überhaupt das Event Handling realisieren können, darf die import Anweisung

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

nicht fehlen. Im package `event` ist beispielsweise das Interface `ActionListener` und die Eventklasse `ActionEvent` deklariert.

Bevor wir aber ausführlicher das Quellprogramm betrachten, ist es sinnvoll, den Programmverlauf zu verstehen. Dieser wird anhand eines Zustandsdiagramms in Abbildung 5-7 illustriert. Hierbei sind die in Abbildung 5-7 bezeichneten Zustände in den screen shots des Beispielprogramms ersichtlich: Abbildung 5-1 zeigt einen screen shot in Zustand A, Abbildung 5-2 und Abbildung 5-4 je einen in Zustand B und Abbildung 5-3 schlussendlich illustriert einen screen shot in Zustand C. Ein Zustandswechsel erfolgt jeweils aufgrund der Aktivierung eines der Buttons `START`, `NEW GAME`, `YOUR TURN`, `TAKE ONE`, `TAKE TWO` und `TAKE THREE`.



**Abbildung 5-7: Zustandsdiagramm für Game**

Zuerst untersucht man, welche Komponenten im Programm Ereignisse erzeugen und welche dieser Ereignisse man behandeln möchte. Aus Abbildung 5-7 geht hervor, dass die Buttons `turn`, `take1`, `take2` und `take3` den Programmverlauf bestimmen. Demzufolge sollte man die durch sie erzeugten Ereignisse abfangen.

Instanzen der Klasse `Button` generieren `ActionEvents`. Der zugehörige Listener ist ein `ActionListener`, welcher die abstrakte Methode `actionPerformed()` implementiert und mit Hilfe der Methode `addActionListener()` beim jeweiligen `Button` registriert wird.

Wenn wir nun die von den Buttons generierten Instanzen der Klasse `ActionEvent` abfangen möchten, müssen wir einerseits eine Klasse deklarieren, welche den `ActionListener` implementiert, und andererseits diese Klasse bei den Buttons als `ActionListener` registrieren.

```
public class UserFrame extends Frame implements ActionListener
```

Das Schlüsselwort **implements** besagt, dass die Klasse UserFrame das Interface ActionListener implementiert. Dies hat zur Folge, dass sie eine Implementation für dessen abstrakte Methode

```
public abstract void actionPerformed(ActionEvent e)
```

liefern muss.

Da die im UserFrame deklarierte Methode actionPerformed() die gleichnamige abstrakte Methode des Interface ActionListener implementiert, ist sie nicht mehr abstrakt. Das Schlüsselwort **abstract** in der Methodenschnittstelle entfällt.

```
public void actionPerformed(ActionEvent event) {  
    String command = event.getActionCommand();  
    if (command.equals("START")) start();  
    else if (command.equals("NEW GAME")) newGame();  
    else if (command.equals("YOUR TURN")) myTurn();  
    else if (command.equals("TAKE ONE")) userTakes(1);  
    else if (command.equals("TAKE TWO")) userTakes(2);  
    else if (command.equals("TAKE THREE")) userTakes(3);  
}
```

Die Methode actionPerformed() erhält den durch den Button generierten ActionEvent als Parameter. Dieser wird innerhalb der Methode mit event bezeichnet. In der Anweisung

```
String command = event.getActionCommand();
```

liefert die Botschaft getActionCommand() die Beschriftung desjenigen Buttons zurück, der das Ereignis ausgelöst hat. Dieser String wird dann der lokalen Variablen command zugewiesen. Die Methode getActionCommand() ist in der Klasse ActionEvent deklariert.

Nun wird in der Anweisungssequenz

```
if (command.equals("START")) start();  
else if (command.equals("NEW GAME")) newGame();  
else if (command.equals("YOUR TURN")) myTurn();  
else if (command.equals("TAKE ONE")) userTakes(1);  
else if (command.equals("TAKE TWO")) userTakes(2);  
else if (command.equals("TAKE THREE")) userTakes(3);
```

in Abhängigkeit vom ereignisgenerierenden Button der Programmverlauf bestimmt: von Fall zu Fall wird eine andere Methode aufgerufen, wodurch der in Abbildung 5-7 ersichtliche Zustandswechsel erfolgt.

Damit nun der ActionListener UserFrame die Ereignisobjekte der Buttons turn, take1, take2 und take3 „hört“, muss er bei ihnen registriert werden. Dies geschieht ebenfalls in der Klasse UserFrame:

```
turn.addActionListener(this);  
take1.addActionListener(this);  
take2.addActionListener(this);  
take3.addActionListener(this);
```

Die in der Klasse `Button` deklarierte Methode `addActionListener()` erfordert als Parameter eine Instanz des `ActionListener`. Da die Klasse `UserFrame` im Beispiel den `ActionListener` implementiert, wird der Methode `addActionListener()` das Schlüsselwort `this`, also eine Instanz der Klasse `UserFrame`, übergeben (siehe Abschnitt 2.3.1.2).

Der Button `turn` ändert jeweils im Programmverlauf seine Beschriftung und hierdurch auch seine Funktion. Wie aus der Methode `actionPerformed()` hervorgeht, wird der Programmverlauf nicht aufgrund des Identifier eines aktivierten Buttons, sondern aufgrund seiner Beschriftung bestimmt. So handelt es sich bei den vermeintlichen Buttons `START`, `NEW GAME` und `YOUR TURN` um eine einzige Instanz, nämlich um den Button `turn`.

Im Programm wird die Benutzeroberfläche vom eigentlichen Kern - den Spielregeln - getrennt. Im Konstruktor `UserFrame()` wird die graphische Benutzeroberfläche generiert, in den Methoden `newGame()`, `start()`, `userTakes()`, `myTurn()` sind die Spielregeln definiert. Koordiniert werden diese beiden Teile in der Methode `actionPerformed()`: sie bestimmt aufgrund eintreffender Events der graphischen Benutzeroberfläche den Spielverlauf, indem sie die entsprechende Methode aufruft. Dank dieser Trennung ist es nun auch möglich, das Aussehen der graphischen Benutzeroberfläche zu ändern, ohne dass schwerwiegende Programmänderungen nötig sind. Grundsätzlich ist auch ein modular vorliegendes Programm, also ein Programm, welches in funktionale Einheiten zerfällt, viel wartungsfreundlicher.

***Siehe auch:** 5.1.2.1*

## 5.1.2 Syntax

### 5.1.2.1 Event Handling

Folgende Zusammenstellung gibt einen Überblick, wie man bei der Umsetzung des **Event Handling** vorgehen kann:

EventHandling
<ol style="list-style-type: none"><li>1. Bestimmen derjenigen Komponenten, deren Events man behandeln möchte.</li><li>2. Bestimmen der Eventtypen, welche die Komponenten generieren.</li><li>3. Implementieren des zu einem Eventtyp gehörenden Interface.</li><li>4. Registrieren der implementierenden Klasse bei der eventerzeugenden Komponente.</li></ol>



Welche Events eine Komponente generiert, ist in einer Referenzquelle bei der jeweiligen Komponente angegeben. Es gibt die folgenden im package `java.awt.event` deklarierten Eventklassen mit zugehörigen Listenerinterfaces:

Eventklasse	Listenerinterface
ActionEvent	ActionListener
AdjustmentEvent	AdjustmentListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener
InputEvent <sup>1</sup>	
ItemEvent	ItemListener
KeyEvent	KeyListener
MouseEvent	MouseListener, MouseMotionListener
PaintEvent <sup>2</sup>	
TextEvent	TextListener
WindowEvent	WindowListener

Wie aus der Tabelle zu entnehmen ist, endet ein **Identifier** einer Eventklasse immer mit dem Wort „Event“, derjenige eines Listenerinterface mit „Listener“. Der erste Teil des Identifier stimmt überein, so heisst beispielsweise der zur Klasse `ActionEvent` zugehörige Listener auch `ActionListener`.

Analoges lässt sich auch zu den Bezeichnern der **Registriermethoden**, welche in der Klasse der ereignisgenerierenden Komponente deklariert sind, erwähnen: sie beginnen mit dem Wort „add“, auf welches der Identifier des zu registrierenden Listener folgt (z.B. `addActionListener()`).

Es ist auch möglich, einen einmal registrierten Listener wieder zu **deaktivieren**. Hierzu verwendet man die in der Klasse der ereignisgenerierenden Komponenten deklarierte „remove“-Methode. Sie beginnt mit dem Wort „remove“, auf welches der Identifier des zu registrierenden Listener folgt (z.B. `removeActionListener()`).

Abbildung 5-8 zeigt die Vererbungshierarchie der Eventklassen.

---

<sup>1</sup> `InputEvent` ist die Oberklasse von `KeyEvent` und `MouseEvent`.

<sup>2</sup> Zu `PaintEvent` gibt es kein Listenerinterface, da solche Events meist nur systemintern verwendet werden.

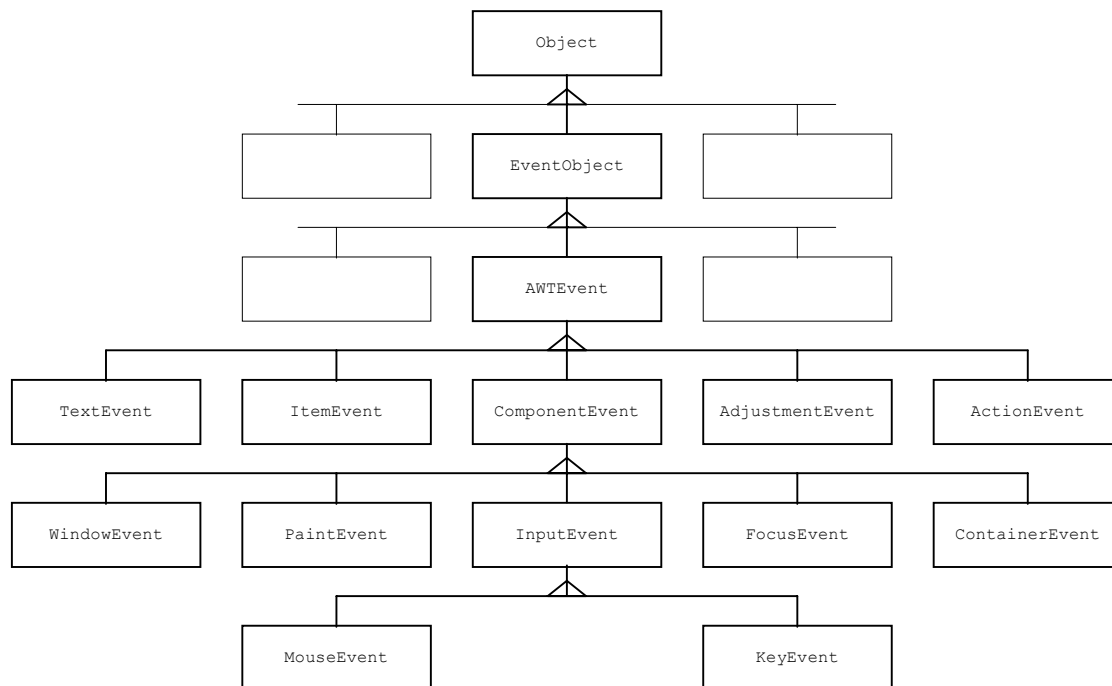


Abbildung 5-8: Einbettung der Eventklassen in die Klassenbibliothek

## 5.2 Zusammenfassung

Das **Event Handling** erlaubt einem Benutzer, mit dem Programm zu interagieren. Die Kommunikation findet über Maus und Tastatur statt:

- ◆ Eine durch den Benutzer aktivierte Komponente erzeugt ein entsprechendes Ereignisobjekt. Dieser **Event** ist eine Instanz einer **Eventklasse**. Für jeden Ereignistyp gibt es eine zugehörige Eventklasse.
- ◆ Ein **Listener-Interface** deklariert abstrakte Methoden und „hört“ Ereignisse eines bestimmten Typs. Es gibt also für jede Eventklasse auch ein zugehöriges Listener-Interface. Aufgrund des Eintreffens eines Ereignisses werden die im Listener-Interface deklarierten Methoden aufgerufen.
- ◆ Die abstrakten Methoden eines Listener-Interface werden von einer Klasse implementiert. Eine solche **implementierende Klasse** ist Instanz des jeweiligen Interface und wird deshalb auch als Listener bezeichnet. Schlussendlich werden die Methoden der implementierenden Klassen ausgeführt.



- ◆ Damit eintreffende Ereignisse wahrgenommen werden können, muss eine Instanz einer implementierenden Klasse bei der jeweiligen ereignisgenerierenden Komponente **registriert** werden.
- ◆ Listeners können auch wieder **deaktiviert** werden. Dies hat zur Folge, dass die im Listener-Interface deklarierten abstrakten Methoden nicht mehr aufgerufen werden.

# 6 Instanz-/Klassenmethoden ItemEvent, ItemListener Visibility Modifiers

In diesem Kapitel wird die Unterteilung von Methoden in Instanz- und Klassenmethoden vollzogen. Es wird ebenfalls auf die Sichtbarkeitsattribuierung mit Hilfe der sogenannten visibility modifiers eingegangen. Das ganze Kapitel basiert auf einem Beispielprogramm, welches in zwei Versionen vorliegt.

## 6.1 Instanzmethode versus Klassenmethode

Das Programm „Random Sentences - Version 1“ implementiert wiederum ein Spiel, bei welchem der Computer die durch den Benutzer eingegebenen Satzbausteine zufällig zu ganzen Sätzen kombiniert.

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

public class Rndm {
    static Random rndm = new Random();

    static public int nextInt(int n) {return Math.abs(rndm.nextInt())%n;}
}

public class UserFrame extends Frame implements ActionListener{
    private Label sentence;
    private TextField subjectField, verbField, objectField;
    private List subjectList, verbList, objectList;

    private void place(Component comp,int x,int y,int width,int height) {
        comp.setBounds(x, y, width, height);
        add(comp);
    }

    public UserFrame() {
        setTitle("Sentences");
    }
}
```

```
        setLayout(null);
        setSize(560,420);
        Button button;
        place(sentence=new Label(),30,30,460,20);
        place(button=new Button("SHOW"),490,30,40,20);
        button.addActionListener(this);
        place(subjectField=new TextField(),30,60,160,20);
        subjectField.addActionListener(this);
        place(verbField=new TextField(),200,60,160,20);
        verbField.addActionListener(this);
        place(objectField=new TextField(),370,60,160,20);
        objectField.addActionListener(this);
        place(subjectList=new List(),30,90,160,300);
        place(verbList=new List(),200,90,160,300);
        place(objectList=new List(),370,90,160,300);
        setVisible(true);
        sentence.setText("Enter phrases and press RETURN!");
        subjectField.requestFocus();
    }

    private void update(List list, TextField textField) {
        if (!textField.getText().equals("")) {
            list.addItem(textField.getText());
            textField.setText("");
        }
    }

    public void actionPerformed(ActionEvent event){
        if (event.getSource() instanceof TextField) { //RETURN
            update(subjectList, subjectField);
            update(verbList, verbField);
            update(objectList, objectField);
            subjectField.requestFocus();
        }
        else if (event.getActionCommand().equals("SHOW")) {
            sentence.setText(
                subjectList.getItem(Rndm.nextInt(subjectList.getItemCount()))+" "+
                verbList.getItem(Rndm.nextInt(verbList.getItemCount()))+" "+
                objectList.getItem(Rndm.nextInt(objectList.getItemCount())));
        }
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 6.1.1 Zum Programm

Die Application bittet den Benutzer, Sätze einzugeben, wobei Subjekt, Verb und Objekt getrennt voneinander in die separaten TextFields geschrieben werden müssen. Durch Drücken der RETURN-Taste werden die Elemente aus den Textfeldern in die

darunterstehenden Lists übertragen. Beim Aktivieren des Buttons SHOW generiert der Computer zufallsmässig aus den eingegebenen Satzfragmenten Sätze (siehe Abbildung 6-1).



**Abbildung 6-1: Random Sentences - Version 1, UserFrame**

### 6.1.1.1 List

In diesem Programm wird die GUI-Komponente **List** verwendet (siehe Abbildung 6-1). Die im package `java.awt` deklarierte Klasse `List` erlaubt das Präsentieren mehrerer Auswahlmöglichkeiten. Je nach Implementierung kann man nur ein oder mehrere Elemente aufs Mal selektieren. Wenn für die Darstellung der Elemente einer List zu wenig Platz vorhanden ist, wird eine Leiste zum Blättern (engl. **scrollbar**) hinzugefügt. Im

Gegensatz zur Checkbox, bei welcher die Angabe eines Zustandes (ein/aus) im Vordergrund steht, erlaubt eine List die Selektion von Elementen (engl. **item**).

In der Klasse `List` sind auch die folgenden, im Programm verwendeten Methoden deklariert:

```
public void addItem(String item)
public String getItem(int index)
public int getItemCount()
```

Die Methode `addItem()` fügt den als Parameter übergebenen String am Ende der List ein. `getItem()` liefert den String, welcher an der angegebenen Position in der List gefunden wird, wobei anzufügen ist, dass die **Indizierung** bei Null beginnt. Die Methode `getItemCount()` ermittelt schlussendlich die Anzahl Elemente einer List.

#### *6.1.1.2 Der instanceof-Operator*

Erneut ist die Klasse `UserFrame` für die Schaffung der graphischen Benutzeroberfläche zuständig. Sie deklariert Instanzvariablen der Klassen `Label`, `TextField` und `List`. Neben ihrem Konstruktor, welcher die Plazierung der Komponenten und die Registrierung des `ActionListener` vornimmt, der Methoden `place()` und `actionPerformed()` enthält sie die Methode

```
private void update(List list, TextField textField) {
    if (!textField.getText().equals("")) {
        list.addItem(textField.getText());
        textField.setText("");
    }
}
```

welche für die Übertragung der Eingaben aus den `TextFields` in die Lists zuständig ist.

Jedes Mal, wenn der Benutzer nach der Eingabe in ein `TextField` die RETURN-Taste oder den SHOW-Button betätigt, generiert die entsprechende Komponente einen `ActionEvent`. Dieser bewirkt den Aufruf der im Listener `UserFrame` deklarierten Methode

```
public void actionPerformed(ActionEvent event){
    if (event.getSource() instanceof TextField) {           //RETURN
        update(subjectList, subjectField);
        update(verbList, verbField);
        update(objectList, objectField);
        subjectField.requestFocus();
    }
    else if (event.getActionCommand().equals("SHOW")) {
        sentence.setText(
            subjectList.getItem(Rndm.nextInt(subjectList.getItemCount()))+" "+
            verbList.getItem(Rndm.nextInt(verbList.getItemCount()))+" "+
            objectList.getItem(Rndm.nextInt(objectList.getItemCount())));
    }
}
```

In ihr wird als erstes überprüft, ob die RETURN-Taste oder der SHOW-Button aktiviert wurde. Trifft ersteres zu, werden die TextFields geleert und deren Inhalte in die Lists übertragen. Falls aber SHOW betätigt wurde, gibt das Label sentence einen zufallsmässig generierten Satz aus.

Der **instanceof-Operator** untersucht, ob der linke Operand eine Instanz des rechten Operanden ist und evaluiert zu einem Booleschen Ergebnis.

Die Methode `getSource()` ist in der Klasse `EventObject` (siehe Abbildung 5-8) enthalten. Sie liefert die eventgenerierende Instanz, also das entsprechende TextField, zurück.

***Siehe auch:*** 6.1.2.1

### 6.1.1.3 Instanzmethoden und Klassenmethoden

Nebst der Klasse `UserFrame` enthält das Programm die Klasse

```
public class Rndm {
    static Random rndm = new Random();

    static public int nextInt(int n) {return Math.abs(rndm.nextInt())%n;}
}
```

Sie deklariert als Klassenvariable eine Instanz `rndm` der bereits in Abschnitt 3.1.1 erläuterten Klasse `Random`.

Die Methode `nextInt()` generiert eine positive, ganze Zufallszahl, die zwischen 0 und (n-1) liegt. In der Methodenschnittstelle wird die Methode als **static** deklariert. Das Schlüsselwort **static** besagt, dass es sich um eine **Klassenmethode** handelt. Wie auch bei den Variablen (siehe Abschnitt 2.4.1.2) unterscheidet man bei den Methoden **Instanz- und Klassenmethoden**. Hierbei ist bei Instanzmethoden der Botschaftsempfänger immer eine Instanz, bei Klassenmethoden jedoch eine Klasse. Instanzmethoden arbeiten auf den Daten eines konkreten Objektes, Klassenmethoden auf denjenigen einer Klasse.

Nicht immer ist die Entscheidung, ob man eine Methode als Instanz- oder Klassenmethode deklarieren soll, eindeutig. Als Faustregel kann man aber festhalten, dass Klassenmethoden ausschliesslich Klassenvariablen manipulieren.

Ob Instanz- oder Klassenmethode, diese Tatsache schlägt sich auch im Aufruf einer Methode nieder. Instanzmethoden richten sich immer an Instanzen. Diese gehen als Empfänger der Botschaft dem Methodenaufruf voraus. Im Gegensatz dazu wird bei Klassenmethoden lediglich die Klasse, in welcher die Methode deklariert ist, vorangestellt.

Der Aufruf der Methode `nextInt()` erfolgt in der Klasse `UserFrame`, in der Methode `actionPerformed()`:

```
sentence.setText(
    subjectList.getItem(Rndm.nextInt(subjectList.getItemCount()))+" "+
    verbList.getItem(Rndm.nextInt(verbList.getItemCount()))+" "+
    objectList.getItem(Rndm.nextInt(objectList.getItemCount())));
```

Vor dem Methodenaufruf steht die empfangende Klasse `Rndm`. Würde die Methode `nextInt()` als Instanzmethode deklariert, müsste man zuerst eine Instanz der Klasse `Rndm` generieren, welche man dann dem Methodenaufruf voranstellen könnte.

**Siehe auch:** 6.1.2.2, 6.1.2.3

#### 6.1.1.4 Satzgenerator

Fügt man dem Programm die folgende Klasse zu

```
public class SentenceGenerator {
    static public String compose(String subject,String verb,String object){
        switch (Rndm.nextInt(5)) {
            case 0: return subject+" "+verb+" "+object+ ".";
            case 1: return "Weshalb "+verb+" "+subject+" "+object+"?";
            case 2: return "Unglaublich! Da "+verb+" "+subject+" doch
                        tatsaechlich "+object+"!";
            case 3: return subject+" "+verb+" niemals "+object+"!";
            case 4: return subject+" "+verb+" schon wieder "+object+"!";
            default: return "";
        }
    }
}
```

kann man nebst der zufälligen Auswahl der Satzelemente auch noch die Satzart zufällig bestimmen. Hierdurch entsteht das Programm „Random Sentences - Version 2“.

In obigen switch statement bewirkt das Schlüsselwort `return` das sofortige Verlassen der ganzen Methode mit Wertrückgabe. Daher ist das Schlüsselwort `break` (siehe Abschnitt 3.1.2.19) überflüssig.

Die Methode `actionPerformed()` muss dann um den Aufruf von `compose()` entsprechend modifiziert werden:

```
public void actionPerformed(ActionEvent event){  
    .  
    .  
    else if (event.getActionCommand().equals("SHOW")) {  
        sentence.setText(SentenceGenerator.compose(  
            subjectList.getItem(Rndm.nextInt(subjectList.getItemCount())),  
            verbList.getItem(Rndm.nextInt(verbList.getItemCount())),  
            objectList.getItem(Rndm.nextInt(objectList.getItemCount()))));  
    }  
}
```

Die Klassenmethode `compose()` erlaubt die zufällige Erzeugung von Satztypen, indem mit dem Aufruf `Rndm.nextInt(5)` eine der fünf möglichen Satzkonstruktionen gewählt wird. Hierbei kann man einfache Sätze, Fragen, Negationen etc. bilden. Abbildung 6-2 illustriert ein Beispiel.



**Abbildung 6-2: Random Sentences - Version 2, UserFrame**



## 6.1.2 Syntax

### 6.1.2.1 Der instanceof-Operator

Der **instanceof-Operator** kann nur im Zusammenhang mit komplexen Datentypen verwendet werden. Er untersucht, ob der linksstehende Operand eine Instanz des rechtsaufgeführten Operandes ist und evaluiert zu einem **Booleschen Ausdruck**. Hierbei muss object eine Instanz eines komplexen Datentyps sein und ReferenceDataType einen komplexen Datentyp bezeichnen.

InstanceofOperator
<code>object instanceof ReferenceDataType</code>

Beispiel:     `(event.getSource() instanceof TextField)`

### 6.1.2.2 Instanzmethoden und Klassenmethoden

In Ergänzung zu den Ausführungen in Abschnitt 3.1.2.7 soll an dieser Stelle gezeigt werden, wie die Deklaration einer Instanz- bzw. Klassenmethode erfolgt:

Ist in der Methodenschnittstelle zusätzlich das Schlüsselwort **static** deklariert, handelt es sich um eine **Klassenmethode**.

ClassMethod
<pre><b>static</b> type Identifier(ParameterList) {     Statements }</pre>

Beispiel:     siehe Programmbeispiel „Random Sentences“,  
              Methode nextInt()

Bei **Instanzmethoden** fehlt hingegen das Schlüsselwort `static`.

### InstanceMethod

```
type Identifier(ParameterList) {  
    Statements  
}
```

Beispiel:    siehe Programmbeispiel „Random Sentences“,  
             Methode update()

### 6.1.2.3 Methodenaufruf

In Ergänzung zu Abschnitt 3.1.2.8 zeigt dieser Abschnitt den Aufruf einer Instanz- bzw. Klassenmethode:

Grundsätzlich kann eine Botschaft nur dann an eine Instanz bzw. Klasse gesandt werden, wenn die entsprechende Methode in dieser Klasse auch deklariert ist.

**Instanzmethoden** haben als Botschaftsempfänger eine Instanz derjenigen Klasse, in welcher die Instanzmethode deklariert ist.

### InstanceMethodCall

```
classIdentifier.instanceIdentifier.Identifier();
```

Beispiele:    button.addActionListener(this);  
             setVisible(true);

Der empfangenden Instanz kann man zusätzlich den Namen der Klasse voranstellen, in welcher sie deklariert wird. Da der Compiler aber implizit annimmt, dass das empfangende Objekt auch in derjenigen Klasse deklariert ist, in welcher der Methodenaufruf erfolgt, ist die Angabe der Klasse nur notwendig, wenn dem nicht so ist, wenn also die Instanz in einer anderen Klasse deklariert worden ist.

Falls keine empfangende Instanz angegeben wird, setzt der Compiler die Referenz **this**. Botschaftsempfänger ist somit also das zum Zeitpunkt des Methodenaufrufs referenzierte Objekt.

**Klassenmethoden** haben als Botschaftsempfänger jene Klasse, in welcher die Klassenmethode deklariert ist.

<b>ClassMethodCall</b>
<code>classIdentifier.Identifier();</code>

Beispiel: `Math.abs(rndm.nextInt())%n);`

Falls die empfangende Klasse dieselbe ist wie diejenige, in welcher der Methodenaufruf erfolgt, muss sie nicht explizit angegeben werden.

In seltenen Fällen ist es aber notwendig, zusätzlich das package in der Form `java.package.~` anzugeben. Dies ist nur dann erforderlich, wenn man die notwendigen Pakete nicht importiert hat oder wenn man innerhalb desselben package zwei Klassen gleich benennt<sup>1</sup>.

#### 6.1.2.4 Attributzugriff

Der Zugriff auf **Instanzvariablen** bzw. **Klassenvariablen** erfolgt in Analogie zum Aufruf einer Instanz- bzw. Klassenmethode (siehe Abschnitt 6.1.2.3).

## 6.2 Visibility Modifiers

Das folgende Programm „Random Sentences - Version 3“ erweitert nun die vorangehende Version um ein Menu. Änderungen sind **fett** markiert.

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

public class Rndm {
    static Random rndm = new Random();

    static public int nextInt(int n) {return Math.abs(rndm.nextInt())%n;}
}

public class SentenceGenerator {
    public String compose(String subject, String verb, String object) {
        return subject+" "+verb+" "+object+".";
    }
```

---

<sup>1</sup> Um Verwirrungen vorzubeugen sollte man aber darauf achten, dass sämtliche Klassen unterschiedliche Namen haben.

```
}

public class NegationGenerator extends SentenceGenerator {
    public String compose(String subject, String verb, String object) {
        return subject+" "+verb+" niemals "+object+"!";
    }
}

public class QuestionGenerator extends SentenceGenerator {
    public String compose(String subject, String verb, String object) {
        return "Weshalb "+verb+" "+subject+" "+object+"?";
    }
}

public class UserFrame extends Frame
    implements ActionListener, ItemListener{
    private Label sentence;
    private TextField subjectField, verbField, objectField;
    private List subjectList, verbList, objectList;
    private SentenceGenerator generator;

    private void place(Component comp,int x,int y,int width,int height) {
        comp.setBounds(x, y, width, height);
        add(comp);
    }

    public UserFrame() {
        setTitle("Sentences");
        setLayout(null);
        setSize(560,440);
        generator = new SentenceGenerator();    //default generator
        Button button;
        MenuBar menuBar;
        Menu preferences;
        MenuItem sentenceItem, negationItem, questionItem;

        menuBar = new MenuBar();
        menuBar.add(preferences = new Menu("Preferences"));
        preferences.add(sentenceItem=new MenuItem("Sentence"));
        sentenceItem.addActionListener(this);
        preferences.add(negationItem=new MenuItem("Negation"));
        negationItem.addActionListener(this);
        preferences.add(questionItem=new MenuItem("Question"));
        questionItem.addActionListener(this);
        setMenuBar(menuBar);

        place(sentence=new Label(),30,50,460,20);
        place(button=new Button("SHOW"),490,50,40,20);
        button.addActionListener(this);
        place(subjectField=new TextField(),30,80,160,20);
        subjectField.addActionListener(this);
        place(verbField=new TextField(),200,80,160,20);
        verbField.addActionListener(this);
        place(objectField=new TextField(),370,80,160,20);
        objectField.addActionListener(this);
        place(subjectList=new List(),30,110,160,300);
    }
}
```

```
        subjectList.addItemListener(this);
        place(verbList=new List(),200,110,160,300);
        verbList.addItemListener(this);
        place(objectList=new List(),370,110,160,300);
        objectList.addItemListener(this);
        setVisible(true);
        sentence.setText("Enter phrases and press RETURN!");
        subjectField.requestFocus();
    }

    private void update(List list, TextField textField) {
        if (!textField.getText().equals("")) {
            list.addItem(textField.getText());
            textField.setText("");
        }
    }

    public void actionPerformed(ActionEvent event){
        if (event.getSource() instanceof TextField) {
            update(subjectList, subjectField);
            update(verbList, verbField);
            update(objectList, objectField);
            subjectField.requestFocus();
        } else if (event.getActionCommand().equals("SHOW")) {
            subjectList.select(Rndm.nextInt(subjectList.getItemCount()));
            verbList.select(Rndm.nextInt(verbList.getItemCount()));
            objectList.select(Rndm.nextInt(objectList.getItemCount()));
            sentence.setText(generator.compose(
                subjectList.getSelectedItem(),
                verbList.getSelectedItem(),
                objectList.getSelectedItem()));
        } else if (event.getActionCommand().equals("Sentence")) {
            generator = new SentenceGenerator();
        } else if (event.getActionCommand().equals("Negation")) {
            generator = new NegationGenerator();
        } else if (event.getActionCommand().equals("Question")) {
            generator = new QuestionGenerator();
        }
    }

    public void itemStateChanged(ItemEvent event){
        sentence.setText(generator.compose(
            subjectList.getSelectedItem(),
            verbList.getSelectedItem(),
            objectList.getSelectedItem()));
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 6.2.1 Zum Programm

Die zwei grundlegendsten Neuerungen im diesem Programm sind die Auswahl von Elementen aus den Lists per Mausklick und die Selektion eines Satzgenerators aus einem Menu.

#### 6.2.1.1 List und ItemListener

In Ergänzung zu den in Abschnitt 6.1.1.1 aufgeführten Methoden werden in diesem Programm zusätzlich noch die in der Klasse **List** deklarierten Methoden

```
public void select(int index)
public synchronized String getSelectedItem()
```

verwendet. Erstere markiert das an der Position `index` befindliche Element. Letztere gibt ein selektiertes Element als String zurück.

Da eine `List` aufgrund der Selektion eines ihrer Elemente einen **ItemEvent** generiert, implementiert die Klasse `UserFrame` nebst einem `ActionListener` für das Abfangen der `ActionEvents` auch einen **ItemListener**. Hierzu muss sie die Methode

```
public abstract void itemStateChanged(ItemEvent e)
```

implementieren.

Die Registrierung des `ItemListener` bei den Lists geschieht im Konstruktor `UserFrame()` in den Zeilen

```
subjectList.addItemListener(this);
verbList.addItemListener(this);
objectList.addItemListener(this);
```

#### 6.2.1.2 Menu

Im Konstruktor `UserFrame()` werden die lokalen Variablen

```
MenuBar menuBar;
Menu preferences;
MenuItem sentenceItem, negationItem, questionItem;
```

deklariert.

Die Klasse **MenuBar** deklariert eine Menuleiste, welche einem Frame mittels

```
public void setMenuBar(MenuBar mb)
```

zugeordnet wird.

Eine Instanz der Klasse `MenuBar` beinhaltet ein oder mehrere **Menus**, welche ihrerseits **MenuItems** oder wiederum Menus (Untermenus) umfassen. Die Klassen `MenuBar`, `Menu` und `MenuItem` sind Unterklassen von **MenuComponent**, welche auf gleicher Hierarchiestufe wie `Component` steht.

#### Die Methode

```
public Menu add(Menu m)
```

fügt einem `MenuBar` ein `Menu` zu und die Methode

```
public MenuItem add(MenuItem mi)
```

einem `Menu` ein `MenuItem`.

#### In den Zeilen

```
menuBar = new MenuBar();
menuBar.add(preferences = new Menu("Preferences"));
preferences.add(sentenceItem=new MenuItem("Sentence"));
sentenceItem.addActionListener(this);
preferences.add(negationItem=new MenuItem("Negation"));
negationItem.addActionListener(this);
preferences.add(questionItem=new MenuItem("Question"));
questionItem.addActionListener(this);
setMenuBar(menuBar);
```

wird ein `Menu` mit der Beschriftung „Preferences“ kreiert, welches die `MenuItems` „Sentence“, „Negation“ und „Question“ aufweist. In der letzten Anweisung wird `menuBar` dem `UserFrame` zugeordnet.

Da `MenuItems` **ActionEvents** generieren, bekommen sie allesamt `UserFrame` als ihren `Listener` registriert.

Im Unterschied zu der in Abschnitt 6.1.1.4 vorgestellten Lösung für die Implementierung eines Satzgenerators wird in diesem Programm die Vererbung eingesetzt: die Klasse `SentenceGenerator` hat die zwei Unterklassen `NegationGenerator` und `QuestionGenerator`, welche beide die Methode `compose()` redefinieren.

#### Die Instanzvariable

```
private SentenceGenerator generator;
```

wird im Programm nun dazu verwendet, den aktuellen Satzgenerator zu referenzieren. Zu Programmbeginn verweist sie auf eine Instanz der Klasse `SentenceGenerator`, welcher auf diese Weise als Standard-Generator definiert wird. Im Programmverlauf kann `generator` aber auch aufgrund der Benutzerinteraktion Instanzen der Klassen `NegationGenerator` und `QuestionGenerator` referenzieren. Die entsprechende

Zuweisung erfolgt in der Methode `actionPerformed()`<sup>1</sup>, also immer dann, wenn der Benutzer im Menu „Preferences“ eine Auswahl trifft.

Vorteil dieser Implementation ist, dass man keine Fallunterscheidung bezüglich des Satzgenerators machen muss. Da `generator` immer auf den momentan aktuellen Satzgenerator verweist, genügt es, die Botschaft `compose()` an `generator` zu schicken: je nach referenziertem Objekt wird dann die Methode der entsprechenden Klasse aufgerufen.

Es ist aber darauf hinzuweisen, dass mit jeder Menuauswahl eine neue Instanz der Generator-Klassen generiert wird. Alte Instanzen werden also nicht mehr referenziert und deshalb von der Garbage Collection (siehe Abschnitt 4.2.1.2) „entsorgt“.

### 6.2.1.3 Sichtbarkeitsattribuierung und Information Hiding

In allen Programmen ist bereits häufig das Schlüsselwort **public** wie auch **private** verwendet worden. Bei diesen handelt es sich um sogenannte **visibility modifiers**. Sie können Klassen, Methoden sowie Instanz- und Klassenvariablen zugeordnet werden und bestimmen hierdurch deren Sichtbarkeit. Die Sichtbarkeit ist in Relation zu einem package bzw. zu einer Klasse definiert und regelt den gegenseitigen Zugriff von Programmelementen.

Das Attribut `public` bezeichnet die uneingeschränkte Sichtbarkeit. Egal von welcher Programmstelle aus, man kann auf ein `public` attribuiertes Element immer zugreifen. Im Gegensatz dazu beschränkt `private` die Sichtbarkeit auf eine Klasse. So sind beispielsweise sämtliche Attribute der Klasse `UserFrame` und auch deren Methode `place()` als `private` deklariert, was zur Folge hat, dass man nur innerhalb der Klasse `UserFrame` auf sie zugreifen bzw. sie aufrufen kann. Werden jedoch entsprechende Methoden zur Verfügung gestellt, ist ein indirekter Zugriff auf `private` deklarierte Attribut dennoch möglich (siehe Abschnitt 7.1.1.2).

Die Einschränkung der Sichtbarkeit entspricht dem Prinzip des **Information Hiding**<sup>2</sup>. Übertragen auf den momentanen Sachverhalt besagt dieses, dass die Klassen untereinander nur das Nötigste an Information preisgeben sollen, wodurch die gegenseitige Unabhängigkeit wächst. Eine solche Unabhängigkeit und Flexibilität ist im Sinne einer erhöhten Wartungsfreundlichkeit anzustreben.

***Siehe auch:*** 6.2.2.1

---

<sup>1</sup> Aufgrund der Vererbungshierarchie kann man `generator` Instanzen der Klassen `NegationGenerator` bzw. `QuestionGenerator` zuweisen (siehe Abschnitt 4.3.2.1).

<sup>2</sup> Der Begriff Information Hiding wurde erstmals 1971 durch David Parnas geprägt.



## 6.2.2 Syntax

### 6.2.2.1 Visibility Modifiers

Die folgende Tabelle illustriert die in Java verfügbaren **visibility modifiers**, welche die Sichtbarkeit von Klassen, Methoden oder Attributen definieren. Wird nicht explizit ein visibility modifier verwendet, wird die Sichtbarkeit standardmässig auf package gesetzt.

<i>Sichtbarkeit</i>	<i>public</i>	<i>protected</i>	<i>package</i>	<i>private</i>
Gleiche Klasse	ja	ja	ja	ja
Klasse im selben Paket	ja	ja	ja	nein
Unterklasse in gleichem oder anderem Paket	ja	ja	nein	nein
Nicht-Unterklasse/ Anderes Paket	ja	nein	nein	nein

Untenstehende Tabelle zeigt den Einfluss der Sichtbarkeitsattribution auf die Vererbung:

<i>Vererbt an</i>	<i>public</i>	<i>protected</i>	<i>package</i>	<i>private</i>
Unterklasse im gleichen Paket	ja	ja	ja	nein
Unterklasse in einem anderen Paket	ja	ja	nein	nein

Die Tabelle besagt beispielsweise, dass eine `protected` attribuierte Methode sowohl für eine Unterklasse im gleichen als auch für eine Unterklasse in einem anderen Paket sichtbar ist.

## 6.3 Zusammenfassung

Methoden lassen sich in Instanz- und Klassenmethoden unterteilen. **Klassenmethoden** operieren ausschliesslich auf Klassenvariablen und beziehen sich immer auf die Klasse selber. **Instanzmethoden** hingegen sind immer mit einem Objekt assoziiert, auf dessen Werte sie zugreifen.

Die Unterscheidung zwischen Instanz- und Klassenmethoden macht sich auch im **Methodenaufruf** sichtbar. Botschaftsempfänger einer Instanzmethode ist immer eine Instanz. Bei Klassenmethoden richtet sich jedoch der Aufruf an die Klasse, welche die entsprechende Methode deklariert.

Mittels **visibility modifiers** kann man die Sichtbarkeit von Klassen, Methoden und Attributen bestimmen. Ihr Einsatz erlaubt die Idee des Information Hiding umzusetzen.

Es wurden die GUI-Bausteine **List** und **Menu** eingeführt.

# 7

## Method Overloading

## Arrays

## Exception Handling

Das vorliegende Kapitel basiert auf einem Programmbeispiel, dessen Funktionalität in insgesamt vier Versionen erweitert wird. Es werden Klassen und Methoden zur Ausgabe der Uhrzeit vorgestellt sowie der Umgang mit der Klasse `Graphics` nochmals aufgegriffen. Anschliessend werden Arrays eingeführt. Zum Abschluss wird eingehend das Exception Handling besprochen.

### 7.1 Calendar

Das Programm „Time - Version 1“ implementiert dank fundamentaler Integerarithmetik eine einfache digitale Uhr.

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

public class Time {
    private int hours, minutes;           //0<=hours<24 0<=minutes<60

    public Time(int h, int m) {           //constructor with parameters
        setTime(h,m);
    }

    public Time() {setTime(0,0);}         //constructor without parameters

    public int getHours() {return hours;}

    public int getMinutes() {return minutes;}

    public int getMinutesAfterMidnight() {
        return hours*60+minutes;
    }

    public void setTime(int h, int m) {   //set instance variables
```

```
        hours = (h+m/60)%24;
        minutes = m%60;
        if (minutes<0) {
            minutes += 60;
            hours--;
        }
        if (hours<0) hours += 24;
    }

    public void increment(int m) {          //add m to minutes
        setTime(0,getMinutesAfterMidnight()+m);
    }

    //return time in standard representation:
    public String toString() {
        return String.valueOf(hours)+":"+
            (minutes<10?"0":"")+String.valueOf(minutes);
    }

    //return true if time equals parameter:
    public boolean equals(Time t) {
        return (hours==t.getHours())&&(minutes==t.getMinutes());
    }

    //return true if time is before parameter:
    public boolean before(Time t) {
        if (hours==t.getHours())
            return (minutes<t.getMinutes());
        else return (hours<t.getHours());
    }
}

public class UserFrame extends Frame implements ActionListener {
    private TextField text;
    private Time time;

    private void place(Component comp,int x,int y,int width,int height) {
        comp.setBounds(x, y, width, height);
        add(comp);
    }

    public UserFrame() {
        Button button;
        setTitle("Clock");
        setLayout(null);
        setSize(200,200);
        setResizable(false);
        place(new Label("Time:"),60,50,80,20);
        place(text=new TextField(),60,70,80,20);
        place(button=new Button("SET"),60,120,30,20);
        button.addActionListener(this);
        place(button=new Button("+"),95,120,20,20);
        button.addActionListener(this);
        place(button=new Button("-"),120,120,20,20);
        button.addActionListener(this);
        setVisible(true);
        time = new Time();
        text.setText(time.toString());
    }
}
```

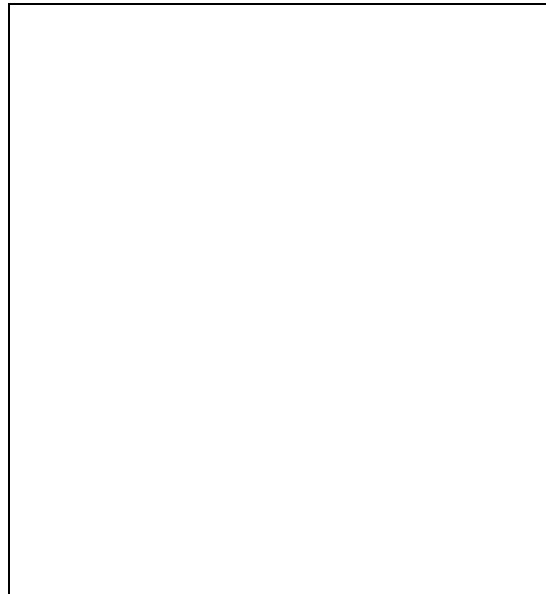
```
        text.requestFocus();
    }

    public void actionPerformed(ActionEvent event){
        //set time to current time:
        if (event.getActionCommand().equals("SET")) {
            Calendar date = Calendar.getInstance();
            time.setTime(
                date.get(Calendar.HOUR_OF_DAY),date.get(Calendar.MINUTE));
            //increment time by one minute:
        } else if (event.getActionCommand().equals("+")) {
            time.increment(1);
            //decrement time by one minute:
        } else if (event.getActionCommand().equals("-")) {
            time.increment(-1);
        }
        text.setText(time.toString());
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 7.1.1 Zum Programm

Wie Abbildung 7-1 zeigt, realisiert die Application „Time - Version 1“ eine einfache, digitale Uhr. Dem Benutzer stehen drei Buttons zur Verfügung: SET initialisiert die Uhr mit der aktuellen Systemzeit, ‘+’ und ‘-’ erlauben eine Addition bzw. Subtraktion in Minutenschritten.



**Abbildung 7-1: Time - Version 1, UserFrame**

Im wesentlichen besteht das Programm aus den Klassen `UserFrame` und `Time`, wobei `UserFrame` die graphische Benutzeroberfläche generiert. `Time` hingegen bietet Methoden zur Zeitmanipulation an.

### *7.1.1.1 Zuweisung mit Operation*

Die Klasse `Time` deklariert die Instanzvariablen `hours` und `minutes`, welche sie `private` deklariert. In der Methode

```
public void setTime(int h, int m) {  
    hours = (h+m/60)%24;  
    minutes = m%60;  
    if (minutes<0) {  
        minutes += 60;  
        hours--;  
    }  
    if (hours<0) hours += 24;  
}
```

werden diese initialisiert.

Aufgrund der Zeilen

```
hours = (h+m/60)%24;  
minutes = m%60;
```

ist gewährleistet, dass sich die Stunden nur zwischen 0 und 23 und die Minuten zwischen 0 und 59 bewegen. Allfälligen Überträge werden ebenfalls durchgeführt.

In den restlichen Zeilen der Methode ist definiert, wie mit negativen Parameterwerten zu verfahren ist: es wird das Komplement zu 60 bzw. 24 berechnet. Hierbei bezeichnet der **'+='-Operator (compound assignment operator)** eine Zuweisung, welche zugleich mit einer arithmetischen Operation verknüpft ist. Der Ausdruck `minutes += 60` ist die verkürzte Schreibweise für:

```
minutes = minutes + 60;
```

***Siehe auch:*** 7.1.2.1

#### *7.1.1.2 Information Hiding*

Da die Klasse `Time` die Instanzvariablen `hours` und `minutes` als `private` deklariert, kann auf diese ausserhalb der Klasse `Time` nicht zugegriffen werden (siehe Abschnitt 6.2.1.3 und 6.2.2.1). Im Gegenzug bietet sie aber Methoden, welche auf den Werten dieser Instanzvariablen operieren.

So deklariert die Klasse `UserFrame` die Methoden `getHours()` und `getMinutes()`, welche die Stunden und Minuten der aktuellen Uhrzeit zurückliefern, sowie die Methode `getMinutesAfterMidnight()`, welche berechnet, wieviele Minuten seit Mitternacht vergangen sind.

Die Methode `increment()` erlaubt die minutenweise Addition bzw. Subtraktion bei negativem Parameter. Beachtenswert hierbei ist, dass die momentane Zeit zuerst in Minuten umgerechnet werden muss, bevor die Addition bzw. Subtraktion erfolgt. Das erhaltene Resultat wird dann der Methode `setTime()` übergeben, welche den Ausdruck wieder zu Stunden und Minuten konvertiert.

Die Methoden `equals()` und `before()` ermöglichen ein als Parameter übergebenes Zeitobjekt mit der momentanen Zeit zu vergleichen. Diese Methoden werden aber zusammen mit `getHours()` und `getMinutes()` nie im Programm aufgerufen. Da sie jedoch eine weitere Zeitmanipulation realisieren, sind sie ebenfalls in der Klasse `Time` deklariert. Hierdurch wird `Time` zu einer universellen Klasse, welche Methoden zur Generierung und Manipulation von Zeitobjekten anbietet, und kann in einem anderen Zusammenhang wiederverwendet werden.

Dadurch, dass die Klasse `Time` den Zugriff auf ihre Instanzvariablen von ausserhalb der Klasse unmöglich macht, dafür aber Methoden anbietet, welche eine genau definierte Manipulation erlauben, wird ein kontrollierter, externer Zugriff realisiert.

#### *7.1.1.3 Conditional Operator*

Die Methode

```
public String toString() {  
    return String.valueOf(hours)+":"+"  
        (minutes<10?"0":"")+String.valueOf(minutes);  
}
```

verwendet den **'?:'-Operator**. Dieser **conditional operator** evaluiert aufgrund eines Booleschen Ausdrucks zu einem Wert. Vor dem Fragezeichen steht die Boolesche Bedingung. Ist diese wahr, liefert der Operator den Wert vor dem Doppelpunkt<sup>1</sup>, ansonsten denjenigen nach dem Doppelpunkt zurück. Ist im Beispiel `minutes` kleiner als 10, ergibt der conditional operator den Wert "0" andernfalls den leeren String.

Die Methode `toString()` realisiert für die Ausgabe in einer graphischen Benutzeroberfläche eine String-Repräsentation der Zeit.

***Siehe auch:*** 7.1.2.1

### 7.1.1.4 Method Overloading

Die Klasse `Time` verwendet zwei Konstruktoren. Beide haben zwar denselben Identifier, unterscheiden sich aber dennoch in ihrer Signatur, da sie eine unterschiedliche Anzahl Parameter deklarieren.

Dieses Phänomen bezeichnet man mit **method overloading**<sup>2</sup>. Darunter versteht man die Möglichkeit, denselben Identifier mehrmals innerhalb einer Klasse für die Bezeichnung von Methoden zu verwenden, sofern sich diese immer in ihrer Signatur unterscheiden. Die Signatur einer Methode wird durch ihren Namen, der Anzahl Parameter sowie deren Typ und deren Reihenfolge bestimmt.

### 7.1.1.5 Calendar

In der Methode `actionPerformed()` des `UserFrames` werden Elemente der Klasse `Calendar` verwendet. Die im package `util` deklarierte Klasse **Calendar** erlaubt die Ermittlung des momentanen Zeitpunktes und dessen Aufbrechung in verschiedene Facetten. So deklariert `Calendar` beispielsweise Attribute für Minuten, Stunden, Wochentage, Monate etc. und auch Methoden, welche auf die Attribute zugreifen, um beispielsweise den heutigen Wochentag zu ermitteln.

---

<sup>1</sup> Der Doppelpunkt ist wie das Schlüsselwort `else` eines `if statement` zu werten.

<sup>2</sup> Der Begriff `method overloading` ist vom Begriff `method overriding` (siehe Abschnitt 2.3.1.4) zu unterscheiden.



Die Klassenmethode `getInstance()` liefert ein `Calendar`-Objekt zurück. Die Attributwerte dieses `Calendar`-Objekts kann man dank der Instanzmethode `get()` ermitteln, indem man als Parameter den Namen des zu untersuchenden Attributs angibt.

In der Methode `actionPerformed()` wird in den Zeilen

```
if (event.getActionCommand().equals("SET")) {
    Calendar date = Calendar.getInstance();
    time.setTime(
        date.get(Calendar.HOUR_OF_DAY), date.get(Calendar.MINUTE));
}
```

zuerst eine Instanz `date` der Klasse `Calendar` generiert. Danach erfolgt der Methodenaufruf `setTime()` mit den zwei Parametern

```
date.get(Calendar.HOUR_OF_DAY)
date.get(Calendar.MINUTE)
```

Bei diesen handelt es sich erneut um Methodenaufrufe. Die Instanzmethoden `get()` der Klasse `Calendar` erlaubt den Zugriff auf die Attribute eines `Calendar`-Objekts, indem es den entsprechenden Attributwert zurückgibt. Die Konstante `HOUR_OF_DAY` enthält die Anzahl Stunden, `MINUTE` die Anzahl Minuten der momentanen Systemzeit.

## 7.1.2 Syntax

### 7.1.2.1 Zuweisung mit Operation

Eine **Zuweisung** kann auch sogleich mit einer **Operation** kombiniert werden.

CompoundAssignmentOperator
Identifier op= Expression

Beispiel:    `minutes += 60;`

Hierbei ist obiger Ausdruck äquivalent zum Ausdruck:

```
Identifier = (type) ((Identifier) op (Expression))
```

wobei `type` den Datentyp von `Identifier` bezeichnet. Beim Operator `op` kann es sich unter anderem um einen der folgenden Operatoren handeln:

`*, /, %, +, -, &, ^, |`

### 7.1.2.2 Conditional Operator

Ein **conditional operator** evaluiert aufgrund des vor dem Fragezeichen stehenden Booleschen Ausdrucks zu einem der nachfolgenden Werte. Ergibt die Bedingung `true`, wird der links vom Doppelpunkt stehende Wert ausgewählt, ansonsten derjenige rechts vom Doppelpunkt. Hierbei ist der Doppelpunkt wie das Schlüsselwort `else` eines `if` statement zu werten.

ConditionalOperator
<code>(Expression? true_value : false_value)</code>

Beispiel: `(minutes<10?"0":"")+String.valueOf(minutes);`

## 7.2 Graphics

Die vorhergehende Programmversion wird nun um die graphische, analoge Darstellung der Uhr erweitert. Neuerungen sind **fett** gedruckt.

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

public class Time {
    private int hours, minutes;           //0<=hours<24 0<=minutes<60

    public Time(int h, int m) {           //constructor with parameters
        setTime(h,m);
    }

    public Time() {setTime(0,0);}         //constructor without parameters

    public int getHours() {return hours;}

    public int getMinutes() {return minutes;}

    public int getMinutesAfterMidnight() {
        return hours*60+minutes;
    }
}
```

---

```

public void setTime(int h, int m) {    //set instance variables
    hours = (h+m/60)%24;
    minutes = m%60;
    if (minutes<0) {
        minutes += 60;
        hours--;
    }
    if (hours<0) hours += 24;
}

public void increment(int m) {        //add m to minutes
    setTime(0, getMinutesAfterMidnight()+m);
}

//return time in standard representation:
public String toString() {
    return String.valueOf(hours)+":"+
        (minutes<10?"0":"")+String.valueOf(minutes);
}

//return true if time equals parameter:
public boolean equals(Time t) {
    return (hours==t.getHours()) && (minutes==t.getMinutes());
}

//return true if time is before parameter:
public boolean before(Time t) {
    if (hours==t.getHours())
        return (minutes<t.getMinutes());
    else return (hours<t.getHours());
}

//draw clock at position x y with radius r
public void draw(Graphics g, int x, int y, int r) {
    // draw circle with diameter = 2*r:
    g.drawOval(x,y,2*r,2*r);
    // draw minute hand with length = 0.9*r:
    g.drawLine(x+r,y+r,
        x+r+(int)Math.round(0.9*r*Math.sin(2*Math.PI*minutes/60)),
        y+r-(int)Math.round(0.9*r*Math.cos(2*Math.PI*minutes/60)));
    // draw hour hand with length = 0.7*r:
    g.drawLine(x+r,y+r,
        x+r+(int)Math.round
            (0.7*r*Math.sin(2*Math.PI*getMinutesAfterMidnight()/720)),
        y+r-(int)Math.round
            (0.7*r*Math.cos(2*Math.PI*getMinutesAfterMidnight()/720)));
}
}

public class UserFrame extends Frame implements ActionListener {
    private TextField text;
    private Time time;

    private void place(Component comp,int x,int y,int width,int height) {
        comp.setBounds(x, y, width, height);
        add(comp);
    }
}

```

```
}

public UserFrame() {
    Button button;
    setTitle("Clock");
    setLayout(null);
    setSize(200,260);
    setResizable(false);
    place(new Label("Time:"), 60,150,80,20);
    place(text=new TextField(), 60,170,80,20);
    place(button=new Button("SET"), 60,200,30,20);
    button.addActionListener(this);
    place(button=new Button("+"), 95,200,20,20);
    button.addActionListener(this);
    place(button=new Button("-"), 120,200,20,20);
    button.addActionListener(this);
    time = new Time();
    text.setText(time.toString());
    text.requestFocus();
    setVisible(true);
}

public void paint(Graphics g) {
    time.draw(g,60,40,40);
}

public void actionPerformed(ActionEvent event){
    //set time to current time:
    if (event.getActionCommand().equals("SET")) {
        Calendar date = Calendar.getInstance();
        time.setTime
            (date.get(Calendar.HOUR_OF_DAY),date.get(Calendar.MINUTE));
    //increment time by one minute:
    } else if (event.getActionCommand().equals("+")) {
        time.increment(1);
    //decrement time by one minute:
    } else if (event.getActionCommand().equals("-")) {
        time.increment(-1);
    }
    text.setText(time.toString());
    repaint();
}

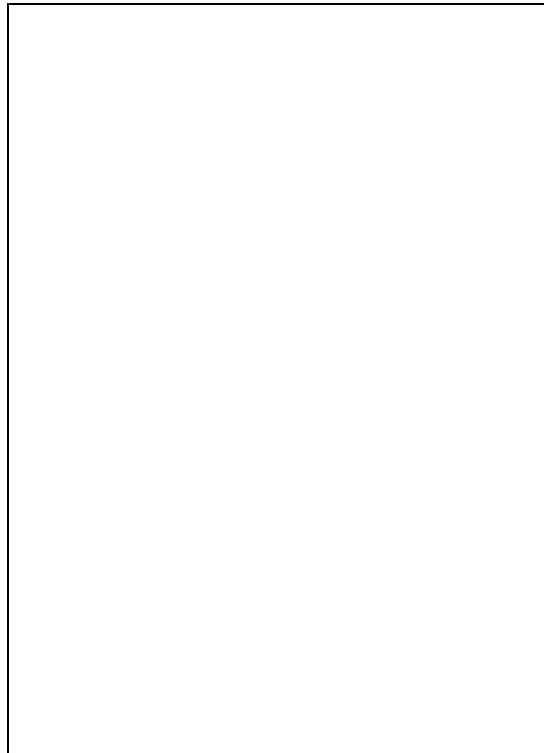
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 7.2.1 Zum Programm

Abbildung 7-2 illustriert die gleichzeitig digitale als auch analoge Darstellung der Uhr. Hierzu wird die in der Klasse Time neu deklarierte Methode `draw()` verwendet, welche

in der Methode `paint()` (siehe Abschnitt 2.3.1.3 und 2.3.1.4) des `UserFrame` aufgerufen wird. Damit gewährleistet ist, dass nach jeder Änderung durch den Benutzer die Uhr erneut gezeichnet wird, muss am Ende von `actionPerformed()` die Methode `repaint()` aufgerufen werden.



**Abbildung 7-2: Time - Version 2, UserFrame**

#### *7.2.1.1 drawOval() und drawLine()*

Wenden wir uns nun der Methode `draw()` zu:

```
//draw clock at position x y with radius r
public void draw(Graphics g, int x, int y, int r) {
    // draw circle with diameter = 2*r:
    g.drawOval(x,y,2*r,2*r);
    // draw minute hand with length = 0.9*r:
    g.drawLine(x+r,y+r,
        x+r+(int)Math.round(0.9*r*Math.sin(2*Math.PI*minutes/60)),
        y+r-(int)Math.round(0.9*r*Math.cos(2*Math.PI*minutes/60)));
    // draw hour hand with length = 0.7*r:
    g.drawLine(x+r,y+r,
        x+r+(int)Math.round
            (0.7*r*Math.sin(2*Math.PI*getMinutesAfterMidnight()/720)),
        y+r-(int)Math.round
            (0.7*r*Math.cos(2*Math.PI*getMinutesAfterMidnight()/720)));
}
```

In der ersten Anweisung erfolgt ein Aufruf der Methode **drawOval()**. Diese in der Klasse Graphics deklarierte Methode erlaubt das Zeichnen von Ellipsen. Die ersten beiden Parameter geben die Koordinaten der linken oberen Ecke desjenigen Rechtecks an, welches die Ellipse genau umgibt. Die letzten beiden Parameter beschreiben schliesslich die Breite und Höhe der Ellipse. Da im Aufruf `drawOval()` sowohl für die Breite als auch für die Höhe die gleichen Werte übergeben werden, entsteht ein Kreis. Die linke obere Ecke dieses Kreises ist an Position  $(x, y)$ , sein Radius beträgt  $r$  pixels (siehe Abbildung 7-3). Dieser Kreis symbolisiert nun unsere Uhr.

Für das Darstellen der Zeiger der Uhr wird die Methode **drawLine()** verwendet. Sie zeichnet eine Linie. Ihre ersten beiden Parameter bezeichnen die Koordinaten des Startpunktes, die letzten zwei Parameter die Koordinaten des Zielpunktes. Die Koordinaten des Startpunktes sind sowohl für den Minutenzeiger als auch für den Stundenzeiger identisch und fallen mit dem Mittelpunkt des Kreises  $(x+r, y+r)$  zusammen.

Abbildung 7-3 illustriert, wie man die Koordinaten eines Zeigerendpunktes berechnen kann. Da wir den Mittelpunkt des Kreises kennen, müssen wir lediglich noch die Strecken  $dx$  und  $dy$  ermitteln, um die Koordinaten des Zeigerendpunktes zu erhalten. Für den dargestellten Fall hat der Zielpunkt dann die x-Koordinate  $x+r+dx$  und die y-Koordinate  $y+r-dy$ .

Wenn wir nun annehmen, dass wir den Winkel  $a$  kennen, können wir mit Hilfe der Winkelfunktionen Sinus und Cosinus die Strecken  $dx$  und  $dy$  berechnen:

$$\begin{aligned}dx &= \sin(a) * l \\ dy &= \cos(a) * l\end{aligned}$$

Hierbei ist  $l$  die Länge des Zeigers.

Da aufgrund seiner Periodizität der Sinus von einem Winkel zwischen 0 und 180 Grad bzw. zwischen einem Vielfachen von 360 und einem Vielfachen von 360 plus 180 Grad immer einen positiven Wert ergibt, erhält man für sämtliche  $dx$ , deren Winkel sich in der rechten Kreishälfte bewegen, einen positiven Wert. Da die x-Werte von links nach rechts

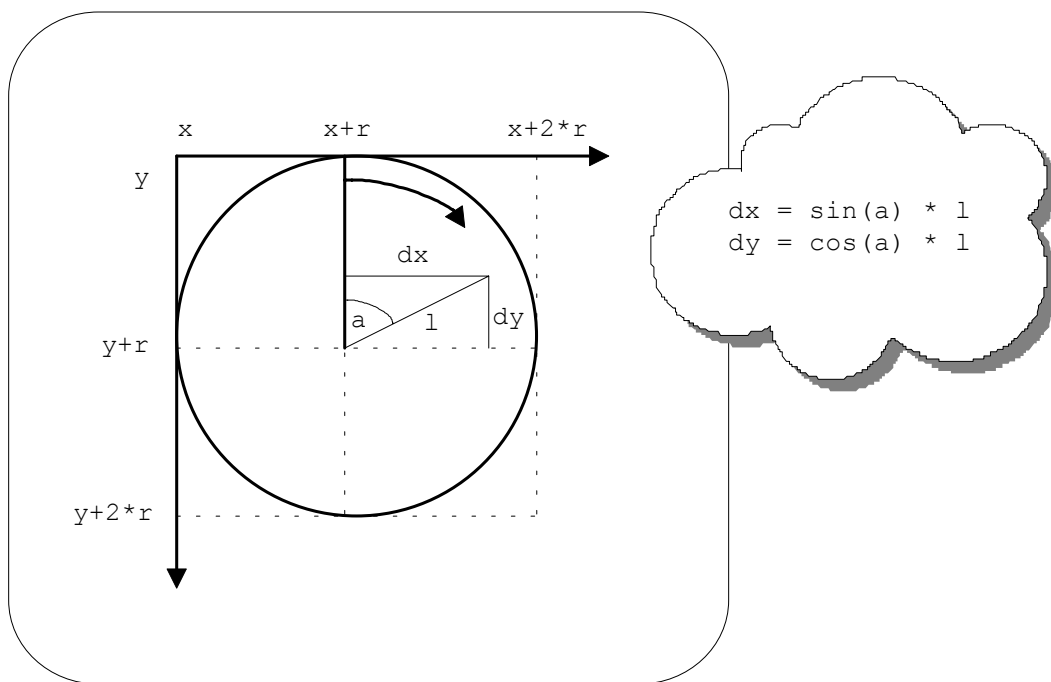
zunehmen, muss man also die positiven  $dx$ -Werte der rechten Kreishälfte zur x-Koordinate des Kreismittelpunktes addieren:

$$x + r + dx$$

Die negativen  $dx$ -Werte der linken Kreishälfte werden hingegen implizit subtrahiert.

Der Cosinus eines Winkels ist jedoch negativ, falls sich dieser zwischen einem Vielfachen von 90 und einem Vielfachen von 90 plus 180 Grad bewegt, womit man für  $dy$  in der unteren Kreishälfte negative Werte erhält. Da aber die y-Werte im Unterschied zu einem konventionellen Koordinatensystem von oben nach unten zunehmen, muss man also in der unteren Kreishälfte einen Zuwachs der y-Koordinate erreichen, weshalb man  $dy$  von der y-Koordinate des Kreismittelpunktes subtrahiert:

$$y + r - dy$$



**Abbildung 7-3: Berechnung des Endpunktes eines Zeigers**

Nun bleibt uns lediglich noch zu klären, wie man den Winkel  $a$  ermittelt. Da der Winkel eines Zeigers proportional mit der verflossenen Zeit zunimmt, muss er in Zusammenhang mit den vergangenen Minuten bzw. Stunden gebracht werden.

Normalerweise erfolgt eine Winkelangabe in Grad. Daneben gibt es auch das sogenannte Bogenmass<sup>1</sup>. Da die in Java definierten Winkelfunktionen Sinus und Cosinus mit dem Bogenmass rechnen, werden wir fortan dieses gebrauchen.

Da der Minutenzeiger in 60 Minuten den ganzen Kreis durchläuft, legt er in 60 Minuten einen Winkel von  $2\pi$  zurück. In einer Minute beträgt somit sein Winkel  $2\pi/60$ , womit wir die generelle Formel

$$\frac{2*\pi*minutes}{60}$$

erhalten. Hierbei gibt *minutes* die Anzahl verstrichener Minuten an.

Im Gegensatz zum Minutenzeiger durchläuft der Stundenzeiger den ganzen Kreis erst in  $12*60 = 720$  Minuten. Somit erhalten wir für die Berechnung des Winkels des Stundenzeigers in Abhängigkeit von der verstrichenen Zeit die Formel:

$$\frac{2*\pi*getMinutesAfterMidnight}{720}$$

Hierbei bezeichnet *getMinutesAfterMidnight* die Anzahl verstrichener Minuten seit Mitternacht.

Wenn wir nun alles zusammensetzen, erhalten wir für die Koordinaten des Minutenzeigerendpunktes:

$$\begin{aligned}x + r + \sin\left(\frac{2*\pi*minutes}{60}\right)*l \\ y + r - \cos\left(\frac{2*\pi*minutes}{60}\right)*l\end{aligned}$$

und für die Koordinaten des Stundenzeigers:

$$\begin{aligned}x + r + \sin\left(\frac{2*\pi*getMinutesAfterMidnight}{720}\right)*l \\ y + r - \cos\left(\frac{2*\pi*getMinutesAfterMidnight}{720}\right)*l\end{aligned}$$

Wenn wir nun die Länge des Minutenzeigers auf 90% des Radius der Uhr festsetzen, dann sieht der Programmcode zum Zeichnen des Minutenzeigers folgendermassen aus:

---

<sup>1</sup> Zur Erinnerung:  $180^\circ$  sind ein  $\pi$  und  $360^\circ$   $2\pi$ .



```
g.drawLine(x+r,y+r,
    x+r+(int)Math.round(0.9*r*Math.sin(2*Math.PI*minutes/60)),
    y+r-(int)Math.round(0.9*r*Math.cos(2*Math.PI*minutes/60)));
```

Die ersten beiden Parameter bezeichnen die Startkoordinaten, also den Mittelpunkt der Uhr, die letzten beiden die Koordinaten der Zeigerspitze, welche analog zu obiger Formel ermittelt werden. Die Klasse `Math` deklariert die Konstante `PI` und die Klassenmethoden `sin()`, `cos()` sowie `round()`, wobei letztere eine etwaige Gleitkommazahl zur nächstgelegenen ganzen Zahl rundet. Da `round()` eine Integer-Zahl vom Typ `long` zurückgibt, `drawLine()` aber Parameter vom Typ `int` erwartet, muss der ganze Ausdruck noch zu einem `int` konvertiert werden. Bei `minutes` handelt es sich natürlich um die in der Klasse `Time` deklarierte Instanzvariable.

Für den Stundenzeiger, welcher 70% des Kreisradius beträgt, ergibt sich der folgende Methodenaufruf:

```
g.drawLine(x+r,y+r,
    x+r+(int)Math.round
    (0.7*r*Math.sin(2*Math.PI*getMinutesAfterMidnight()/720)),
    y+r-(int)Math.round
    (0.7*r*Math.cos(2*Math.PI*getMinutesAfterMidnight()/720)));
```

Um die Anzahl der verstrichenen Minuten seit Mitternacht zu berechnen, wird die in der Klasse `Time` deklarierte Methode `getMinutesAfterMidnight()` aufgerufen.

## 7.3 Arrays

Vorliegende Programmversion „Time - Version 3“ ermöglicht zusätzlich die viertelstündliche Ausgabe der Zeit in Form eines Texts. Änderungen sind wiederum **fett** hervorgehoben.

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

public class Time {
    private int hours, minutes;           // 0<=hours<24 0<=minutes<60

    public Time(int h, int m) {           //constructor with parameters
        setTime(h,m);
    }

    public Time() {setTime(0,0);}         //constructor without parameters

    public int getHours() {return hours;}
```

```
public int getMinutes() {return minutes;}

public int getMinutesAfterMidnight() {
    return hours*60+minutes;
}

public void setTime(int h, int m) {    //set instance variables
    hours = (h+m/60)%24;
    minutes = m%60;
    if (minutes<0) {
        minutes += 60;
        hours--;
    }
    if (hours<0) hours += 24;
}

public void increment(int m) {          //add m to minutes
    setTime(0,getMinutesAfterMidnight()+m);
}

//return time in standard representation:
public String toString() {
    return String.valueOf(hours)+":"+
        (minutes<10?"0":"")+String.valueOf(minutes);
}

//return true if time equals parameter:
public boolean equals(Time t) {
    return (hours==t.getHours())&&(minutes==t.getMinutes());
}

//return true if time is before parameter:
public boolean before(Time t) {
    if (hours==t.getHours())
        return (minutes<t.getMinutes());
    else return (hours<t.getHours());
}

//draw clock at position x y with radius r
public void draw(Graphics g, int x, int y, int r) {
    // draw circle with diameter = 2*r:
    g.drawOval(x,y,2*r,2*r);
    // draw minute hand with length = 0.9*r:
    g.drawLine(x+r,y+r,
        x+r+(int)Math.round(0.9*r*Math.sin(2*Math.PI*minutes/60)),
        y+r-(int)Math.round(0.9*r*Math.cos(2*Math.PI*minutes/60)));
    // draw hour hand with length = 0.7*r:
    g.drawLine(x+r,y+r,
        x+r+(int)Math.round
            (0.7*r*Math.sin(2*Math.PI*getMinutesAfterMidnight()/720)),
        y+r-(int)Math.round
            (0.7*r*Math.cos(2*Math.PI*getMinutesAfterMidnight()/720)));
}

//return time in words if applicable:
public String toFancyString() {
    String word[]={"midnight","one","two","three","four","five","six",
        "seven","eight","nine","ten","eleven","noon",
```

```

        "one","two","three","four","five","six",
        "seven","eight","nine","ten","eleven"};
    if (minutes%15==0)
        switch (minutes/15) {
            case 0: return word[hours]+((hours%12!=0)? " o'clock":"");
            case 1: return "quarter past "+word[hours];
            case 2: return "half past "+word[hours];
            case 3: return "quarter to "+word[hours+1];
        }
    return "";
}
}

public class UserFrame extends Frame implements ActionListener {
    private Label message;
    private TextField text;
    private Time time;

    private void place(Component comp,int x,int y,int width,int height) {
        comp.setBounds(x, y, width, height);
        add(comp);
    }

    public UserFrame() {
        Button button;
        setTitle("Clock");
        setLayout(null);
        setSize(200,260);
        setResizable(false);
        place(message=new Label(),60,120,120,20);
        place(new Label("Time:"),60,150,80,20);
        place(text=new TextField(),60,170,80,20);
        place(button=new Button("SET"),60,200,30,20);
        button.addActionListener(this);
        place(button=new Button("+"),95,200,20,20);
        button.addActionListener(this);
        place(button=new Button("-"),120,200,20,20);
        button.addActionListener(this);
        time = new Time();
        text.setText(time.toString());
        text.requestFocus();
        setVisible(true);
    }

    public void paint(Graphics g) {
        time.draw(g,60,40,40);
    }

    public void actionPerformed(ActionEvent event){
        //set time to current time:
        if (event.getActionCommand().equals("SET")) {
            Calendar date = Calendar.getInstance();
            time.setTime
                (date.get(Calendar.HOUR_OF_DAY),date.get(Calendar.MINUTE));
            //increment time by one minute:
        } else if (event.getActionCommand().equals("+")) {
            time.increment(1);

```

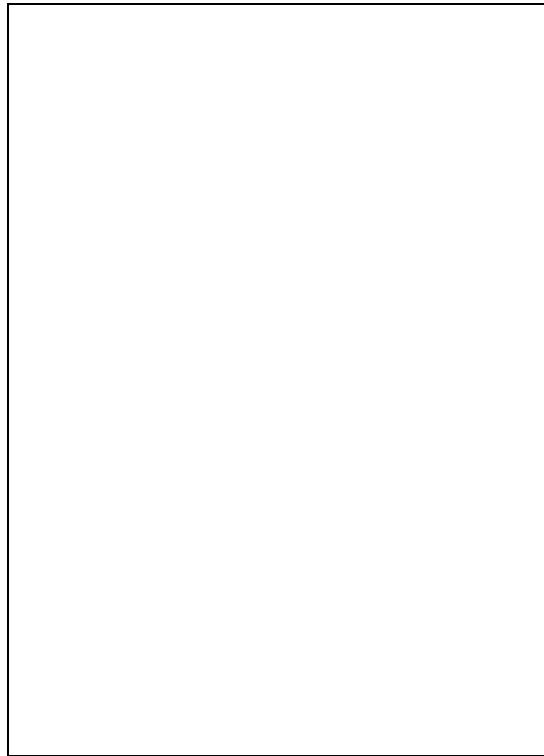
```
        //decrement time by one minute:
        } else if (event.getActionCommand().equals("-")) {
            time.increment(-1);
        }
        text.setText(time.toString());
        message.setText(time.toFancyString());
        repaint();
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 7.3.1 Zum Programm

Diese Programmversion erlaubt die Ausgabe der Zeit zu jeder Viertelstunde in Form eines Textes (siehe Abbildung 7-4). Hierzu wird die in der Klasse `Time` neu deklarierte Instanzmethode `toFancyString()` verwendet, welche den String im Label `message` anzeigt. Damit nach jeder Benutzerinteraktion eine allfällige textuelle Zeitangabe gemacht wird, erfolgt am Schluss der Methode `actionPerformed()` der Aufruf

```
message.setText(time.toFancyString());
```



**Abbildung 7-4: Time - Version 3, UserFrame**

#### *7.3.1.1 Arrays*

Die Instanzmethode `toFancyString()` verwendet einen bisher unbekannten, komplexen Datentyp, nämlich die im package `java.lang.reflect` deklarierte Klasse **Array**. Diese Datenstruktur erlaubt **Elemente gleichen Typs** so anzuordnen, dass jedes Element durch einen **Index** identifiziert wird. Aufgrund dieser Indizierung wird ein direkter Elementzugriff möglich.

In der Methode `toFancyString()` wird die folgende Instanz der Klasse `Array` deklariert und sogleich mit Elementen, also Werten initialisiert:

```
String word[]={ "midnight", "one", "two", "three", "four", "five", "six",  
               "seven", "eight", "nine", "ten", "eleven", "noon",  
               "one", "two", "three", "four", "five", "six",  
               "seven", "eight", "nine", "ten", "eleven" };
```

Die **eckigen Klammern** machen deutlich, dass es sich um einen Array handelt. Vor diesen steht der Name des Array, also `word`, und ganz zu Beginn der Zeile der Typ der Array-Elemente: der Array `word` beinhaltet hiermit Elemente vom Typ `String`.

Der Array wird gleichzeitig mit seiner Deklaration initialisiert, indem ihm seine Elemente in **geschweiften Klammern**, durch Kommata separiert, zugewiesen werden.

Die **Indizierung** der Elemente erfolgt in der Reihenfolge ihrer Erwähnung, ist ganzzahlig und beginnt bei Null: das erste Element "midnight" erhält den Index 0, "one" den Index 1, "two" den Index 2 etc.

Ist ein Array einmal kreiert, ist die **Anzahl** seiner Elemente **fix**. Da die Indizierung bei Null beginnt, ist die Länge eines Array um eins grösser als der Index seines letzten Elementes.

Um auf ein Element eines Array zuzugreifen, setzt man seinen Identifier und gibt in den eckigen Klammern den Index des jeweiligen Elementes an. Hierbei muss der angegebene Index zu einer Integer-Zahl evaluieren. In der Methode `toFancyString()` bezeichnet also `word[hours]` dasjenige Element, welches im Array `word` an der durch den Integer-Wert `hours` bezeichneten Position vorliegt.

```
public String toFancyString() {  
    String word[]={ "midnight", "one", "two", "three", "four", "five", "six",  
                   "seven", "eight", "nine", "ten", "eleven", "noon",  
                   "one", "two", "three", "four", "five", "six",  
                   "seven", "eight", "nine", "ten", "eleven" };  
    if (minutes%15==0)  
        switch (minutes/15) {  
            case 0: return word[hours]+((hours%12!=0)? " o'clock":"");  
            case 1: return "quarter past "+word[hours];  
            case 2: return "half past "+word[hours];  
            case 3: return "quarter to "+word[hours+1];  
        }  
    return "";  
}
```

Der Array `word` in der Methode `toFancyString()` ist so aufgebaut, dass der Wert eines Elementes mit dessen Index übereinstimmt: an der Position null, also um null Uhr, findet man den String "midnight" vor, an Position eins "one", an Position zwei "two" etc.

Da immer nur zur vollen Viertelstunde eine textuelle Zeitangabe erfolgen soll, wird in der Anweisung

```
if (minutes%15==0)
```

geprüft, ob `minutes` ein Vielfaches von fünfzehn ist. Trifft dies zu, gibt es vier zu unterscheidende Fälle: die Stunde ist voll, es ist Viertel nach, die Hälfte der Stunde ist angeschnitten oder es ist Viertel vor. Die entsprechenden vier Fälle werden mit einem `switch` statement ausgewertet. Hierbei ist das conditional statement im ersten Fall zu betrachten, welches sicherstellt, dass bei der Ausgabe von „midnight“ bzw. „noon“ kein „o’ clock“ folgt. Sollte die Bedingung des obigen `if` statement jedoch nicht erfüllt sein, wird der leere String zurückgegeben.

***Siehe auch:** 7.3.2.1*

## 7.3.2 Syntax

### 7.3.2.1 Arrays

Ein **Array**<sup>1</sup> (dt: **Feld**) realisiert eine Datenstruktur, bei welcher auf **Elemente gleichen Typs** durch einen ganzzahligen **Index** zugegriffen wird.

In Java werden Arrays durch Objekte realisiert. Die Indizierung beginnt bei Null, weshalb die Länge des Array somit um eins grösser ist als der Index des letzten Elementes. Ist ein Array einmal instanziiert, so ist die **Anzahl der Elemente fix**. Es ist aber selbstverständlich möglich, die Werte der einzelnen Elemente eines Array zu verändern.

Man kann einen Array auf folgende zwei Weisen deklarieren:

<b>ArrayDeclaration</b>
<pre>type Identifier[];</pre>

Beispiel:    `String word[];`

---

<sup>1</sup> Ein Array ist als Konzept zur Strukturierung von Daten auch in anderen Programmiersprachen zu finden.

ArrayDeclaration
<pre>type [] Identifier;</pre>

Beispiel: `String [] word;`

Die **eckigen Klammern** machen deutlich, dass es sich um einen Array handelt, `type` bezeichnet den Typ der Array-Elemente.

Es gibt zwei Möglichkeiten einen Array zu generieren<sup>1</sup>:

ArrayCreation
<pre>type Identifier[] = new type[n];</pre>

Beispiel: `String word[] = new String[24];`

Man verwendet das Schlüsselwort **new** und gibt in eckigen Klammern die Anzahl Elemente des Array an, oder man initialisiert ihn sogleich mit den Elementwerten:

ArrayCreation
<pre>type Identifier[] = {w<sub>0</sub>, w<sub>1</sub>, w<sub>2</sub>, ..., w<sub>(n-1)</sub>};</pre>

Beispiel: 

```
String word[] =
    {"midnight", "one", "two", "three", "four",
     "five", "six", "seven", "eight", "nine",
     "ten", "eleven", "noon", "one", "two",
     "three", "four", "five", "six", "seven",
     "eight", "nine", "ten", "eleven"};
```

Auf ein Element eines Array **greift man zu**, indem man den Index des Elementes in eckigen Klammern dem Namen des Array nachstellt. Hierbei muss der angegebene Indexwert eine positive Integerzahl oder gleich Null sein.

---

<sup>1</sup> Natürlich gelten die Angaben für beide Deklarationsarten. Sollte der Array bereits zu einem früheren Zeitpunkt deklariert worden sein, genügt es, den `Identifier` zu setzen.



<b>ArrayAccess</b>
Identifizier[index]

Beispiele:   word[3]  
              word[hours]  
              word[hours+1]

In Java gibt es auch noch die Möglichkeit, **mehrdimensionale Arrays** zu deklarieren. In der Deklaration entspricht dann die Anzahl der eckigen Klammerpaare der Dimension des Array (siehe Abschnitt 7.3).

## 7.4 Exception Handling

In diesem Abschnitt wird das vorhergehende Programm um eine weitere Methode ergänzt. Diese Änderung erlaubt dem Benutzer, die digitale Zeitangabe zu editieren.

In der Klasse Time wird neu die Methode

```
public void setTime (String s) throws NumberFormatException {
    String hString, mString;
    StringTokenizer tokenizer = new StringTokenizer(s, ":");
    try {
        hString = tokenizer.nextToken();
        mString = tokenizer.nextToken();
    }
    catch (NoSuchElementException e) {
        throw new NumberFormatException();
    }
    setTime(Integer.parseInt(hString), Integer.parseInt(mString));
}
```

deklariert, welche in der Klasse UserFrame in der leicht modifizierten Methode

```
public void actionPerformed(ActionEvent event){
    //set time to current time:
    if (event.getActionCommand().equals("SET")) {
        Calendar date = Calendar.getInstance();
        time.setTime
            (date.get(Calendar.HOUR_OF_DAY),date.get(Calendar.MINUTE));
    //increment time by one minute:
    } else if (event.getActionCommand().equals("+")) {
        time.increment(1);
    //decrement time by one minute:
    } else if (event.getActionCommand().equals("-")) {
        time.increment(-1);
    // check and set time according to changes in text:
    } else {
        try {time.setTime(text.getText());}
        catch (NumberFormatException e) {
            message.setText("Invalid Format");
            return;
        }
    }
    text.setText(time.toString());
    message.setText(time.toFancyString());
    repaint();
}
```

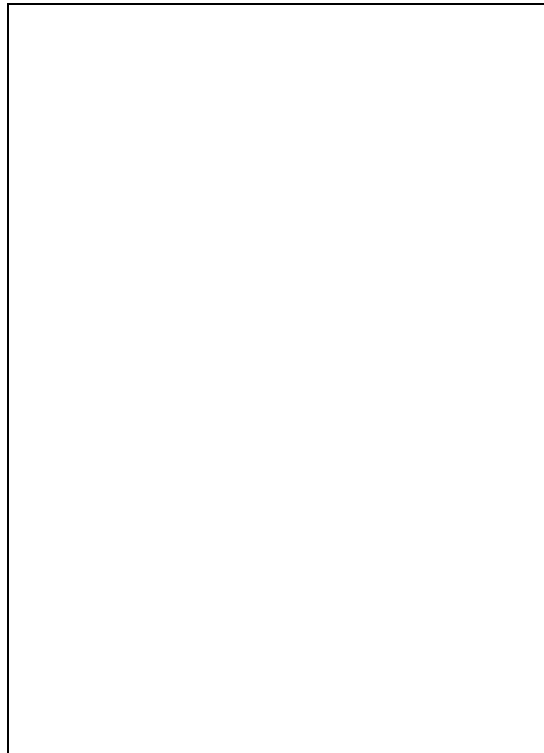
aufgerufen wird.

Zusätzlich erfolgt im Konstruktor `UserFrame()` noch die Registrierung des `ActionListener` beim `TextField text`:

```
text.addActionListener(this);
```

### 7.4.1 Zum Programm

In dieser Programmversion ist es dem Benutzer nun möglich, das `TextField text` zu editieren. Hierbei ist im Programm gewährleistet, dass nur ein korrektes Zeitformat angezeigt wird. Ansonsten wird im `Label message` angezeigt, dass die Eingabe nicht zulässig ist (siehe Abbildung 7-5).



**Abbildung 7-5: Time - Version 4, UserFrame**

#### 7.4.1.1 *String Tokenizer*

Ein **String Tokenizer** erlaubt es, einen String aufgrund von Separatoren (engl. delimiter) in Teilstrings zu zerlegen.

In der Methode `setTime()` bewirkt das Programmstück<sup>1</sup>

```
String hString, mString;  
StringTokenizer tokenizer = new StringTokenizer(s, ":");  
hString = tokenizer.nextToken();  
mString = tokenizer.nextToken();
```

die Zerlegung einer durch `<Stunden>:<Minuten>` dargestellte Uhrzeit in die beiden lokalen Variablen `hString` und `mString`. Dem Konstruktor `StringTokenizer()` wird der aufzusplittende String `s` als erster Parameter übergeben. Der zweite Parameter definiert den für die Zerlegung von `s` zu verwendenden Separator `":"`. Der Aufruf der Instanzmethode `nextToken()` bewirkt nun, dass der String von links nach rechts bis

---

<sup>1</sup> Das Schlüsselwort `try` sowie das geschweifte Klammerpaar wurde aus didaktischen Gründen weggelassen. Hierauf wird in Abschnitt 7.4.1.2 eingegangen.

zum ersten Separator durchlaufen und der so erhaltenen Teilstring zurückgegeben wird. Ein weiterer Aufruf von `nextToken()` retourniert denjenigen Teilstring, der zwischen der momentanen Position im String und dem nächsten Separator (oder dem String-Ende) liegt.

In den obigen Zeilen wird also der erste Teilstring von `s` dem String `hString` und der zweite Teilstring dem String `mString` zugewiesen. Ist die Benutzereingabe korrekt, also beispielsweise von der Form „16:15“, dann enthält `hString` nach der Zuweisung die Stunden und `mString` die Minuten der neu zu setzenden Uhrzeit.

### 7.4.1.2 Exception Handling

*So wie Johnny Gernimwind für seine Rotkäppchen-Inszenierung damit rechnen muss, dass irgend etwas „schiefgehen“ kann, sollte auch ein Programm für etwaige ausserordentliche Vorkommnisse gerüstet sein.*

*Was kann da nicht alles im Puppentheater passieren: ein Puppenspieler könnte seinen Text vergessen, der Faden einer Marionette könnte reißen oder es könnte plötzlich zu regnen beginnen. Für all diese Fälle ist unser Regisseur vorbereitet: er hat eine Souffleuse, eine grosse Rolle starken Nylonfadens und einen Klavierspieler, der die Pause überbrücken würde sowie für die Schlechtwettervariante einen Raum im Stadthaus zur Verfügung.*

In Java macht das sogenannte **Exception Handling** die Handhabung ausserordentlicher Vorkommnisse möglich.

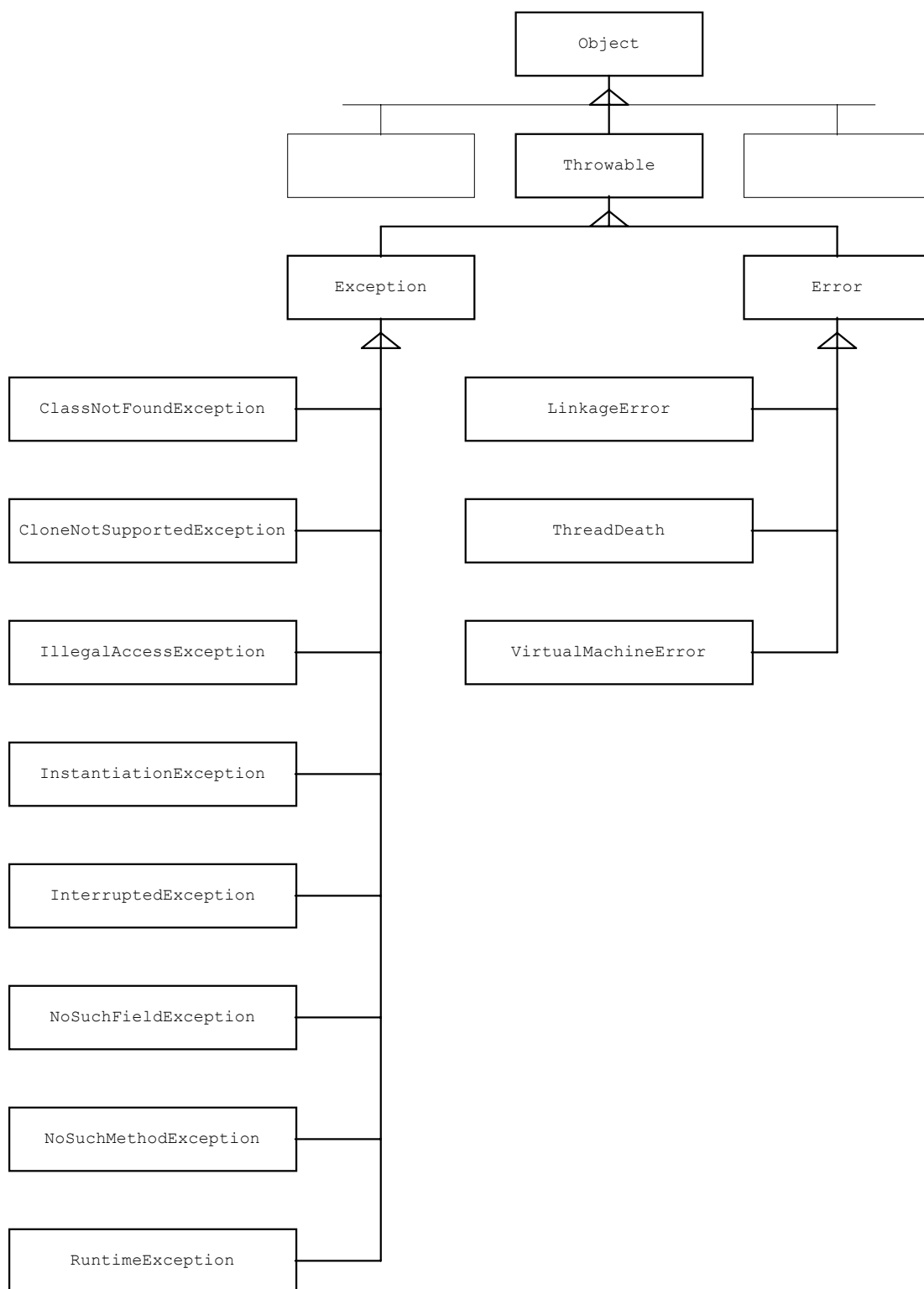
Das Auftreten eines Fehlers oder eines ausserordentlichen Vorkommnisses wird in einem Java-Programm - wie könnte es auch anders sein - durch Objekte, den sogenannten **Exceptions**, signalisiert. Eine Exception ist immer eine Instanz einer **Exception-Klasse**. Durch die in der Klassenbibliothek bereits deklarierten Exception-Klassen ist es möglich, ausserordentliche Zustände zu kategorisieren.

Wie Abbildung 7-6 zeigt, ist die Oberklasse aller Exception-Klassen die Klasse **Throwable**. Ihre direkten Unterklassen bilden die **Klasse Exception**<sup>1</sup> und die **Klasse Error**. Hierbei repräsentiert die Klasse Exception mitsamt Unterklassen solche ausserordentliche Zustände, welche durchaus handhabbar sind. Behebt man solche Exceptions, ist ein weiterer, reibungsloser Programmverlauf möglich. Im Gegensatz dazu bezeichnen Instanzen der Klasse Error bzw. ihrer Unterklassen Fehlervorkommnisse, die praktisch nicht mehr auffangbar sind und zumeist den Abbruch des Programms durch den Interpreter zur Folge haben. Aus diesem Grund kann man auch mit Hilfe des Exception

---

<sup>1</sup> Im Skript wird, wie auch allgemein üblich, der Begriff Exception für ein Objekt, welches einen ausserordentlichen Zustand oder Fehler signalisiert, verwendet. Die Klasse, von welcher es Instanz ist, wird mit Exception-Klasse bezeichnet. Spricht man hingegen von der Klasse Exception, ist die in Abbildung 7-6 ersichtliche Unterklasse von Throwable gemeint.

Handling Instanzen der Klasse `Error` bzw. ihrer Unterklassen praktisch nicht abfangen. Es ist noch zu erwähnen, dass Abbildung 7-6 nicht alle verfügbaren Exception-Klassen illustriert. So hat einerseits die Klasse `RuntimeException` noch etliche Unterklassen und andererseits sind auch die Klassen `LinkageError` und `VirtualMachineError` erweitert.



**Abbildung 7-6: Einbettung der Exception-Klassen in die Klassenbibliothek**

Wenn es im Programmverlauf zu einem ausserordentlichen Vorkommnis kommt, wird der normale Programmverlauf abgebrochen und eine Instanz der entsprechenden Exception-Klasse generiert. Man kann sich nun vorstellen, dass diese Exception gewissermassen ins Programm **geworfen** wird und darauf wartet, von jemandem aufgefangen zu werden. Wer ist aber der Fänger?

Im Puppentheater ist es beispielsweise der Requisiteur, welcher, sobald ein Faden reisst, das Malheur beseitigt. Im Programm ist das Schlüsselwort **catch** der Fänger. Dem Schlüsselwort `catch` folgt ein geschweiftes Klammerpaar, welches statements zur Handhabung der Situation deklariert. Diese statements werden jedoch nur beim Eintritt einer Exception ausgeführt, bei „normalem“ Programmverlauf aber übersprungen.

Während den Proben fürs Marionettenspiel wurde festgelegt, welche Person für welche Panne zuständig ist; es ist also der Requisiteur und nicht die Souffleuse, welcher den Nylonfaden ersetzt. Auch im Programm ist definiert, welches catch statement welche Exception behandelt: ein catch statement deklariert in runden Klammern den Exception-Typ, den es auffängt.

Wenn nun ein catch statement eine eintreffende Exception nicht aufnimmt, weil es Exceptions solchen Typs nicht behandelt, wirft es die Exception wie eine „heisse Kartoffel“ zum nächsten catch statement. Falls sich dieses der Exception wiederum nicht annimmt, geht das Spiel weiter, bis die Exception schlussendlich auf ein catch statement trifft, welches die Exception auffängt.

Betrachten wir die Reise einer Exception genauer: Wenn die Exception generiert wird, sucht sie zuerst im geschweiften Klammerpaar ihres Ursprungs nach einem catch statement. Ist dort keines oder kein passendes zu finden, verlässt sie dieses und geht zu demjenigen geschweiften Klammerpaar, welches das vorhergehende enthält. Auf diese Weise arbeitet sich die Exception durch die Blockstruktur einer Methode. Verlässt sie diese, siedelt sie zu demjenigen geschweiften Klammerpaar über, in welchem der Methodenaufruf stattgefunden hat. Dort arbeitet sie sich erneut durch die ganze Blockstruktur. Auf ihrer Suche nach einem catch statement durchquert eine Exception das ganze Programm, indem sie sukzessive das zuletzt ausgeführte geschweifte Klammerpaar nach einem geeignetem catch statement durchforstet und hierdurch die Anweisungsreihenfolge des Interpreters von hinten her aufrollt. Trifft eine Exception auf ihrem Weg zurück bis zur `main()` Methode nie auf ein erforderliches catch statement, gibt der Interpreter eine entsprechende Fehlermeldung aus.

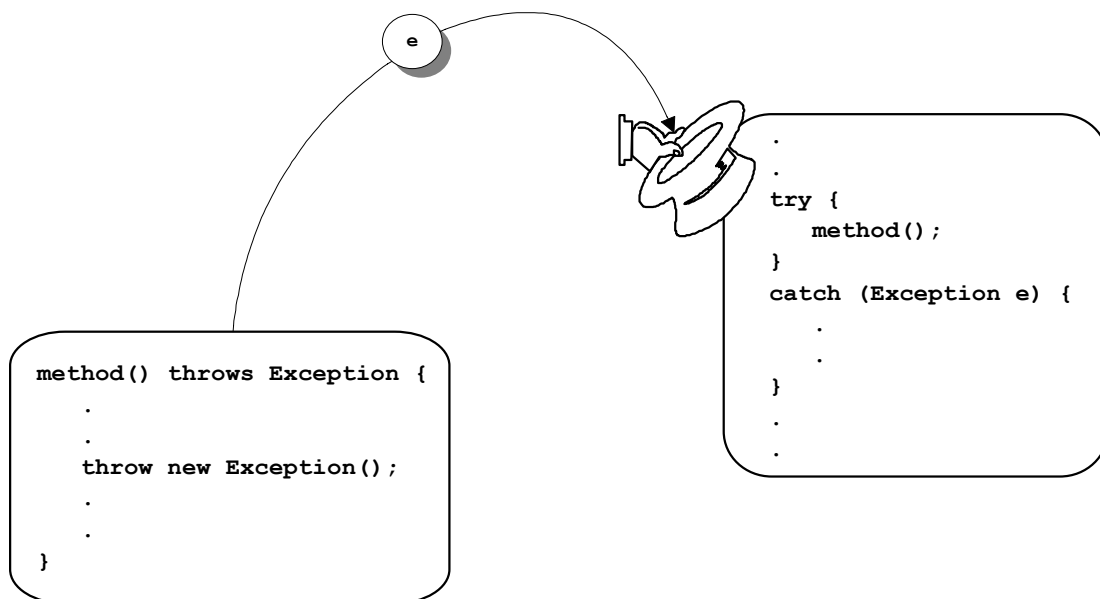
Wer wirft nun aber eine Exception? Eine Exception wird innerhalb einer Methode generiert und mit dem Schlüsselwort **throw** geworfen.

Falls nun eine solche Methode die generierte Exception nicht behandelt, also kein entsprechendes catch statement aufweist, wirft sie die Exception weiter. Dies muss sie in ihrer Signatur mit dem Schlüsselwort **throws** deklarieren.

In einem Programm können also Exceptions explizit mit dem Schlüsselwort `throw` generiert werden. Andererseits ist es aber auch möglich, im Programm kein einziges `throw` statement zu verwenden und dennoch mit Exceptions konfrontiert zu werden. Dies ist dann der Fall, wenn eine Methode aufgerufen wird, die in ihrer Signatur ein `throws` statement deklariert. In einer solchen Situation entsteht eine Exception implizit.

Will man eine Exception - sei sie nun explizit oder implizit generiert - sinngemäss behandeln, muss man einerseits ein `catch` statement zur Verfügung stellen und andererseits diejenigen Anweisungen, welche Exceptions provozieren, in geschweifte Klammern setzen und mit dem Schlüsselwort `try` bezeichnen. Ein **try** statement markiert jene Anweisungen, welche den Auswurf einer Exception hervorrufen können. Hierbei garantiert ein `try` statement, dass eine allfällige Exception in einem geeigneten `catch` statement gehandhabt wird. Für ein `try` statement gibt es also immer ein entsprechendes `catch` statement, welches nach diesem zu finden ist.

Abbildung 7-7 demonstriert, wie eine Exception explizit geworfen und am Ort des Methodenaufrufs aufgefangen wird.



**Abbildung 7-7: Illustration Exception Handling**

Wenden wir uns nun dem Exception Handling im Beispielprogramm zu:

Die Methode



```
public void setTime (String s) throws NumberFormatException {
    String hString, mString;
    StringTokenizer tokenizer = new StringTokenizer(s,":");
    try {
        hString = tokenizer.nextToken();
        mString = tokenizer.nextToken();
    }
    catch (NoSuchElementException e) {
        throw new NumberFormatException();
    }
    setTime(Integer.parseInt(hString), Integer.parseInt(mString));
}
```

setzt die Uhrzeit gemäss der Eingabe durch den Benutzer. Hierbei lässt sie aber nur eine korrekte Zeitangabe zu.

In der Methodenschnittstelle wird deklariert, dass die Methode `setTime()` eine Instanz der Klasse `NumberFormatException` werfen kann. `NumberFormatException` ist eine Unterklasse der Klasse `IllegalArgumentException`, die ihrerseits wiederum eine Unterklasse von `RuntimeException` ist. Sie signalisiert, dass der Versuch einen `String` zu einem `Integer`-Datentyp zu konvertieren missglückt ist, da der `String` nicht das erforderliche Format aufweist.

Ein Blick in den Methodenrumpf verrät, dass eine Instanz der Klasse `NumberFormatException` explizit geworfen wird:

```
throw new NumberFormatException();
```

Da sich diese Anweisung aber innerhalb eines `catch` statement befindet, muss ihre Durchführung vom Auftreten einer weiteren `Exception` abhängig sein. Diese `Exception` muss gemäss Deklaration vom Typ `NoSuchElementException` sein.

Es fragt sich nun, welche Anweisungen den Auswurf einer Instanz von `NoSuchElementException` provoziert haben. Diese müssen durch das Schlüsselwort `try` gekennzeichnet sein und entweder explizit oder implizit eine `NoSuchElementException` werfen.

Oberhalb des `catch` statement ist auch das erwartete `try` statement zu finden; es kapselt in sich die Anweisungen

```
try {
    hString = tokenizer.nextToken();
    mString = tokenizer.nextToken();
}
```

Da in diesen Anweisungen kein `throw` statement vorhanden ist, muss die `NoSuchElementException` implizit generiert werden, es muss also eine Methode aufgerufen werden, welche ihrerseits eine solche `Exception` generiert.

Die in der Klasse `StringTokenizer` (siehe Abschnitt 7.4.1.1) deklarierte Methode `nextToken()` wirft eine Instanz der Klasse `NoSuchElementException`, wenn

keine Tokens mehr im String vorhanden sind. Wenn also beispielsweise ein Benutzer „8.00“ anstatt „8:00“ eingibt, würde eine `NoSuchElementException` generiert, da der `StringTokenizer` gar keinen Separator vorfindet.

In der Methode `setTime()` wird wohl eine durch `nextElement()` hervorgerufene `NoSuchElementException` abgeblockt, dafür aber eine `NumberFormatException` generiert, die nicht in der Methode selber behandelt wird.

Am Ende der Methode `setTime()` erfolgt der Aufruf von `setTime()`<sup>1</sup>. Hierbei werden die Parameter

```
Integer.parseInt(hString), Integer.parseInt(mString)
```

übergeben. Die in der Klasse `Integer`<sup>2</sup> deklarierte Klassenmethode `parseInt()` konvertiert einen als Parameter erhaltenen String zu einem `int`. Dieser wird als Wert zurückgegeben. Sollten jedoch die im String enthaltenen Zeichen - mit Ausnahme des ersten Zeichens, welches auch ein Minuszeichen darstellen kann - keine Zahlen sein, wird eine `NumberFormatException` geworfen.

In der Methode `setTime()` kann also einmal explizit im `catch` statement und einmal implizit durch die Methode `parseInt()` eine `NumberFormatException` generiert werden. Da innerhalb der Methode aber kein passender `catch` statement vorhanden ist, wird die `NumberFormatException` weitergeworfen, was das Schlüsselwort `throws` in der Signatur bestätigt. Schauen wir nun in der aufrufenden Methode nach, ob diese ein `catch` statement für eine `NumberFormatException` deklariert.

In den geschweiften Klammern des letzten `else` statement<sup>3</sup> der Methode `actionPerformed()` in der Klasse `UserFrame` wird die Methode `setTime()` aufgerufen:

```
else {
    try {time.setTime(text.getText());}
    catch (NumberFormatException e) {
        message.setText("Invalid Format");
        return;
    }
}
```

An dieser Stelle im Programm sehen wir auch den Methodenaufruf von `setTime()` mit dem Schlüsselwort `try` versehen und finden das gesuchte `catch` statement. Das

---

<sup>1</sup> Wieder ein Beispiel für method overloading. Dieses Mal ist die Methode `setTime()` gemeint, welche Parameter vom Typ `int` erwartet.

<sup>2</sup> Siehe Abschnitt 4.3.3.

<sup>3</sup> Da alle anderen Fälle in den vorhergehenden `else` statements abgeklärt wurden, muss es sich beim letzten `else` statement um den Fall handeln, dass der Benutzer die RETURN-Taste im `TextField` gedrückt hat.

Schlüsselwort `return` bewirkt, dass die Programmkontrolle an den Ort des Methodenaufrufs zurückgeht und somit die Methode `actionPerformed()` verlassen wird. Wenn der Benutzer also in irgendeiner Weise ein ungültiges Zeitformat eingibt, wird die Nachricht „Invalid Format“ durch das Label `message` ausgegeben. Ansonsten nimmt das Programm seinen üblichen Verlauf, die eingegebene Zeit wird angezeigt.

*Siehe auch:* 7.4.2.1

## 7.4.2 Syntax

### 7.4.2.1 Exception Handling

Das Konzept des **Exception Handling** erlaubt, ausserordentliche Vorkommnisse innerhalb eines Programms zu erfassen und zu beheben, so dass ein weiterer, ungestörter Programmverlauf möglich ist. Hierbei werden sowohl die kritischen Anweisungen als auch diejenigen zur Behebung der Situation speziell mit den Schlüsselwörtern `try` bzw. `catch` markiert und heben sich dadurch als Bestandteil des Exception Handling vom übrigen Programm-Code ab.

Das Eintreten einer Exception bewirkt, dass der normale Programmfluss unterbrochen wird: die momentan auszuführenden Anweisungen werden unvollendet verlassen, damit nach einem geeigneten Fänger für die Exception gesucht werden kann. Die Suche nach einem passenden Fänger beginnt dort, wo sich die Exception ereignet hat, geht über zur aufrufenden Methode und erstreckt sich schliesslich über das ganze Programm. Sie entspricht der umgekehrten Ausführungsreihenfolge des Interpreters. Nach Handhabung einer Exception wird mit der Programminterpretation nach demjenigen `catch` statement weitergefahren, welches die Exception behandelt hat. Wird eine Exception jedoch nirgends im Programm abgefangen, gibt der Interpreter eine entsprechende Fehlermeldung auf der Konsole aus.

Eine Instanz einer Exception-Klasse kann entweder explizit durch das Schlüsselwort **throw** oder implizit durch Verwendung einer exception-generierenden Methode erzeugt werden.

Generiert eine Methode eine Exception, welche sie aber nicht handhabt, muss sie dies mittels dem Schlüsselwort **throws** in ihrer Signatur deklarieren. Hierzu bilden aber Exceptions der Klassen `RuntimeException` bzw. `Error` eine Ausnahme. Da diese häufig vorkommen, muss ihr Auftreten nicht durch das Schlüsselwort `throws` deklariert werden.

Will man eine Exception beheben, muss man jene Anweisungen, welche eine Exception hervorrufen können, in geschweifte Klammern setzen, mit dem Schlüsselwort `try` markieren und ein passendes **catch** statement deklarieren. Dieses ist nach dem `try`

statement zu finden und dient als Fänger eines klar definierten Exception-Typs. Die in ihm gekapselten Anweisungen helfen, die „Notsituation“ zu beheben und werden nur ausgeführt, wenn eine entsprechende Exception eintrifft.

ExceptionHandling
<pre>try {     .     .     throw new Exception();     oder     method(); } catch(Exception e) {     .     . }</pre>

Beispiel: 

```
try {time.setTime(text.getText());}
catch (NumberFormatException e) {
    message.setText("Invalid Format");
    return;
}
```

Es kann auch vorkommen, dass auf ein try statement mehrere catch statements folgen. Hierbei unterscheiden sich die catch statements im Typ der Exceptions, die sie behandeln. Dies ist dann der Fall, wenn ein try statement mehrere, unterschiedliche Exceptions generiert.

Unter Umständen gibt es Anweisungen, die, egal ob eine Exception vorliegt oder nicht, immer ausgeführt werden sollten. Solche Anweisungen kann man in einem sogenannten **finally** statement kapseln, welches den catch statements nachgestellt ist. Es ist garantiert, dass ein finally statement immer ausgeführt wird. Es wird unmittelbar, bevor die Ausführungskontrolle das try statement verlässt, aufgerufen<sup>1</sup>.

---

<sup>1</sup> Ausserhalb des Exception Handling kann ein try statement ohne eine catch statement auch nur zusammen mit einem finally statement verwendet werden. Ein try statement muss immer entweder mit einem catch oder finally statement oder mit beiden verwendet werden.

**ExceptionHandling**

```
try {}  
catch(Exception e) {}  
finally {}
```

Es ist auch möglich, seine eigenen Exception-Klassen zu bilden, indem man sie von der Klasse `Throwable` bzw. `Exception` erben lässt.

## 7.5 Zusammenfassung

Der erste Teil dieses Kapitels befasst sich vor allem mit der Klasse **Calendar** und dem Umgang mit **Graphics**-Objekten bzw. dem Zeichnen einer Uhr mittels trigonometrischer Funktionen. Hierbei wurde der **compound assignment operator** und der **conditional operator** eingeführt. Das Prinzip des **method overloading** wurde ebenfalls erwähnt.

Anschliessend wurde die Datenstruktur Array behandelt. Ein **Array** realisiert eine Datenstruktur, bei welcher auf **Elemente gleichen Typs** durch einen ganzzahligen **Index** zugegriffen wird. In Java werden Arrays durch Objekte realisiert. Die Indizierung beginnt bei Null, die **Anzahl der Elemente** eines Array ist **fix**.

Der Schluss des Kapitels widmet sich dem **Exception Handling**. Dieses erlaubt die Handhabung ausserordentlicher Vorkommnisse eines Programms. Hierbei werden die exceptiongenerierenden Anweisungen und die Anweisungen zu deren Handhabung speziell mit den Schlüsselwörtern `try` und `catch` markiert:

- ◆ Das Auftreten eines aussergewöhnlichen Vorkommnisses wird im Programm durch das „Werfen“ einer **Exception** symbolisiert. Der normale Programmverlauf wird unterbrochen, damit man den Zustand beheben kann. Eine Exception ist Instanz einer **Exception-Klasse**.
- ◆ Eine Exception kann entweder explizit mit dem Schlüsselwort **throw** oder implizit aufgrund eines Methodenaufrufs geworfen werden. Führt der Aufruf einer Methode zum Auswurf einer Exception, muss die Methode dies in ihrer Signatur durch das Schlüsselwort **throws** deklarieren.
- ◆ Eine Exception wandert so lange durch die Blockstruktur eines Programms, bis sie einen geeigneten Fänger findet. Sollte sie nirgends gefangen werden, gibt der Interpreter eine entsprechend Fehlermeldung aus.

- ◆ Zur Handhabung einer Exception muss man die exceptiongenerierenden Anweisungen mit dem Schlüsselwort **try** markieren und ein geeignetes **catch** statement zur Verfügung stellen. Die im catch statement gekapselten Anweisungen werden nur aufgrund einer Exception ausgeführt, bei normalem Programmverlauf aber übersprungen.

# 8

## Abstrakte Klassen Abstrakte Methoden

Dieses Kapitel befasst sich nun ausführlich mit abstrakten Methoden und abstrakten Klassen. Zur Illustration wird ein anschauliches Beispielprogramm verwendet.

### 8.1 Abstrakte Methoden und Klassen

Das Programm „Calculator“ erlaubt das Rechnen mit Zahlen verschiedener Zahlensysteme.

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

abstract public class NumberSystem{
    abstract int toInt(String s);
    abstract String toString(int value);
}

abstract public class ArabicSystem extends NumberSystem {
    public int radix;

    public int toInt(String s) {
        int n=0;
        for (int i=0;i<s.length();i++)
            n=n*radix+Character.digit(s.charAt(i),radix);
        return n;
    }

    public String toString(int value) {
        String s="";
        do {
            s = Character.toUpperCase(Character.forDigit(value%radix,radix))+s;
            value=value/radix;
        }while (value>0);
        return s;
    }
}
```

```
    }
}

public class HexNumber extends ArabicSystem {
    public HexNumber() {radix=16;}
}

public class BinNumber extends ArabicSystem {
    public BinNumber() {radix=2;}
}

public class OctNumber extends ArabicSystem {
    public OctNumber() {radix=8;}
}

public class DecNumber extends ArabicSystem {
    public DecNumber() {radix=10;}
}

public class RomanNumber extends NumberSystem {
    static private final char symbol[] = {'M','D','C','L','X','V','I'};
    static private final int val[] = {1000, 500, 100, 50, 10, 5, 1};

    public RomanNumber() {}

    public int toInt(String s) {
        int n=0;
        for (int i=0;i<s.length();i++)
            for (int j=0;j<7;j++)
                if (s.charAt(i)==symbol[j]) {
                    n=n+val[j];
                    break;
                }
        return n;
    }

    public String toString(int value) {
        String s="";
        for (int i=0; i<7; i++) {
            while (value>=val[i]) {
                s=s+symbol[i];
                value = value-val[i];
            }
        }
        return s;
    }
}

public class NumberField extends TextField{
    private NumberSystem numberSystem;

    public NumberField(NumberSystem n) {numberSystem=n;}

    public int getNumber() {return numberSystem.toInt(getText());}

    public void setNumber(int value) {
        setText(numberSystem.toString(value));
    }
}
```



```
public void changeBase(NumberSystem n) {
    int value=getNumber();
    numberSystem=n;
    setNumber(value);
}

}

public class UserFrame extends Frame
    implements ActionListener, ItemListener{
private NumberSystem base[] = new NumberSystem[5];
private NumberField op1, op2, result;
private Choice choice;

private void place(Component comp,int x,int y,int width,int height) {
    comp.setBounds(x, y, width, height);
    add(comp);
}

public UserFrame() {
    Button button;
    setTitle("Calculator");
    setLayout(null);
    setSize(220,360);
    setResizable(false);
    base[0] = new DecNumber();
    base[1] = new BinNumber();
    base[2] = new OctNumber();
    base[3] = new HexNumber();
    base[4] = new RomanNumber();
    place(choice=new Choice(),70,60,100,20);
    choice.addItemListener(this);
    choice.addItem("Decimal");
    choice.addItem("Binary");
    choice.addItem("Octal");
    choice.addItem("Hex");
    choice.addItem("Roman");
    choice.select(0);
    place(op1=new NumberField(base[0]),60,180,100,20);
    place(op2=new NumberField(base[0]),60,205,100,20);
    place(button=new Button("+"),60,230,20,20);
    button.addActionListener(this);
    place(button=new Button("-"),85,230,20,20);
    button.addActionListener(this);
    place(button=new Button("*"),110,230,20,20);
    button.addActionListener(this);
    place(button=new Button("/"),135,230,20,20);
    button.addActionListener(this);
    place(new Label("result:"),60,260,100,20);
    place(result=new NumberField(base[0]),60,280,100,20);
    op1.setBackground(Color.white);
    op2.setBackground(Color.white);
    result.setBackground(Color.white);
    result.setEditable(false);
    setVisible(true);
}

public void actionPerformed(ActionEvent event){
```

```
        if (event.getActionCommand().equals("+"))
            result.setNumber(op1.getNumber()+op2.getNumber());
        else if (event.getActionCommand().equals("-"))
            result.setNumber(op1.getNumber()-op2.getNumber());
        else if (event.getActionCommand().equals("*"))
            result.setNumber(op1.getNumber()*op2.getNumber());
        else if (event.getActionCommand().equals("/"))
            result.setNumber(op1.getNumber()/op2.getNumber());
    }

    public void itemStateChanged(ItemEvent event){
        NumberSystem b=base[choice.getSelectedIndex()];
        op1.changeBase(b);
        op2.changeBase(b);
        result.changeBase(b);
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 8.1.1 Zum Programm

Wie die untenstehenden screen shots illustrieren, ermöglicht das Programm „Calculator“ die Addition, Subtraktion, Multiplikation oder Division zweier Zahlen. Hierbei werden dezimale, binäre, oktale, hexadezimale und römische Zahlen unterstützt.

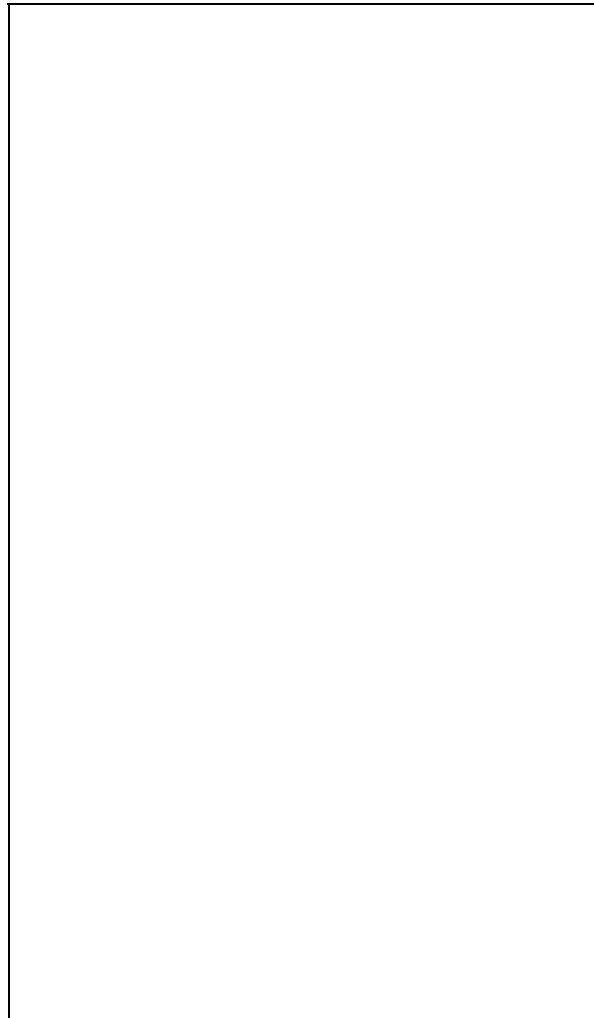
Abbildung 8-1 zeigt die Addition zweier Dezimalzahlen, Abbildung 8-2 die Addition zweier Binärzahlen und Abbildung 8-3 schlussendlich die Addition von römischen Zahlen.



**Abbildung 8-1: Calculator, UserFrame**



**Abbildung 8-2: Calculator, UserFrame**



**Abbildung 8-3: Calculator, UserFrame**

Im Programm werden die Klassen `NumberSystem`, `ArabicSystem`, `HexNumber`, `BinNumber`, `OctNumber`, `DecNumber` und `RomanNumber` für die Darstellung der verschiedenen Zahlensysteme verwendet. Die graphische Benutzeroberfläche sowie das Event Handling wird wiederum durch die Klasse `UserFrame` realisiert. Die Klasse `NumberField` schlussendlich bildet eine Art Vermittler zwischen den „Zahlensystem-Klassen“ und der graphischen Benutzeroberfläche.

#### *8.1.1.1 Choice*

Für die Auswahl des Zahlensystems wird eine **Choice** verwendet (siehe Abbildung 8-1 bis Abbildung 8-3). Die im package `awt` deklarierte Klasse `Choice` ermöglicht, wie auch eine `List` (siehe Abschnitt 6.1.1.1), die Auswahl von Elementen (engl. `item`). Im Unterschied zu dieser zeigt aber eine `Choice` normalerweise nur ein Element, nämlich das

ausgewählte, aufs Mal an. Für die Selektion eines Elementes kann man die Choice aber wie ein Menu „aufklappen“, wodurch eine List sichtbar wird, weshalb man auch von **pull-down lists** spricht. Eine Choice erlaubt die Selektion nur eines Elementes aufs Mal (gegenseitiger Ausschluss).

Im Programm werden neben ihrem Konstruktor die folgenden Methoden der Klasse Choice verwendet:

```
public synchronized void addItemListener(ItemListener l)
public synchronized void addItem(String item)
public synchronized void select(int pos)
public int getSelectedIndex()
```

Da ein selektiertes Item einen ItemEvent generiert, erfordert eine Choice die Registrierung eines ItemListener mittels addItemListener().

Die Methode addItem() erlaubt das Hinzufügen eines neuen Elementes, select() das Selektieren des sich an der bezeichneten Position befindlichen Items und getSelectedIndex() schlussendlich liefert den Index des momentan selektierten Elementes<sup>1</sup>.

### 8.1.1.2 Abstrakte Methoden

Wie Abbildung 8-4 zeigt, ist die Klasse NumberSystem die Oberklasse aller „Zahlensystem-Klassen“. Sie hat die direkten Unterklassen ArabicSystem und RomanNumber. Die Klasse ArabicSystem hat ihrerseits wiederum die Unterklassen HexNumber, BinNumber, OctNumber und DecNumber. Die gewählte Vererbungshierarchie hat ihren Grund in den unterschiedlichen Eigenschaften der arabischen (Binär, Oktal, Dezimal und Hexadezimal) und römischen Zahlen<sup>2</sup>.

---

<sup>1</sup> Der Index des ersten Elementes ist null.

<sup>2</sup> Der Wert einer römischen Zahl ergibt sich durch eine reine Addition ihrer Ziffern. Im Gegensatz dazu ist der Wert einer arabischen Zahl eine Summe von Produkten: die Ziffern werden noch jeweils mit einer Potenz ihrer Basis multipliziert, bevor sie addiert werden (Beispiel: die Binärzahl 0101 hat den Wert  $1*2^0+0*2^1+1*2^2+0*2^3 = 5$ ).

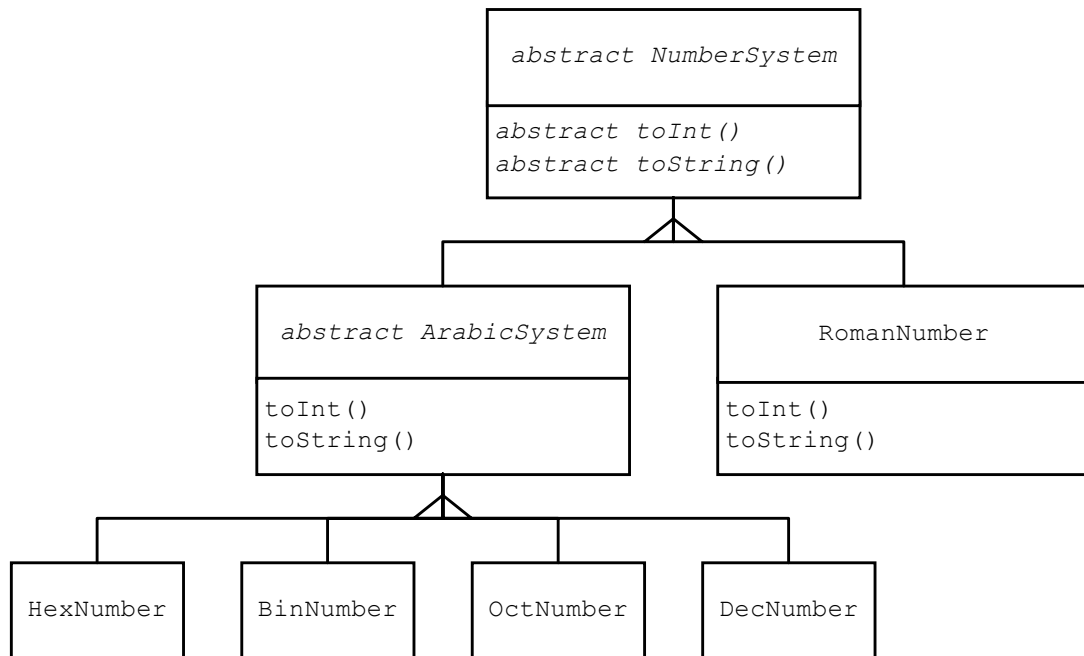


Abbildung 8-4: Vererbungshierarchie der "Zahlensystem-Klassen"

Da sämtliche „Zahlensystem-Klassen“ einerseits den als String repräsentierten Zahlenwert, welcher der Benutzer im GUI eingibt, zu einem Integer-Wert konvertieren und andererseits für die Ausgabe in der graphischen Benutzeroberfläche einen `int` Wert als String darstellen müssen, verfügt die Klasse `NumberSystem` über die abstrakten Methoden

```

abstract int toInt(String s);
abstract String toString(int value);

```

Eine **abstrakte Methode** verwendet das Schlüsselwort **abstract** und deklariert lediglich die Methodenschnittstelle ohne deren Rumpf. Dies macht es möglich, wohl die Signatur einer Methode zu bestimmen, jedoch ihre Implementation noch offen zu lassen.

Abstrakte Methoden werden an ihre Unterklassen weitervererbt. Im Programmbeispiel erbt die Klasse `ArabicSystem` sowie die Klasse `RomanNumber` von `NumberSystem`. Beide Unterklassen implementieren die abstrakten Methoden `toInt()` und `toString()`, wobei sich aber ihre Implementationen aufgrund der unterschiedlichen Eigenschaften der Klassen voneinander unterscheiden.

Hätte man bereits in der Oberklasse `NumberSystem` die beiden Methoden implementiert, müsste man sich für eine der beiden möglichen Implementationen entscheiden und dann entsprechend in einer der Unterklassen die Methoden redefinieren. In diesem Fall ist also die Verwendung von abstrakten Methoden ein eleganterer und effizienterer Ansatz.

### Die in der Klasse `ArabicSystem` deklarierte Methode

```
public int toInt(String s) {
    int n=0;
    for (int i=0;i<s.length();i++)
        n=n*radix+Character.digit(s.charAt(i),radix);
    return n;
}
```

wandelt den als `String` repräsentierten Zahlenwert in einen `int` Wert um, indem sie den `String` von links her bis zum Ende durchläuft und dabei jedes Zeichen, basierend auf dem jeweiligen Zahlensystem, als `int` interpretiert. Die erhaltenen Integer-Werte werden sukzessive addiert, wobei jedoch der Wert des vorhergehenden Durchlaufs mit der jeweiligen Basis multipliziert wird. Die in der Klasse `String` deklarierte Methode `charAt()` liefert das Zeichen an der angegebenen Position im `String`, die Klassenmethode `digit()` der Klasse `Character` gibt den numerischen Wert eines `char` aufgrund der als Parameter übergebenen Basis `radix` zurück.

Zur Umwandlung eines `int`-Wertes in einen `String` wird in der Klasse `ArabicSystem` die Methode

```
public String toString(int value) {
    String s="";
    do {
        s = Character.toUpperCase(Character.forDigit(value%radix,radix))+s;
        value=value/radix;
    }while (value>0);
    return s;
}
```

deklariert. Hierbei wird der Integer-Wert von rechts her durchlaufen, wobei jede Ziffer in ihre Stringrepräsentation umgewandelt wird. Der Rückgabestring der Methode wird in jedem Schleifendurchlauf erweitert, indem die neue Zahl links an den bisherigen `String` angehängt wird. Es werden dazu die beiden in der Klasse `Character` deklarierten Klassenmethoden `forDigit()` und `toUpperCase()` verwendet. Erstere bestimmt die Characterrepräsentation einer Zahl unter Berücksichtigung der angegebenen Basis, letztere wandelt einen allfälligen Kleinbuchstaben in den entsprechenden Grossbuchstaben um.

Die Klasse `RomanNumber` benötigt für die Implementation der beiden abstrakten Methoden `toInt()` und `toString()` die als Konstanten deklarierten Arrays

```
static private final char symbol[] = {'M','D','C','L','X','V','I'};
static private final int val[] = {1000, 500, 100, 50, 10, 5, 1};
```

Der Array `val` gibt den Wert des jeweils an gleicher Position befindlichen römischen Zeichens im Array `symbol` an.

Die Methode



```
public int toInt(String s) {
    int n=0;
    for (int i=0;i<s.length();i++)
        for (int j=0;j<7;j++)
            if (s.charAt(i)==symbol[j]) {
                n=n+val[j];
                break;
            }
    return n;
}
```

durchläuft den als Parameter übergebenen String, vergleicht das vorgefundene Zeichen mit den Werten des Array `symbol` und setzt bei Übereinstimmung den entsprechenden `int`-Wert des Array `val`. Die gewonnenen Integer-Werte werden sukzessive addiert. Das `break` statement bewirkt das Verlassen der inneren `for` Schleife.

Die Generierung der römischen Stringrepräsentation eines `int` passiert in der Methode

```
public String toString(int value) {
    String s="";
    for (int i=0; i<7; i++) {
        while (value>=val[i]) {
            s=s+symbol[i];
            value = value-val[i];
        }
    }
    return s;
}
```

Hier wird der Array `val` von links nach rechts durchlaufen und dessen Elemente mit dem zu konvertierenden `int`-Wert verglichen. Ist der zu konvertierende Wert grösser oder gleich dem Arrayelement, wird im auszugebenden String das entsprechende Element des Array `symbol` gesetzt und der `int`-Wert um den Wert des Arrayelementes `val` reduziert. Der auszugebende String wird mit jedem Durchlauf der `while` Schleife erweitert, indem der gewonne Wert rechts an den vorhergehenden angefügt wird.

***Siehe auch:** 8.1.2.1*

### 8.1.1.3 Abstrakte Klassen

Sobald eine Klasse eine oder mehrere abstrakte Methoden deklariert, ist sie selber abstrakt. **Abstrakte Klassen** können nicht instanziiert werden. Erst eine Unterklasse, welche für sämtliche abstrakten Methoden einen Rumpf aufweist, kann instanziiert werden.

Im Programmbeispiel ist die Klasse `NumberSystem` aufgrund ihrer abstrakten Methoden abstrakt. Sie wird mit dem Schlüsselwort **abstract** deklariert

```
abstract public class NumberSystem{
    abstract int toInt(String s);
    abstract String toString(int value);
}
```

Da die Methoden `toInt()` und `toString()` in der Klasse `NumberSystem` abstrakt sind, ist es plausibel, dass die Klasse `NumberSystem` nicht instanziiert werden kann.

Aus dem Programm ist aber auch zu entnehmen, dass die Klasse `ArabicSystem` ebenfalls eine abstrakte Klasse ist, obwohl sie keine abstrakte Methoden deklariert<sup>1</sup>. Es ist möglich, eine Klasse als **abstract** zu deklarieren, auch wenn diese keine abstrakten Methoden enthält. Hierdurch lässt sich eine Instanziierung verhindern.

Im Beispiel können lediglich die Unterklassen der Klasse `ArabicSystem` instanziiert werden. Da erst in den Unterklassen die bereits in der Oberklasse deklarierte Instanzvariable `radix` initialisiert wird, wäre eine Instanziierung von `ArabicSystem` wenig sinnvoll.

Die Klasse `NumberField`, eine Unterklasse von `TextField`, wird für die Textfelder der graphischen Benutzeroberfläche verwendet. Sie deklariert Methoden, um den `int`-Wert einer Zahl, welche der Benutzer eingegeben hat, zu ermitteln oder eine Zahl als `String` im GUI anzuzeigen. Andererseits ermöglicht sie auch die Repräsentation der Zahlen in den unterschiedlichen Zahlenformaten.

Nach eingehender Betrachtung merkt man aber, dass die besagten Methoden lediglich die Methoden `toInt()` und `toString()` aufrufen. Der Empfänger `numberSystem` der Botschaften `toInt()` und `toString()` wird als Instanz der Klasse `NumberSystem` deklariert und referenziert immer eine Instanz derjenigen „Zahlensystem-Klasse“, welche momentan in der `Choice` selektiert ist<sup>2</sup>.

Entscheidend dabei ist, dass `numberSystem` vom Typ `NumberSystem` deklariert wurde, da hierdurch eine konversionslose Zuweisung gewährleistet ist. Hätte man `numberSystem` als Instanz der Klasse `DecNumber` deklariert, so könnte man ihr nicht eine Instanz der Klasse `RomanNumber` zuweisen, da ein `type cast` erforderlich wäre.

Dies illustriert auch einen weiteren Vorteil für die Verwendung von abstrakten Methoden bzw. Klassen: Deren Einsatz zwingt den Programmierer, bereits auf höherer Hierarchiestufe die Signatur von weiter unten implementierten Methoden festzulegen,

---

<sup>1</sup> Die Unterscheidung zwischen abstrakten Klassen und nicht-abstrakten Klassen wird im Programm zusätzlich noch durch den Identifier unterstrichen: abstrakte Klassen werden als `~System` und sonstige Klassen als `~Number` bezeichnet.

<sup>2</sup> In der Klasse `UserFrame` wird ein Array `base` verwendet, welcher gewissermassen „auf Vorrat“ Instanzen der fünf „Zahlensystem-Klassen“ enthält. Eine solche Instanz wird der Klasse `NumberField` einerseits zu Programmbeginn durch Aufruf ihres Konstruktors und andererseits durch die Methode `changeBase()` übergeben.

wodurch deren Zugriff vereinheitlicht wird. Es ist beispielsweise möglich, in der Klasse `NumberField` lediglich den Aufruf `numberSystem.toInt()` zu setzen. Die virtuelle Maschine ruft dann diejenige Methode auf, welche in der Klasse der jeweiligen Instanz deklariert ist. Hätte man keine abstrakten Methoden deklariert und würde zusätzlich die Signatur der jeweiligen Methoden in den Klassen `ArabicSystem` und `RomanNumber` unterschiedlich ausfallen, müsste in der Klasse `NumberField` für jede Instanz der Methodenaufruf separat gemacht werden.

*Siehe auch: 8.1.2.2*

## 8.1.2 Syntax

### 8.1.2.1 Abstrakte Methoden

Eine **abstrakte Methode** deklariert lediglich die Signatur der Methode, nicht aber deren Rumpf. Die Deklaration verwendet das Schlüsselwort **abstract** und wird mit einem Semikolon abgeschlossen.

<b>AbstractMethod</b>
<b>abstract</b> type Identifier(ParameterList);

Beispiele: `abstract int toInt(String s);`  
`abstract String toString(int value);`

Abstrakte Methoden werden an Unterklassen weitervererbt. Eine Unterklasse, welche einen Methodenrumpf für eine abstrakte Methode deklariert, **implementiert** diese.

### 8.1.2.2 Abstrakte Klassen

**Abstrakte Klassen** werden mit dem Schlüsselwort **abstract** bezeichnet und **können nicht instanziiert werden**.

<b>AbstractClass</b>
<b>abstract class</b> Identifier {}

Beispiel:    `abstract public class NumberSystem{  
                  abstract int toInt(String s);  
                  abstract String toString(int value);  
          }`

Hierbei ist eine Klasse, welche eine oder mehrere abstrakten Methoden deklariert, automatisch selber abstrakt.

Eine Klasse kann aber auch ohne, dass sie abstrakte Methoden enthält, das Schlüsselwort `abstract` verwenden, was dann ihre Instanzierung verhindert (siehe Beispielpogramm „Calculator“, Klasse `ArabicSystem`).

## 8.2 Zusammenfassung

Der Schwerpunkt dieses Kapitels bilden die abstrakten Methoden und Klassen:

- ♦ Eine **abstrakte Methode** ist eine Methode, welche nur einen Methodenkopf deklariert, aber keinen Rumpf aufweist. Sie bietet die Möglichkeit, allein die Schnittstelle einer Methode festzulegen, aber deren Implementierung noch offen zu lassen.
- ♦ Eine **abstrakte Klasse** kann nicht instanziiert werden. Weist eine Klasse eine oder mehrere abstrakte Methoden auf, ist sie automatisch auch selber abstrakt. Die Deklaration von abstrakten Methoden ist jedoch nicht zwingend.

Als Baustein einer graphischen Benutzeroberfläche wurde die **Choice** vorgestellt.

# 9 Interfaces Adapter-Klassen

Das vorliegende Kapitel setzt sich nun ausführlich mit Interfaces auseinander. Aufgrund der Einführung der Adapter-Klassen wird das Event Handling nochmals kurz aufgegriffen. Beide Konzepte werden an einem Beispielprogramm illustriert.

## 9.1 Interfaces und Adapter-Klassen

Die Application „Word Guess“ realisiert ein Spiel für zwei Personen, bei welchem der Eine ein Wort erraten muss, das sich der Andere ausgedacht hat.

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

// specifies methods needed by WordManager:
public interface WordInterface {
    public String getWord();
    public void setWord(String s);
    public void setMessage(String s);
}

public class WordManager extends KeyAdapter {
    private WordInterface userFrame;
    private String word;           //secret word
    private String pattern;        //displays characters found so far
    private String characters;     //string of characters tried so far

    public WordManager(WordInterface userFrame) {
        this.userFrame = userFrame;
        word = userFrame.getWord();
        pattern = "";
        for (int i=0;i<word.length();i++)
            pattern = pattern+'\u0080';
        characters = "";
    }
}
```

```
}

//return true if x==y regardless of case:
private boolean isEqual(char x, char y) {
    return Character.toLowerCase(x) == Character.toLowerCase(y);
}

public void keyTyped(KeyEvent event) {
    char c = event.getKeyChar();    //character guessed
    characters = characters+c;
    for (int i=0;i<word.length();i++)
        //if word[i]==c replace pattern[i] by word[i]:
        if (isEqual(word.charAt(i),c))
            pattern=pattern.substring(0,i)
                +word.charAt(i)+pattern.substring(i+1);
    userFrame.setWord(pattern);
    userFrame.setMessage(characters.length()+" characters tried: "
        +characters);
}
}

//supplies methods needed by WordManager:
public class UserFrame extends Frame
    implements ActionListener, WordInterface {
    private Label message;
    private TextField text;
    private WordManager wordManager;

    private void place(Component comp,int x,int y,int width,int height) {
        comp.setBounds(x, y, width, height);
        add(comp);
    }

    public UserFrame() {
        Button button;
        setTitle("Word Guess");
        setLayout(null);
        setSize(260,200);
        setResizable(false);
        place(message=new Label("Enter a word and press RETURN:"),
            30, 50, 200, 20);
        place(text=new TextField(),30,70,200, 20);
        text.addActionListener(this);
        text.setFont(new Font("Monaco",Font.PLAIN,9));
        place(button=new Button("NEW GAME"),80,120,80,20);
        button.addActionListener(this);
        text.setEchoChar('●');
        text.requestFocus();
        setVisible(true);
        text.requestFocus();
    }

    public void actionPerformed(ActionEvent event){
        if (event.getSource() instanceof TextField) {
            //let user guess:
            addKeyListener(wordManager = new WordManager(this));
            setMessage("");
            text.setEchoChar((char)0);    //disable echo
        }
    }
}
```

```
        text.setEditable(false);
    } else if (event.getActionCommand().equals("NEW GAME")) {
        //let user enter a new word:
        removeKeyListener(wordManager);
        setMessage("Enter a word and press RETURN:");
        text.setText("");
        text.setEchoChar('●');
        text.setEditable(true);
        text.requestFocus();
    }
}

public String getWord() {return text.getText();}

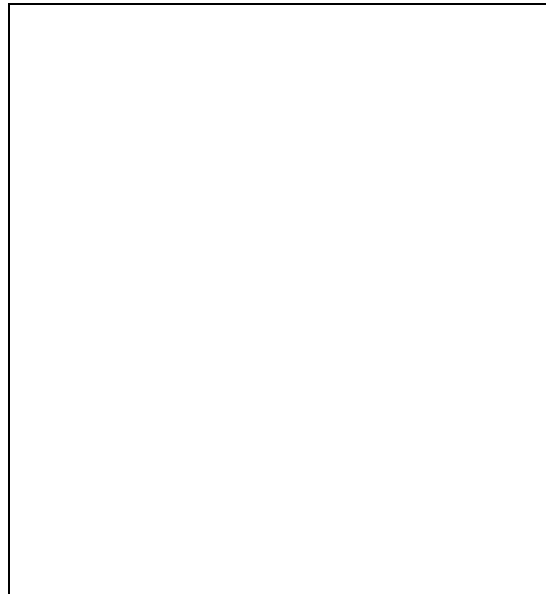
public void setWord(String s) {text.setText(s);}

public void setMessage(String s) {message.setText(s);}
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 9.1.1 Zum Programm

„Word Guess“ implementiert ein Ratespiel, bei welchem zwei Spieler gegeneinander antreten. Der eine Benutzer gibt in das in Abbildung 9-1 ersichtliche TextField das zu erratende Wort ein. Hierbei wird seine Eingabe maskiert. Nun beginnt der zweite Spieler zu raten, indem er sukzessive Buchstaben eingibt. Ist ein Buchstabe im zu erratenden Wort enthalten, wird er an der entsprechenden Stelle im TextField angezeigt. Oberhalb des TextField wird ausserdem angegeben, welche Buchstaben bereits erfragt wurden.



**Abbildung 9-1: Word Guess, UserFrame**

### 9.1.1.1 Interfaces

Das Programm „Word Guess“ deklariert das Interface

```
// specifies methods needed by WordManager:
public interface WordInterface {
    public String getWord();
    public void setWord(String s);
    public void setMessage(String s);
}
```

Ein **Interface** ist ein komplexer Datentyp, welcher nur abstrakte Methoden<sup>1</sup> deklariert<sup>2</sup>. Es kann zusätzlich auch Attribute enthalten, wobei diese aber nur Konstanten sein dürfen. Aufgrund der abstrakten Methoden kann ein Interface konsequenterweise nicht instanziiert werden.

Das Schlüsselwort **interface** besagt, dass es sich bei `WordInterface` um ein solches handelt. Das Interface `WordInterface` umfasst die drei abstrakten Methoden `getWord()`, `setWord()` und `setMessage()`.

---

<sup>1</sup> Die Verwendung des Schlüsselwortes `abstract` ist für die Methodendeklaration nicht zwingend, da bei einem Interface alle Methoden implizit abstrakt sind.

<sup>2</sup> Ein Interface geht also noch einen Schritt weiter als eine abstrakte Klasse, bei welcher nur eine Methode abstrakt sein muss oder auch gar keine abstrakte Methode aufweisen muss, wenn sie mit dem Schlüsselwort `abstract` deklariert wird.



Die Klasse `UserFrame` macht aufgrund des Schlüsselwortes **implements** deutlich, dass sie das Interface `WordInterface` implementiert:

```
public class UserFrame extends Frame
    implements ActionListener, WordInterface {
```

Dies bedeutet, dass sie sämtliche im Interface deklarierten abstrakten Methoden implementieren muss, was sie auf folgende Weise macht:

```
public String getWord() {return text.getText();}

public void setWord(String s) {text.setText(s);}

public void setMessage(String s) {message.setText(s);}
```

Die Methoden `getWord()`, `setWord()` und `setMessage()` werden im Programm also dazu verwendet, die Benutzereingabe zu ermitteln bzw. einen String im `TextField` oder im Label der graphischen Benutzeroberfläche auszugeben.

Ähnlich der Vererbung ist eine Instanz einer implementierenden Klasse immer auch eine Instanz des jeweiligen Interface. Im Gegensatz zur Vererbung darf aber eine Klasse auch mehrere Interfaces implementieren.

***Siehe auch:*** 9.1.2.1

#### 9.1.1.2 Adapter-Klassen

Wie ausführlich in Abschnitt 5.1.1.1 erläutert wurde, bedarf es für die Realisation des Event Handling immer der Implementation eines Listener. Da es sich bei sämtlichen Listeners um Interfaces handelt, erfordert die Implementation eines Listener die Implementation der in ihm deklarierten, abstrakten Methoden.

In der Klassenbibliothek gibt es etliche Listeners, welche mehrere abstrakte Methoden deklarieren. So gibt es beispielsweise den `KeyListener`, der `KeyEvents` abfängt und drei abstrakte Methoden deklariert: die Methode `keyPressed()` wird aufgerufen, wenn eine Taste gedrückt wird, `keyReleased()` wird ausgeführt wenn eine Taste losgelassen wird und `keyTyped()` schliesslich gibt das eingegebenen Zeichen an.

Möchte man nun aber - wie im Programm „Word Guess“ - lediglich das durch den Benutzer eingegebene Zeichen ermitteln, würde die Methode `keyTyped()` völlig ausreichen. Die Implementation des Interface `KeyListener` erzwingt aber auch die Implementation der beiden anderen abstrakten Methoden.

Um hier Abhilfe zu schaffen, deklariert Java für jeden Listener, welcher mehr als eine abstrakte Methode enthält, eine sogenannte Adapter-Klasse. Eine **Adapter-Klasse** ist eine

abstrakte Klasse<sup>1</sup>, welche einen Listener implementiert, indem sie für jede abstrakte Methode einen leeren Rumpf deklariert. Die in einer Adapter-Klasse deklarierten Methoden „bewirken“ also nichts und haben hierdurch auch keinen Einfluss auf das Event Handling. Indem man von einer Adapter-Klasse erbt, hat man die Möglichkeit, wohl einen Listener zu implementieren, muss jedoch nur jene Methode(n) redefinieren, die man tatsächlich benötigt.

Im Programm wird die Klasse `WordManager` deklariert, welche von einer Klasse `KeyAdapter` erbt. Die Klasse `KeyAdapter` ist die Adapter-Klasse für den `KeyListener` und implementiert dessen abstrakten Methoden `keyPressed()`, `keyReleased()` und `keyTyped()`. Da man für das Ratespiel aber lediglich wissen muss, welche Taste gedrückt wurde, redefiniert die Klasse `WordManager` nur die Methode `keyTyped()`, welche jedes Mal, wenn der Benutzer einen Buchstaben rät, aufgerufen wird:

```
public void keyTyped(KeyEvent event) {
    char c = event.getKeyChar();    //character guessed
    characters = characters+c;
    for (int i=0;i<word.length();i++)
        //if word[i]==c replace pattern[i] by word[i]:
        if (isEqual(word.charAt(i),c))
            pattern=pattern.substring(0,i)
                +word.charAt(i)+pattern.substring(i+1);
    userFrame.setWord(pattern);
    userFrame.setMessage(characters.length()+" characters tried: "
                        +characters);
}
```

Die in der Klasse `KeyEvent` deklarierte Methode `getKeyChar()` liefert das durch den Benutzer gedrückte Zeichen. Falls dieses mit irgendeinem Zeichen des zu erratenden Worts übereinstimmt, wird es im maskierten String an der entsprechenden Stelle ausgegeben. Hierzu wird die in der Klasse `String` deklarierte Methode

```
public String substring(int beginIndex, int endIndex)
```

verwendet, welche denjenigen Teilstring zurückgibt, der an der Position `beginIndex` beginnt und an Position `endIndex-1` endet<sup>2</sup>.

Den Vergleich des durch den Benutzer eingegebenen Zeichens mit irgendeinem Zeichen des zu erratenden Wortes vollführt die Methode

---

<sup>1</sup> Eine Adapter-Klasse ist lediglich als `abstract` deklariert, um eine Instanzierung zu vermeiden, weist aber keine abstrakten Methoden auf.

<sup>2</sup> Die Methode `substring(int beginIndex, int endIndex)` ist von der in Abschnitt 5.1.1 erläuterten Methode `substring(int beginIndex)` zu unterscheiden, da sie im Gegensatz zu dieser zwei Parameter deklariert.

```
private boolean isEqual(char x, char y) {  
    return Character.toLowerCase(x) == Character.toLowerCase(y);  
}
```

Damit dieser Vergleich unabhängig von Gross- oder Kleinschreibung ist, verwendet sie die Methode `toLowerCase()`, welche den übergebenen Buchstaben in einen Kleinbuchstaben umwandelt.

Die Instanzvariable `userFrame` referenziert die graphische Benutzeroberfläche, also den `UserFrame`. Hierbei ist zu beachten, dass sie vom Typ des Interface `WordInterface` deklariert wurde. Aufgrund dieser Deklaration ist es auch möglich, der Variablen `userFrame` eine Instanz einer anderen Klasse, welche das Interface `WordInterface` implementiert, zuzuweisen. Dies illustriert die Flexibilität, welche die Verwendung eines Interface bietet: ungeachtet von der Vererbungshierarchie kann man ein Protokoll bestimmen, welches zur gegenseitigen Kommunikation mehrerer Klassen dient. Daher auch der Name Interface (dt. Schnittstelle).

Ein Adapter wird auf die gleiche Weise wie ein Interface registriert. In der Methode `actionPerformed()` der Klasse `UserFrame` geschieht dies in der Zeile

```
addKeyListener(wordManager = new WordManager(this));
```

Da man aber nicht will, dass die eingegebenen Zeichen mit dem zu erratenden Wort des vorhergehenden Durchgangs verglichen werden, wenn der Benutzer ein neues Wort eingibt, muss der `KeyListener` vorübergehend „taub“ gemacht werden.

```
removeKeyListener(wordManager);1
```

***Siehe auch:*** 5.1.2.1, 9.1.2.2

## 9.1.2 Syntax

### 9.1.2.1 Interfaces

Ein **Interface** ist ein komplexer Datentyp, dessen Methoden allesamt abstrakt sind und der, falls er Attribute hat, nur Konstanten aufweist. Er wird mit dem Schlüsselwort **interface** deklariert.

---

<sup>1</sup> Leider ist für dieses Programm die Portabilität nicht vollständig gewährleistet. So hat sich gezeigt, dass man für einen Windows-PC den Listener beim `TextField` `text` registrieren bzw. dort auch deaktivieren muss.

### Interface

```
interface Identifier {  
    Constants  
    AbstractMethods  
}
```

Beispiel: 

```
public interface WordInterface {  
    public String getWord();  
    public void setWord(String s);  
    public void setMessage(Strings s);  
}
```

Die Methoden eines Interface müssen nicht zwingenderweise das Schlüsselwort **abstract** verwenden, da sie implizit abstrakt sind.

So wie eine Klasse Unterklasse einer anderen Klasse sein kann, kann sie auch ein Interface **implementieren**, was sie mit dem Schlüsselwort **implements** besagt. In einem solchen Fall muss sie sämtliche abstrakte Methoden des Interface implementieren.

### InterfaceImplementation

```
class Identifier implements InterfaceName {}
```

Beispiel: Siehe Beispielprogramm „Word Guess“, Klasse UserFrame

Eine Instanz einer Klasse, welche ein Interface implementiert, ist automatisch auch eine Instanz des jeweiligen Interface.

In Java kann eine Klasse mehrere Interfaces implementieren.

Bestimmungen bezüglich dem type casting von Interfaces finden sich in Abschnitt 4.3.2.2.

### 9.1.2.2 Adapter-Klassen

Eine **Adapter-Klasse** ist eine Klasse, welche einen entsprechenden Listener implementiert. Hierbei liefert sie für sämtliche im Listener deklarierte Methoden einen leeren Methodenrumpf. Eine Adapter-Klasse ist, obwohl sie keine abstrakten Methoden enthält, abstrakt, um ihre Instanzierung zu vermeiden.

Muss man zwecks Realisation des Event Handling einen Listener implementieren, welcher mehrere abstrakte Methoden deklariert, benötigt aber lediglich eine oder nur einen Teil von diesen, dann ist es vorteilhaft, von der entsprechenden Adapter-Klasse zu erben. In diesem Fall muss man nur gerade die erfordernten Methoden redefinieren und implementiert dennoch das Interface.

Beim Namen einer Adapter-Klasse stimmt der erste Teil mit dem zugehörigen Listener überein: die Adapter-Klasse für einen `KeyListener` beispielsweise heisst dementsprechend `KeyAdapter`. Ein Adapter wird - wie auch ein Listener - bei der ereignisgenerierenden Komponente mittels `addXListener()` registriert bzw. mittels `removeXListener()` deaktiviert.

Die nachfolgende Tabelle zeigt, welche Interfaces in der Java-Klassenbibliothek durch Adapter-Klassen implementiert werden:

Listenerinterface	Adapter-Klasse
<code>ComponentListener</code>	<code>ComponentAdapter</code>
<code>ContainerListener</code>	<code>ContainerAdapter</code>
<code>FocusListener</code>	<code>FocusAdapter</code>
<code>KeyListener</code>	<code>KeyAdapter</code>
<code>MouseListener</code>	<code>MouseAdapter</code>
<code>MouseMotionListener</code>	<code>MouseMotionAdapter</code>
<code>WidowListener</code>	<code>WidowAdapter</code>

## 9.2 Zusammenfassung

In diesem Kapitel stehen die Interfaces und Adapter-Klassen im Vordergrund:

- ◆ Ein **Interface** ist ein abstrakter Datentyp, welcher nur abstrakte Methoden umfasst. Deklariert er zusätzlich noch irgendwelche Attribute, dann nur Konstanten.
- ◆ Eine Klasse, welche für sämtliche in einem Interface deklarierten abstrakten Methoden einen Rumpf aufweist, **implementiert** das Interface. Hierdurch ist eine Instanz einer solchen implementierenden Klasse eine Instanz des Interface.
- ◆ Eine Klasse kann mehrere Interfaces zugleich implementieren.

- ◆ Eine **Adapter-Klasse** ist eine Klasse, welche einen zugehörigen Listener implementiert. Obwohl sie keine abstrakten Methoden aufweist, ist sie abstrakt, um ihre Instanzierung zu verhindern.
- ◆ Sämtliche Methoden einer Adapter-Klasse weisen einen **leeren Rumpf** auf und haben somit keinen Einfluss auf das Event Handling.
- ◆ Will man nicht alle Methoden eines Listener implementieren, kann man von der entsprechenden Adapter-Klasse **erben**, um dort die entsprechenden Methoden zu redefinieren.
- ◆ Eine Adapter-Klasse wird - wie ein Listener - bei der ereignisgenerierenden Komponente **registriert** bzw. **deaktiviert**.

# 10 Arrays Schleifen

Da in den vorangehenden Kapiteln sämtliche einzuführende Konzepte bereits erläutert wurden, sollen die nun folgenden Kapitel anhand von anspruchsvolleren Programmbeispielen aufzeigen, was sonst noch möglich ist. Die nachfolgenden Beispiele sind als Anregung zu verstehen. Sie werden nicht mehr ausführlich besprochen, es werden lediglich die grundlegendsten Ideen vermittelt.

## 10.1 Zweidimensionaler Array

Die Application „Puzzle“ ist ein Geduldspiel, bei welchem in einer Matrix angeordnete Spielsteine sortiert werden müssen.

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

public class Tile extends Button implements ActionListener{
    public static final Color tileColor = Color.lightGray;
    public static final Color emptyColor = Color.black;
    private Tile[] neighbour;

    public Tile(String label) {
        setLabel(label);
        if (isEmpty()) setBackground(emptyColor);
        else setBackground(tileColor);
        addActionListener(this);
    }

    public void setNeighbour(Tile[] t) {neighbour = t;}

    private boolean isEmpty() {return (getLabel().equals("0"));}

    private void moveTo(Tile t) {
        t.setBackground(tileColor);
```

```

        t.setLabel(getLabel());
        setBackground(emptyColor.black);
        setLabel("0");
    }

    public void actionPerformed(ActionEvent event){
        for(int i=0;i<4;i++){
            if (neighbour[i].isEmpty())
                moveTo(neighbour[i]);
        }
    }
}

public class UserFrame extends Frame {
    static private int n=4;
    static private int size=40;

    private void place(Component comp,int x,int y,int width,int height) {
        comp.setBounds(x, y, width, height);
        add(comp);
    }

    //generate a random permutation of array p:
    private void shuffle(int[] p) {
        Random rndm = new Random();
        for (int k=0;k<n*n;k++) {
            int i=Math.abs(rndm.nextInt())%(n*n);
            int j=Math.abs(rndm.nextInt())%(n*n);
            int temp=p[i];
            p[i]=p[j];
            p[j]=temp;
        }
    }

    public UserFrame() {
        setTitle("Puzzle");
        setLayout(null);
        setSize(size*n,20+size*n);
        setResizable(false);
        setVisible(true);
        Tile[][] tile = new Tile[n][n];
        Tile border = new Tile(String.valueOf(n*n));
        int[] perm = new int[n*n];
        for (int i=0;i<n*n;i++) perm[i]=i;
        shuffle(perm);
        for (int i=0;i<n;i++)
            for (int j=0;j<n;j++)
                place(tile[i][j]=new Tile(String.valueOf(perm[n*i+j])),
                    size*i,20+size*j,size,size);
        for (int i=0;i<n;i++)
            for (int j=0;j<n;j++)
                tile[i][j].setNeighbour(new Tile[] {
                    i>0?tile[i-1][j]:border,
                    j<n-1?tile[i][j+1]:border,
                    i<n-1?tile[i+1][j]:border,
                    j>0?tile[i][j-1]:border
                });
    }
}

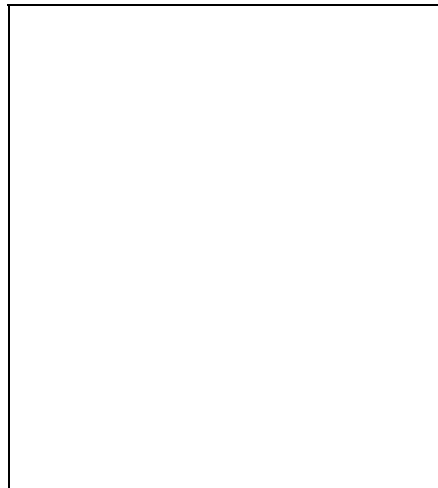
```



```
public class TestProg {  
    public static void main(String[] args) {new UserFrame();}  
}
```

### 10.1.1 Zum Programm

Wie Abbildung 10-1 zeigt, realisiert das Programm „Puzzle“ ein Spiel, bei welchem man die numerierten Spielsteine aufsteigend von links nach rechts bzw. von oben nach unten sortieren muss. Hierbei können die Steine aufgrund des leeren Feldes ausgetauscht werden.



**Abbildung 10-1: Puzzle, UserFrame**

Obwohl es für den Benutzer den Anschein hat, dass er beim Sortieren die Spielsteine tatsächlich verschiebt, ist dem in Wirklichkeit nicht so. Im Programm sind die Spielsteine, die sogenannten `Tiles`, lediglich unterschiedlich eingefärbt: ein Leerraum wird durch einen schwarzen Spielstein symbolisiert und mit "0" beschriftet. Wenn ein Benutzer einen Stein vermeintlich verschiebt, wird allein die Farbe von hellgrau nach schwarz gewechselt und die Beschriftung verändert.

Die Spielmatrix wird im Programm durch einen **zweidimensionalen Array**, bestehend aus `Tiles`, dargestellt:

```
Tile[][] tile = new Tile[n][n];
```

Dieser wird im Konstruktor `UserFrame()` im Programmstück

```
for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
        place(tile[i][j]=new Tile(String.valueOf(pern[n*i+j])),
            size*i,20+size*j,size,size);
```

mit Tiles initialisiert, welche sogleich im UserFrame plaziert werden. Hierbei indiziert der Ausdruck

`tile[i][j]`

dasjenige Element des zweidimensionalen Array `tile`, welches sich in der Zeile `i` und Spalte `j` befindet.

Dank der **ineinandergeschachtelten for Schleife**<sup>1</sup> wird die Spielmatrix erzeugt: die äussere Schleife iteriert über die Zeilen, die innere Schleife über die Spalten der Matrix, so dass das Spielfeld Zeile für Zeile generiert wird.

Jedes `Tile` verfügt über einen vierelementigen Array `neighbour`, welcher seine unmittelbaren Nachbarn, also wiederum `Tiles` beinhaltet. Hierbei indiziert das erste Element von `neighbour` den oberen Nachbar, das zweite den rechten, das dritte den unteren und das letzte Element schlussendlich den links liegenden Nachbarn.

Im Konstruktor `UserFrame()` in den Zeilen

```
for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
        tile[i][j].setNeighbour(new Tile[] {
            i>0?tile[i-1][j]:border,
            j<n-1?tile[i][j+1]:border,
            i<n-1?tile[i+1][j]:border,
            j>0?tile[i][j-1]:border
        });
```

werden die Nachbarn für jedes `Tile` der Spielmatrix festgesetzt. Falls ein Spielstein am Rand des Spielfeldes liegt, hat er nur drei und im Falle eines Ecksteines nur zwei effektive Nachbarn. In diesen Fällen werden die restlichen Nachbarn mit `border` initialisiert. Das `Tile border` ist ein unsichtbarer Spielstein, welcher für die Darstellung des das Spielfeld umgebenden Randes verwendet wird.

Die Methode `setNeighbour()` verlangt als Parameter einen Array, welcher `Tiles` enthält. In ihrem Aufruf erhält sie einen sogenannten anonymen Array.

Ein **anonymer Array** ist namenlos. Er kann gleichzeitig mit seiner Generierung ohne Zuweisung initialisiert werden:

---

<sup>1</sup> Der Begriff macht deutlich, dass sich die innere for Schleife im Rumpf der äusseren befindet. Dies hat zur Folge, dass mit einem Durchlauf der äusseren for Schleife die innere Schleife einmal vollständig durchlaufen wird.

```
new Tile[] {i>0?tile[i-1][j]:border,j<n-1?tile[i][j+1]:border,  
            i<n-1?tile[i+1][j]:border,j>0?tile[i][j-1]:border}
```

# 11 Turtle-Geometrie

## Rekursion

### Stack

Die nun folgenden Programme basieren auf der sogenannten Turtle-Geometrie von Logo. **Logo** ist eine Programmiersprache, die anfangs 70er Jahre entwickelt wurde und als Einstieg ins Programmieren für Anfänger, insbesondere für Jugendliche, dienen soll.

Die Grundidee der **Turtle-Geometrie** ist die Erzeugung graphischer Darstellungen durch die programmierte Bewegung einer Turtle (dt. Schildkröte). Hierbei rührt die Idee ursprünglich von der Vorstellung einer sich im Sand fortbewegenden Schildkröte her: dabei wird die Spur, die sie hinterlässt, als Graphik interpretiert. Grundsätzlich kann sich die Turtle immer nur geradeaus, also in Richtung ihres Kopfes, bewegen. Will man ihre momentane Ausrichtung ändern, muss man sie drehen.

## 11.1 Turtle-Geometrie

Das vorliegende Programm „Turtle - Version 1“ illustriert die Turtle-Geometrie.

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

public class Turtle {
    private Graphics graphics;
    private Point position;
    private int orientation;

    public Turtle(Graphics g) {
        graphics = g;
        Rectangle clip = g.getClipBounds();
        position = new Point(clip.x + clip.width/2, clip.y + clip.height/2);
        orientation = 0;
    }

    public void forward(int distance) {
        int x = position.x +
```

```
        (int)Math.round(distance*Math.sin(Math.PI*orientation/180));
int y = position.y -
        (int)Math.round(distance*Math.cos(Math.PI*orientation/180));
graphics.drawLine(position.x, position.y, x, y);
position.setLocation(x, y);
    }

    public void right(int angle) {
        orientation = (orientation + angle)%360;
    }
}

public class UserFrame extends Frame{

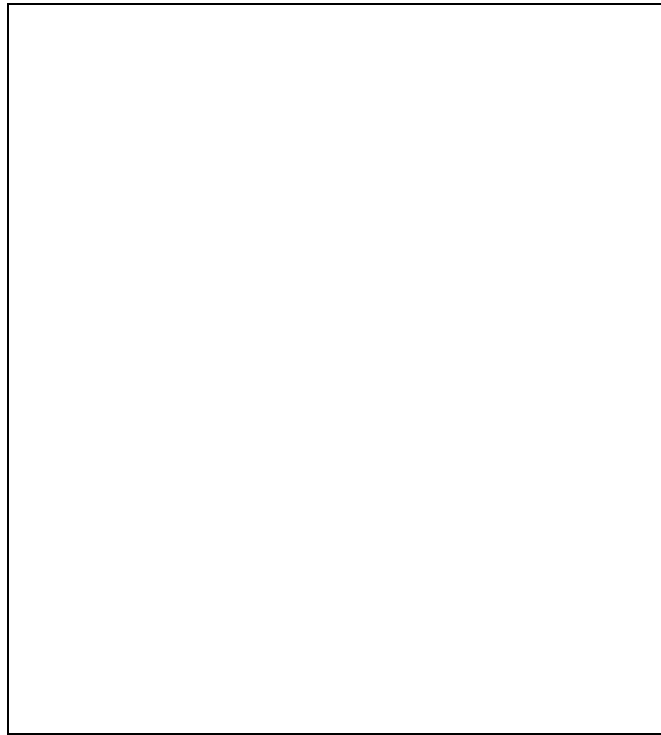
    public UserFrame() {
        setTitle("Drawing");
        setSize(400,400);
        setVisible(true);
    }

    public void paint(Graphics g) {
        Turtle turtle = new Turtle(g);
        for (int i=0;i<5;i++) {
            turtle.forward(100);
            turtle.right(144);
        }
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 11.1.1 Zum Programm

Abbildung 11-1 zeigt die graphische Ausgabe, die durch die im Programm definierte Fortbewegung einer Turtle entsteht.



**Abbildung 11-1: Turtle - Version 1, UserFrame**

Wie Abbildung 11-2 illustriert, verwendet das Programm „Turtle - Version 1“ zur Fortbewegung der Schildkröte die zwei Methoden `forward()` und `right()`. Die Positionierung der Turtle erfolgt mittels **Polarkoordinaten**.

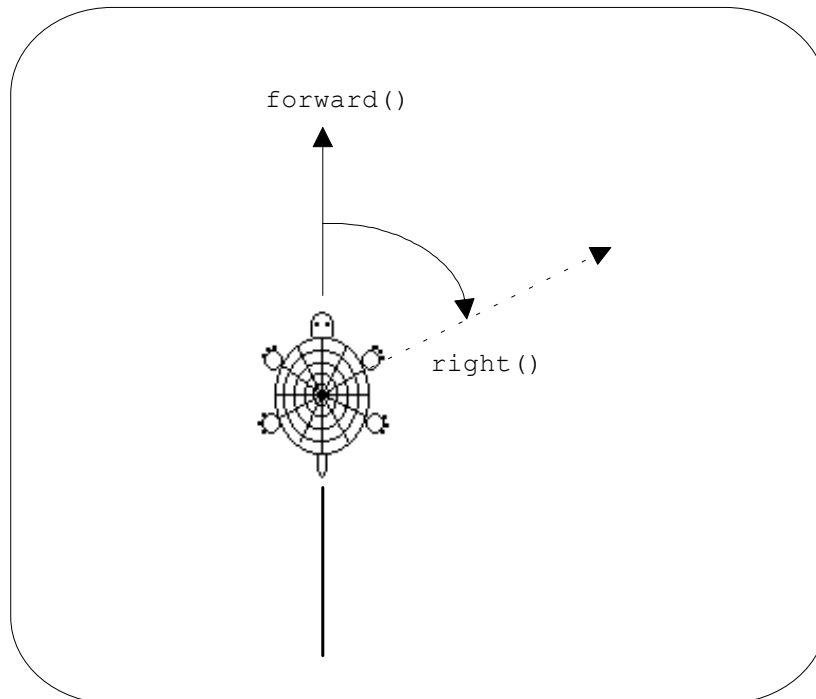


Abbildung 11-2: Turtle-Geometrie

## 11.2 Vererbung

Die vorangehende Programmversion wurde um die Klasse `PowerTurtle` zum Programm „Turtle - Version 2“ erweitert.

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

public class Turtle {
    private Graphics graphics;
    private Point position;
    private int orientation;

    public Turtle(Graphics g) {
        graphics = g;
        Rectangle clip = g.getClipBounds();
        position = new Point(clip.x + clip.width/2, clip.y + clip.height/2);
    }
}
```

```
        orientation = 0;
    }

    public void forward(int distance) {
        int x = position.x +
            (int)Math.round(distance*Math.sin(Math.PI*orientation/180));
        int y = position.y -
            (int)Math.round(distance*Math.cos(Math.PI*orientation/180));
        graphics.drawLine(position.x, position.y, x, y);
        position.setLocation(x, y);
    }

    public void right(int angle) {
        orientation = (orientation + angle) % 360;
    }
}

public class PowerTurtle extends Turtle {
    public PowerTurtle(Graphics g) {super(g);}

    public void polygon(int n, int size) {
        for (int i=0;i<n;i++) {
            forward(size);
            right(360/n);
        }
    }
}

public class UserFrame extends Frame{
    public UserFrame() {
        setTitle("Drawing");
        setSize(400,400);
        setVisible(true);
    }

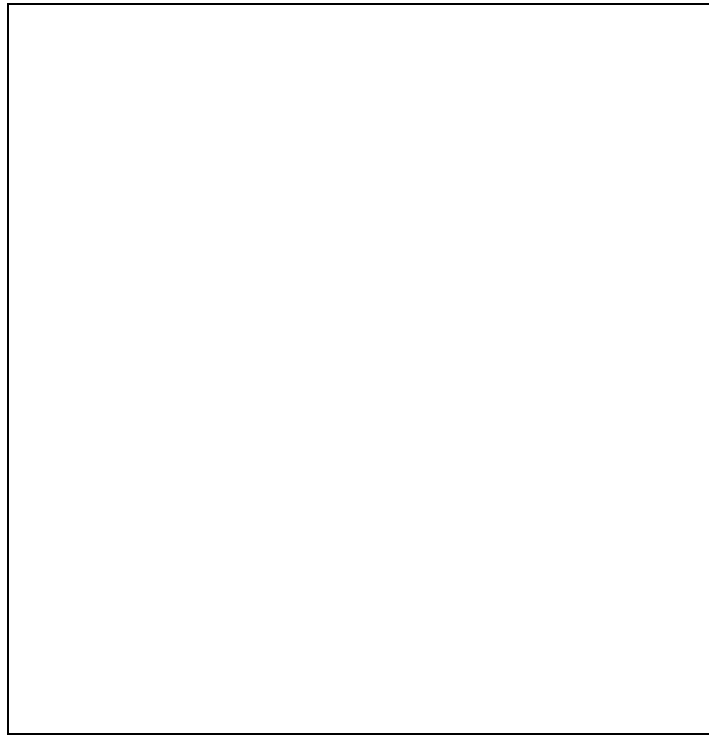
    public void paint(Graphics g) {
        PowerTurtle turtle = new PowerTurtle(g);
        turtle.polygon(8,50);
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 11.2.1 Zum Programm

Dieses Mal hinterlässt die Turtle die in Abbildung 11-3 ersichtliche Graphik.





**Abbildung 11-3: Turtle - Version 2, UserFrame**

Die neu deklarierte Klasse `PowerTurtle` enthält nun eine Methode zum Zeichnen von Polygonen. Die Klasse `PowerTurtle` ist Unterklasse von `Turtle`.

## 11.3 Rekursion

Das Programm „Turtle - Version 3“ erweitert die vorangehende Version. Änderungen sind wie immer **fett** markiert.

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

public class Turtle {
    private Graphics graphics;
    private Point position;
    private int orientation;

    public Turtle(Graphics g) {
        graphics = g;
        Rectangle clip = g.getClipBounds();
```

```

        position = new Point(clip.x + clip.width/2, clip.y + clip.height/2);
        orientation = 0;
    }

    public void forward(int distance) {
        int x = position.x +
            (int)Math.round(distance*Math.sin(Math.PI*orientation/180));
        int y = position.y -
            (int)Math.round(distance*Math.cos(Math.PI*orientation/180));
        graphics.drawLine(position.x, position.y, x, y);
        position.setLocation(x, y);
    }

    public void right(int angle) {
        orientation = (orientation + angle) % 360;
    }

    public void back(int distance) {forward(-distance);}

    public void left(int angle) {right(-angle);}
}

public class PowerTurtle extends Turtle {
    public PowerTurtle(Graphics g) {super(g);}

    public void star(int n, int size) {
        for (int i=0; i<n; i++) {
            forward(size);
            right(720/n);
        }
    }

    public void spiral(int size) {
        if (size>0) {
            forward(size);
            right(60);
            spiral(size-3);
        }
    }

    public void tree(int size) {
        if (size>0) {
            forward(size);
            left(60);
            tree(size/2);
            right(120);
            tree(size/2);
            left(60);
            back(size);
        }
    }
}

public class UserFrame extends Frame{
    public UserFrame() {
        setTitle("Drawing");
    }
}

```

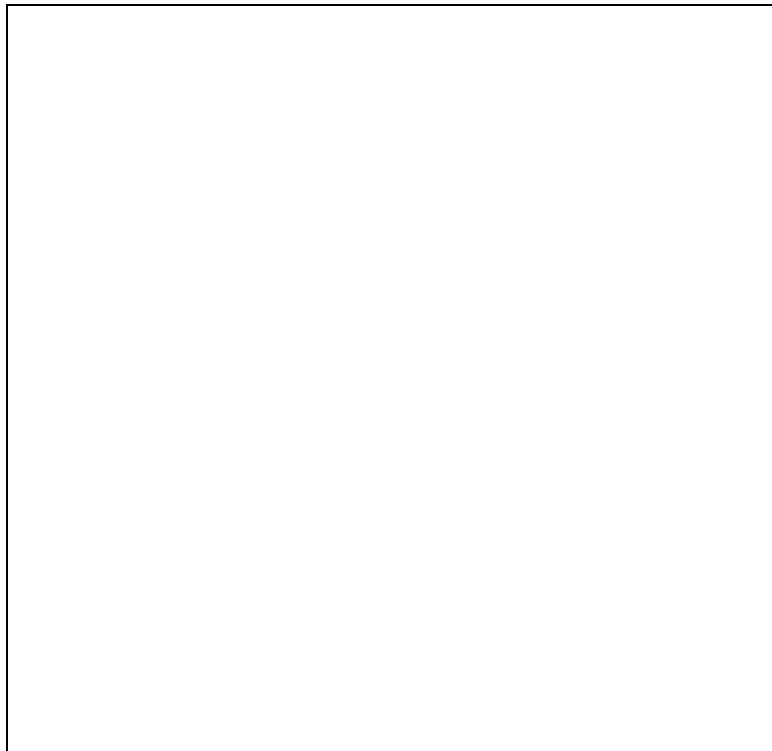
```
        setSize(400,400);
        setVisible(true);
    }

    public void paint(Graphics g) {
        PowerTurtle turtle = new PowerTurtle(g);
        turtle.spiral(80);
    }
}

public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 11.3.1 Zum Programm

Abbildung 11-4 illustriert die Ausgabe des Programmes.



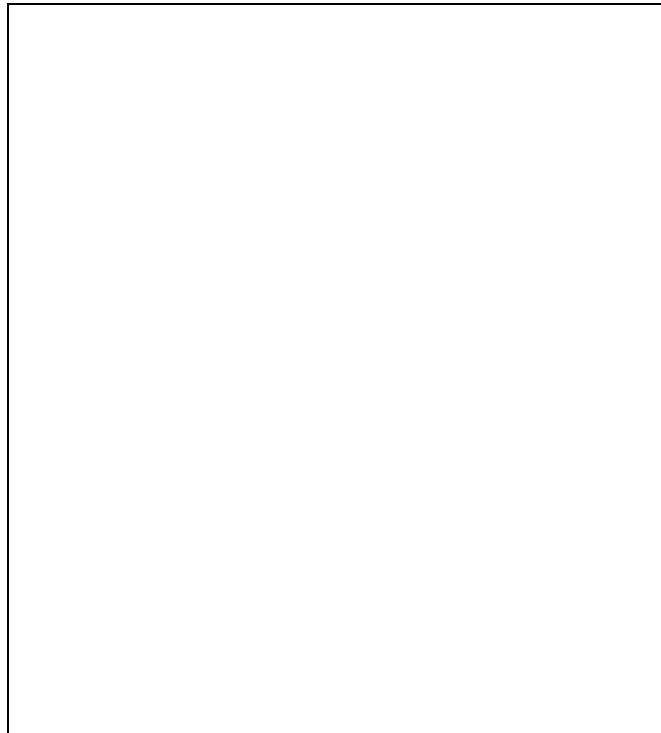
**Abbildung 11-4: Turtle - Version 3, UserFrame**

Das Programm „Turtle - Version 3“ deklariert nun zusätzlich die Methoden `back()` und `left()`, wobei diese invers zu den Methoden `forward()` bzw. `right()` sind.

Die Klasse `PowerTurtle` verfügt über die Methoden `spiral()`, zum Zeichnen einer Spirale, und `tree()`, zum Zeichnen eines Baumes. Beide Methoden sind rekursiv.

Unter **Rekursion** versteht man den Aufruf einer Methode in der Methode selber. In der Methode `spiral()` erfolgt als letztes statement der rekursive Aufruf. Die Methode `tree()` verwendet sogleich zweimal einen rekursiven Aufruf. Damit die Rekursion nicht endlos andauert, deklarieren beide Methoden die Laufbedingung (`size>0`).

Ein Aufruf von `turtle.tree(64)` in der Methode `paint()` ergibt den in Abbildung 11-5 ersichtliche Baum.



**Abbildung 11-5: Turtle - Version 3, UserFrame**

Der in Abbildung 11-5 dargestellte Baum visualisiert auch graphisch die beiden rekursiven Aufrufe der Methode `tree()`: ein Baum besteht aus einem Stamm und einem linken und rechten Teilbaum, welcher wiederum aus einem Stamm und einem linken und rechten Teilbaum besteht etc.

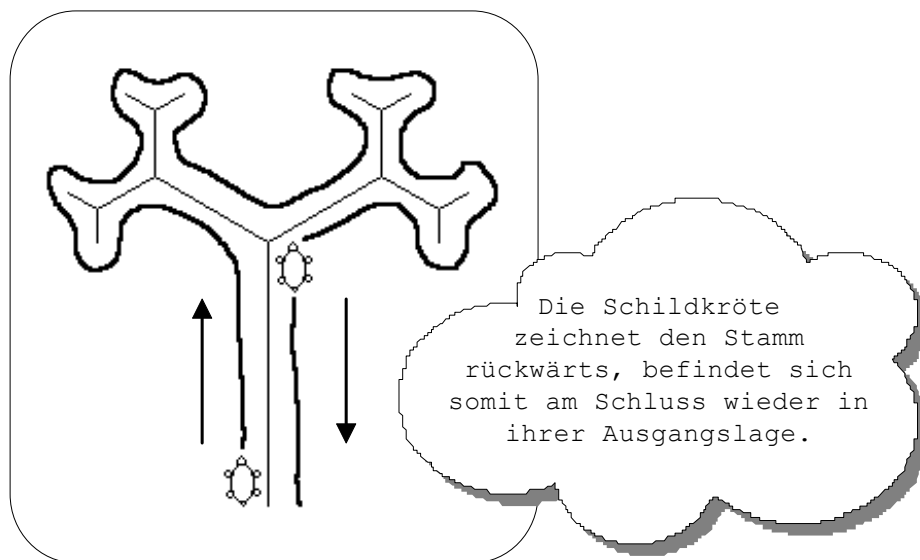
Analog zum rekursiven Aufbau eines Baumes sieht die Methode `tree()` folgendermassen aus:

```

public void tree(int size) {
    if (size>0) {
        forward(size);
        left(60);
        tree(size/2);
        right(120);
        tree(size/2);
        left(60);
        back(size);
    }
}

```

Falls die Laufbedingung noch erfüllt ist, wird der Stamm gezeichnet, darauf rekursiv der linke und dann rekursiv der rechte Teilbaum. Abbildung 11-6 veranschaulicht anhand eines Baumes mit weniger Ästen, wie die Rekursion abgearbeitet wird, das heisst, wie die Turtle den Baum zeichnet.



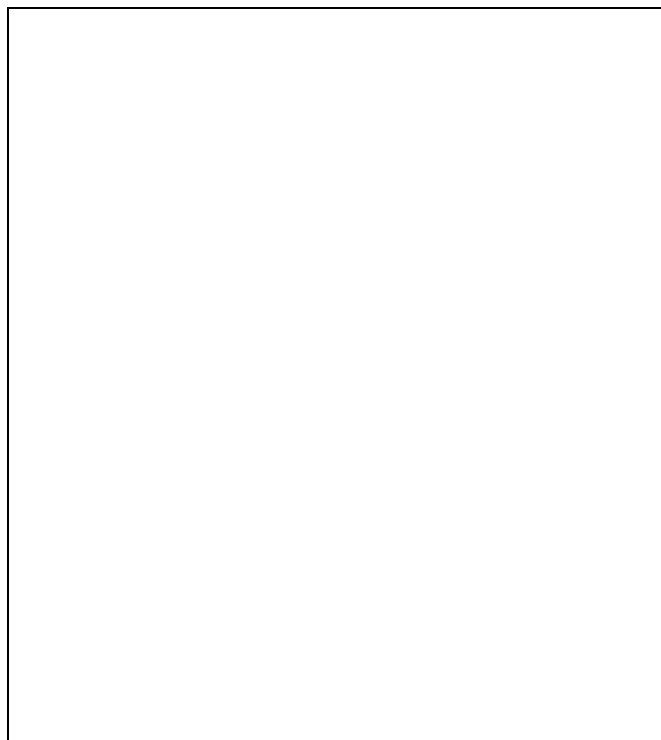
**Abbildung 11-6: Rekursive Struktur eines Baumes**

Tatsächlich zeichnet die Schildkröte den Stamm des Baumes bzw. jeden Stamm jedes Teilbaumes doppelt: einmal Vorwärts und einmal rückwärts. Am Schluss befindet sie sich wieder dort, wo sie mit dem Zeichnen angefangen hat, in derselben Ausrichtung.

Dass der Anfangszustand mit dem Endzustand der Turtle übereinstimmt, kann man auch aus der Symmetrie des Programmes entnehmen: sukzessive werden die Anweisungen des ersten Programmteils im zweiten Programmteil invertiert. Der Anweisung `forward(size)` steht die Anweisung `back(size)` gegenüber, die Aufrufe

`tree(size/2)` gruppieren sich spiegelbildlich um die Achse `right(120)`, und die zwei Linksdrehungen werden durch `right(120)` rückgängig gemacht.

Obwohl aufgrund des Programmes die Schildkröte beim Zeichnen jedes Teilbaumes immer wieder in ihre Ausgangslage zurückgelangen sollte, können Rundungsfehler in der Methode `forward()` diese Symmetrie zerstören. Abbildung 11-7 zeigt, welche Auswirkungen Rundungsfehler auf den Programmverlauf haben können. Es wurde die Methode `tree()` mit dem Parameter 48 aufgerufen.



**Abbildung 11-7: Turtle - Version 3, UserFrame**

## 11.4 Stack

Die Application „Turtle - Version 4“ weist nun neu die Klasse Logo auf.

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

---

```

import java.awt.event.*;

public class Turtle {
    private Graphics graphics;
    private Point position;
    private int orientation;

    public Turtle(Graphics g) {
        graphics = g;
        Rectangle clip = g.getClipBounds();
        position = new Point(clip.x + clip.width/2, clip.y + clip.height/2);
        orientation = 0;
    }

    public void forward(int distance) {
        int x = position.x +
            (int)Math.round(distance*Math.sin(Math.PI*orientation/180));
        int y = position.y -
            (int)Math.round(distance*Math.cos(Math.PI*orientation/180));
        graphics.drawLine(position.x, position.y, x, y);
        position.setLocation(x, y);
    }

    public void right(int angle) {
        orientation = (orientation + angle) % 360;
    }
}

public class Logo {
    private Sequence code;
    private Turtle turtle;

    public abstract class Command {
        abstract public void run();
        abstract public String toString();
    }

    public class Forward extends Command {
        private int argument;
        public Forward(int n) {argument = n;}
        public void run() {turtle.forward(argument);}
        public String toString() {return "forward "+argument+"\n";}
    }

    public class Back extends Command {
        private int argument;
        public Back(int n) {argument = n;}
        public void run() {turtle.forward(-argument);}
        public String toString() {return "back "+argument+"\n";}
    }

    public class Right extends Command {
        private int argument;
        public Right(int n) {argument = n;}
        public void run() {turtle.right(argument);}
        public String toString() {return "right "+argument+"\n";}
    }
}

```

```
public class Left extends Command {
    private int argument;
    public Left(int n) {argument = n;}
    public void run() {turtle.right(-argument);}
    public String toString() {return "left "+argument+"\n";}
}

public class Sequence extends Command {
    private Stack commands;
    public Sequence() {commands = new Stack();}
    public void add(Command c) {commands.push(c);}
    public void undo() {if (!commands.empty()) commands.pop();}

    public void run() {
        int size = commands.size();
        for (int i=0;i<size;i++)
            ((Command)commands.elementAt(i)).run();
    }

    public String toString() {
        int size = commands.size();
        String result = "";
        for (int i=0;i<size;i++)
            result = result+commands.elementAt(i);
        return result;
    }
}

public Logo() {clear();}

public void clear() {code = new Sequence();}

public void add(String name, int arg) {
    if (name.equals("FD")) code.add(new Forward(arg));
    else if (name.equals("BK")) code.add(new Back(arg));
    else if (name.equals("RT")) code.add(new Right(arg));
    else if (name.equals("LT")) code.add(new Left(arg));
}

public void undo() {code.undo();}

public String getText() {return code.toString();}

public void run(Graphics g) {
    turtle = new Turtle(g);
    code.run();
}
}

public class UserFrame extends Frame implements ActionListener {
    private TextArea text;
    private Button button;
    private CheckboxGroup commands, arguments;
    private Drawing drawing;
    private Logo logo;

    public class NumCheckbox extends Checkbox {
        int num;
    }
}
```



```

    public NumCheckbox(int n, boolean state, CheckboxGroup group) {
        super(String.valueOf(n), state, group);
        num = n;
    }

    public int getNum() {return num;}
}

public class Drawing extends Frame {
    public Drawing() {
        setTitle("Drawing");
        setBounds(400,0,400,500);
        setVisible(true);
    }

    public void paint(Graphics g) {logo.run(g);}
}

private void place(Component comp,int x,int y,int width,int height) {
    comp.setBounds(x, y, width, height);
    add(comp);
}

public UserFrame() {
    setTitle("Editor");
    setLayout(null);
    setBounds(0,0,400,515);
    setResizable(false);
    commands = new CheckboxGroup();
    arguments = new CheckboxGroup();
    place(text = new TextArea(),30,80,340,340);
    text.setBackground(Color.white);
    place(button = new Button("CLEAR"),30,450,60,20);
    button.setBackground(Color.white);
    button.addActionListener(this);
    place(new Checkbox("FD",true,commands),120,440,40,15);
    place(new Checkbox("BK",false,commands),120,455,40,15);
    place(new Checkbox("RT",false,commands),120,470,40,15);
    place(new Checkbox("LT",false,commands),120,485,40,15);
    place(new NumCheckbox(5,true,arguments),170,440,40,15);
    place(new NumCheckbox(30,false,arguments),170,455,40,15);
    place(new NumCheckbox(50,false,arguments),170,470,40,15);
    place(new NumCheckbox(90,false,arguments),170,485,40,15);
    place(button = new Button("DO"),240,450,60,20);
    button.setBackground(Color.white);
    button.addActionListener(this);
    place(button = new Button("UNDO"),310,450,60,20);
    button.setBackground(Color.white);
    button.addActionListener(this);
    logo = new Logo();
    drawing = new Drawing();
    setVisible(true);
}

public void actionPerformed(ActionEvent event) {
    if (event.getActionCommand().equals("CLEAR")) {
        logo.clear();
    }
}

```

```
    } else if (event.getActionCommand().equals("DO")) {
        String name = commands.getSelectedCheckbox().getLabel();
        int arg = ((NumCheckbox)
            arguments.getSelectedCheckbox()).getNum();
        logo.add(name, arg);
    } else if (event.getActionCommand().equals("UNDO"))
        logo.undo();
    drawing.repaint();
    text.setText(logo.getText());
}

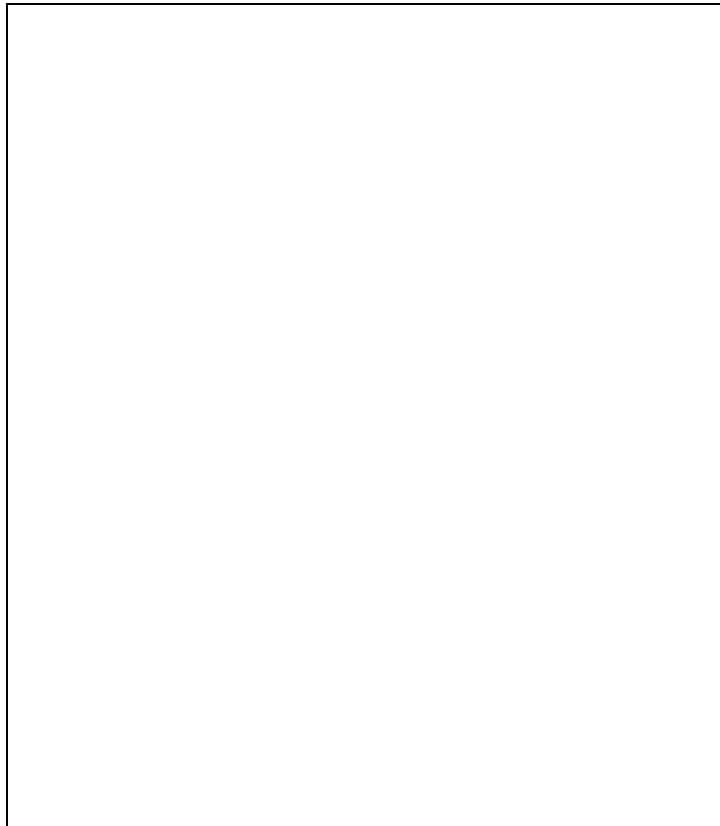
public class TestProg {
    public static void main(String[] args) {new UserFrame();}
}
```

### 11.4.1 Zum Programm

Diese Programmversion bietet nun dem Benutzer die Möglichkeit, selber Turtle-Graphiken zu erstellen. Hierzu steht ihm ein Editorfenster (siehe Abbildung 11-8) zur Verfügung, welches die zur Fortbewegung der Turtle notwendigen Befehle forward, back, right und left in Form von radio buttons aufweist. Der Button UNDO erlaubt, den zuletzt eingegebenen Befehl rückgängig zu machen. Abbildung 11-9 zeigt die Graphik, die aufgrund der im Editor ersichtlichen Befehlsfolge entsteht: die Laterne ist in einem Zug gezeichnet.



**Abbildung 11-8: Turtle - Version 4, UserFrame**



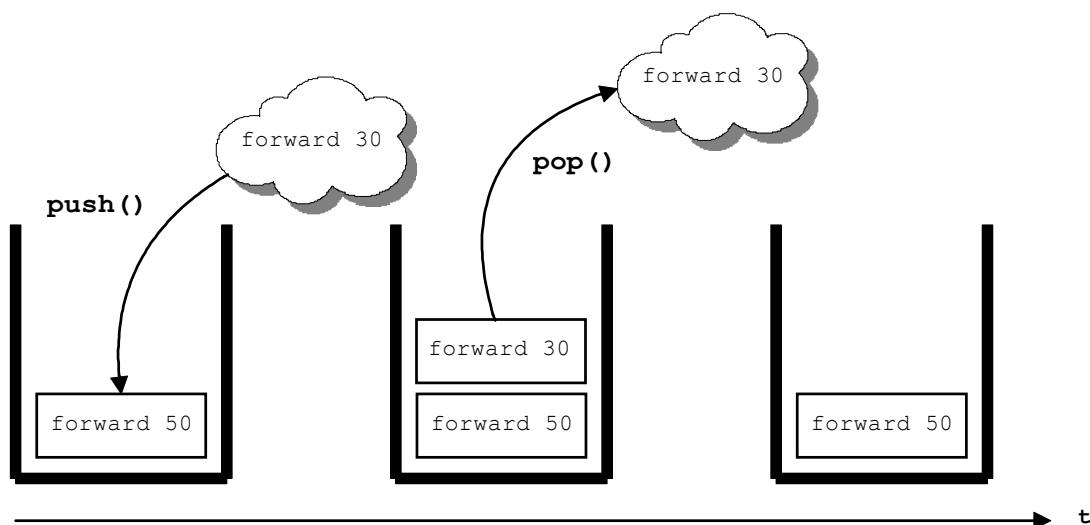
**Abbildung 11-9: Turtle - Version 4, Drawing**

Die Klasse `Logo` bildet einen in Java eingebetteten **Interpreter** für die Programmiersprache Logo. Sie führt die durch die Checkboxes generierten Anweisungen direkt aus, ohne ein wiederverwendbares Maschinenprogramm zu erzeugen.

Im Programm verwenden die Klassen `Logo` und `UserFrame` sogenannte innere Klassen. Eine **innere Klasse** (engl. **inner class**) ist eine Klasse, welche wie ein Attribut innerhalb einer anderen Klasse deklariert wird.

Für die Implementation des Buttons UNDO wird ein Stack verwendet. Ein **Stack** (Stapel) ist eine Datenstruktur, welche das Ablegen und Auffinden von Daten ermöglicht. Hierbei hat man immer nur auf das zuletzt abgelegte Element Zugriff. Will man also auf das vorletzte Element zugreifen, muss man zuerst das zuletzt abgelegte Element wegnehmen.

Die Klasse `Stack` realisiert eine solche Datenstruktur. Sie deklariert die für einen Stack charakteristischen Methoden `push()` und `pop()`. Wenn man sich vorstellt, dass die zu speichernden Daten aufeinandergestapelt werden, dann legt die Methode `push()` das als Parameter übergebene Element zuoberst auf dem Stapel ab und die Methode `pop()` nimmt das oberste Element vom Stapel weg (siehe Abbildung 11-10).



**Abbildung 11-10: Illustration eines Stack**

Die innere Klasse `Sequence` verwendet für die Speicherung der zum Zeichnen der Graphik benötigten Anweisungen einen Stack. Aufgrund der Methode `pop()` ist es ihr dann möglich, die Anweisungen Schritt für Schritt rückgängig zu machen.

# 12

## Binärer Baum Hashtabelle

Die Programmreihe „Family“ illustriert anhand eines Familienstammbaumes die Datenstruktur eines binären Baumes.

### 12.1 Binärer Baum

Das untenstehende Programm ist ein Ausschnitt aus dem Beispielprogramm „Family - Version 3“.

```
public class Person {
    static Hashtable table = new Hashtable(100);
    static EmptyPerson empty = new EmptyPerson();

    static boolean isKnown(String name) {return table.containsKey(name);}
    static Person getPerson(String name) {return (Person)table.get(name);}

    String name;
    boolean male;
    Person father, mother;

    public Person(String string) {
        name=string;
        table.put(name, this);
    }

    public boolean isEmpty() {return false;}
    public void setSex(String sex) {male = sex.equals("male");}
    public void setFather(Person person) {father=person;}
    public void setMother(Person person) {mother=person;}
    public String getName() {return name;}
    public Person getFather() {return father;}
    public Person getMother() {return mother;}

    public String getSex() {
        if (male) return "male";
    }
}
```

```

        else return "female";
    }

    public String getFathersName() {return getFather().getName();}
    public String getMothersName() {return getMother().getName();}
    public String toString() {return name+" (" +father+", " +mother+")";}
    public int count() {return 1+father.count()+mother.count();}

    public int maxDepth() {
        return 1+Math.max(father.maxDepth(), mother.maxDepth());
    }

    public boolean treeContains(Person person) {
        if (this==person) return true;
        else
            return father.treeContains(person) || mother.treeContains(person);
    }

    public boolean isRelatedTo(Person person) {
        if (person.treeContains(this)) return true;
        else
            return father.isRelatedTo(person) || mother.isRelatedTo(person);
    }

    public void paint(Graphics g, int x, int y, int dist) {
        g.fillOval(x-3,y-3,6,6);
        g.drawString(name,x+6,y+3);
        if ((!father.isEmpty()) & (y>60)) {
            g.drawLine(x,y,x-dist,y-50);
            father.paint(g,x-dist,y-50,dist/2);
        }
        if ((!mother.isEmpty()) & (y>60)) {
            g.drawLine(x,y,x+dist,y-50);
            mother.paint(g,x+dist,y-50,dist/2);
        }
    }

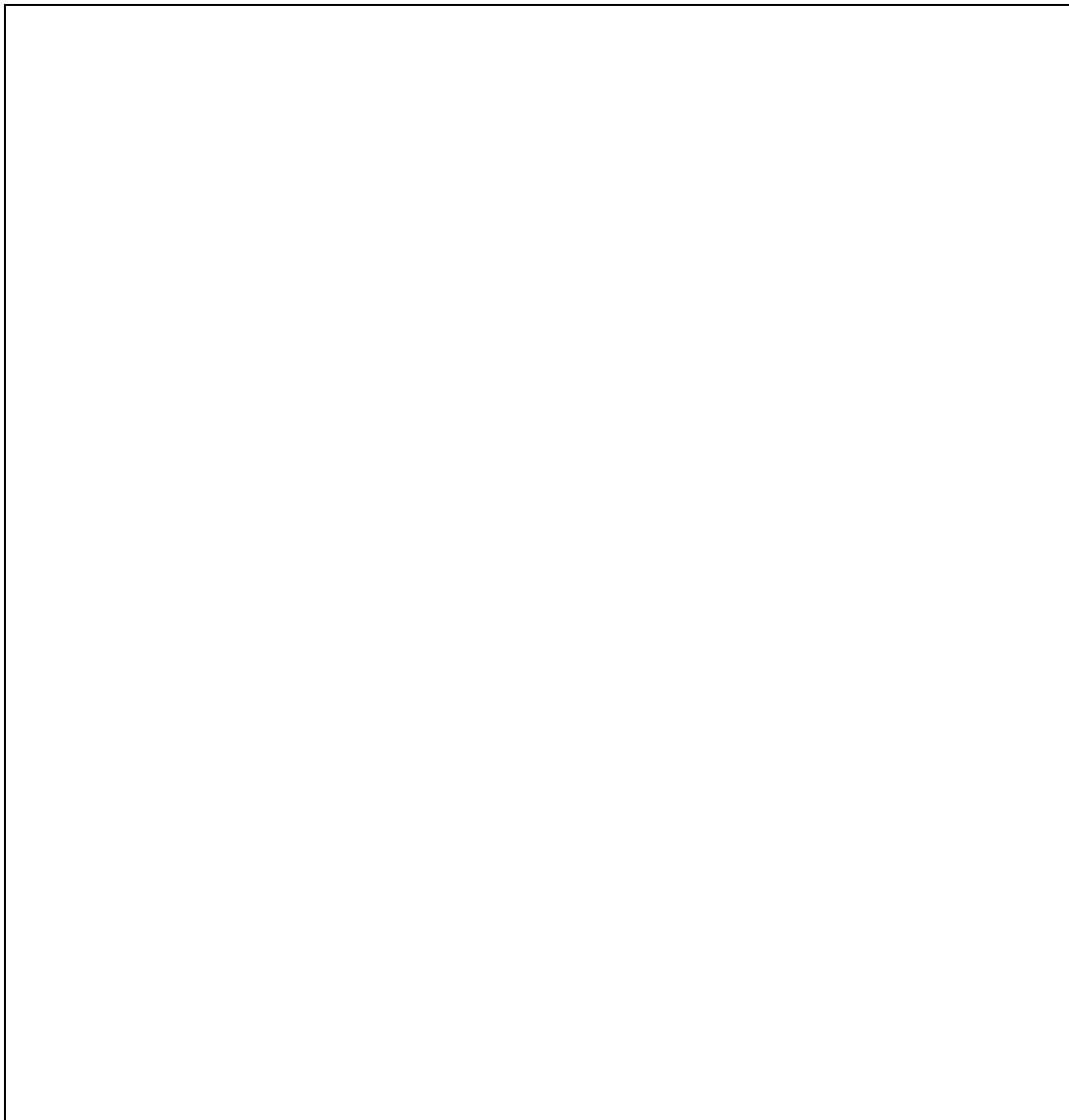
    public String toPostOrder() {
        return father.toPostOrder()+mother.toPostOrder()+
            getName()+"\n"+getSex()+"\n"+getFathersName()+
            "\n"+getMothersName()+"\n";
    }
}

public class EmptyPerson extends Person{
    public EmptyPerson() {super("");}
    public boolean isEmpty() {return true;}
    public String getName() {return "";}
    public Person getFather() {return empty;}
    public Person getMother() {return empty;}
    public String toString() {return "";}
    public int count() {return 0;}
    public int maxDepth() {return 0;}
    public boolean treeContains(Person person) {return false;}
    public boolean isRelatedTo(Person person) {return false;}
    public void paint(Graphics g, int x, int y, int dist) {return;}
    public String toPostOrder() {return "";}
}

```

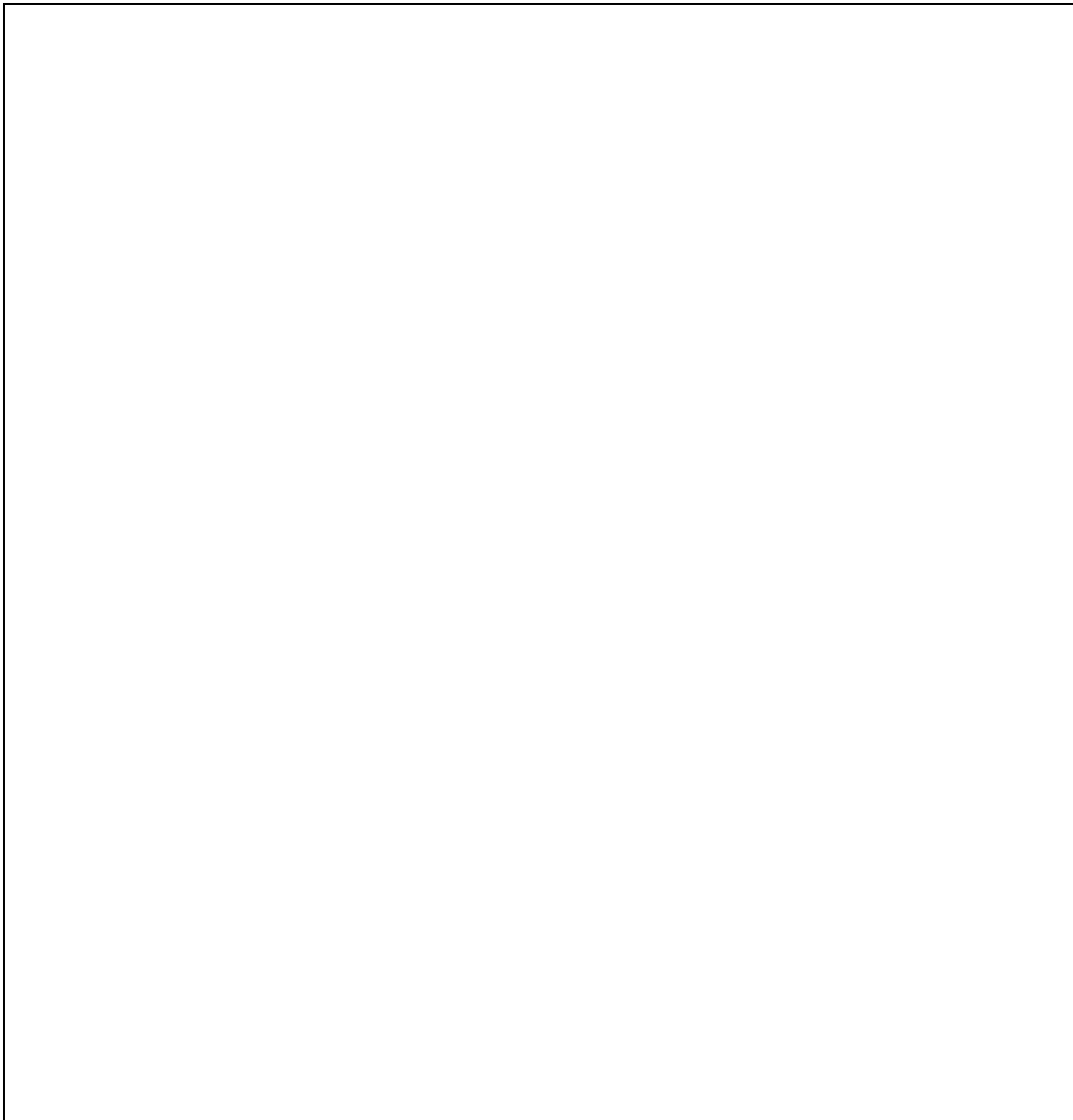
### 12.1.1 Zum Programm

Das Programm „Family - Version 3“ erlaubt die Erfassung eines Familienstammbaumes. Abbildung 12-1 und Abbildung 12-2 zeigen das Aussehen des Programms anhand der englischen Königsfamilie.



**Abbildung 12-1: Family, UserFrame**





**Abbildung 12-2: Family, DrawingFrame**

Es ist für einen Stammbaum charakteristisch, dass jede Person immer eine Mutter und einen Vater hat. Aufgrund dieser Eigenschaft werden im Programm die Personen und deren Beziehungen untereinander als binärer Baum abgebildet. Ein **binärer Baum** erlaubt die Strukturierung von Daten (Personen), indem er ein Datum (Kind) immer mit maximal zwei anderen Daten assoziiert (Mutter, Vater). Sein Name verdankt er seinem Aussehen.

Die Klasse `Person` verwendet für die Speicherung und Auffindung von erfassten Personen eine **Hashtabelle**. Auf die in einer Hashtabelle abgespeicherten Elemente (Personen) kann aufgrund eines Schlüsselwortes (String) zugegriffen werden. Die in der Klassenbibliothek deklarierte Klasse `Hashtable` weist sämtliche Funktionen einer Hashtabelle auf.

In der Klasse `Person` sind etliche rekursive Methoden deklariert. Die Methode `toString()` beispielsweise ist wegen der String Concatenation implizit rekursiv: der `'+'`-Operator ruft, falls es sich bei einem Operanden um keinen String handelt, automatisch dessen Methode `toString()` auf. Die Methoden `count()`, `maxDepth()`, `treeContains()` und `isRelatedTo()` sind hingegen alle explizit rekursiv.

Die Klasse `Empty` realisiert das Konzept eines **sentinel** (dt. Aufpasser): für die imaginäre Urmutter des Stammbaumes setzt man das Objekt `empty`. Hierbei redefiniert die Klasse `Empty` als Unterklasse von `Person` sämtliche Methoden und passt sie speziell an die Gegebenheiten der obersten Hierarchiestufe an, wodurch man sich Fallunterscheidungen in den Methoden der Klasse `Person` ersparen kann.

# A

## nhänge

### 13.1 Anhang A: Java Syntax

#### 13.1.1 Schlüsselwörter

Java kennt die folgenden **Schlüsselwörter** (engl. **keywords**). Sie sind reserviert und dürfen nicht für die Vergabe von Identifiern verwendet werden. Momentan werden die Schlüsselwörter `const` und `goto` nicht eingesetzt.

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

### 13.1.2 Einfache Datentypen

Java definiert die folgenden **einfachen Datentypen** (engl. **primitive data types**). Hierbei lassen sich sämtliche einfachen Datentypen in drei Gruppen unterteilen: **boolean**, Integer-Datentypen und Floating-Point-Datentypen. Zu der Gruppe der **Integer-Datentypen** zählt man die Typen `char`, `byte`, `short` und `long`, wobei `char` den Unicode repräsentiert. Die Typen `float` und `double` werden unter der Bezeichnung **Floating-Point-Datentypen** zusammengefasst.

<i>Typ</i>	<i>Wertebereich</i>	<i>Default</i>	<i>Grösse</i>
<code>boolean</code>	<b>true</b> und <b>false</b>	<b>false</b>	1 bit
<code>char</code>	von <code>\u0000</code> bis <code>\uFFFF</code>	<code>\u0000</code>	16 bit
<code>byte</code>	von -128 bis 127	0	8 bit
<code>short</code>	von -32768 bis 32767	0	16 bit
<code>int</code>	von -2147483648 bis 2147483647	0	32 bit
<code>long</code>	von -9223372036854775808 bis 9223372036854775807	0	64 bit
<code>float</code>	von $\pm 3.40282347E+38$ bis $\pm 1.40239846E-45$	0.0	32 bit
<code>double</code>	von $\pm 1.79769313486231570E+308$ bis $\pm 4.94065645841246544E-324$	0.0	64 bit

### 13.1.3 Operatoren

Die folgende Tabelle zeigt eine Übersicht der im Skript behandelten **Operatoren**. Prior. gibt die relative Priorität des Operators an, hält also fest, welche Operation vor einer anderen auszuführen ist. Haben zwei Operationen die gleiche Priorität, dann bestimmt deren Assoziativität (Assoz.), ob der Ausdruck von links nach rechts (L) oder von rechts nach links (R) zu evaluieren ist.

<i>Prior.</i>	<i>Operator</i>	<i>Beschreibung</i>	<i>Assoz.</i>
1	++	unäres Postfix- oder Präfix-Inkrement	R
	--	unäres Postfix- oder Präfix-Dekrement	R
	+, -	unäres Plus, unäres Minus (Vorzeichen)	R
	!	Negation	R
	(type)	Type Cast	R
2	*, /, %	Multiplikation, Division, Modulo	L
3	+, -	Addition, Subtraktion	L
	+	String Concatenation	L
4	<	kleiner als	L
	<=	kleiner gleich	L
	>	grösser als	L
	>=	grösser gleich	L
	instanceof	instanceof-Operator	L
5	==	gleich (identisch)	L
	!=	ungleich (nicht-identisch)	L
6	&	Konjunktion	L
7	^	exklusives Oder (XOR)	L
8		Disjunktion	L
9	&&	Konjunktion	L
10		Disjunktion	L
11	?:	Conditional Operator	R
12	=	Assignment	R
	op=	Compound Assignment Operator	R

## 13.2 Anhang B: Übersicht Klassenbibliothek

Nachfolgende Tabelle zeigt eine alphabetische Übersicht über die **Klassenbibliothek** von Java. Es werden jedoch nur Elemente aufgeführt, welche in den Beispielprogrammen verwendet wurden. Hierbei erläutert die untenstehende Legende, wie die Beispielprogramm in der Übersichtstabelle referenziert werden, und auf welchen Seiten sie sich im Skript befinden.

Nr.	Programm	Seite
1	Business Cards - Version 1	22
2	Business Cards - Version 2	37
3	Business Cards - Version 3	40
4	Business Cards - Version 4	51
5	Multiple Choice	58
6	Unicode	105
7	Game	125
8	Random Sentences - Version 1	147
9	Random Sentences - Version 3	156
10	Time - Version 1	164
11	Time - Version 2	171
12	Time - Version 3	178
13	Time - Version 4	186
14	Calculator	199
15	Word Guess	213
16	Puzzle	223
17	Turtle - Version 1	228
18	Turtle - Version 2	231
19	Turtle - Version 3	233
20	Turtle - Version 4	238

Index	Signatur	Zugehörigkeit	Programm
abs(int a)	public static int	Class java.lang.Math	5, 8, 9, 16
actionPerformed(ActionEvent e)	public abstract void	Interface java.awt.event.ActionListener	1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20
add(Component comp)	public Component	Class java.awt.Container	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20
addActionListener(ActionListener l)	public synchronized void	Class java.awt.Button	1, 2, 3, 4, 5, 7, 8, 9, 14, 15, 16, 20
addActionListener(ActionListener l)	public synchronized void	Class java.awt.TextField	8, 9, 15
addActionListener(ActionListener l)	public synchronized void	Class java.awt.MenuItem	9
addItem(String item)	public void	Class java.awt.List	8, 9
addItem(String item)	public synchronized void	Class java.awt.Choice (E)	14
addItemListener(ItemListener l)	public synchronized void	Class java.awt.List	9
addItemListener(ItemListener l)	public synchronized void	Class java.awt.Choice	14
addKeyListener(KeyListener l)	public synchronized void	Class java.awt.Component	15
append(String str)	public synchronized void	Class java.awt.TextArea	6
black	public static final Color	Class java.awt.Color	16
BOLD	public static final int	Class java.awt.Font	3, 4
Button()	public	Class java.awt.Button	7
Button(String label)	public	Class java.awt.Button	1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20
charAt(int index)	public char	Class java.lang.String	14, 15
Checkbox(String label)	public	Class java.awt.Checkbox	5
Checkbox(String label, boolean state, CheckboxGroup group)	public	Class java.awt.Checkbox	2, 3, 4, 20
CheckboxGroup()	public	Class java.awt.CheckboxGroup	2, 3, 4, 20
Choice()	public	Class java.awt.Choice	14
cos(double a)	public static native double	Class java.lang.Math	11, 12, 13, 17, 18, 19, 20
digit(char ch, int radix)	public static int	Class java.lang.Character	14
drawLine(int x1, int y1, int x2, int y2)	public abstract void	Class java.awt.Graphics	11, 12, 13, 17, 18, 19, 20
drawOval(int x, int y, int width, int height)	public abstract void	Class java.awt.Graphics	11, 12, 13
drawString(String str, int x, int y)	public abstract void	Class java.awt.Graphics	3, 4
elementAt(int index)	public final synchronized Object	Class java.util.Vector	20
empty()	public boolean	Class java.util.Stack	20
equals(Object anObject)	public boolean	Class java.lang.String	4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20

Font(String name, int style, int size)	public	Class java.awt.Font	3, 4, 6, 15
forDigit(int digit, int radix)	public static char	Class java.lang.Character	14
get(int field)	public final int	Class java.util.Calendar	10, 11, 12, 13
getActionCommand()	public String	Class java.awt.event.ActionEvent	4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20
getClipBounds()	public abstract Rectangle	Class java.awt.Graphics	17, 18, 19, 20
getInstance()	public static synchronized Calendar	Class java.util.Calendar	10, 11, 12, 13
getItem(int index)	public String	Class java.awt.List	8
getItemCount()	public int	Class java.awt.List	8, 9
getKeyChar()	public char	Class java.awt.event.KeyEvent	15
getLabel()	public String	Class java.awt.Checkbox	2, 3, 4, 20
getLabel()	public String	Class java.awt.Button	16
getSelectedCheckbox()	public Checkbox	Class java.awt.CheckboxGroup	2, 3, 4, 20
getSelectedIndex()	public int	Class java.awt.Choice	14
getSelectedItem()	public synchronized String	Class java.awt.List	9
getSource()	public Object	Class java.util.EventObject	8, 9, 15
getState()	public boolean	Class java.awt.Checkbox	5
getText()	public synchronized String	Class java.awt.TextComponent	1, 2, 3, 4, 7, 8, 9, 13, 14, 15
height	public int	Class java.awt.Rectangle	17, 18, 19, 20
HOURL_OF_DAY	public static final int	Class java.util.Calendar	10, 11, 12, 13
itemStateChanged(ItemEvent e)	public abstract void	Interface java.awt.event.ItemListener	9, 14
keyTyped(KeyEvent e)	public void	Class java.awt.event.KeyAdapter	15
Label()	public	Class java.awt.Label	5, 7, 8, 9, 12, 13
Label(String text)	public	Class java.awt.Label	1, 2, 3, 4, 5, 10, 11, 12, 13, 14, 15
length()	public int	Class java.lang.String	7, 14, 15
lightGray	public static final Color	Class java.awt.Color	16
List()	public	Class java.awt.List	8, 9
Menu add(Menu m)	public	Class java.awt.MenuBar	9
Menu(String label)	public	Class java.awt.Menu	9
MenuBar()	public	Class java.awt.MenuBar	9
MenuItem add(MenuItem mi)	public	Class java.awt.Menu	9
MenuItem(String label)	public	Class java.awt.MenuItem	9
MINUTE	public static final int	Class java.util.Calendar	10, 11, 12, 13
nextInt()	public int	Class java.util.Random	5, 8, 9, 16
nextToken()	public String (E)	Class java.util.StringTokenizer	13
NumberFormatException()	public	Class java.lang.NumberFormatException	13
out	public static final PrintStream	Class java.lang.System	1, 2, 3, 4
paint(Graphics g)	public void	Class java.awt.Container	3, 4, 11, 12, 13, 17, 18, 19, 20
parseInt(String s)	public static int (E)	Class java.lang.Integer	13



PI	public static final double	Class java.lang.Math	11, 12, 13, 17, 18, 19, 20
PLAIN	public static final int	Class java.awt.Font	3, 4, 15
Point(int x, int y)	public	Class java.awt.Point	17, 18, 19, 20
pop()	public synchronized Object	Class java.util.Stack	20
println(String x)	public void	Class java.io.PrintStream	1, 2, 3, 4
processEvent(AWTEvent e)	protected void	Class java.awt.Window	1, 2, 3, 4, 5
push(Object item)	public Object	Class java.util.Stack	20
Random()	public	Class java.util.Random	5, 8, 9, 16
removeKeyListener(KeyListener l)	public synchronized void	Class java.awt.Component	15
repaint()	public void	Class java.awt.Component	11, 12, 13, 20
requestFocus()	public void	Class java.awt.Component	7, 8, 9, 10, 11, 12, 13, 15
round(double a)	public static long	Class java.lang.Math	11, 12, 13, 17, 18, 19, 20
select(int index)	public void	Class java.awt.List	9
select(int pos)	public synchronized void (E)	Class java.awt.Choice	14
setBackground(Color c)	public void	Class java.awt.Component	6, 14, 16, 20
setBounds(int x, int y, int width, int height)	public void	Class java.awt.Component	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20
setEchoChar(char c)	public synchronized void	Class java.awt.TextField	7, 15
setEditable(boolean b)	public synchronized void	Class java.awt.TextComponent	7, 14, 15
setFont(Font f)	public synchronized void	Class java.awt.Component	6, 15
setFont(Font font)	public abstract void	Class java.awt.Graphics	3, 4
setLabel(String label)	public synchronized void	Class java.awt.Button	7, 16
setLayout(LayoutManager mgr)	public void	Class java.awt.Container	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20
setLocation(int x, int y)	public void	Class java.awt.Point	17, 18, 19, 20
setMenuBar(MenuBar mb)	public void	Class java.awt.Frame	9
setResizable(boolean resizable)	public synchronized void	Class java.awt.Frame	1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 20
setSize(int width, int height)	public void	Class java.awt.Component	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
setState(boolean state)	public void	Class java.awt.Checkbox	5
setText(String t)	public synchronized void	Class java.awt.TextComponent	7, 8, 9, 10, 11, 12, 13, 14, 15, 20
setText(String text)	public synchronized void	Class java.awt.Label	5, 7, 8, 9, 12, 13, 15

setTitle(String title)	public synchronized void	Class java.awt.Frame	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
setVisible(boolean b)	public void	Class java.awt.Component	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
sin(double a)	public static native double	Class java.lang.Math	11, 12, 13, 17, 18, 19, 20
size()	public final int	Class java.util.Vector	20
Stack()	public	Class java.util.Stack	20
StringTokenizer(String str, String delim)	public	Class java.util.StringTokenizer	13
substring(int beginIndex)	public String (E)	Class java.lang.String	7, 15
substring(int beginIndex, int endIndex)	public String (E)	Class java.lang.String	15
System.out.println(String)	siehe out und println(String)		1, 2, 3, 4
TextArea()	public	Class java.awt.TextArea	6, 20
TextField()	public	Class java.awt.TextField	1, 2, 3, 4, 7, 8, 9, 10, 11, 12, 13, 15
toHexString(int i)	public static String	Class java.lang.Integer	6
toLowerCase(char ch)	public static char	Class java.lang.Character	15
toUpperCase(char ch)	public static char	Class java.lang.Character	14
valueOf(char c)	public static String	Class java.lang.String	6
valueOf(int i)	public static String	Class java.lang.String	5, 10, 11, 12, 13, 16, 20
white	public static final Color	Class java.awt.Color	6, 14, 20
width	public int	Class java.awt.Rectangle	17, 18, 19, 20

## 13.3 Anhang C: Klassendiagramme

Die folgende Graphik erklärt die Notationsform von **Klassendiagrammen**.

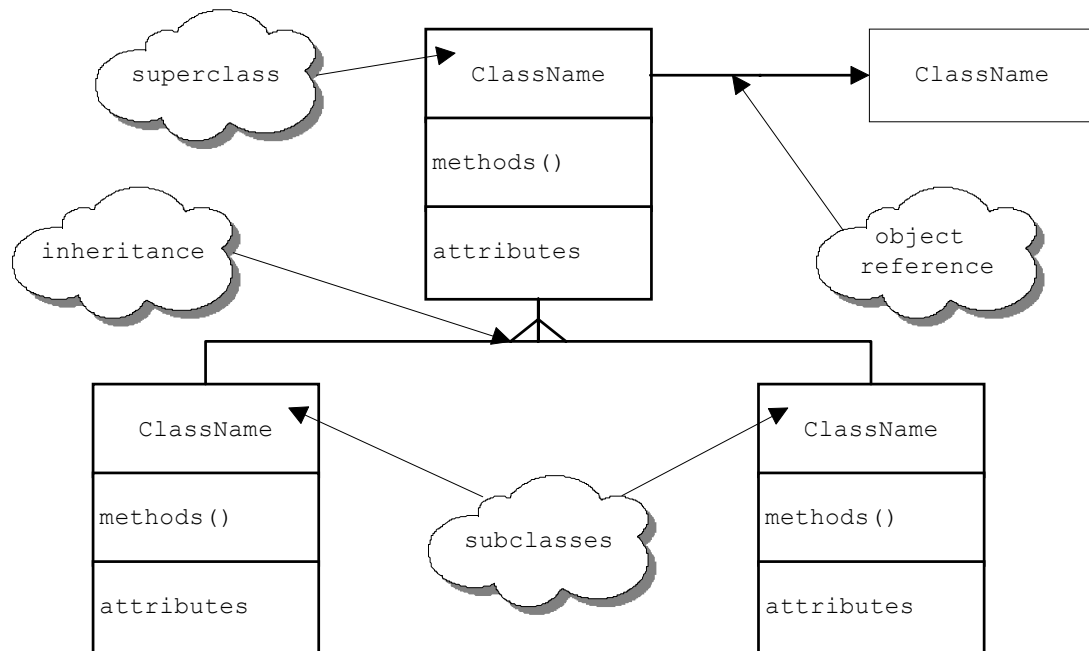


Abbildung 13-1: Notationsform von Klassendiagrammen

## 13.4 Anhang D: Prinzipien guten Programmierens

Die folgend aufgeführten Punkte listen allgemein akzeptierte Regeln auf, welche zur Verständlichkeit eines Programms beitragen. Die Reihenfolge ihrer Erwähnung entspricht keiner Rangordnung hinsichtlich ihrer Wichtigkeit:

- ◆ **Pretty Printing:** Die ineinanderverschachtelte Programmstruktur soll auch in der Darstellung des Programmes ersichtlich sein. Man kann beispielsweise mit jeder Verschachtelungstiefe die Programmzeilen entsprechend einrücken. Es empfiehlt sich auch, eine schliessende geschweifte Klammer mit derjenigen Zeile, welche die öffnende geschweifte Klammer enthält, bündig zu setzen (siehe Abbildung 2-1).
- ◆ **Gross-/Kleinschreibung:** Die Gross- bzw. Kleinschreibung soll ebenfalls als stilistisches Mittel eingesetzt werden. Im Skript werden die Identifier von Klassen und Interfaces im Gegensatz zu den Namen von Attributen und Methoden mit einem grossen Anfangsbuchstaben geschrieben. Wird jedoch ein Identifier aus mehreren Wörtern zusammengesetzt, werden für die Wörter innerhalb des Identifier grosse Anfangsbuchstaben verwendet.

- ◆ **Kommentar:** Die Verwendung von Kommentaren macht ein Programm verständlicher. Hierbei kann ein Kommentar an unterschiedlichen Stellen im Programm, für unterschiedliche Zwecke gesetzt werden.

Bei der Variablendeklaration legt ein Kommentar die Bedeutung der jeweiligen Variablen fest:

```
double width, height; //Breite und Höhe in Metern
```

Zugleich kann er auch deren Wertebereich angeben:

```
int n;          //Anzahl der Elemente, n≥0
int index;      //0≤index<n
```

Hilfreich ist auch ein Kommentar bei einer ganzen Anweisungsfolge, wenn er deren Bedeutung darlegt:

```
if (a[j-1]>a[j]) { //swap(a[j-1],a[j])
    int t = a[j-1];
    a[j-1] = a[j];
    a[j] = t;
}
```

Wenn immer möglich, sollte ein Kommentar eine Zusicherung (siehe Abschnitt 5.1.1) sein:

```
void selectionSort(int[] a) {
    int n = a.length;
    for (int i=0;i<n-1;i++) {
        //alle Elemente im Intervall a[0,i-1] sind nichtfallend sortiert
        //und alle Elemente a[0,i-1] ≤ alle Elementen a[i,n-1]
        int imin = i;
        int min = a[imin];
        for (int j=i+1;j<n;j++)
            if (a[j]<min) {
                imin=j;
                min = a[imin];
            }
        //min = a[imin] ist das kleinste Element von a[i,n-1]
        a[imin] = a[i];
        a[i] = min;
        //alle Elemente im Intervall a[0,i] sind nichtfallend sortiert
        //und alle Elemente a[0,i] ≤ alle Elementen a[i+1,n-1]
    }
    //alle Elemente im Intervall a[0,n-2] sind nichtfallend sortiert
    //und alle Elemente a[0,n-2] ≤ a[n-1], daher gilt
    //alle Elemente im Intervall a[0,n-1] sind nichtfallend sortiert
}
```

Bei Schleifen sollte man in Form eines Kommentars angeben, warum sie terminieren:

```
//binäres Suchen von x im steigend sortierten Array a:
while (i+1<j) {
    int m = (i+j)/2;
    if (a[m]<=x) i = m;
    else j = m;
} //Terminationsfunktion t:(j-i) wird solange halbiert bis t=1 ist
```

Obwohl der Einsatz von Kommentaren äusserst empfehlenswert ist, rät es sich aber davon ab, in die andere Richtung des Extrems zu fallen. Ein Programm kann auch überkommentiert werden:

```
i++; //i wird um eins erhöht
```

- ◆ **Mnemotechnische Identifier:** Als Identifier sollten sprechende Namen verwendet werden, die Sinn und Funktion der entsprechenden Klasse/Attribut/Methode ausdrücken. Um Programme international austauschen zu können, ist die Verwendung von englischen Identifiers empfehlenswert.

Die Verwendung mnemotechnischer Identifier macht einen entsprechenden Kommentar oft überflüssig:

```
area = width*height;
```

- ◆ **Konstanten:** Der Einsatz mnemotechnischer Konstanten kann die Verwendung von Kommentar erübrigen. Die folgenden Programmausschnitte illustrieren die Berechnung des Eintrittspreises in Abhängigkeit vom Alter eines Besuchers. Sie liegen in vier Versionen vor und werden schrittweise verbessert.

Die untenstehende erste Version muss zur Verständlichkeit sämtliche Anweisungen kommentieren. Zudem werden im Fall von Kleinkindern die Anweisungen der ersten beiden if statements ausgeführt:

```
public int eintritt(int alter, int preis) {
    int ergebnis = preis;           //voller Preis
    if (alter<=6) ergebnis=0;        //Kleinkinder sind gratis
    if (alter<16) ergebnis /= 2;     //Jugendliche zahlen halben Preis
    if (alter>=60) ergebnis /= 2;    //Senioren zahlen halben Preis
    return ergebnis;
}
```

Die folgenden zwei Version beheben nun den vorangehenden Makel, dass bei Kleinkindern die ersten beiden if statements ausgeführt werden:

```
public int eintritt(int alter, int preis) {
    if (alter<=6) return 0;           //Kleinkinder sind gratis
    if (alter<16) return preis/2;    //Jugendliche zahlen halben Preis
    if (alter>=60) return preis/2;   //Senioren zahlen halben Preis
    return preis;                    //sonst voller Preis
}

public int eintritt(int alter, int preis) {
    int ergebnis = preis;           //voller Preis
    if (alter<=6) ergebnis = 0;     //Kleinkinder sind gratis
    else if (alter<16) ergebnis = preis/2; //Jugendliche zahlen
                                         //halben Preis
    else if (alter>=60) ergebnis = preis/2; //Senioren zahlen
                                         //halben Preis
    return ergebnis;
}
```

Die letzte Version schlussendlich verwendet mnemotechnische Konstanten und kann ganz auf den Einsatz von Kommentaren verzichten:

```
static final int KIND = 0;
static final int JUGENDLICH = 1;
static final int ERWACHSEN = 2;
static final int SENIOR = 3;

int kategorie(int alter) {
    if (alter<=6) return KIND;
    if (alter>=60) return SENIOR;
    if (alter<16) return JUGENDLICH;
    return ERWACHSEN;
}

public int eintritt(int alter, int preis) {
    switch (kategorie(alter)) {
        case KIND: return 0;
        case JUGENDLICH: return preis/2;
        case ERWACHSEN: return preis;
        case SENIOR: return preis/2;
        default: return preis;
    }
}
```

Verwendet man zusätzlich einen Array zur Speicherung der Ermässigung, dann zeichnen sich die folgenden Änderungen ab:

```
static final int[] prozent = {0,50,100,50};

public int eintritt(int alter, int preis) {
    return preis*prozent[kategorie(alter)]/100;
}
```

- ◆ **Keine Tricks:** Hinsichtlich einer besseren Verständlichkeit und Wartbarkeit eines Programmes sollten Tricks vermieden werden.

Obwohl beide Programmausschnitte das Vertauschen zweier Zahlen bewirken, sind die linksstehenden Anweisungen im Unterschied zu den rechtsstehenden schwer verständlich:

```
y = x+y;           int t =x;
x = y-x;           x = y;
y = y-x;           y = t;
```

- ◆ **Keep it simple:** Man sollte nie allzulange, unüberschaubare Ausdrücke verwenden. Statt dessen bieten sich Hilfsvariablen an, die, falls sie mnemotechnisch geschickt gewählt sind, zudem Kommentar ersparen.

Die Anweisung

```
message.setText((firstNameField.getText()=="?"?:
    firstNameField.getText().charAt(0))+". "+
    familyNameField.getText());
```

wird durch den Einsatz von Hilfsvariablen verständlicher:

```
String firstName = firstNameField.getText();
String familyName = familyNameField.getText();
String name;
if (firstName=="")
    name = familyName;
else {
    char initial = firstName.charAt(0);
    name = initial+". "+familyName;
}
message.setText(name);
```

- ◆ **Schleifen:** Die verschiedenen Arten von Schleifen (for, while, do) sollten aufgrund ihrer unterschiedlichen Eigenschaften gezielt eingesetzt werden.

Die untenstehende Schleife würde beispielsweise besser als while Schleife formuliert werden, da for Schleifen für den Einsatz eines Zählers prädestiniert sind. Zudem ist die untenstehende for Schleife endlos:

```
for (int m=(i+j)/2; i+1!=j; a[m]<=x?i=m:j=m);
```

- ◆ **Boolesche Ausdrücke:** Boolesche Ausdrücke sollten prinzipiell direkt verwendet werden und nicht auf `true` bzw. `false` überprüft werden.

Der rechtsstehende Ausdruck setzt im Vergleich zum linksstehenden Ausdruck direkt den Booleschen Wert ein:

```
boolean ordered = ...;           boolean ordered = ...;
while (ordered==false) {...}      while (!ordered) {...}
```

Die folgenden drei Programmausschnitte illustrieren den Vergleich zweier Uhrzeiten. Hierbei ist die erste Version zwar korrekt, aber aufwendig:

```
boolean before(Time x, Time y) {
    if (x.hours<y.hours)
        return true;
    else if ((x.hours==y.hours)&(x.minutes<y.minutes))
        return true;
    else return false;
}
```

Auch die folgende Version ist wohl nur für einen Profi der Booleschen Algebra verständlich und deshalb nicht empfehlenswert:

```
boolean before(Time x, Time y) {
    return (x.hours<y.hours)||((x.hours==y.hours)&
        (x.minutes<y.minutes))
}
```

Die letzte Version schlussendlich ist im Gegensatz zur vorhergehenden leicht verständlich und im Unterschied zur ersten Version effizienter:

```
boolean before(Time x, Time y) {
    if (x.hours!=y.hours)
        return x.hours<y.hours;
    else
        return x.minutes<y.minutes;
}
```

- ◆ **Seiteneffekte:** Typed methods sollten keine Seiteneffekte zur Folge haben sondern primär einen Wert zurückliefern.

Als typed method liefert beispielsweise die in der Klasse `Stack` deklarierte Methode `pop()` einen Wert zurück. Zusätzlich verändert sie aber noch den Stack, was für den Verwender der Methode aufgrund der Methodenschnittstelle nicht ersichtlich ist. Ob der Text - „Nuss“ oder „Baum“ - zuerst gedruckt wird, hängt davon ab, in welcher Reihenfolge der Ausdruck abgearbeitet wird:

```
Stack stack = new Stack();
stack.push("Nuss");
stack.push("Baum");
System.out.println(stack.pop()+"-"+stack.pop());
```

- ◆ **Datenstrukturen:** Eine grosszügig angelegte Datenstruktur kann das Programm oft vereinfachen.

Die folgenden drei Programmausschnitte illustrieren die Zuordnung eines Monatsnamens zu seiner entsprechenden Nummer:



```
static final String[] name = {"Jaenner", "Februar", "Maerz", "April",  
                             "Mai", "Juni", "Juli", "August",  
                             "September", "Oktober", "November",  
                             "Dezember"};  
  
//Umwandlung des Monatsnamens in die entsprechende Nummer:  
static int numberOfMonth(String string) {  
    for (int i=1; i<=12; i++)  
        if (string.equals(name[i-1]))  
            return i; //voller Monatsname gefunden  
    return 0;        //kein gueltiger Monatsname  
}
```

Dadurch, dass man nun an erster Position im Array den Leerstring einfügt, braucht man den Index *i* im if statement der for Schleife nicht mehr um eins zu verringern:

```
static final String[] name = {"", "Jaenner", "Februar", "Maerz", "April",  
                             "Mai", "Juni", "Juli", "August",  
                             "September", "Oktober", "November",  
                             "Dezember"};  
  
//Umwandlung des Monatsnamens in die entsprechende Nummer:  
static int numberOfMonth(String string) {  
    for (int i=1; i<=12; i++)  
        if (string.equals(name[i]))  
            return i; //voller Monatsname gefunden  
    return 0;        //kein gueltiger Monatsname  
}
```

Die letzte Version ist zwar sehr effizient aber auch tückisch:

```
static String[] name = {"", "Jaenner", "Februar", "Maerz", "April",  
                      "Mai", "Juni", "Juli", "August",  
                      "September", "Oktober", "November",  
                      "Dezember"};  
  
//Umwandlung des Monatsnamens in die entsprechende Nummer:  
static int numberOfMonth(String string) {  
    name[0] = string;  
    int i=12;  
    while (!string.equals(name[i])) i--;  
    return i; //Funktionswert Null bedeutet ungeltiger Monatsname  
}
```

- ◆ **Lokalität:** Im Sinne des Information Hiding soll man wenn immer möglich, Variablen lokal einsetzen und deklarieren. Es macht beispielsweise keinen Sinn, eine Schleifenvariable als Instanzvariable zu deklarieren.
- ◆ **Erweiterbarkeit:** Grundsätzlich soll so programmiert werden, dass einzelne Programmteile leicht erweitert und für andere Programme wiederverwendet werden können.
- ◆ **Dokumentation:** Gerade grössere Softwareprojekte sollten unbedingt dokumentiert werden. Eine solche Dokumentation enthält sämtliche Dokumente der Programm-Implementierung von den eigentlichen Anforderungen des Kunden bis hin zu den

letzten Testergebnissen. Sie ist von der Benutzerdokumentation, die dem Kunden als „Bedienungsanleitung“ zum Produkt abgegeben wird, zu unterscheiden.

## 13.5 Anhang E: Übersicht Programmiersprachen

Die nachfolgende Graphik zeigt in Form eines Stammbaumes die Entwicklungsgeschichte der Programmiersprachen.

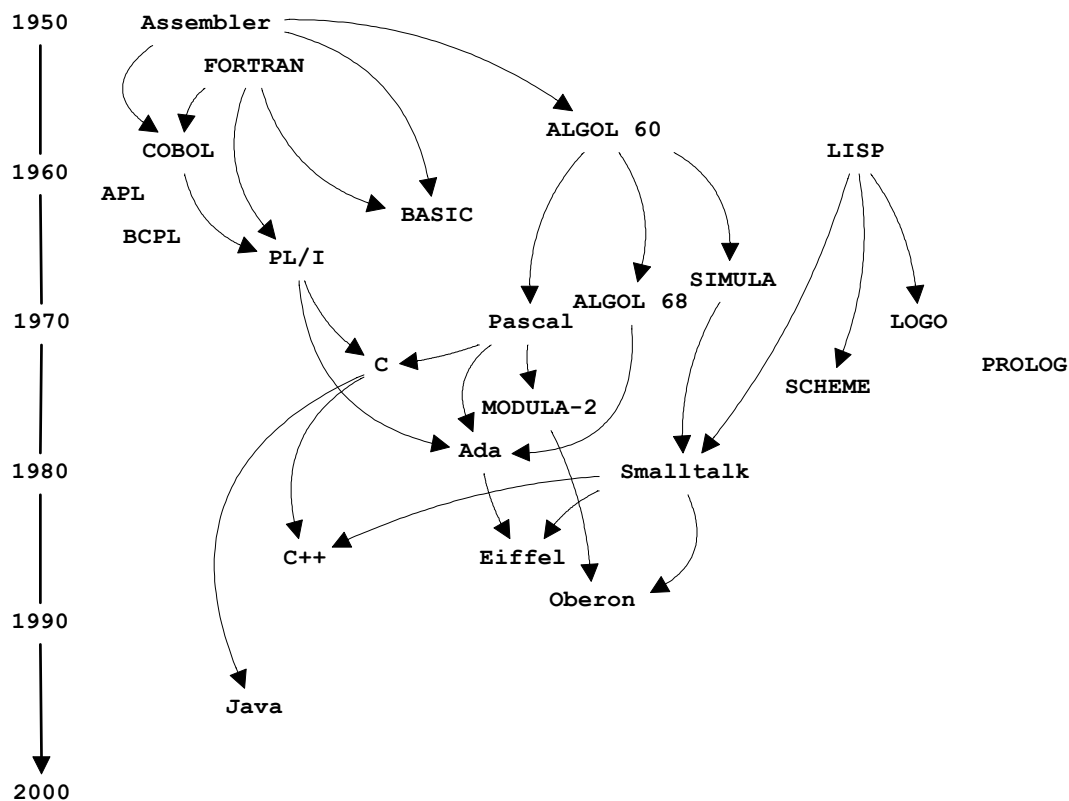


Abbildung 13-2: Stammbaum der Programmiersprachen

# S

## tichwortverzeichnis

Da die aus der Klassenbibliothek verwendeten Elemente in Anhang C aufgeführt werden, sind sie im folgenden Stichwortverzeichnis nicht enthalten.

### 14.1 Stichwortverzeichnis

,	awt, Paket.....	12
‘!=’-Operator.....	64	
‘+’-Operator.....	27; 113	
‘+=’-Operator.....	164	
‘=’-Operator.....	42	
‘==’-Operator.....	65; 116	
‘?:’-Operator.....	165	
<b>A</b>		
abstract, Schlüsselwort.....	204; 206; 208	
abstrakte Klasse.....	206; 208	
abstrakte Methode.....	135; 204; 208	
Abstraktion.....	9	
Adapter-Klasse.....	215; 218	
Addition.....	62	
Anweisung.....	7; 17; 29	
applet.....	14	
Application.....	14; 30	
arithmetische Operatoren.....	86	
arithmetischen Operatoren.....	62	
Array.....	178; 180	
Array, anonym.....	224	
Array, mehrdimensional.....	223	
ASCII-Code.....	105	
assertion.....	<i>Siehe</i> Zusicherung	
assignment.....	<i>Siehe</i> Zuweisung	
Attribut.....	4; 22	
attribute.....	<i>Siehe</i> Attribut	
Attributzugriff.....	152	
<b>B</b>		
bedingte Anweisung.....	8; 100	
bedingter Ausdruck.....	71	
Betriebssystem.....	12	
Betriebssystemunabhängigkeit.....	12	
Binärcode.....	12	
binäre Sprache.....	12	
binärer Baum.....	247	
boolean, Datentyp.....	63; 87; 249	
Botschaft.....	5	
break, Schlüsselwort.....	77; 101	
Button.....	21	
byte, Datentyp.....	86; 108; 249	
Bytecode.....	13	
<b>C</b>		
C++, Programmiersprache.....	7	
C, Programmiersprache.....	7	
case, Schlüsselwort.....	77; 101	
catch statement.....	188; 192	
char, Datentyp.....	105; 108; 249	
Checkbox.....	33	
CheckboxGroup.....	33	
Choice.....	202	
class.....	<i>Siehe</i> Klasse	
class, Schlüsselwort.....	18; 30	
Compiler.....	12	
Component.....	21	

compound assignment operator ..... 164; 166  
 Concatenation ..... 113  
 Concatenation, String ..... 27  
 conditional operator ..... 165; 167  
 conditional statement .. *Siehe* bedingte Anweisung  
 console ..... *Siehe* Konsole  
 constructor ..... *Siehe* Konstruktor  
 Container ..... 21

### D

Datentyp ..... 61  
 decrement ..... *Siehe* Dekrement  
 default, Schlüsselwort ..... 77; 101  
 Dekrement ..... 74  
 do Schleife ..... 97  
 doppelte Anführungszeichen ..... 112  
 double, Datentyp ..... 108; 249

### E

eckige Klammern ..... 179; 181  
 einfache Anführungszeichen ..... 106; 109  
 einfache Datentypen ..... 61; 114; 249  
 else statement ..... 49; 72; 100  
 escape sequence ..... 106; 107; 109  
 Event ..... 134  
 Event Handling ..... 26; 133; 139  
 Exception ..... 185  
 Exception Handling ..... 185; 192  
 Exception-Klasse ..... 185  
 exklusives Oder ..... 75  
 extends, Schlüsselwort ..... 37; 45

### F

false ..... 64  
 final, Schlüsselwort ..... 105; 108  
 finally statement ..... 193  
 float, Datentyp ..... 108; 249  
 Floating-Point-Datentypen ..... 108; 249  
 for Schleife ..... 66; 98  
 for Schleife, ineinandergeschachtelt ..... 223  
 Frame ..... 21

### G

Garbage Collection ..... 115  
 geschweifte Klammern ..... 18; 22; 29  
 graphical user interface ..... 10  
 graphische Benutzeroberfläche ..... 10  
 GUI ..... *Siehe* graphical user interface  
 Gültigkeitsbereich einer Variablen ..... 81

### H

Hashtabelle ..... 247  
 hierarchisches System ..... 3

### I

identifizier ..... 22; 28  
 Identität ..... 65  
 if statement ..... 48; 71; 100  
 Information Hiding ..... 164  
 Implementierung ..... 135  
 implements, Schlüsselwort ..... 138; 214; 217  
 import, statement ..... 18; 29  
 increment ..... *Siehe* Inkrement  
 Information Hiding ..... 157  
 inheritance ..... *Siehe* Vererbung  
 Inkrement ..... 74  
 instance ..... *Siehe* Instanz  
 instanceof-Operator ..... 147; 150  
 Instanz ..... 4  
 Instanzmethode ..... 147; 150; 151  
 Instanzvariable ..... 49; 81  
 int, Datentyp ..... 61; 85; 108  
 Integer-Datentypen ..... 108; 249  
 Interaktion ..... 10  
 Interface ..... 135; 213; 217  
 interface, Schlüsselwort ..... 213; 217  
 Internet ..... 14  
 Interpreter ..... 13  
 Iteration ..... 9; 65; 95

### J

Java ..... 2; 7; 11

### K

keyword ..... *Siehe* Schlüsselwort  
 Klasse ..... 3; 4; 30  
 Klasse, innere ..... 241  
 Klassenbibliothek ..... 11; 253  
 Klassendeklaration ..... 30  
 Klassendiagramm ..... 37; 255  
 Klassenmethode ..... 147; 150; 151  
 Klassenvariable ..... 49; 81  
 Klassifikation ..... 2  
 Komma ..... 22  
 Kommentar ..... 60; 80  
 komplexe Datentypen ..... 61; 114  
 Konsole ..... 20  
 Konstante ..... 105; 108  
 Konstruktor ..... 41; 46  
 Kopf, Methodendeklaration ..... 22; 83

### L

Label ..... 21  
 lang, Paket ..... 30  
 List ..... 146  
 Listener ..... 134  
 logische Operatoren ..... 87

Logo, Programmiersprache ..... 7; 225  
lokale Variable ..... 67; 82  
long, Datentyp ..... 86; 108; 249  
loop ..... *Siehe* Schleife

**M**

main() Methode ..... 27  
Maschinenprogramm ..... 12  
Maschinensprache ..... 12  
math, Paket ..... 12  
Menu ..... 156  
MenuBar ..... 155  
MenuItem ..... 156  
message ..... *Siehe* Botschaft  
method overloading ..... 165  
method overriding ..... 44  
Methoden ..... 9  
Methodenaufruf ..... 6; 9; 24; 85; 151  
Methodendeklaration ..... 22; 24; 83  
Methodschnittstelle, Methodendeklaration ..... 22  
Modulo-Operator ..... 62

**N**

new, Schlüsselwort ..... 42; 46  
Nicht-Identität ..... 64  
null-Wert ..... 115

**O**

Oberklasse ..... 3; 6  
object ..... *Siehe* Objekt  
Object, Klasse ..... 38  
Objekt ..... 4  
objektorientierte Programmiersprache ..... 2; 11  
objektorientiertes Programm ..... 7  
Objektorientierung ..... 2  
Operatoren ..... 251

**P**

package ..... *Siehe* Paket  
Paket ..... 12  
Parameter ..... 22; 84  
Pascal, Programmiersprache ..... 7  
pass by reference ..... 116  
pass by value ..... 115  
pixel ..... *Siehe* Rasterpunkte  
Portabilität ..... 12  
postfix expression ..... *Siehe* Postfix-Notation  
Postfix-Notation ..... 74  
Präfix-Notation ..... 74  
prefix expression ..... *Siehe* Präfix-Notation  
primitive data types ..... *Siehe* einfache Datentypen  
private, Schlüsselwort ..... 157  
Programm ..... 7

Programmieren ..... 255  
Programmierersprache ..... 7  
Programmierersprachen ..... 262  
public, Schlüsselwort ..... 46; 157

**Q**

Quellprogramm ..... 12

**R**

radio buttons ..... 33  
Rasterpunkte ..... 23  
Redefinition ..... 44  
reference data types... *Siehe* komplexe Datentypen  
Referenzsemantik ..... 114  
Rekursion ..... 233  
return, Schlüsselwort ..... 63; 84  
Rumpf, Methodendeklaration ..... 22; 83  
runde Klammern ..... 22

**S**

Schleife ..... 65  
Schleifen ..... 95  
Schlüsselwort ..... 29  
Schlüsselwörter ..... 248  
Selektion ..... 8; 48; 71; 100  
Semikolon ..... 29  
sentinel ..... 248  
Sequenz ..... 7  
short, Datentyp ..... 86; 108; 249  
Signatur, Methodendeklaration ..... 22; 83  
Stack ..... 241  
statement ..... *Siehe* Anweisung  
static, Schlüsselwort ..... 50; 82; 105; 108; 147; 150  
String ..... 27; 69; 111  
String Tokenizer ..... 184  
subclass ..... *Siehe* Unterklasse  
superclass ..... *Siehe* Oberklasse  
switch statement ..... 77; 101  
switch, Schlüsselwort ..... 77

**T**

TextArea ..... 103  
TextField ..... 21  
this, Schlüsselwort ..... 42; 151  
throw, Schlüsselwort ..... 188; 192  
throws, Schlüsselwort ..... 188; 192  
true ..... 64  
try statement ..... 189; 192  
Turtle-Geometrie ..... 225  
type cast ..... *Siehe* Typenkonvertierung  
typed method ..... 84  
Typenkonvertierung ..... 117; 120

### **U**

Unicode .....	105
Unterklasse .....	3; 6
util, Paket .....	59

### **V**

Variable .....	50
Variablendeklaration .....	80
Vererbung .....	6; 37; 45
einfache Vererbung .....	37
Mehrfachvererbung .....	37
Vergleichsoperatoren .....	87
virtuellen Maschine .....	13

visibility modifiers .....	157; 158
void method .....	84
void, Schlüsselwort .....	63; 84

### **W**

Wertsemantik .....	114
while Schleife .....	96

### **Z**

Zusicherung .....	131
Zustand .....	4
Zustandsdiagramm .....	136
Zuweisung .....	42; 82; 83