Inhaltsverzeichnis

STICHWORTVERZEICHNIS....VI

1 OBJEKTE UND KLASSEN....1

1.1 Die Begriffe Objekt und Klasse....1

1.2 Methoden....1

1.3 Datentypen....1

1.4 Der Zustand von Objekten....2

1.5 Java-Quelltext....2

2 KLASSENDEFINITIONEN....3

2.1 Grobaufbau von Klassen....3
2.1.1 Datenfelder....3

2.2 Kommentare....4

2.3 Konstruktoren sind Methoden....4

2.4 Zuweisungen....5

2.5 Sondierende Methoden....5

2.6 Bildschirmausgaben in Java....6

3 LOKALE VARIABLEN, DATENFELDER UND PARAMETER....7

3.1 Eigenschaften der Variablentypen:....7
3.1.1 Datenfelder....7
3.1.2 Formale Parameter....7
3.1.3 Lokale Variablen....7

4 BEDINGTE ANWEISUNGEN....8

4.1 Logische Operatoren....8

4.2 Allgemeine Form der Alternative (If-Anweisung zweiseitig)....8

4.3 Allgemeine Form der Alternative (switch-Anweisung zweiseitig)....9

4.4 Die while-Schleife....9

4.5 Die do-Schleife....10

4.6 Die for-Schleife....10

5 OBJEKTINTERAKTION....11

5.1 Abstraktion und Modularisierung....................11

5.2 Klassendiagramme und Objektdiagramme....................12

5.3 Objekttypen
....................12

5.4 Arithmetische Operatoren....................13

5.5 new-Anweisung....................14

5.6 Modifierer....................14
5.6.1 Public und Private Eigenschaften....................14
5.6.2 Der Zugriffs-Modifier protected....................15
5.6.3 Modifier Tabelle....................15
5.6.4 Die unterschiedlichen Ebenen des Zugriffsschutzes....................16

5.7 Konstruktoren überladen....................16

5.8 Methodenaufrufe....................16
5.8.1 Interne Methodenaufrufe....................16
5.8.2 Externe Methodenaufrufe....................16

5.9 Das Schlüsselwort this....................17

5.10 Programmtest mit dem Debugger....................17

6 KLASSENBIBLIOTHEK 18

6.1 Dokumentation von Klassen....................18

6.2 Laden einer Klasse aus der Klassenbibliothek....................20

6.3 Wichtige Klassen....................21
6.3.1 String....................21
6.3.2 StringTokenizer....................25
6.3.3 Random....................26
6.3.4 Wrapper-Klassen....................27
6.3.5 Math....................28

6.4 Wichtige Interface....................30
6.4.1 Comparable....................30

7 OBJEKTSAMMLUNGEN....................32

7.1 Objektstrukturen mit Sammlungen....................32

7.2 Sammlungen mit fester Größe (Arrays)....................32
7.2.1 Array-Objekte deklarieren und Erzeugen....................32
7.2.2 Array-Objekte benutzen....................33
7.2.2.1 Direkter Zugriff....................33
7.2.2.2 Länge des Arrays....................33

7.3 Objektsammlungen mit flexibler Größe....................34
7.3.1 List....................34
7.3.1.1 ArrayList....................34
7.3.1.2 LinkedList....................36
7.3.1.3 Stack....................38
7.3.2 Map Abbildungen....................39

7.3.2.1 Hash Map....39
7.3.2.1.1 Abbildungen und Map-Klassen....39
7.3.2.2 TreeMap....40
7.3.3 Set - Menge....42
7.3.3.1 HashSet....42
7.3.3.2 TreeSet....43

7.4 Iterator 45

7.5 Der Cast-Operator....46

7.6 Die null-Referenz....46

8 FEHLER VERMEIDEN - FEHLER FINDEN....48

8.1 Testen und Fehlerbeseitigung....48

8.2 Modultests in BlueJ....48

8.3 Automatisierte Tests....49

8.4 Manuelle Ausführung....49

8.5 print-Anweisungen....50

8.6 Debugger....50

9 KLASSENENTWURF....51

9.1 Kopplung und Kohäsion....51

9.2 Code-Duplizierung....51

9.3 Entwurf nach Zuständigkeiten....52

9.4 Kapselung der Daten....52

9.5 Implizite Kopplung....52

9.6 Programmausführung ohne BlueJ....53
9.6.1 Klassenmethoden....53

9.7 Die main-Methode....53

10 VERERBUNG - POLYMORPHIE....54

10.1 Syntax....54

10.2 Klassenhierarchie....55

10.3 Subklassen und Subtypen....56

10.4 Polymorphe Variablen....57

10.5 Statischer und dynamischer Typ....59

10.6 Überschreiben von Methoden....59
10.6.1 Dynamische Methodensuche....59

10.7 super-Aufrufe in Methoden....61

10.8 Methoden-Polyphormie....61

10.9 Aus Object geerbte Methode: toString....61

11 ABSTRAKTE KLASSEN....62

11.1 Abstrakte Methoden....63

11.2 Interfaces....63

12 FEHLERBEHANDLUNG: EXCEPTIONS....65

12.1 Prinzip der Ausnahmebehandlung....65

12.2 Behandlung von geprüften Exceptions....67
12.2.1 Auffangen der Exception....67
12.2.2 Propagieren (weiterreichen) der Exception....69

12.3 Behandlung von ungeprüften Exceptions....69

13 EIN- UND AUSGABE VON TEXTDATEIEN....70

13.1 Textausgabe mit FileWriter....70
13.1.1 Escape-Codes für Sonderzeichen....71

13.2 Texteingabe mit FileReader....71

13.3 Texteingabe von der Konsole....72

14 GRAFIK UND FENSTER IM AWK....73

14.1 Die Klasse Frame: Top-Level Grafikfenster....73

14.2 Die Klasse Panel: Ein Grafik-Container....75

14.3 Die Klasse Graphics....76
14.3.1 Das Koordinatensystem von Graphics....81
14.3.2 Zeichnen und Füllen....81
14.3.3 Linien....81
14.3.4 Rechtecke....81
14.3.5 Polygone (Vielecke)....82
14.3.6 Ovale und Bögen....82

14.4 Grafische Textausgabe und Schrifttypen (Font)....83

14.5 Die Farben der Grafikausgabe (Color)....83

15 EVENTS UND EVENTHANDLER....85

15.1 Eventhandler....86

15.2 Listener Registration....87
15.2.1 Klassenimplimentation....87
15.2.2 Innere Klassen....88
15.2.3 Anonyme innere Klassen....90

15.3 Panel mit "Gedächtnis" 90

15.4 Event – Kurzübersicht....91
15.4.1 Focus-Ereignisse....91
15.4.2 Key-Ereignisse....91
15.4.3 Mouse-Ereignisse....91
15.4.4 MouseMotion-Ereignisse....92
15.4.5 Component-Ereignisse....92
15.4.6 Container-Ereignisse....93
15.4.7 Window-Ereignisse....93
15.4.8 Action-Ereignisse....93
15.4.9 Adjustment-Ereignisse....94
15.4.10 Item-Ereignisse....94
15.4.11 Text-Ereignisse 94

16 KOMPONENTEN EINES GUI....96

16.1 Die Komponenten des AWT....96

16.2 Die vier Schritte für die Programmierung eines GUI:....98
16.2.1 Die benötigten Komponenten instanzieren....98
16.2.2 Die Komponenten in einen Container logisch einfügen....98
16.2.3 Die Komponenten im Container räumlich anordnen....98
16.2.4 Die Komponentenereignisse in Eventmethoden verarbeiten....98

16.3 Layout-Management....99
16.3.1 FlowLayout....99
16.3.2 GridLayout....100
16.3.3 Die Größe der Komponenten 100
16.3.4 BorderLayout....101

16.4 Dialoge....103

16.5 Filedialoge....106

17 APPLETS....107

17.1 Unterschiede zwischen Applets und Applikationen....107

17.2 Sicherheitseinschränkungen von Applets....107

17.3 Erstellen von Applets....107
17.3.1 Initialisieren....108
17.3.2 Starten....108
17.3.3 Stoppen....108
17.3.4 Zerstören....108
17.3.5 Zeichnen (paint)....109

17.4 Applet in eine Webseite einfügen....110

17.5 Java-Archive....111

17.6 Parameter an Applets weitergeben....111

Stichwortverzeichnis

A
Abstrakte Klassen....62
Abstrakte Methoden....63
Abstraktion....11, 14
Adapterklassen....88
add-Methode....98
aktueller Parameter....4
Anonyme innere Klassen....90
Applets....107
Applikation....73
Arithmetische Operatoren....13
Array-Objekte....32
ArrayList....34
Attribute....2
B
Bildschirmausgaben in Java....6
Binärdateien....70
Block....5
Bögen....82
BorderLayout....101
Bottom-Up-Design....11
BufferedReader....71
Bufferwriter....71
Bytecode....2
C
Call-back-Methode....85
Cast-Operator....46
catch....67
ClassCastException....46
Code-Duplizierung....51
CODEBASE....110
Color....83
Comparable....30
Compiler....2
Container....75, 85
D
Datenfelder....2, 3, 7, 14
Datentypen....1
Debugger....17
Der Zustand von Objekten....2
destroy()....108
Dialoge....103
Die Begriffe Objekt und Klasse....1
do-Schleife....10
dynamischen Typs....59
dynamischer Typ....59
E
Ein- und Ausgabe von Textdateien....70
Ersetzbarkeit....56
Erweiterbarkeit....51
Escape-Codes....71
Event....74
Event sources....85
Eventhandler....85
Events....85
Exception....65
Exceptions....65
Externe Methodenaufrufe....16
F
Farben....83
Filedialoge....106
FileReader....71
FileWriter....70
finally....68
FlowLayout....99
for-Schleife....10
Formale Parameter....7
Frame....73
Füllen....81
G
Gedächtnis....90
Geprüfte Exceptions....66
Graphics....76
GridLayout....100
Grobaufbau von Klassen....3
GUI....96
H
Hash Map....39
HashSet....42
I
If-Anweisung....8, 9
Implizite Kopplung....52
init()....108, 109
Innere Klassen....88
inneren Klassen....88
Inspektor-Methoden....5
Instanz....1
Interface....30
Interfaces....63
Interne Methodenaufrufe....16
ist-ein-Beziehung....54
Iterator....45
J
Java-Quelltext....2
K
Kapselung der Daten....52
Klassendefinitionen....3
Klassendiagramm....12
Klassenhierarchie....55
Klassenimplimentation....87
Klassenmethoden....53
Kohäsion....51
Kommentare....4
Komponenten....85, 96
Konsole....72
Konstruktoren sind Methoden....4
Konstruktoren überladen....16
Koordinatensystem....81
Kopplung....51
L
LayoutManager....98
Linien....81
List....34
Logische Fehler....48
Logische Operatoren....8
Lokale Variablen....7
Lokale Variablen, Datenfelder und Parameter....7
lose gekoppelt....14
lose Kopplung....14
M
main-Methode....53
Map....34

Methoden..1
Methodenaufrufe...16
Methodenkopf..4
milestone-Methoden'..107
modal..103
Modifier Tabelle..15
Modifierer...14
Modularisierung...11
Module..11
Modultests...48
MouseEvent...86
Mutator-Methoden..6
N
Negative Tests..49
new-Anweisung..14
Nichtterminalsymbole...3
null-Referenz..46
O
Objektdiagramm...12
Objekte und Klassen...1
Objektsammlungen mit flexibler Größe.......................34
Objekttypen..12
Ovale..82
P
package..21
Panel..75
Polymorphe Variablen...57
Polymorphie...54
Positive Tests...49
Private..14
protected..15
Pseudocode..3
Public...14
R
Rechtecke...81
Referenzvariable..56
Regressionstests...49
RGB-Wert...83
Rückgabeanweisung\;...5
RuntimeException..66
S
Schrifttypen..83
Set34
Sichtbarkeit von Variablen...5
Signatur..1
Sondierende Methoden...5
source code...2
Stack...38
start()...108
statischen Typ...59
statischer Typ..59
Statischer und dynamischer Typ..................................59
String..21
StringTokenizer..25
Subklassen...54, 56
Subtypen...56
super-Aufrufe in Methoden..61
Superklasse...54
switch...9
Syntaxfehler..48
T
Teile und Herrsche...11
Terminalsymbolen..3
Textausgabe..70
Textdateien...70
this...17, 53
throw..65
Top-Down-Design..11
toString...61
TreeMap..40
TreeSet..43
try 67
U
Überschreiben von Methoden......................................59
Ungeprüfte Exceptions...66
V
Verändernde Methoden..6
Vererbung...54
Vielecke..82
Vorteile durch Vererbung...55
W
Wartbarkeit...51
while-Schleife...9
WindowListener..74
Wrapper...34
Wrapper-Klassen...27
Z
Zeichnen..81
Zugriff mit Index oder Iterator?.................................46
Zuweisungen...5
Zuweisungsoperator..5

1 Objekte und Klassen

Eine **Klasse** ist eine abstrakte Beschreibung von Objekte einer Kategorie.

Anders ausgedrückt:
- Eine Klasse beschreibt, wie alle Objekte der entsprechenden Kategorie aufgebaut sind.
- Eine Klasse ist somit der abstrakte „Bauplan“ für ihre Objekte.

1.1 Die Begriffe Objekt und Klasse

Ein konkretes **Objekt** nennt man eine **Instanz** einer Klasse.
Jedes Objekt in der OOP ist Instanz (mindestens) einer Klasse.

1.2 Methoden

Jede Methode definiert, welche Art von Parametern man an sie übergeben darf und muss.
In obigen Dialogfenster wird dies durch die Zeile

```
void horizontalBewegen(int entfernung)
```

zum Ausdruck gebracht. Man nennt das die **Signatur** der Methode.

1.3 Datentypen

Der Typ legt fest, welche Art von Informationen als Parameter übergeben werden kann.

Primitive Datentypen

Typname	Länge	Wertebereich	Standardwert
boolean	1	true, false	false
char	2	Alle Unicode-Zeichen	\u0000
byte	1	$-2^{7}...2^{7}-1$	0
short	2	$-2^{15}...2^{15}-1$	0
int	4	$-2^{31}...2^{31}-1$	0
long	8	$-2^{63}...2^{63}-1$	0
float	4	$+/-3.40282347 * 10^{38}$	0.0
double	8	$+/-1.79769313486231570 * 10^{308}$	0.0

1.4 Der Zustand von Objekten

Die **Eigenschaften** eines Objekts werden auch seine **Attribute** genannt, hier im Beispiel etwa x- und y-Position, Größe, Farbe, Sichtbarkeit.
In Java heißen diese Attribute von Objekten auch **Datenfelder** oder kurz **Felder** (engl. *fields*).

Die Menge der momentanen Werte aller Attribute eines Objekts wird als der **Zustand** des Objekts bezeichnet.

1.5 Java-Quelltext

Jede Klasse wird durch einen Java-Quelltext definiert. (engl. : **source code**)

Er enthält das Programm, geschrieben in der Programmiersprache Java.
In BlueJ erreicht man den Quelltext durch das Kommando *bearbeiten* für ein Klassensymbol.

Im Java-Quelltext jeder Klasse:
- Definition der Datenfelder
- Definition der Methoden und ihrer Parameter
- Definition der Aktionen bei Aufruf der Methode

Kunst des Programmierens:
- Klassendefinitionen erstellen
- Hier in Java

Nach Änderung des Quelltextes muss die Klasse neu übersetzt werden.
Der **Compiler** (das Übersetzungsprogramm) erzeugt aus dem Quelltext den **„Bytecode**“, der zur Ausführung des Programms von der virtuellen Java-Maschine verwendet wird.

(**jvm** = java virtual machine)

Die virtuelle Java-Maschine ist ein Programm, das den Java-Bytecode interpretieren kann (ein „Interpreter“).

2 Klassendefinitionen

2.1 Grobaufbau von Klassen

Jede Klassendefinition besteht aus Kopf und Rumpf.
Der Kopf enthält das Schlüsselwort class und den Namen der Klasse.
Das Schlüsselwort private ist optional. Mehr darüber später.

Der Rumpf der Klassendefinition ist in geschweifte Klammern eingeschlossen.
Er enthält
- die Datenfelder für die Speicherung der Daten der Objekte,
- die Konstruktoren für die Erzeugung neuer Instanzen der Klasse,
- die Methoden zur Realisierung des Verhaltens der Objekte.

allgemeiner Aufbau:

```
public class Klassenname
{
   Datenfelder
   Konstruktoren
   Methoden
}
```

Hinweise zur Schreibweise:

Programmcode ist hier in Schreibmaschinenschrift dargestellt.
Die *kursiv* geschiebenen Wörter sind Platzhalter, die durch einen konkreten Programmtext ersetzt werden müssen. In manchen Büchern werden solche Platzhalter in spitze Klammern geschrieben, z.B. <Klassenname>.

In der formalen Beschreibung von Programmiersprachen nennt man diese Platzhaltersymbole auch **Nichtterminalsymbole**, weil sie noch nicht endgültig (terminal) stehen bleiben sondern ersetzt werden.
Sie stehen im Gegensatz zu **Terminalsymbolen** die endgültig so stehen bleiben können, hier z.B. `public` und `class` .

Wir verwenden hier kursiv geschriebnen Text in der Abbildung von Programmen auch, um ein ausgelassenes Programmstück zu beschreiben. Man spricht dann auch von **Pseudocode**.
Es ist klar, dass der Compiler Pseudocode nicht übersetzen kann.

2.1.1 Datenfelder

Die Datenfelder sind Speicherzellen im Objekt, in denen Werte gespeichert werden können.
Jedes Objekt (jede Instanz) hat seine eigenen Speicherzellen für seine Datenfelder.
Die Methoden und Konstruktoren eines Objekts können auf die Werte der Datenfelder zugreifen.

Die Werte der Datenfelder einer Instanz kann man in BlueJ mit dem Objektinspektor untersuchen. (Durch Doppelklick auf das Objekt in der Objektleiste!)

2.2 Kommentare

Text für Erläuterungen des Programmierers an menschliche Leser.

Einzeilige Kommentare beginnen mit `//` und enden am Ende der aktuellen Zeile.

Mehrzeilige Kommentare beginnen mit `/*` und enden mit `*/`. Sie können sich über mehrere Zeilen erstrecken.

Dokumentationskommentare beginnen mit `/**` und enden mit `*/` und können sich ebenfalls über mehrere Zeilen erstrecken.

2.3 Konstruktoren sind Methoden

Konstruktoren sind spezialisierte Methoden.
Sie werden unmittelbar dann für ein Objekt aufgerufen, wenn es neu erzeugt wird.
Konstruktoren bringen die Objekte in einen gültigen Anfangszustand.
Sie "initialisieren" den Zustand des Objekts.
Konstruktoren haben immer denselben Namen wie die Klasse.

Hinweis:
In Java werden alle Datenfelder automatisch mit einem vordefinierten Wert initialisiert, abhängig vom Typ. int-Felder werden z.B. mit 0 initialisiert. Trotzden sollte man die Initialisierung explizit hinschreiben um zu dokumentieren, dass man sie nicht vergessen hat.

Datenübergabe mit Parametern

Methoden, und auch Konstruktoren sind Methoden, können Werte über Parameter erhalten.
Die Parameter einer Methode werden im **Methodenkopf** definiert, z.B. hier im Konstruktor:

```
public Ticketautomat(int ticketpreis);
```

Beim Aufruf der Methode (des Konstruktors) wird ein Wert in den Parameter geschrieben.
Der Parameter ist ein Speicherplatz der Methode, der ihr solange zur Verügung steht, wie ihre Ausführung dauert. Wir sprechen von Methodenspeicher, d.h. Speicher in der Methode. Das folgende Bild zeigt die Übergabe des Parameterwerts über den Parameter `ticketpreis` des Konstruktors und die Eintragung ins Datenfeld `preis`.

Der Wert des Parameters außerhalb der Methode, der übergeben wird, heißt **aktueller Parameter**.
Formale Parameter speichern Werte in einer Methode, sind also auch Variablen, allerdings eine andere Art als die Datenfelder der Objekte. Im obigen Bild sind alle Variablen durch weiße Kästen dargestellt.

Sichtbarkeit von Variablen:
Ein formaler Parameter kann nur innerhalb **seiner** Methode angesprochen werden. Man sagt, er ist nur im Rumpf seiner Methode sichtbar.
Im Gegensatz dazu die Datenfelder: Sie können in der kompletten Klassendefinition angesprochen werden. Ihr Sichbarkeitsbereich ist die ganze Klasse.
Lebensdauer von Variablen:
Ein formaler Parameter existiert nur solange **seine** Methode ausgeführt wird. Nach Beendigung der Methode wird der Methodenspeicher und damit der formale Parameter gelöscht. Seine Lebensdauer ist also die Ausführungszeit der Methode. Wird die Methode erneut aufgerufen, wird neuer Speicher zur Verfügung gestellt. Der alte Wert ist weg!
Die Lebensdauer eines Datenfeldes hingegen ist solange wie die Existenz seines Objekts.

2.4 Zuweisungen

Zuweisungen sind Anweisungen, die einen Wert in eine Variablen speichern.
(Slang: die einen Wert in eine Variable schreiben.)
Eine Zuweisung enthält einen Zuweisungsoperator, z.B. das Gleichheitszeichen "=" in

```
preis = ticketpreis;
```

Der Wert rechts vom Zuweisungsoperator wird in die Variable links vom **Zuweisungsoperator** gespeichert, hier der Wert der Variable `ticketpreis` in die Variable `preis`. Die rechte Seite kann auch ein beliebig komplizierter Ausdruck sein.
Der Wert des Ausdrucks wird zuerst berechnet und anschließend an die Variable links zugewiesen.
Regel: Der Typ des Ausdrucks rechts muss zum Typ der Variablen links "passen".
Die gleiche Regel gilt für aktuelle und formale Parameter von Methoden.
Der Typ des aktuelle Parameters muss zum Typ des formale Parameters passen.

Hinweis: "Passen" heißt nicht unbedingt "gleich sein". Näheres dazu später.

2.5 Sondierende Methoden

Eine Menge von Anweisungen und Deklarationen, die in geschweifte Klammern eingeschlossen sind, heißt **Block**. Der Rumpf jeder Methode ist ein Block.
Vergleiche die Signaturen des Konstruktors und obiger Methode:

```
public Ticketautomat(int ticketpreis)
public int gibPreis()
```

Der Konstruktor hat keinen Ergebnistyp, die Methode hier hat den Ergebnistyp `int`.
Konstruktoren erkennt man daran, dass nie einen Ergebnistyp haben.
Methoden, die kein Ergebnis liefern, haben den Ergebnis typ `void`.
Methoden und Konstruktoren können beliebig viele Parameter haben.
Im Rumpf von `gibpreis` steht hier nur die eine Anweisung `return preis;`
Dies ist eine **Rückgabeanweisung**; der Wert des hinter dem Schlüsselwort stehenden Ausdrucks wird als Ergebnis zurückgegeben. Der Typ des Ausdrucks muss zum deklarierten Rückgabetyp passen. Der Ausdruck ist hier ganz einfach: `preis`. Zurückgegeben wird also unmittelbar der Wert des Datenfeldes Preis.
Methoden, die Werte des Zustands zurückliefern heißen sondierende Methoden oder auch ***Inspektor-Methoden***.

Verändernde Methoden

Viele Methoden ändern den Zustand eines Objektes. Man nennt diese deshalb verändernde Methoden oder **Mutator-Methoden.**
Nach Ausführung einer verändernden Methoden verhält sich ein Objekt u.U. anders als vorher.

2.6 Bildschirmausgaben in Java

siehe Beispiel: die Methode `ticketDrucken()`
- Ergebnistyp void, keine Paramter.
- Aufruf von `System.out.println("# Die BlueJ-Linie")` gibt den Text in Anführungszeichen aus.
- `System.out.println` ist eine Methode mit einem String-Parameter.
- Erneuter Aufruf von `System.out.println` sorgt für Ausgabe in neuer Zeile (Zeilenvorschub).
- Aufruf von `System.out.println` sorgt für Ausgabe in neuer Zeile (Zeilenvorschub).
- Aufruf von `System.out.println("# " + preis + " Cent.")` enthält als aktuellen Parameter einen Ausdruck.
- der Ausdruck `"# " + preis + " Cent."` ist vom Typ String.

Wenn in a+b der Operand a oder der Operand b von Typ String ist, ist auch das Ergebnis vom Typ String.
Ist einer der Operanden nicht vom Typ String, wird er erst in String gewandelt.
Ergebnis ist ein String, der aus der Verkettung der beiden Strings besteht.

3 Lokale Variablen, Datenfelder und Parameter

Die Sichtbarkeit von lokalen Variablen ist auf den Block beschränkt, in dem sie deklariert wurde.
Die Lebensdauer ist die Zeit, in der der Block ausgeführt wird. Der Block kann der gesamte Rumpf der Methode sein.
Man kann die lokale Variable auch gleich bei der Deklaration initialisieren:

```
int wechselgeld=bisherGezahlt;
```

3.1 Eigenschaften der Variablentypen:

3.1.1 Datenfelder

- Sie werden außerhalb von Methoden (und Konstruktoren) deklariert
- klassenweite Sichtbarkeit
- wenn `private` deklariert, von außerhalb der Klasse nicht sichtbar
- speichern den Zustand des Objekts
- Lebensdauer wie Objekt
- werden automatisch initialisiert
- bilden den Zustand des Objekts

3.1.2 Formale Parameter

- Sie werden im Methodenkopf deklariert
- Sichtbarkeit ist auf die definierende Methode beschränkt
- Lebensdauer ist die Ausführungszeit ihrer definierenden Methode
- danach gehen die Werte verloren: also nur zur temporären Speicherung geeignet
- Sie bekommen ihre Werte von außen beim Aufruf der Methode von den aktuellen Parametern

3.1.3 Lokale Variablen

- Sie werden in einem Block einer Methode deklariert
- Sichtbarkeit ist auf diesen definierenden Block beschränkt
- Lebensdauer ist die Ausführungszeit des deklarierenden Blockes
- Sie müssen explizit initialisiert werden vor der Benutzung in einem Ausdruck
- Sie werden also nicht automatisch initialisiert
- Beim Verlassen des Blocks gehen ihre Werte verloren: nur temporäre Speicherung

4 Bedingte Anweisungen

4.1 Logische Operatoren

<table>
<tr><th>Operator</th><th>Bezeichnung</th><th>Bedeutung</th></tr>
<tr><td>!</td><td>Logisches NICHT</td><td>!a ergibt false, wenn a wahr ist, und true, wenn a falsch ist.</td></tr>
<tr><td>&&</td><td>UND mit Short-Circuit-Evaluation</td><td>a && b ergibt true, wenn sowohl a als auch b wahr sind. Ist a bereits falsch, so wird false zurückgegeben und b nicht mehr ausgewertet.</td></tr>
<tr><td>||</td><td>ODER mit Short-Circuit-Evaluation</td><td>a || b ergibt true, wenn mindestens einer der beiden Ausdrücke a oder b wahr ist. Ist bereits a wahr, so wird true zurückgegeben und b nicht mehr ausgewertet.</td></tr>
<tr><td>&</td><td>UND ohne Short-Circuit-Evaluation</td><td>a & b ergibt true, wenn sowohl a als auch b wahr sind. Beide Teilausdrücke werden ausgewertet.</td></tr>
<tr><td>|</td><td>ODER ohne Short-Circuit-Evaluation</td><td>a | b ergibt true, wenn mindestens einer der beiden Ausdrücke a oder b wahr ist. Beide Teilausdrücke werden ausgewertet.</td></tr>
<tr><td>(Ziel_Typ) Objekt</td><td>cast</td><td>Konvertiert Objekte</td></tr>
<tr><td>instanceof</td><td>Prüft auf Instanzäquivalenz</td><td>Instanz1 instanceof Instanz2</td></tr>
<tr><td>^</td><td>Exklusiv-ODER</td><td>a ^ b ergibt true, wenn beide Ausdrücke einen unterschiedlichen Wahrheitswert haben.</td></tr>
</table>

4.2 Allgemeine Form der Alternative (If-Anweisung zweiseitig)

boolesche Ausdrücke können Logische Operationen enthalten.

```
if (boolescher Ausdruck) {
   Anweisungen falls true
} else {
   Anweisungen falls false
}
```

4.3 Allgemeine Form der Alternative (switch-Anweisung zweiseitig)

Beispielprogramm
Programmname: Energieverbrauch
Verzeichnis: 10
Klasse: Energiebilanz ; grün

Die `switch`-Anweisung ist eine einfache Form der Mehrfachverzweigung. Sie vergleicht nacheinander den Ausdruck hinter dem `switch` (ein primitiver Typ wie `byte`, `char`, `short` oder `int`) mit jedem einzelnen Fallwert. Alle Fallwerte müssen unterschiedlich sein. Stimmt der Ausdruck mit der Konstanten überein, so wird die Anweisung beziehungsweise die Anweisungen hinter der Sprungmarke ausgeführt.
Gibt es keine Übereinstimmung mit einer Konstanten, so lässt sich optional die Sprungmarke `default` einsetzen

```
 switch ( op )
{
  case '+':   // addiere
    break;

  case '-':   // subtrahiere
    break;

  case '*':   // multipliziere
    break;

  case '/':   // dividiere
    break;

  default:
    System.err.println( "Operand nicht definiert!" );
}
```

4.4 Die while-Schleife

Eine Schleife erlaubt es, einen Block von Anweisungen mehrfach auszuführen.
Eine von mehreren Formen der Schleife ist die ***while***-Schleife:

Syntax

```
while (Ausdruck) {
     Anweisung
}
```

Das Schlüsselwort ***while*** (engl. für solange) leitet die Schleife ein.
In der runden Klammer folgt die **Bedingung** für eine (erneute) Ausführung der Anweisungen.

In der geschweiften Klammer stehen die zu wiederholenden Anweisungen, der "**Schleifenrumpf**".
Ist die Wiederholungsbedingung erfüllt, wird der Schleifenrumpf einmal ausgeführt.
Danach wird die Bedingung erneut geprüft, und im positiven Fall der Rumpf erneut ausgeführt.

Ist die Bedingung bei Prüfung am Anfang oder später nicht erfüllt, wird das Programm mit der Anweisung fortgesetzt, die auf den Schleifenrumpf folgt.

4.5 Die do-Schleife

Syntax

```
do{
  Anweisung;
}
while (Ausdruck) ;
```

Bedeutung

Die `do`-Schleife arbeitet *nichtabweisend*, d.h. sie wird mindestens einmal ausgeführt. Da zunächst die Schleifenanweisung ausgeführt und erst dann der Testausdruck überprüft wird, kann die `do`-Schleife frühestens nach einem Durchlauf regulär beendet werden. Die Bearbeitung der Schleife wird immer dann beendet, wenn der Test des Schleifenausdrucks `false` ergibt.

4.6 Die for-Schleife

Eine for-Schleife ist besonders geeignet, wenn
- die Anzahl der Wiederholungen von vornherein feststeht,
- innerhalb der Schleife eine Variable benötigt wird, die sich bei jeder Iteration ändert.

Eine for-Schleife wird oft benutzt, um über Array-Elemente zu iterieren.
Allgemeine Form:

Syntax:

```
for(initialisierung    ;    Bedingung    ;    Aktion nach Rumpf) {
   Anweisungen ;
}
```

5 Objektinteraktion

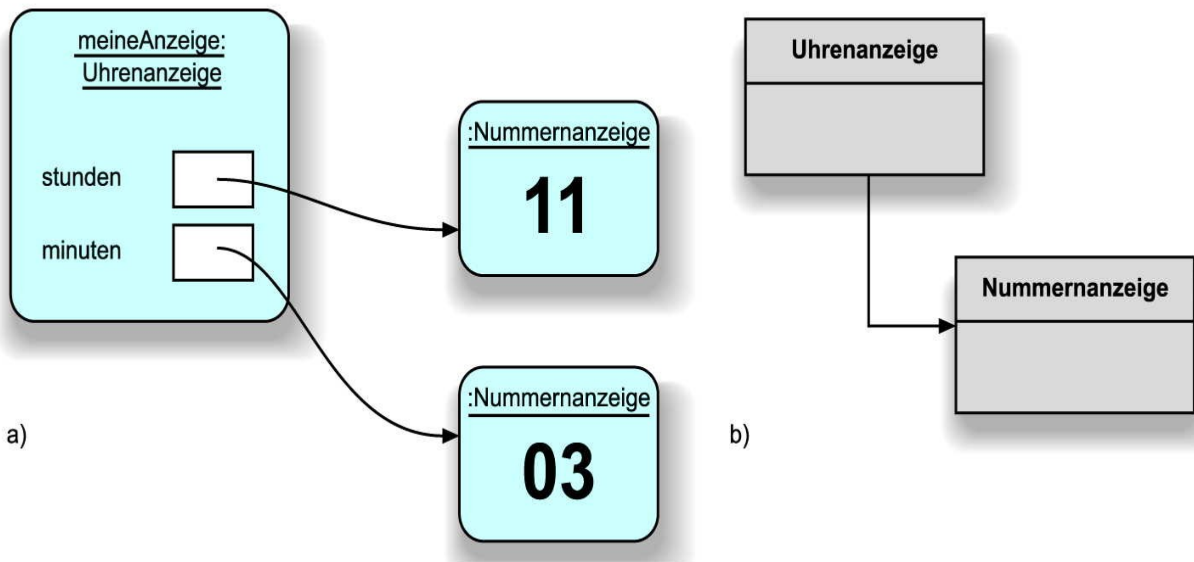
Die allermeisten Programme erfordern das Zusammenspiel mehrerer Objekte unterschiedlichen Typs.

5.1 Abstraktion und Modularisierung

Das Gesamtsystem wird in Teilsysteme zerlegt,
diese werden wiederum in Teilsysteme zerlegt usw.
solange, bis dabei kleine Teilsysteme entstehen,
die überschaubar genug sind um als Einheit entwickelt zu werden.
Herunterbrechen in handhabbare Teilaufgaben: "**Top-Down-Design**".
Prinzip: **Teile und Herrsche**.
Die Teilsysteme nennt man **Module**, den Prozess **Modularisierung**.
Module können von mehreren Entwicklern parallel entwickelt werden.
Nach der Entwicklung der Module einer Stufe:
daraus die Teilsysteme der nächst höheren Stufe zusammensetzen usw.
bis man auf der obersten Stufe das Gesamtsystem zusammensetzen kann.
Zusammensetzen von Teileinheiten: "**Bottom-Up-Design**".
In Summe spricht man vom **Top-Down/Bottom-Up-Design.**
Der Entwickler eines Moduls kennt natürlich alle Details der Realisierung seines Moduls.
Der Entwickler einer höheren Teileinheit, die dieses Modul nutzt, braucht dessen Realisierungsdetails nicht alle zu kennen.
Er **abstrahiert** von den Details der niedrigeren Ebene.
Er behält den Überblick durch **Abstraktion**.
Abstraktion ist die Fähigkeit, Details zu ignorieren, um das Gesamtbild erfassen zu können.
Abstraktion und Modularisierung werden auch bei der Softwareentwicklung angewandt:
Teilkomponenten finden, die unabhängig implementiert werden können.
In der objektorientierten Programmierung (OOP) sind die Teilkomponenten Objekte.

5.2 Klassendiagramme und Objektdiagramme

Die Struktur der Uhrenanzeige verdeutlicht ein Objektdiagramm:



Objektdiagramm a) - dynamische Sicht (zur Laufzeit):
Zeigt die Situation, wenn man ein Objekt vom Typ Uhrenanzeige erzeugt.
Das Objektdiagramm macht deutlich:
Die Variable `stunden` im Uhrenanzeige-Objekt enthält nicht das Nummernanzeige-Objekt selbst sondern eine ***Referenz*** auf das Objekt, veranschaulicht durch einen ***Pfeil*** im Objektdiagramm.

Klassendiagramm b) - statische Sicht (zur Programmierzeit):
Die Klasse Uhrenanzeige nutzt die Klasse Nummernanzeige,
die Klasse Uhrenanzeige hängt von Nummernanzeige ab.

5.3 Objekttypen

Objekttypen, sind Typen, die durch Klassen definiert werden.
Es gibt einige in Java vordefinierte Klassen z.B. `String`, andere definieren wir selbst.
Ein Objekt wird nicht direkt in einer Variablen gespeichert, sondern lediglich eine Referenz auf das Objekt.

5.4 Arithmetische Operatoren

Operator	**Bezeichnung**	**Bedeutung**
+	Positives Vorzeichen	+n ist gleichbedeutend mit n
-	Negatives Vorzeichen	-n kehrt das Vorzeichen von n um
+	Summe	a + b ergibt die Summe von a und b
-	Differenz	a - b ergibt die Differenz von a und b
*	Produkt	a * b ergibt das Produkt aus a und b
/	Quotient	a / b ergibt den Quotienten von a und b
%	Restwert, Modulo	a % b ergibt den Rest der ganzzahligen Division von a durch b. In Java läßt sich dieser Operator auch auf Fließkommazahlen anwenden.
++	Präinkrement	++a ergibt a+1 und erhöht a um 1
++	Postinkrement	a++ ergibt a und erhöht a um 1
--	Prädekrement	--a ergibt a-1 und verringert a um 1
--	Postdekrement	a-- ergibt a und verringert a um 1

% ist der Modulo-Operator.
Der Ausdruck a%b liefert als Ergebnis den Rest der ganzzahligen Division von a durch b.

15%10 = 5
6%5 = 1
14%7 = 0
23%24 = 23
24%24 = 0
59%60 = 59
60%60 = 0

5.5 new-Anweisung

Allgemeine Form der new-Anweisung:

```
new Klassenname ( Parameterliste )
```

Sie bewirkt drei Dinge:
- Sie erzeugt ein neues Objekt der angegebenen Klasse,
- Sie ruft den Konstruktor dieser Klasse auf und übergibt ihm die aktuellen Parameter der Parameterliste.
- Sie liefert als Wert eine Referenz auf das neu erzeugte Objekt.

5.6 Modifierer

5.6.1 Public und Private Eigenschaften

Die Schlüsselwörter `public` und `private` regeln die Zugriffsmöglichkeit auf Datenfelder und Methoden. Sie heißen deshalb **Zugriffsmodifikatoren**.
Zugriffsmodifikatoren steuern die Sichtbarkeit von Datenfeldern und Methoden.

`public`	Zugriff auch von außerhalb
`private`	Zugriff nur innerhalb der definierenden Klasse

Die **Schnittstelle** der Klasse ist der öffentliche Teil einer Klasse.
Ihre Beschreibung muss deshalb alle **public** Datenfelder und Methoden enthalten.
Die **Implementierung** der Klasse legt fest, wie im Detail die Klasse ihre Augabe erfüllt.
Die Implementierung ist der private Teil der Klasse.
Die meisten Datenfelder (bisher alle) gehören zum privaten Teil der Klasse.
Der private Teil der Klasse sollte für den Anwender der Klasse verborgen sein.
Man sagt, die private Information der Klasse ist gekapselt.
Schlagwörter: **Kapselung, information hiding, Geheimnisprinzip.**
Das Geheimnisprinzip hat zwei Aspekte:
1) Prinzip **Abstraktion:**
Nutzer (Programmierer) der Klasse sollte die Interna einer Klasse nicht kennen müssen, um die Klasse nutzen zu können.
Beispiel: Wir konnten die Klasse HashMap ganz einfach verwenden, weil wir die Klasse als einen abstrakten Modul betrachtet haben.
2) Prinzip **lose Kopplung:**
Eine nutzende Klasse sollte keine Kenntnis über die Interna genutzen Klasse haben, damit die genutzte Klasse jederzeit geändert werden kann, ohne dass die Nutzer das wissen müssen. Bei einer Änderung muss lediglich die Schnittstelle beibehalten werden. Man sagt auch, eine Klasse und Ihre Nutzer sind nur **lose gekoppelt**.
Beispiel: Wenn die interne Implementierung der Klasse HashMap geändert wird, arbeitet sie trotzdem weiter mit unserer Kundendienst-Klasse zusammen. Es besteht nur eine **lose Kopplung**.
Lose Kopplung erleichtert die Wartung eines Software-Systems ganz erheblich.

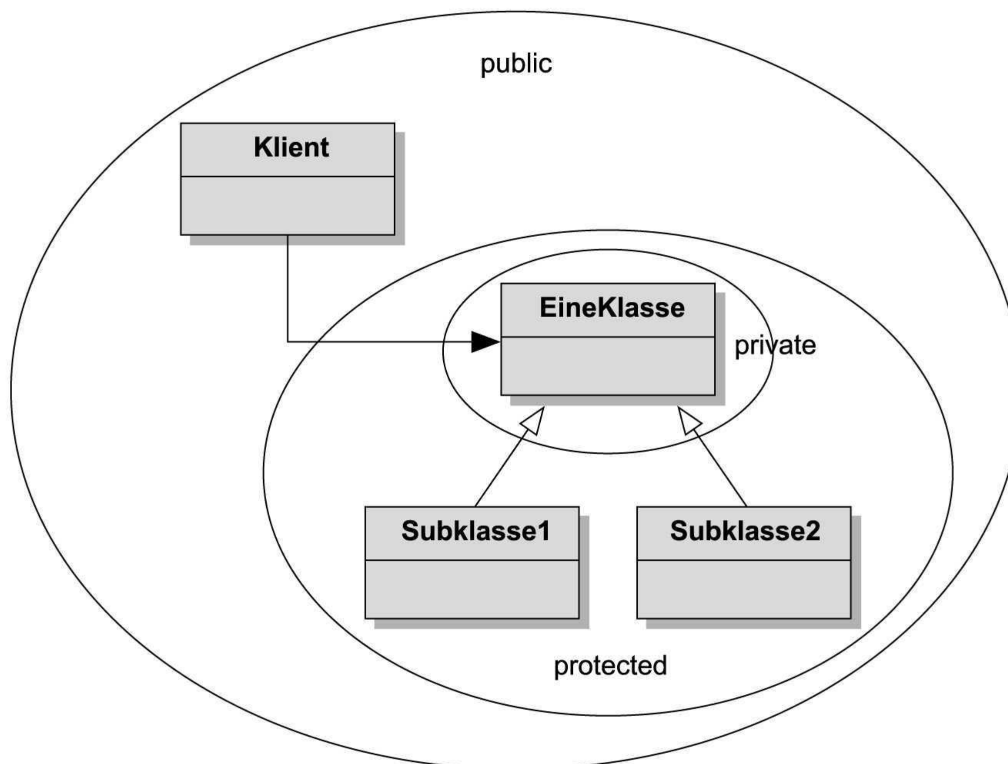
Datenfelder

Bis jetzt waren alle Datenfelder in unseren Projekten `private.`
Der Grund liegt liegt im **Geheimnisprinzip**.
Java erlaubt auch public Datenfelder, im allgemeinen ist aber davon abzuraten.

Dadurch dass Datenfelder private sind hat ein Objekt selbsz die Kontrolle über seinen Zustand,
d.h. die Datenfelder können nur durch Methoden der Klasse des Objekts verändert werden.
Fazit: Datenfelder sollten immer privat sein. (Ausnahmen später)

5.6.2 Der Zugriffs-Modifier *protected*

protected ermöglicht die Zugreifbarkeit für (direkte und indirekte) Subklassen.



5.6.3 Modifier Tabelle

public	Keine Einschränkung der Zugriffsrechte auf Klassen, Methoden oder Variablen
package	Standardzugriff: Verfügbarkeit nur für die Klassen innerhalb des selben Paketes (Angabe optional)
private	höchste Schutzebene für Methoden und Variablen, Sichtbarkeit nur innerhalb der eigenen Klasse
protected	wie package, jedoch können gegenwärtige und zukünftige Subklassen (auch aus anderen Paketen) ebenfalls zugreifen
static	Erzeugung von Klassenmethoden und –variablen, ohne Instanz
final	verhindert, dass Klassen abgeleitet und Methoden überschrieben werden können und macht Variablen zu Konstanten: Variable Großbuchstabe
abstract	Dient der Erstellung abstrakter Klassen und Methoden

5.6.4 Die unterschiedlichen Ebenen des Zugriffsschutzes

Sichtbarkeit	public	protected	Default	private
Innerhalb derselben Klasse	Ja	Ja	Ja	Ja
Von einer bel. Klasse im selben Paket	Ja	Ja	Ja	Nein
Von einer bel. Klasse außerhalb des Pakets	Ja	Nein	Nein	Nein
Von einer Subklasse im selben Paket	Ja	Ja	Ja	Nein
Von einer Subklasse außerhalb des Pakets	Ja	Ja	Nein	Nein

5.7 Konstruktoren überladen

Manche Klassen haben mehrere Konstruktoren, so z.B. auch Uhrenanzeige.
Damit kann man ein Objekt auf verschiedene Arten initialisieren.
z.B.

```
public Uhrenanzeige()
```

Die Anzeige wird implizit auf 00:00 gesetzt.

```
public Uhrenanzeige(int stunde, int minute)
```

Die Anzeige wird explizit auf die übergebenen Werte gesetzt.
Es ist klar, dass alle Konstruktoren den selben Namen haben, nämlich den der Klasse.
Ihre Signaturen müssen sich in der Parameterliste unterscheiden.
Wenn es mehrere Konstruktoren gibt, spricht man vom **Überladen** des Konstruktors.
Das gleiche Prinzip ist auch bei allen anderen Methoden möglich:
Eine Methode kann dadurch **überladen** werden, dass weitere Methoden
- des gleichen Namens aber
- unterschiedlicher Parameterliste

definiert werden.

Java unterscheidet Methoden nicht allein aufgrund ihres Namens, sondern aufgrund ihrer Signatur.

5.8 Methodenaufrufe

5.8.1 Interne Methodenaufrufe

Allgemein bewirkt ein Methodenaufruf ***methodenname (parameterliste)*** folgendes:
- Die Werte der aktuellen Parameter werden den formalen Parametern zugewiesen.
- Danach werden die Anweisungen der entsprechenden Methode ausgeführt.
- Danach wird zur nächsten Anweisung *hinter* dem Methodenaufruf zurückgekehrt.

5.8.2 Externe Methodenaufrufe

Allgemeine Syntax des **externen** Methodenaufrufs (Punktnotation):

```
objekt . methodenname ( parameterliste )
```

Vor dem Punkt steht der Name des **Objekts**, für das die Methode augferufen wird.
Siehe Beispiel oben:

5.9 Das Schlüsselwort this

Mit `this` spricht man in einer Methode das jeweilige Objekt an, für das die Methode gerade aufgerufen wurde.

5.10 Programmtest mit dem Debugger

Debugger: Eine interaktive Testhilfe
Haltepunkte setzen
Einzelne Anweisung ausführen
Methode als eine Anweisung betrachten (Schritt über)
In eine Methode springen (Schritt hinein)

6 Klassenbibliothek

6.1 Dokumentation von Klassen

Die Dokumentation einer Klasse sollte genau die Information enthalten,
die ein **anderer** Programmierer braucht,
um die Klasse **anwenden** zu können,
ohne den Quellcode lesen zu müssen.
Diese Dokumentation beschreibt die Schnittstelle (engl. interface) der Klasse,
deshalb auch die Bezeichnung **application programmer interface** Dokumentation bei der Java-Klassenbibliothek,
kurz API-Dokumentation.
Das Java SDK enthält das Programm **javadoc**, das die Dokumentation automatisch aus dem Quellcode erzeugen kann. Voraussetzung: man muss einige Konventionen bei den Kommentaren beachten.
In Bluej wird javadoc intern benutzt:
- im Quellcodefenster von *implementation* auf *interface* wechseln, oder
- im Menu Werkzeuge "Dokumentation erzeugen" auswählen.

Elemente einer Klassendokumentation:

Klassenname
Zweck und Eigenschaften der Klasse (Kommentar)
Versionsnummer (@version)
Autor (@author)
Beschreibung der Konstuktoren und Methoden.
Dokumentation von Methoden (incl. Konstruktoren)
Name der Methode
Ergebnistyp (außer bei Konstruktoren)
Namen und Typen der Parameter
Zweck und Arbeitsweise der Methode (Kommentar)
Beschreibung eines jeden Parameters (@param)
Beschreibung des Ergebnisses (@result)
Zusätzlich sollte jedes Projekt einen Projektkommentar enthalten, in der Datei "ReadMe.txt"
Kommentare zur Auswertung durch javadoc beginnen mit /** und enden mit */:

Beispiel:

```
/**
 * Klassennamensbeschreibung
 *
 * @author (Jens)
 * @version (1)
 */
public class Test
{
   // Kommentar der nicht erkannt wird nur eine Zeile
   private int x;
   /*
    *Kommentar der nicht erkannt wird, über mehrere Zeilen
    *
    */


   /**
    * Kommentar vor dem Konstructor
    */
   public Test()
   {
      // Kommentar der nicht erkannt wird nur eine Zeile
      x = 0;
   }

   /**
    *Kommentar für eine Methode
    */
   public int sampleMethod(int y)
   {

      return x + y;
   }
}
```

Class Test

```
java.lang.Object
  |
  +--Test
```

public class **Test**
extends java.lang.Object

Klassennamensbeschreibung

Version:
(1)
Author:
(Jens)

Constructor Summary

Test() Kommentar vor dem Konstructor	

Method Summary

int	**sampleMethod** (int y) Kommentar für eine Methode

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Test

`public Test()`

Kommentar vor dem Konstructor

Method Detail

sampleMethod

`public int sampleMethod(int y)`

Kommentar für eine Methode

6.2 Laden einer Klasse aus der Klassenbibliothek

Die gesamte Bibliothek ist unterteilt in thematisch zusammengehörende Pakete (engl. ***package***).
Das vereinfacht für den Programmierer und für den Compiler das Auffinden der Klassen.
Damit eine Klasse verwendet werden kann, muß angegeben werden, in welchem Paket sie liegt. Hierzu gibt es zwei unterschiedliche Möglichkeiten:

a) **Die Klasse wird über ihren vollen (qualifizierten) Namen angesprochen**:

```
java.util.Date d = new java.util.Date();
```

b) **Am Anfang des Programms werden die gewünschten Klassen mit Hilfe einer `import`-Anweisung eingebunden**:

Die `import`-Anweisung gibt es in zwei unterschiedlichen Ausprägungen:

1) Gesamtes Packet:

```
import java.util.*;
...
Date d = new Date();
```

2) Mit der Syntax `import Paket.Klasse;` wird genau eine Klasse importiert. Alle anderen Klassen des Pakets bleiben unsichtbar:

```
import java.util.Date;
...
Date d = new Date();
```

6.3 Wichtige Klassen

6.3.1 String

Method Summary	
char	**charAt**(int index) Returns the character at the specified index.
int	**compareTo**(Object o) Compares this String to another Object.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(String str) Concatenates the specified string to the end of this string.

boolean	**contentEquals**(StringBuffer sb) Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
static String	**copyValueOf**(char[] data) Returns a String that represents the character sequence in the array specified.
static String	**copyValueOf**(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified.
boolean	**endsWith**(String suffix) Tests if this string ends with the specified suffix.
boolean	**equals**(Object anObject) Compares this string to the specified object.
boolean	**equalsIgnoreCase**(String anotherString) Compares this String to another String, ignoring case considerations.
byte[]	**getBytes**() Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
void	**getBytes**(int srcBegin, int srcEnd, byte[] dst, int dstBegin) **Deprecated.** *This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the the* getBytes() *method, which uses the platform's default charset.*
byte[]	**getBytes**(String charsetName) Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
void	**getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array.
int	**hashCode**() Returns a hash code for this string.
int	**indexOf**(int ch) Returns the index within this string of the first occurrence of the specified character.
int	**indexOf**(int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
int	**indexOf**(String str) Returns the index within this string of the first occurrence of the specified substring.
int	**indexOf**(String str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
String	**intern**() Returns a canonical representation for the string object.
int	**lastIndexOf**(int ch) Returns the index within this string of the last occurrence of the

	specified character.
int	**lastIndexOf**(int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int	**lastIndexOf**(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
int	**lastIndexOf**(String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int	**length**() Returns the length of this string.
boolean	**matches**(String regex) Tells whether or not this string matches the given regular expression.
boolean	**regionMatches**(boolean ignoreCase, int toffset, String other, int ooffset, int len) Tests if two string regions are equal.
boolean	**regionMatches**(int toffset, String other, int ooffset, int len) Tests if two string regions are equal.
String	**replace**(char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
String	**replaceAll**(String regex, String replacement) Replaces each substring of this string that matches the given regular expression with the given replacement.
String	**replaceFirst**(String regex, String replacement) Replaces the first substring of this string that matches the given regular expression with the given replacement.
String[]	**split**(String regex) Splits this string around matches of the given regular expression.
String[]	**split**(String regex, int limit) Splits this string around matches of the given regular expression.
boolean	**startsWith**(String prefix) Tests if this string starts with the specified prefix.
boolean	**startsWith**(String prefix, int toffset) Tests if this string starts with the specified prefix beginning a specified index.
CharSequence	**subSequence**(int beginIndex, int endIndex) Returns a new character sequence that is a subsequence of this sequence.
String	**substring**(int beginIndex) Returns a new string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a new string that is a substring of this string.

char[]	**toCharArray**() Converts this string to a new character array.
String	**toLowerCase**() Converts all of the characters in this String to lower case using the rules of the default locale.
String	**toLowerCase**(Locale locale) Converts all of the characters in this String to lower case using the rules of the given Locale.
String	**toString**() This object (which is already a string!) is itself returned.
String	**toUpperCase**() Converts all of the characters in this String to upper case using the rules of the default locale.
String	**toUpperCase**(Locale locale) Converts all of the characters in this String to upper case using the rules of the given Locale.
String	**trim**() Returns a copy of the string, with leading and trailing whitespace omitted.
static String	**valueOf**(boolean b) Returns the string representation of the boolean argument.
static String	**valueOf**(char c) Returns the string representation of the char argument.
static String	**valueOf**(char[] data) Returns the string representation of the char array argument.
static String	**valueOf**(char[] data, int offset, int count) Returns the string representation of a specific subarray of the char array argument.
static String	**valueOf**(double d) Returns the string representation of the double argument.
static String	**valueOf**(float f) Returns the string representation of the float argument.
static String	**valueOf**(int i) Returns the string representation of the int argument.
static String	**valueOf**(long l) Returns the string representation of the long argument.
static String	**valueOf**(Object obj) Returns the string representation of the Object argument

6.3.2 StringTokenizer

```
java.lang.Object
  |
  +--java.util.StringTokenizer
```

Ziel: Einen Eingabesatz in seine Wörter zerlegen und als **Menge** von Wörtern speichern.
Die Wörter eines Satzes werden in der Theorie der Programmiersprachen auch **token** genannt.

Bedeutung: kleinste interessierende Einheit.
Es gibt ein künstliches englisches Fachwort: **to tokenize** = in Enheiten (token) zerlegen.
Einer der zerlegt, ist ein **tokenizer** (deutsch: ein Zerleger, ein Parser).
Zum Zerlegen von Texten, also von Strings, gibt es die Bibliotheksklasse ***StringTokenizer***.
Ein ***StringTokenizer-***Objekt liefert die Token eines Strings
ähnlich wie ein Iterator die Objekte einer Sammlung:

für Sammlungen: ***Iterator*** hasNext() next()	für Texte: ***StringTokenizer*** hasMoreTokens() nextToken()

Damit kann man die *gibEingabe*-Methode der Klasse Eingabeleser so umschreiben,
dass sie als Ergebnis die Menge der Wörter des Satzes zurückgibt:

Constructor Summary	
StringTokenizer(String str) Constructs a string tokenizer for the specified string.	

Method Summary	
int	**countTokens**() Calculates the number of times that this tokenizer's `nextToken` method can be called before it generates an exception.
boolean	**hasMoreElements**() Returns the same value as the `hasMoreTokens` method.
boolean	**hasMoreTokens**() Tests if there are more tokens available from this tokenizer's string.
Object	**nextElement**() Returns the same value as the `nextToken` method, except that its declared return value is `Object` rather than `String`.
String	**nextToken**() Returns the next token from this string tokenizer.
String	**nextToken**(String delim) Returns the next token in this string tokenizer's string.

6.3.3 Random

```
java.lang.Object
  |
  +--java.util.Random
```

In der Java-Klassenbibliothek gibt es eine Klasse, die einen Zufallszahlengenerator darstellt.
Die Klassse ***Random*** enthält Methoden, die **Pseudo-Zufallszahlen** liefern.
(random, engl. für zufällig)
Warum nur "pseudo"-zufällig?
Ein Computer arbeitet hoffentlich immer deterministisch. Der Zufall ist geplant!
Siehe API-Dokumentation der Klasse ***Random*** im Package java.util
Um eine Zufallszahl zu erzeugen muss man
- eine Instanz von *Random* erzeugen und
- für diese Instanz eine Methode aurufen, die die Zufallszahl zurückgibt.

Beispiel:

```
random zufallsGenerator;

zufallsgenerator = new Random();

int index = zufallsGenerator.nextint();
```

Diese Zufallszahlen liegen im Wertebereich von int-Werten: -2 147 483 648 bis +2 147 483 647
Zufallszahlen mit eingeschränktem Wertebereich:
Die Methode ***nextInt(int max)*** der Random-Klasse liefert **0 <= x < max**.
Mit einer Zufallszahl kann man eine zufällige Antwort generieren,

Constructor Summary
Random()   Creates a new random number generator.
Random(long seed)   Creates a new random number generator using a single `long` seed:

Method Summary	
protected int	**next**(int bits) Generates the next pseudorandom number.
boolean	**nextBoolean**() Returns the next pseudorandom, uniformly distributed `boolean` value from this random number generator's sequence.
void	**nextBytes**(byte[] bytes) Generates random bytes and places them into a user-supplied byte array.
double	**nextDouble**()

	Returns the next pseudorandom, uniformly distributed `double` value between `0.0` and `1.0` from this random number generator's sequence.
`float`	**nextFloat**() Returns the next pseudorandom, uniformly distributed `float` value between `0.0` and `1.0` from this random number generator's sequence.
`double`	**nextGaussian**() Returns the next pseudorandom, Gaussian ("normally") distributed `double` value with mean `0.0` and standard deviation `1.0` from this random number generator's sequence.
`int`	**nextInt**() Returns the next pseudorandom, uniformly distributed `int` value from this random number generator's sequence.
`int`	**nextInt**(int n) Returns a pseudorandom, uniformly distributed `int` value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.
`long`	**nextLong**() Returns the next pseudorandom, uniformly distributed `long` value from this random number generator's sequence.
`void`	**setSeed**(long seed) Sets the seed of this random number generator using a single `long` seed.

6.3.4 Wrapper-Klassen

Die Elemente einer Sammlung können beliebige Objekte, jedoch keine primitiven Typen wie z.B. *int* oder *double* sein. Will man beispielsweise ganzzahlige Werte in einer *ArrayList* speichern, hat man ein Problem.
Abhilfe:
Es gibt zu jedem primitiven Typ eine Klasse in der Klassenbibliothek; diese Klasse enthält als Datenfeld eine Variable des primitiven Typs. Auf diese Weise wird die primitve Variable in ein Object "eingepackt" (engl. to wrap).
Die entsprechenden Klassen nennt man deshalb **Wrapper-Klassen:**

primitiver Typ:	Wrapper-Klasse:
int	**Integer**
float	**Float**
double	**Double**
char	**Char**
boolean	**Boolean**
byte	**Byte**

Beispiel zum Einfügen in eine Sammlung (Integer):
Ein *Integer*-Objekt enthält einen *int*-Wert und kann in eine Sammlung aufgenommen werden.

```
int i = 18;
Integer iwrap = new Integer(i);
...
meineSammlung.add(iwrap);
...

oder

meineSammlung.add(new Interger(i));
```

Beispiel zum Umwandeln von Interger in eine int Zahl

```
Integer element = (Integer) meineSammlung.get(0);
int wert = element.intValue();
```

Die Wrapper-Klassen enthalten weitere nützliche Methoden: Siehe Java API-Dokumentation.
Besonders nützlich: Methoden zur Wandlung von Text in einen numerischen Wert.

Beispiel: Wandlung von Text in einen Integer-Wert:

```
String xyz = "4711";
int i;
i = Integer.parseInt(xyz);
```

Entsprechend funktionieren die Klassenmethoden *Double.parseDouble, Float.parseFloat, Byte.parseByte*.

6.3.5 Math

Method Summary	
static double	**abs**(double a) Returns the absolute value of a `double` value.
static float	**abs**(float a) Returns the absolute value of a `float` value.
static int	**abs**(int a) Returns the absolute value of an `int` value.
static long	**abs**(long a) Returns the absolute value of a `long` value.
static double	**acos**(double a) Returns the arc cosine of an angle, in the range of 0.0 through *pi*.

static double	**asin**(double a) Returns the arc sine of an angle, in the range of -*pi*/2 through *pi*/2.
static double	**atan**(double a) Returns the arc tangent of an angle, in the range of -*pi*/2 through *pi*/2.
static double	**atan2**(double y, double x) Converts rectangular coordinates (x, y) to polar (r, *theta*).
static double	**ceil**(double a) Returns the smallest (closest to negative infinity) double value that is not less than the argument and is equal to a mathematical integer.
static double	**cos**(double a) Returns the trigonometric cosine of an angle.
static double	**exp**(double a) Returns Euler's number *e* raised to the power of a double value.
static double	**floor**(double a) Returns the largest (closest to positive infinity) double value that is not greater than the argument and is equal to a mathematical integer.
static double	**IEEEremainder**(double f1, double f2) Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
static double	**log**(double a) Returns the natural logarithm (base *e*) of a double value.
static double	**max**(double a, double b) Returns the greater of two double values.
static float	**max**(float a, float b) Returns the greater of two float values.
static int	**max**(int a, int b) Returns the greater of two int values.
static long	**max**(long a, long b) Returns the greater of two long values.
static double	**min**(double a, double b) Returns the smaller of two double values.
static float	**min**(float a, float b) Returns the smaller of two float values.
static int	**min**(int a, int b) Returns the smaller of two int values.
static long	**min**(long a, long b) Returns the smaller of two long values.
static double	**pow**(double a, double b) Returns of value of the first argument raised to the power of the second argument.
static double	**random**() Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
static double	**rint**(double a)

	Returns the `double` value that is closest in value to the argument and is equal to a mathematical integer.
static long	**round**(double a) Returns the closest `long` to the argument.
static int	**round**(float a) Returns the closest `int` to the argument.
static double	**sin**(double a) Returns the trigonometric sine of an angle.
static double	**sqrt**(double a) Returns the correctly rounded positive square root of a `double` value.
static double	**tan**(double a) Returns the trigonometric tangent of an angle.
static double	**toDegrees**(double angrad) Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
static double	**toRadians**(double angdeg) Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

6.4 Wichtige Interface

6.4.1 Comparable

Das Interface *Comparable*, enthält nur die Methodendeklaration:

```
int compareTo(Object o);
```

Das "*this*"-Objekt wird mit dem als Parameter angegebenen Objekt auf seine Ordnung verglichen und zurückgegeben.

Returnwert < 0 : *this* ist kleiner als o
Returnwert > 0 : *this* ist größer als o
Returnwert = 0 : *this* ist gleich o

Beispiel/ Syntax:

Allgemein:

```
private Objekt object;

public int compareTo(Object object)
{
```

```
 return  this.object .compareTo(object) ;
 }
```

Für Alphabetische Sortierung:

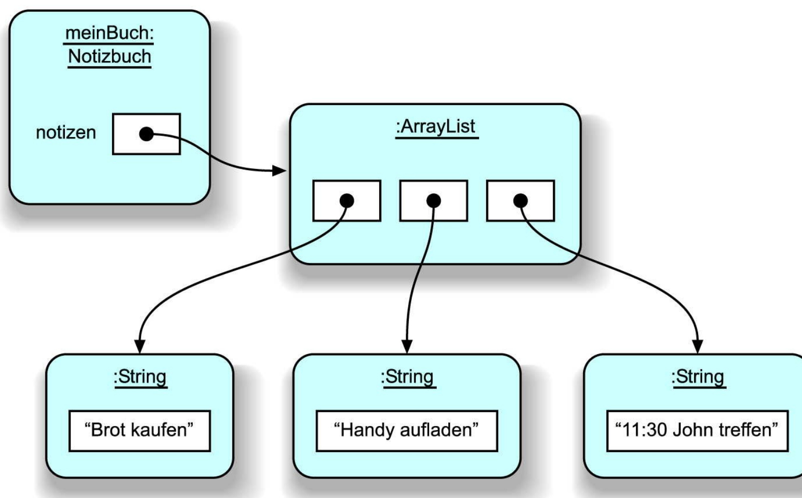
```
String name;

public int compareTo(Object object)
{
 return  name.compareTo(object) ;
 }
```

7 Objektsammlungen

In den meisten Programmen wird eine größere Anzahl von Objekten benötigt.
Es wichtig, diese Objekte zu einer Sammlung (Engl. collection) zusammenfassen zu können.

7.1 *Objektstrukturen mit Sammlungen*

Ein Klasesen-Objekt enthält eine Referenz auf ein Sammlungs-Objekt,
und erst dieses enthält Referenzen auf die eigentlichen Instanzen, welche gehalten werden sollen. Am Beispiel einer ArrayList:



7.2 *Sammlungen mit fester Größe (Arrays)*

Bei manchen Aufgabenstellungen ist
- die Anzahl der Elemente einer Sammlung von vornherein bekannt und
- ändert sich nicht mehr während der Lebensdauer der Sammlung.

Eine Sammlung mit fester Größe heißt Array. Sie hat mindestens folgende Vorteile:
- Der Zugriff auf die Elemente erfolgt viel schneller, also viel effizienter.
- In *Arrays können nicht nur Objekte, sondern auch primitive Typen* gespeichert werden.

Für den Zugriff auf Array-Elemente gibt es eine spezielle Syntax in Java, die es vergleichbar in allen Programmiersprachen gibt.

7.2.1 Array-Objekte deklarieren und Erzeugen

Durch die Deklaration einer Array-Variablen wird noch kein Objekt erzeugt, sondern erst durch eine ***new***-Anweisung, wie bei anderen Objekten auch!

Allgemeine Form für die Deklaration eines Array-Objekts:

```
    private Typ[] Instanzname;
```

Allgemeine Form für die Erzeugung eines Array-Objekts:

```
new Typ[int-Ausdruck]
```

Dabei ist

Typ	der Basistyp der Elemente
int-Ausdruck	die Anzahl der Elemente

Der Basistyp kann ein beliebiger Typ sein: ein primitiver Typ wie im obigen Beispiel, oder ein Klassentyp, wie z.B.

7.2.2 Array-Objekte benutzen

7.2.2.1 Direkter Zugriff

Auf die einzelnen Elemente greift man mit einem Index zu.
Der Index ist ein ganzzahliger Ausdruck in eckigen Klammern.
Beispiele:

```
zugriffeInStunde[12]
automaten[9]
studenten[3*i+k]
```

Fehlermöglichkeit:
Indizes beginnen bei 0. Der größte Index ist um eins kleiner als die Größe des Arrays. Verwendet man einen Index außerhalb dieses gültigen Bereichs, führt das zu einer **ArrayIndexOutOfBoundsException**.
Ausdrücke, die ein Element aus einem Array wählen, können an allen Stellen stehen, an denen auch Werte des Basistyps des Arrays stehen dürfen.

7.2.2.2 Länge des Arrays

Länge des Arrays:

```
int laenge = name.length;
```

7.3 *Objektsammlungen mit flexibler Größe*

Oft ist die Anzahl der Objekte einer Sammlung zu Beginn noch nicht bekannt oder kann sich später noch ändern.

Wir wollen uns zunächst mit der grundlegenden Arbeitsweise der Collection-Typen vertraut machen:

> Eine **`List`** ist eine beliebig große Liste von Elementen beliebigen Typs, auf die sowohl wahlfrei als auch sequentiell zugegriffen werden kann.
>
> Eine **`Map`** ist eine Abbildung von Elementen eines Typs auf Elemente eines anderen Typs, also eine Menge zusammengehöriger Paare von Objekten.
>
> Ein **`Set`** ist eine (doublettenlose) Menge von Elementen, auf die mit typischen Mengenoperationen zugegriffen werden kann.

Achtung Primitive Typen, müssen in Wrapper Klassen gepackt werden!

7.3.1 List

7.3.1.1 ArrayList

Drei Eigenschaften einer ArrayList fallen auf:
- Es kann seine Kapazität vergrößern um weitere Objekte zu speichern.
- Es führt Buch über Anzahl der aktuell gespeicherten Werte.
- Es behält die Reihenfolge der gespeicherten Objekte bei.

Syntax

Referenzvariable deklarieren und erzeugen:

ArrayList liste = new ArrayList();

Method Summary	
`void`	**`add`**`(int index, Object element)` Inserts the specified element at the specified position in this list.
`boolean`	**`add`**`(Object o)` Appends the specified element to the end of this list.
`boolean`	**`addAll`**`(Collection c)` Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection's Iterator.
`boolean`	**`addAll`**`(int index, Collection c)`

	Inserts all of the elements in the specified Collection into this list, starting at the specified position.
`void`	**`clear`**`()` Removes all of the elements from this list.
`Object`	**`clone`**`()` Returns a shallow copy of this `ArrayList` instance.
`boolean`	**`contains`**`(Object elem)` Returns `true` if this list contains the specified element.
`void`	**`ensureCapacity`**`(int minCapacity)` Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
`Object`	**`get`**`(int index)` Returns the element at the specified position in this list.
`int`	**`indexOf`**`(Object elem)` Searches for the first occurence of the given argument, testing for equality using the `equals` method.
`boolean`	**`isEmpty`**`()` Tests if this list has no elements.
`int`	**`lastIndexOf`**`(Object elem)` Returns the index of the last occurrence of the specified object in this list.
`Object`	**`remove`**`(int index)` Removes the element at the specified position in this list.
`protected void`	**`removeRange`**`(int fromIndex, int toIndex)` Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
`Object`	**`set`**`(int index, Object element)` Replaces the element at the specified position in this list with the specified element.
`int`	**`size`**`()` Returns the number of elements in this list.
`Object[]`	**`toArray`**`()` Returns an array containing all of the elements in this list in the correct order.
`Object[]`	**`toArray`**`(Object[] a)` Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.
`void`	**`trimToSize`**`()` Trims the capacity of this `ArrayList` instance to be the list's current size.

7.3.1.2 LinkedList

```
java.lang.Object
  |
  +--java.util.AbstractCollection
        |
        +--java.util.AbstractList
              |
              +--java.util.AbstractSequentialList
                    |
                    +--java.util.LinkedList
```

ähnlich verwendbar wie *ArrayList*, ist verkettete Liste,
erlaubt einfügen auch am Anfang und Ende:

```
void addFirst(Object o)
void addLast(Object o)
Object getFirst()
Object getLast()
```

Constructor Summary
LinkedList()   Constructs an empty list.
LinkedList(Collection c)   Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Method Summary	
void	**add**(int index, Object element) Inserts the specified element at the specified position in this list.
boolean	**add**(Object o) Appends the specified element to the end of this list.
boolean	**addAll**(Collection c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean	**addAll**(int index, Collection c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**addFirst**(Object o) Inserts the given element at the beginning of this list.
void	**addLast**(Object o)

	Appends the given element to the end of this list.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this `LinkedList`.
boolean	**contains**(Object o) Returns `true` if this list contains the specified element.
Object	**get**(int index) Returns the element at the specified position in this list.
Object	**getFirst**() Returns the first element in this list.
Object	**getLast**() Returns the last element in this list.
int	**indexOf**(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
int	**lastIndexOf**(Object o) Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
ListIterator	**listIterator**(int index) Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
Object	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(Object o) Removes the first occurrence of the specified element in this list.
Object	**removeFirst**() Removes and returns the first element from this list.
Object	**removeLast**() Removes and returns the last element from this list.
Object	**set**(int index, Object element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.
Object[]	**toArray**() Returns an array containing all of the elements in this list in the correct order.
Object[]	**toArray**(Object[] a) Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.

7.3.1.3 Stack

```
java.lang.Object
  |
  +--java.util.AbstractCollection
        |
        +--java.util.AbstractList
              |
              +--java.util.Vector
                    |
                    +--java.util.Stack
```

Stapelspeicher, Kellerspeicher, "last in, first out"

Constructor Summary
Stack() Creates an empty Stack.

Method Summary	
boolean	**empty**() Tests if this stack is empty.
Object	**peek**() Looks at the object at the top of this stack without removing it from the stack.
Object	**pop**() Removes the object at the top of this stack and returns that object as the value of this function.
Object	**push**(Object item) Pushes an item onto the top of this stack.
int	**search**(Object o) Returns the 1-based position where an object is on this stack.

7.3.2 Map Abbildungen

7.3.2.1 Hash Map

7.3.2.1.1 Abbildungen und Map-Klassen

Eine Abbildung, engl. Map, wird in Java realisiert durch die Java-Bibliotheksklasse ***Map***.
Ein ***Map***-Objekt ist eine **Sammlung** von Schlüssel-Wert-Paaren.
Sowohl die Schlüssel als auch die Werte einer *Map* sind Objekte.
Vergleich *ArrayList* <-> *Map*

ArrayList:	Map:
- Sammlung von Objekten - beliebige Anzahl von Einträgen - Zugriff auf Wert über Index - Index wird auf Wert abgebildet	- Sammlung von Objektpaaren - beliebige Anzahl von Einträgen aus jeweils Schlüsselobjekt und Wertobjekt - Zugriff auf Wert über Schlüssel - Schlüssel wird auf Wert abgebildet

Konkretes Beispiel für eine Map: Telefonbuch.
Einträge sind Paare aus Name (Schlüssel) und Telefonnummer (Wert).

Eine ***HashMap*** ist eine spezielle Implementierung einer einer ***Map*** .

Die Referenzvariable antwortMap deklarieren:

```
private HashMap antwortMap;
```

Eine Instanz von HashMap erzeugen:

```
antwortMap = new HashMap();
```

Method Summary	
void	**clear**() Removes all mappings from this map.
Object	**clone**() Returns a shallow copy of this `HashMap` instance: the keys and values themselves are not cloned.
boolean	**containsKey**(Object key) Returns `true` if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns `true` if this map maps one or more keys to the specified value.
Set	**entrySet**() Returns a collection view of the mappings contained in this map.

Object	**get**(Object key) Returns the value to which the specified key is mapped in this identity hash map, or `null` if the map contains no mapping for this key.
boolean	**isEmpty**() Returns `true` if this map contains no key-value mappings.
Set	**keySet**() Returns a set view of the keys contained in this map.
Object	**put**(Object key, Object value) Associates the specified value with the specified key in this map.
void	**putAll**(Map t) Copies all of the mappings from the specified map to this map These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
Object	**remove**(Object key) Removes the mapping for this key from this map if present.
int	**size**() Returns the number of key-value mappings in this map.
Collection	**values**() Returns a collection view of the values contained in this map.

7.3.2.2 TreeMap

```
java.lang.Object
  |
  +--java.util.AbstractMap
        |
        +--java.util.TreeMap
```

Map, ähnlich *HashMap*, aber nach den Schlüsseln aufsteigend sortiert.
Die Schlüssel müssen das *Comparable* Interface implementieren.

TreeMap(Map m) Konstruktor,

erzeugt aus den Schlüssel/Wert-Paaren von ***m*** eine **sortierte** Map.

Object put(Object key, Object value)

sortiert das Paar *key/value* gemäß ***key*** in die Map ein

Object firstKey()

Object lastKey()

Constructor Summary
TreeMap() Constructs a new, empty map, sorted according to the keys' natural order.
TreeMap(Comparator c) Constructs a new, empty map, sorted according to the given comparator.
TreeMap(Map m) Constructs a new map containing the same mappings as the given map, sorted according to the keys' *natural order*.

TreeMap(SortedMap m)
Constructs a new map containing the same mappings as the given `SortedMap`, sorted according to the same ordering.

Method Summary

void	**clear**() Removes all mappings from this TreeMap.
Object	**clone**() Returns a shallow copy of this `TreeMap` instance.
Comparator	**comparator**() Returns the comparator used to order this map, or `null` if this map uses its keys' natural order.
boolean	**containsKey**(Object key) Returns `true` if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns `true` if this map maps one or more keys to the specified value.
Set	**entrySet**() Returns a set view of the mappings contained in this map.
Object	**firstKey**() Returns the first (lowest) key currently in this sorted map.
Object	**get**(Object key) Returns the value to which this map maps the specified key.
SortedMap	**headMap**(Object toKey) Returns a view of the portion of this map whose keys are strictly less than `toKey`.
Set	**keySet**() Returns a Set view of the keys contained in this map.
Object	**lastKey**() Returns the last (highest) key currently in this sorted map.
Object	**put**(Object key, Object value) Associates the specified value with the specified key in this map.
void	**putAll**(Map map) Copies all of the mappings from the specified map to this map.
Object	**remove**(Object key) Removes the mapping for this key from this TreeMap if present.
int	**size**() Returns the number of key-value mappings in this map.
SortedMap	**subMap**(Object fromKey, Object toKey) Returns a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive.
SortedMap	**tailMap**(Object fromKey) Returns a view of the portion of this map whose keys are greater than or

	equal to `fromKey`.
`Collection`	`values()` Returns a collection view of the values contained in this map.

7.3.3 Set - Menge

7.3.3.1 HashSet

Definition von Menge
Eine Menge ist eine Zusammenfassung von wohlbestimmten und wohlunterschiedenen Dingen zu einem Ganzen.
Eine Menge (engl. **set**) ist in Java-Terminologie eine einfache Form einer Sammlung (engl. collection).
Es gibt verschiedene Java-Bibliotheks-Klassen zur Darstellung von Mengen.
In Java sind die Elemente von Mengen Objekte.
Wir verwenden hier die Klasse **`HashSet`**.

Syntax

```
HashSet meineMenge;            // Referenzvariable deklarieren

meineMenge = new HashSet();//  eine Instanz von HashSet erzeugen
```

Liste:	Menge:
- Die Elemente haben eine feste Reihenfolge. - Zugriff auf Elemente über Index möglich. - Ein Element kann mehrfach enthalten sein.	- Keine definierte Reihenfolge. - Zugriff auf Elemente nur über Itarator. - Jedes Element ist höchstens einmal enthalten.

Das Interface *Comparable*, enthält nur die Methodendeklaration

```
int compareTo(Object o);
```

vergleicht das "*this*"-Objekt mit dem als Parameter angegebenen Objekt auf ihre Ordnung.

Returnwert < 0 : *this* ist kleiner als o
Returnwert > 0 : *this* ist größer als o
Returnwert = 0 : *this* ist gleich o

Method Summary	
`boolean`	`add(Object o)` Adds the specified element to this set if it is not already present.

void	**clear**() Removes all of the elements from this set.
Object	**clone**() Returns a shallow copy of this `HashSet` instance: the elements themselves are not cloned.
boolean	**contains**(Object o) Returns `true` if this set contains the specified element.
boolean	**isEmpty**() Returns `true` if this set contains no elements.
Iterator	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present.
int	**size**() Returns the number of elements in this set (its cardinality).

7.3.3.2 TreeSet

```
java.lang.Object
  |
  +--java.util.AbstractCollection
        |
        +--java.util.AbstractSet
              |
              +--java.util.TreeSet
```

Aufsteigend sortierte Menge

TreeSet(Collection c) Konstruktor,

erzeugt aus den Elementen der Sammlung c eine **sortierte** Menge
Elemente von c müssen das *Comparable* Interface implementieren.

boolean add(Object o)

sortiert das Objekt o in die Menge ein

Object first()

SortedSet tailSet(Object fromElement)

Teilmenge der Elemente hinter *fromElement*. (tail =Schwanz)

Constructor Summary
TreeSet() Constructs a new, empty set, sorted according to the elements' natural order.
TreeSet(Collection c) Constructs a new set containing the elements in the specified collection, sorted according to the elements' *natural order*.
TreeSet(Comparator c)

Constructs a new, empty set, sorted according to the specified comparator.
TreeSet(SortedSet s) Constructs a new set containing the same elements as the specified sorted set, sorted according to the same ordering.

Method Summary

boolean	**add**(Object o) Adds the specified element to this set if it is not already present.
boolean	**addAll**(Collection c) Adds all of the elements in the specified collection to this set.
void	**clear**() Removes all of the elements from this set.
Object	**clone**() Returns a shallow copy of this `TreeSet` instance.
Comparator	**comparator**() Returns the comparator used to order this sorted set, or `null` if this tree set uses its elements natural ordering.
boolean	**contains**(Object o) Returns `true` if this set contains the specified element.
Object	**first**() Returns the first (lowest) element currently in this sorted set.
SortedSet	**headSet**(Object toElement) Returns a view of the portion of this set whose elements are strictly less than `toElement`.
boolean	**isEmpty**() Returns `true` if this set contains no elements.
Iterator	**iterator**() Returns an iterator over the elements in this set.
Object	**last**() Returns the last (highest) element currently in this sorted set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present.
int	**size**() Returns the number of elements in this set (its cardinality).
SortedSet	**subSet**(Object fromElement, Object toElement) Returns a view of the portion of this set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive.
SortedSet	**tailSet**(Object fromElement) Returns a view of the portion of this set whose elements are greater than or equal to `fromElement`.

7.4 Iterator

In Programmen kommt es sehr häufig vor, dass man alle Elemente einer Sammlung nacheinander bearbeiten muss.
Man sagt auch, über alle Elemente **iterieren**. Bei Sammlungen, welche keine Reihenfolge aufweisen, ist dies die einzige Möglichkeit.
Es gibt deshalb eine explizite Möglichkeit, über alle Elemente einer Sammlung zu iterieren.

```
Iterator it = meineSammlung.iterator();
```

Ein solches ***Iterator***-Objekt bietet zwei Methoden an, um über die Elemente einer Sammlung zu iterieren:

Method Summary	
boolean	**hasNext**() Returns true if the iteration has more elements.
Object	**next**() Returns the next element in the iteration.
void	**remove**() Removes from the underlying collection the last element returned by the iterator (optional operation).

Damit kann man wie folgt über alle Elemente iterieren, hier allgemein als Pseudocode:

```
Iterator it = meineSammlung.iterator();
while (it.hasNext()) {
    mit it.next() das nächste Element abholen
    etwas mit diesem Objekt tun
}
```

Bemerkenswert: Es wird hier kein Index benötigt, um auf die Elemente zuzugreifen.
Will man die Elemente in der Ausgabe nummerieren, kann man eine Zählvariable i einführen:

```
    int i = 0;
    Iterator it = notitzen.iterator();
    while (it.hasNext()) {
        System.out.println(i+": "+it.next());
```

```
            i++;
        }
```

Bei Verwendung von *Iterator* muss man die Definition am Programmanfang erst importieren.

```
import java.util.Iterator;
```

Zugriff mit Index oder Iterator?
Es gibt andere Sammlungs-Typen, bei denen kein Index zur Verfügung steht.
Der Iterator ist die allgemeinere Zugriffsart auf Sammlungselemente.

7.5 Der Cast-Operator

Das Problem liegt in der Universalität von Sammlungen: Sie können alle möglichen Objekte speichern.
Der Rückgabetyp von *get()* und *next()* ist allgemein *Object* !
Man muss dem Compiler deshalb explizit sagen, von welchem Typ das aus der Sammlung stammende Objekt ist.
Beispiel:

```
Klassenname Variable;
Variable = (Klassenname) Intanz.get(1);
```

oder

```
Klassenname Variable;
while (it.hasNext()) {
   Variable = (Klassenname) it.next();
}
```

Da der Rückgabewert vom Typ `Object` ist, muß er in der Regel in den tatsächlich erforderlichen Typ konvertiert werden. Hierbei kann ein Fehler auftreten (Falscher Typ) der eine Exception von Typ `ClassCastException` auslöst.

dem cast-Operator. (cast, engl. für formen)
Er besteht aus einem Typname in runden Klammern.
Er "wandelt" den Typ des Objekts, vor dem er steht, entsprechend um.
Der cast-Operator muss in Verbindung mit Sammlungen häufig benutzt werden.

7.6 Die null-Referenz

Solange eine Referenzvariable noch keine Referenz auf ein konkretes Objekt enthält, referiert sie garnichts, ihr Wert ist **`null`**. null ist hier nicht der numerische Wert 0 sondernd ein Java-Schlüsselwort, das die **Nullreferenz** bezeichnet.
Datenfelder, die Referenzvariablen sind, d.h. die einen Klassentyp haben, werden vom Compiler automatisch mit der Nullreferenz *null* initialisiert.
Referenzvariablen kann man explizit auf *null* abfragen, z.B.

```
    if (notizen == null) {
        notizen = new ArrayList();
    }
```

Der Sinn dieser Anweisungen könnte sein:
"Wenn die Referenzvariable *notizen* schon ein *ArrayList*-Objekt referenziert, dann nehmen wir dieses, ansonsten erzeugen wir ein neues *ArrayList*-Objekt und referenzieren dieses mit der Referenzvariable *notizen*".

8 Fehler vermeiden - Fehler finden

Man unterscheidet zwei Arten von Fehlern:
- Syntaxfehler
- logische Fehler

Syntaxfehler
Bei Verstößen gegen die formalen Regeln der Programmiersprache spricht man von Syntaxfehlern. Syntaxfehler kann der Java-Compiler entdecken und durch Fehlermeldungen anzeigen. Deshalb sind Syntaxfehler recht einfach zu finden.

Logische Fehler
Ein syntaktisch korrektes Programm kann vom Java-Kompiler übersetzt werden.
Das heißt leider noch nicht, dass das Programm das gewünschte Ergebnis liefert.
Logische Fehler (auch "semantische" Fehler genannt) sind oft sehr schwer zu finden.
Selbst verbreitete komerzielle Programme enthalten oft noch logische Fehler (z.B. auch Windows).
Es ist wichtig, Fehler zu vermeiden, gegebenenfalls zu erkennen und auf ein Minimum zu reduzieren.
Um das zu erreichen muss man testen, gegebenenfalls Fehlerursachen suchen (= "debuggen"),

und den Sourcecode möglichst wartungsfreundlich gestalten.

8.1 Testen und Fehlerbeseitigung

Testen und Fehlerbeseitigung sind grundlegende Tätigkeiten bei der Softwareentwickung.
Der dafür erforderliche Zeitaufwand ist oft größer als der für das Entwickeln des Quellcodes.
Oft ist man damit beschäftigt, seine eigenen Programme zu testen, z.B. hier im Programmierkurs.
Viele Entwickler sind aber auch für den Test und die Anpassung von Programmen Anderer zuständig.
Die dazu erforderlichen Techniken sind ähnlich:
Modultests
automatisierte Tests
manuelle Ausführung
print-Anweisungen
Debugger

8.2 Modultests in BlueJ

Das Testen einzelner Einheiten steht im Gegensatz zum Akzeptanztest, dem Testen des Gesamtprogramms.
Eine Einheit kann sein: eine Klasse, eine Gruppe von Klassen, eine Methode.
In BlueJ können die Klassen und Methoden sehr einfach einzeln getestet werden.
Es ist immer sinnvoll, einzelne Teile so früh wie möglich zu testen. Gründe:
- Kleine Teile lassen sich einfacher überschauen und deshalb einfacher testen.
- Man kann sich beim Zusammensetzen zu größeren Einheiten auf zuverlässige - weil geprüfte - Komponenten verlassen.
- Man kann Testfälle sammeln, die man wiederholt überprüft, wenn das Programm wächst.

Vorgehen beim Test:
- Objektinspektoren nutzen.
- Grenzbereiche testen.

Positive Tests: Prüfen, ob das Programm die gewünschte Funktion richtig ausführt.
Negative Tests: Prüfen, wie das Programm mit Fehlersituationen umgeht.
Wichtig: Negative Tests nicht vergessen!
Oft sind die negativen Tests schwieriger als die positiven.

8.3 Automatisierte Tests

Beim Korrigieren von Programmfehlern wird oft das aktuelle Problem behoben, aber an anderer Stelle ein neuer Fehler eingebaut.
Man muss deshalb nach jeder Programmveränderung die Tests wiederholen, um nachzuweisen, dass das Programm noch richtig arbeitet. Solche wiederholten Tests nennt man **Regressionstests**.
Um den Zeitaufwand für Regressionstests zu begrenzen ist es sinnvoll, die Tests zu automatisieren.
Die Testschritte, die wir bisher "von Hand" in BlueJ ausgeführt haben, kann man dazu in ein Programm packen.
Dadurch entsteht eine **automatische Testumgebung.** (Testgerüst, testbench, "Prüfbock").

8.4 Manuelle Ausführung

Logische Fehler sind oft sehr schwer zu finden.
Manchmal hilft es bei der Fehlersuche, einen völlig anderen Betrachtungswinkel zu wählen.
Eine Möglichkeit besteht darin, den Quellcode auszudrucken und - fern vom Rechner - die Abläufe mit Bleistift und Papier nachzuvollziehen, und dadurch den Rechner zu simulieren.
Beim manuellen Ausführen der Methoden sollte man stets die Zustände der Objekte protokollieren.
Oft ist es sinnvoll, den Zustand vor und nach der Ausführung einer Methode zu dokumentieren.

Eine **Empfehlung** aus eigener Erfahrung für die Suche hartnäckiger logischer Fehler:
Diskutieren Sie mit einem Kommilitonen über das Problem.
Der Erfolg kann sich dabei auf zwei Arten einstellen:
Der Außenstehende ist nicht wie Sie "betriebsblind" und erkennt deshalb den Fehler.

Beim Versuch, das Problem einem Außenstehenden zu erklären,
werden einem selbst die Zusammenhänge erst vollständig klar und
man erkennt dadurch den Fehler selbst.

Aus Zeitgründen können wir hier kein Beispiel für diesen aufwändigen Prozess vorführen.
Interessierte finden ein Beispiel für die Fehlersuche im Projekt *Recheneinheit*,
das auch im Buch beschrieben ist.

8.5 print-Anweisungen

Eine verbreitete Vorgehensweise zum finden von Fehlern:
Die wichtigen Schritte der Programmausführung durch ***print-Anweisungen*** dokumentieren.
- Aufrufe von Methoden,
- die Werte von Parametern,
- die Rückgabewerte,
- die Werte von Vaiablen und Datenfeldern an wichtigen Stellen.

Print-Anweisungen können sehr hilfreich sein, um ein Programm zu verstehen.
Nachteile:
- Man muss die richtigen Stellen finden, wo die Information nützlich ist.
- Es entseht leicht eine Informationsflut. Vorsicht bei print in Schleifen!
- Wenn man die Fehler gefunden hat, muss man die prints wieder elimimieren.
- Manchmal braucht man sie nach dem Löschen doch wieder.

Ein- und Ausschalten von Print-Anweisungen

Boolesches Datenfeld als Schalter für Print-Anweisungen einführen:
.

```
boolean test = true;
public void testausgabe(String info)
{
    if (test) {
        System.out.println(info);
    }
}
```

8.6 Debugger

Erster Schritt:
Haltepunkt setzen auf die Stelle, ab der das Verhalten des Programms untersucht werden soll.
Dann Einzelschrittausführung,
Schritt über: führt ganze Methode als ein Schritt aus. (von Details abstrahieren)
Schritt hinein: führt Methode als Einzelschritt aus. (Details analysieren)
Weiter setzt Programm fort bis zum nächsten Haltepunkt.
Beobachten Sie:
Zustand der Objekte: Instanzvariablen (= Datenfelder)
Statische Variable = Klassenvariable
Lokale Variablen der jeweiligen Methoden
Aufruffolge = aktuelle Aufrufhierarchie der Methoden.
Durch Anklicken einer Methode kann man deren lokale Variablen sehen.

9 Klassenentwurf

Wie strukturiert man eine Anwendung und teilt die Teilaufgaben auf Klassen auf?
Es gibt oft viele Möglichkeiten, eine Aufgabenstellung in Software zu lösen.
Alle Lösungen, die bei der Ausführung tatsächlich funktionieren, sind möglicherweise von unterschiedlicher Qualität.
Qualitätskriterien:
leichte Erweiterbarkeit
gute Wartbarkeit

9.1 Kopplung und Kohäsion

Hier werden einige Begriffe zur Diskussion der Qualität eines Klassenentwurfs eingeführt.
Wir werden Sie bei der Untersuchung des Quellcodes benutzen.

Kopplung

Der Begriff Kopplung beschreibt den Grad der **Abhängigkeit** zwischen Klassen.
Man sollte für die Implementierung eine möglichst **lose Kopplung** anstreben.
Eine Klasse sollte möglichst unabhängig von anderen Klassen sein.
Klassen sollten nur über möglichst schmale, wohldefinierte Schittstellen miteinander kommunizieren.
Enge Kopplung: Änderung in einer Klasse macht Änderungen in anderen Klassen erforderlich.
Lose Kopplung: Änderung in einer Klasse lokal möglich.

Kohäsion

Der Begriff Kohäsion einer Programmeinheit beschreibt die **Zusammengehörigkeit** der in der Einheit erledigten Aufgaben.
Eine Einheit kann dabei eine Klasse oder auch nur eine Methode sein.
Eine Programmeinheit sollte nur für *eine* wohl definierte Aufgabe verantwortlich sein und so eine **hohe Kohäsion** besitzen.
Hohe Kohäsion: Wahrscheinlichkeit der Wiederverwendbarkeit (der Klasse, der Methode) ist höher.

9.2 Code-Duplizierung

Das Kopieren von Quellcode-Abschnitten ist ein Hinweis auf schlechten Entwurfsstil.
Abhilfe: Eine separate Methode für die gemeinsame Aufgabe einführen und jeweils aufrufen.
Das verbessert die Kohäsion.

9.3 Entwurf nach Zuständigkeiten

Das Ziel: Klassen mit möglichst loser Kopplung.
Der Weg: Entwurf der Klassen nach Zuständigkeiten
Entwurf nach Zuständigkeiten weist jeder Klasse eine klare Verantwortung für ihre Daten zu. So kann leichter festgelegt werden, in welche Klasse eine Funktionalität eingebaut werden sollte.

9.4 Kapselung der Daten

Das Ziel: Klassen mit möglichst loser Kopplung.
Der Weg: Kapselung - alle Datenfelder privat.

In objektorientierten Programmiersprachen wird eine Klasse durch die Zusammenfassung einer Menge von Daten und darauf operierender Funktionen (die nun *Methoden* genannt werden) definiert. Die Daten werden durch einen Satz Variablen repräsentiert, der für jedes instanziierte Objekt neu angelegt wird (diese werden als *Attribute*, *Membervariablen*, *Instanzvariablen* oder *Instanzmerkmale* bezeichnet). Die Methoden sind im ausführbaren Programmcode nur einmal vorhanden, operieren aber bei jedem Aufruf auf den Daten eines ganz bestimmten Objekts (das Laufzeitsystem übergibt bei jedem Aufruf einer Methode einen Verweis auf den Satz Instanzvariablen, mit dem die Methode gerade arbeiten soll).

Die Instanzvariablen repräsentieren den *Zustand* eines Objekts. Sie können bei jeder Instanz einer Klasse unterschiedlich sein und sich während seiner Lebensdauer verändern. Die Methoden repräsentieren das *Verhalten* des Objekts. Sie sind - von gewollten Ausnahmen abgesehen, bei denen Variablen bewußt von außen zugänglich gemacht werden - die einzige Möglichkeit, mit dem Objekt zu kommunizieren und so Informationen über seinen Zustand zu gewinnen oder diesen zu verändern. Das Verhalten der Objekte einer Klasse wird in seinen Methodendefinitionen festgelegt und ist von dem darin enthaltenen Programmcode und dem aktuellen Zustand des Objekts abhängig.

Diese Zusammenfassung von Methoden und Variablen zu Klassen bezeichnet man als *Kapselung*. Sie stellt die zweite wichtige Eigenschaft objektorientierter Programmiersprachen dar. Kapselung hilft vor allem, die Komplexität der Bedienung eines Objekts zu reduzieren. Um eine Lampe anzuschalten, muß man nicht viel über den inneren Aufbau des Lichtschalters wissen. Sie vermindert aber auch die Komplexität der Implementierung, denn undefinierte Interaktionen mit anderen Bestandteilen des Programms werden verhindert oder reduziert.

9.5 Implizite Kopplung

Von impliziter Kopplung von Klassen spricht man, wenn die Kopplung nicht offensichtlich ist.
Beispiel: Einbau des neuen Befehls "look" in das Spiel.
Der neue Befehl "look" soll die Ausgabe der Rauminfo veranlassen.
Offensichtlich muss man die Klasse ***Befehlswörter*** ändern, um den neuen Befehl in das Array ***gueltigeBefehle[]*** einzutragen.

In der Klasse ***Spiel*** muss der neue Befehl in die Methode ***verarbeiteBefehl(Befehl befehl)*** eingebaut werden.
Nach diesen Änderungen und der Neu-Compilation kann der Befehl "look" benutzt werden.
Problem: Wir haben die Anpassung des Hilfetextes vergessen!
Diese Kopplung ist implizit, d.h. nicht offensichtlich.
Abhilfe: Die Hilfetexte sollten vom Parser erzeugt werden, z.B. durch *parser.gibBefehle()* .
Diese Methode kann die Aufgabe an die *Befehlswoerter*-Klasse weiterleiten.

9.6 Programmausführung ohne BlueJ

9.6.1 Klassenmethoden

Alle bisher geschriebenen Methoden beziehen sich auf ein Objekt, d.h. eine Instanz einer Klasse.
Deshalb nennt man diese Methoden auch **Instanzmethoden**.
Eine Instanzmethode hat Zugriff auf die Datenfelder ihrer Instanz, z.B auch explizit mit dem Schlüsselwort **this.**
Eine Instanzmethode wird aufgerufen durch

```
instanzname.methodenname(parameterliste)
```

Klassenmethoden beziehen sich nicht auf eine konkrete Instanz einer Klasse.
Sie können deshalb nicht auf die Datenfelder eines Objekts zugreifen und keine Instanzmethoden aufrufen.
Das Konzept der Klassenmethoden ist mit dem der Klassenvariablen (= statische Datenfelder) verwandt.
Bei der Definition werden Klassenmethoden durch das Schlüsselwort **`static`** gekennzeichnet.
Eine Klassenmethode wird aufgerufen durch

```
Klassenname.methodenname(parameterliste)
```

oder durch

```
instanzname.methodenname(parameterliste)
```

wobei ***`instanzname`*** der Name einer beliebigen Instanz der Klasse sein darf.

9.7 Die main-Methode

Wir haben Programme bisher immer wie folgt gestartet:
In BlueJ haben wir "von Hand" zunächst ein Objekt als Instanz einer Klasse erzeugt und dann für das Objekt eine oder mehrere Methoden mit Hilfe von BlueJ augerufen.
Ohne BlueJ brauchen wir eine Klasse, die eine statische Methode ***main(...)*** enthält.
Alle Aufrufe, die wir "von Hand" in BlueJ erledigt haben, müssen in der *main*-Methode enthalten sein.
Die *main*-Methode hat die Signatur

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

Der Aufruf erfolgt auf der Kommandozeile in der Form

```
>java KlassenName
```

wobei ***KlassenName*** der Name einer Klasse sein muss, die eine main-Methode enthält.
Die zugehörige class-Datei, das Ergebnis des Compilierens, muss auf dem aktuellen Verzeichnis vorhanden sein.

10 Vererbung - Polymorphie

Entwurfsziele für Programme: Gute Erweiterbarkeit und Wartbarkeit..
Neue OOP-Konzepte in diesem Kapitel zum Erreichen der Ziele: Vererbung und Polymorphie.

Idee: Die gemeinsamen Teile von zwei (oder mehr) Klassen definiert man nur einmal in einer eigenen Klasse und "verwendet" diese dann zweimal.

Terminologie:
Medium ist **Superklasse** von CD und Video.
CD und Video sind **Subklassen** von Medium.
CD und Video **erben** von Medium.
CD und Video **erweitern (engl. extend)** ihre Superklasse Medium.

Eine Subklasse ist eine Spezialisierung ihrer Superklasse.
Eine solche Vererbungsbeziehung wird auch als ***ist-ein***-Beziehung bezeichnet,
z.B. eine CD ist ein Medum; ein Video ist ein Medium.

Die Objekte der Subklasse haben auch alle Datenfelder, die in der *Superklasse* deklariert sind.
Das gleiche gilt für die Methoden: Instanzen von Subklassen verfügen über alle Methoden der Superklasse und natürlich über die eigenen Methoden.
Vorteil der Vererbung: Man Klassen mit großer Ähnlichkeiten ohne Codeduplizierung definieren.

10.1 Syntax

Keine Besonderheit in der Superklasse.
Bei der Definition einer Subklassen ist die Superklasse, also diejenige die erweitert werden soll, hinter dem Schlüsselwort ***extends*** anzugeben.

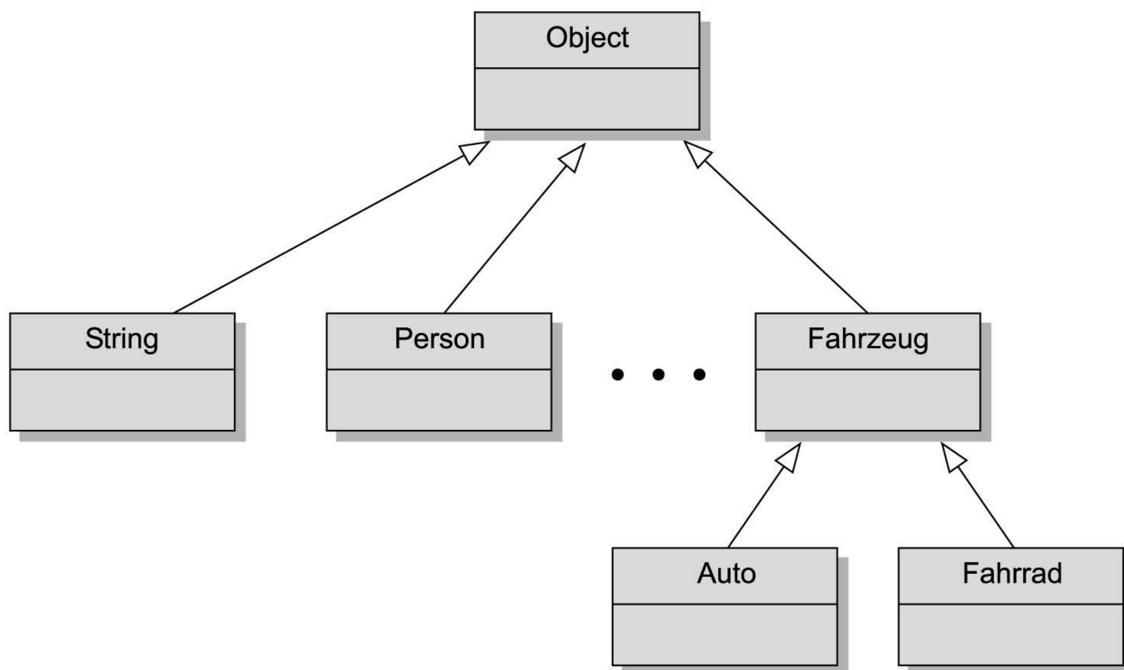
```
public class Subklasse extends Superklasse
 {
   Datenfeld;

public Subklasse()
{
 super();
}

}
```

Das Schlüsselwort `super` bewirkt hier den Aufruf des Konstruktors der Superklasse.
Der Aufruf des Konstruktors der Superklasse muss immer die erste Anweisung im Konstruktor der Subklasse sein.

Wenn kein expliziter *super*-Aufruf im Konstruktor steht, ruft Java implizit den Konstruktor *super()* ohne Parameter auf.

10.2 Klassenhierarchie

Eine Subklasse kann selbst Superklasse einer neuen Subklassse sein.
Im folgenden Beispiel ist Spiel als eine neue Art von Medien eingeführt.
Verschiedene Arten von Spielen werden hier als Subklassen von Spiel modelliert:



In Java haben sogar **alle** Klassen eine Superklasse.
Klassen ohne explizites *extends*-Schlüsselwort erben implizit von der Klasse ***Object,*** die in der Java-Klassenbibliothek enthalten ist.
Damit sind **alle** Klassen direkte oder indirekte Subklassen der Klasse *Objekt*.
Alle Java-Klassen bilden eine große **Vererbungshierarchie**.
Beispiel:

Vorteile durch Vererbung (bis hierher)

Vermeidung von Code-Duplizierung.

Wiederverwertung von Quelltext.

Einfachere Wartbarkeit.

Einfachere Erweiterbarkeit.

10.3 Subklassen und Subtypen

Variablen mit einer Klasse als Typ sind **Referenzvariable**.
Mit Hilfe dieser Referenzvariablen kann man auf Objekte zugreifen, die Instanzen der betreffenden Klasse sind.
An eine (Referenz-)Variable kann man ein Objekt vom betreffenden Klassentyp zuweisen, z.B. ein mit Hilfe des Konstruktors neu erzeugtes Objekt

```
liste1 = new ArrayList();
```

oder ein bereits existierendes durch Zuweisung einer Variablen des gleichen Typs, z.B.

```
liste2 = liste1;
```

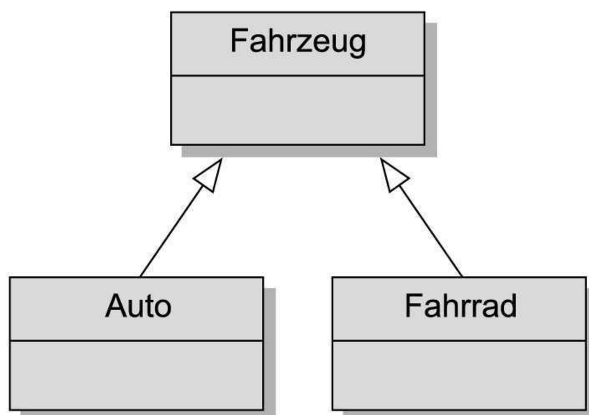
Der Typ einer Variable legt fest, welche Objekte sie referieren kann.
Statt referieren sagt man auch **"halten"**. Eine Referenzvariable **"hält"** ein Objekt.

Prinzip der **Ersetzbarkeit** in der OOP: An einer Stelle, wo ein Objekt einer Klasse X erwartet wird, kann dieses durch ein Objekt einer Subklasse von X ersetzt werden. Z.B.:

- Ein Fahrrad kann verwendet werden, wo ein Fahrzeug gefordert ist.
- Eine CD kann verwendet werden, wo ein Medium gefordert ist.

Bei einer Zuweisung muss der zugewiesene Wert zum Typ der Variable passen.
Der zugewiesene Wert passt dann zum Typ der Variable, wenn er **vom gleichen Typ oder von einem Subtyp** ist.
Demnach sind alle folgenden Zuweisungen legal:

```
Fahrzeug f1 = new Fahrzeug();     // gleicher Typ
Fahrzeug f2 = new Auto();         // Subtyp
Fahrzeug f3 = new Fahrrad();      // Subtyp
```



Umgekehrt gilt das natürlich nicht:

```
Auto meinAuto = new Fahrzeug()     // Fehler!
```

Für einen formalen Parameter vom Typ X darf ein aktueller Parameter vom Typ X oder irgend einem Subtyp von X übergeben werden.

10.4 Polymorphe Variablen

Variablen mit einem Klassentyp sind polymorph (deutsch: vielgestaltig),
d.h. diese Variablen können Objekte von verschiedenen Typen referieren,
nämlich vom Typ der Variable selbst und von allen Subtypen.
Eine Variable vom Typ *Object* kann folglich jedes beliebige Objekt halten, da *Object* unmittelbar oder mittelbar (über mehrere Hierarschiestufen) Superklasse **aller** Klassen ist.
Von diesem Umstand machen Sammlungstypen wie z.B. *ArrayList*, *HashSet* und *HashMap* Gebrauch:
Die Elemente der Sammlungen sind vom Typ *Object*. Deshalb kann man Objekte von beliebigem Typ in ihnen halten.
Man sieht das z.B. an der Signatur der *add*-Methode:

```
public void add(Object element);
```

Das hat den Vorteil, dass man die Sammlungsklassen unverändert für die verschiedensten Zwecke benutzen kann.
Man könnte dadurch sogar völlig verschiedene Objekttypen in **einer einzigen** Sammlung speichern, obwohl diese Mischung im Allgemein aber keinen Sinn macht.
Umgekehrt erhält man die Objekte aus einer Sammlung deshalb leider "anonym" als vom Typ *Object* zurück.
Siehe z.B. die Signatur der *get*-Methode der *ArrayList*:

```
public Object get(int index);
```

Der Typ des Rückgabewertes ist *Object*, das Objekt das er referiert ist aber natürlich noch immer von dem Typ, von dem es war, als es in die Sammlung eingefügt wurde. Der Compiler kann das aber nicht wissen.
Will man es einer Variablen vom Originaltyp zuweisen, benötigt man deshalb den Cast-Operator.
Dieser Effekt tritt nicht nur bei Sammlungen auf, sondern auch bei "gewöhnlichen" Referenzvariablen.

Beispiel:

```
Fahrzeug f;
Auto a;
a = new Auto();
f = a;
a = f ;  // das ergibt einen Fehler beim Compilieren
a = (Auto) f;
```

Der Typ von *f* ist Supertyp von ***a***.
Man kann eine Subtyp- an eine Supertyp-Variable zuweisen (*f = a*),
aber nicht umgekehrt (*a = f*).
Weiteres Beispiel:

```
Fahrzeug f;
Auto a;
Fahrrad rad;
a = new Auto();
f = a;
rad = (Fahrrad) a ;   // Fehler beim Compilieren
rad = (Fahrrad) f ;   // Fehler beim Ausführen
```

Beim Übersetzen der vorletzten Zeile erkennt der Compiler, dass ***a*** ein Auto ist und nie und nimmer ein Fahrrad sein kann. Deshalb kann er sofort einen Fehler melden.

Beim Übersetzen der letzten Zeile erkennt der Compiler, dass ***f*** ein Fahrzeug ist und unter Umständen ein Objekt vom Typ Fahrrad referieren könnte. Er meldet deshalb vorerst noch keinen Fehler. Bei der Ausführung des Programms fällt natürlich auf, dass in ***f*** ein Auto referiert wird und dass die Zuweisung nicht erlaubt ist.

10.5 Statischer und dynamischer Typ

Der Variablentyp ist schon beim Programmieren festgelegt: **statischer Typ**.
Die Typen der referierten Objekte ergeben sich erst bei der Programmausführung: **dynamischer Typ**.

10.6 Überschreiben von Methoden

Wenn in den Subklassen Methoden angeboten werden, welche die gleiche Signatur aufweist, sagt man, die Subklasse überschreibt die Methode der Superklasse.

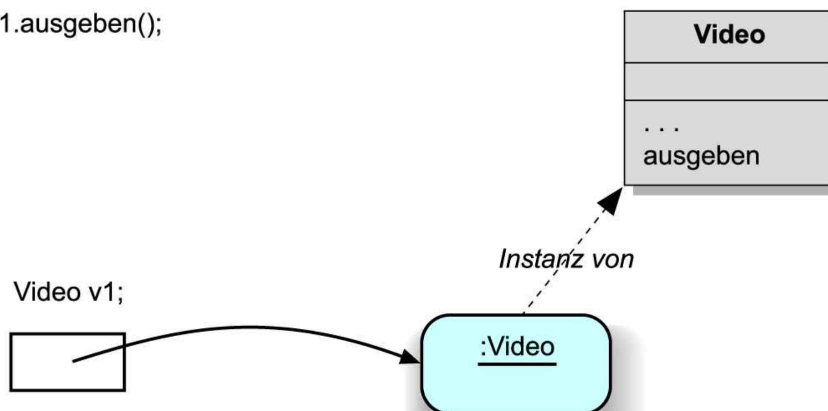
10.6.1 Dynamische Methodensuche

- Die Typüberprüfung berücksichtigt den **statischen Typ.**

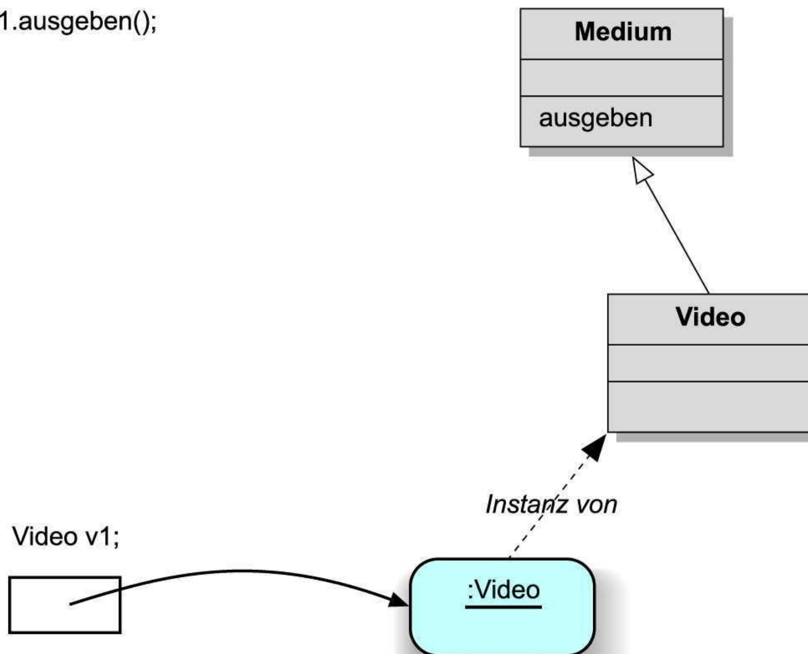
- Zur Laufzeit werden die Methoden des **dynamischen Typs** ausgeführt.

Wir untersuchen drei Szenarien:

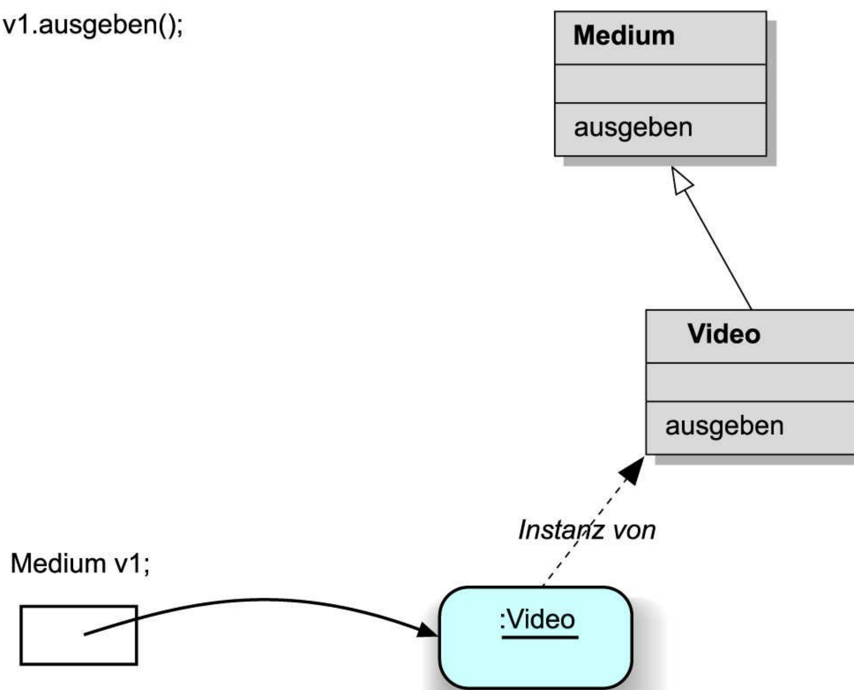
Szenario 1: Variable vom Typ *Video* enthält ein Objekt vom Typ *Video*:
Methode *ausgeben()* in *Video*.



Szenario 2: Variable vom Typ *Video* enthält ein Objekt vom Typ *Video,* das Subklasse von *Medium*:
Methode *ausgeben()* in *Medium*.



Szenario 3: Variable vom Typ *Medium* enthält ein Objekt vom Typ *Video,* das Subklasse von *Medium*:
Methode *ausgeben()* in *Medium* und *Video*.



10.7 super-Aufrufe in Methoden

In der Methode der Subklasse kann man die überschriebene Methode der Superklasse aufrufen, mit hilfe des super-Aufrufs.

Beispiel: in der Klasse *CD*

```
public void ausgeben()
{
    super.ausgeben();
    System.out.println("    " + kuenstler);
    System.out.println("    " + titelanzahl + titel);
}
```

10.8 Methoden-Polyphormie

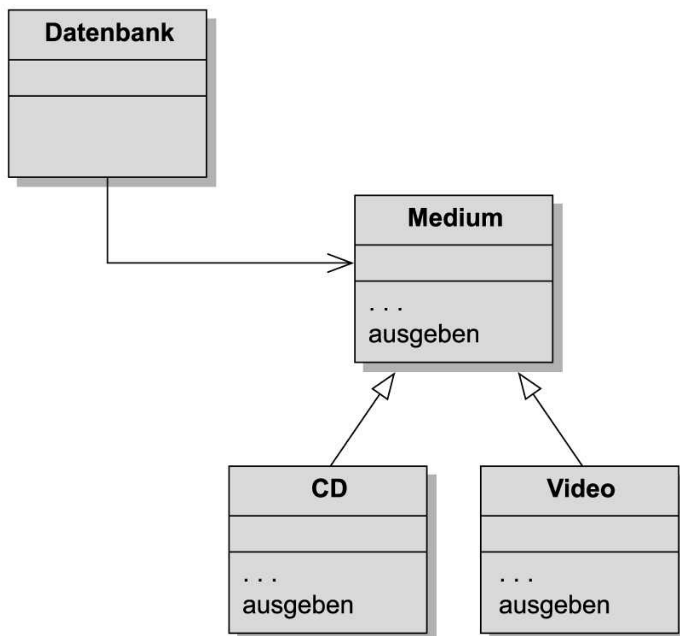
Methodenaufrufe sind in Java polymorph: Man spricht von polymorpher Methodensuche.
Derselbe Methodenaufruf kann zu verschiedenen Zeitpunkten verschiedene Methoden aufrufen,
abhängig vom dynamischen Typ der Variablen, mit der der Aufruf durchgeführt wird.

10.9 Aus Object geerbte Methode: toString

Die Klasse *Objekt* ist Superklasse aller Klassen, entweder direkt oder indirekt.
Alle Klassen erben also die Methoden von *Objekt,* z.B. die Methode *toString()*.
Erprobung: An einer beliebigen Instanz einer Klasse, z.B. Video in BlueJ.
Erklärung: Siehe Interfacebeschreibung von *Object* in der Java-Dokumentation.
Die geerbte Methode *toString()* liefert einen String bestehend aus Klassenname und Adresse des Objekts.

11 Abstrakte Klassen

Im Projekt DoME wird die Superklasse *Medium* nur benutzt,
um die **Gemeinsamkeiten** der Medienobjekte (z.B. CD und Video) an zentraler Stelle zusammenzufassen.
Es ist nicht daran gedacht, wirklich Objekte von Typ Medium zu erzeugen.
Alle Medien sind Objekte einer konkreten Subklasse der allgemeineren Klasse *Medien*.



Eine solche allgemeinere Superklasse nennt man **abstrakt**.
Von einer abstrakten Klasse gibt es keine Instanzen.
Durch den Modifier ***abstract*** bei der Klassendefinition vereinbart man eine abstrakte Klasse.
Der Java Compiler stellt sicher, dass *abstract*-Klassen nicht instanziiert werden können.

Syntax:

```
public abstract class Klassenname
{
  Konstruktor() {}

  Methoden() {}
}
```

11.1 Abstrakte Methoden

Eine abstrakte Methode enthält keine Implementierung sondern deklariert nur die Signatur.
In konkreten Subklassen muß die Methode implementiert werden.
Eine Klasse, die abstrakte Methoden enthält, muss selbst *abstract* deklariert werden.
Hält eine Klasse eine eine abstrakte Methode, so stellt dies eine Verpflichtung für die konkreten Subklassen dar:
In konkreten Subklassen muss die abstrakte Methoden gemäß der Signatur implementiert werden.

Syntax:

```
public abstract class Klassenname
{
  Konstruktor() { }

  Methoden() { }


  public abstract void monatsAbrechnung();

}
```

11.2 Interfaces

Unsere Datenbank kann ausschließlich Objekte vom Typ *Medium* speichern.
Worin ist diese Einschränkung begründet?
Die Medienobjekte werden in einer *ArrayList* gehalten, die kann ja alle Objekttypen speichern.
Das ist also keine Einschränkung.

Die Medienobjekte werden von der Methode *erfasseMedium* in die Datenbank eingetragen.
Einschränkung: Der Typ der einzutragenden Objekte ist durch den Typ des Parameters festgelegt.

Die Objekte der Datenbank werden von der Methode *auflisten* durch Aufruf ihrer Methode *ausgeben* ausgedruckt.
Einschränkung: Nur für Medienobjekte ist das Vorhandensein einer Methode *ausgeben* gewährleistet.
Der Typ des Objektes aus der *ArrayList* wird deshalb per Type-cast festgelegt.
Diese Einschränkungen gehen unnötig weit!
Es genügt sicherzustellen, dass nur Objekte von einem Typ übergeben werden, der eine Methode *ausgeben* hat
Ein solcher Typ kann statt durch eine Klasse durch ein Interface festgelegt werden, z.B.

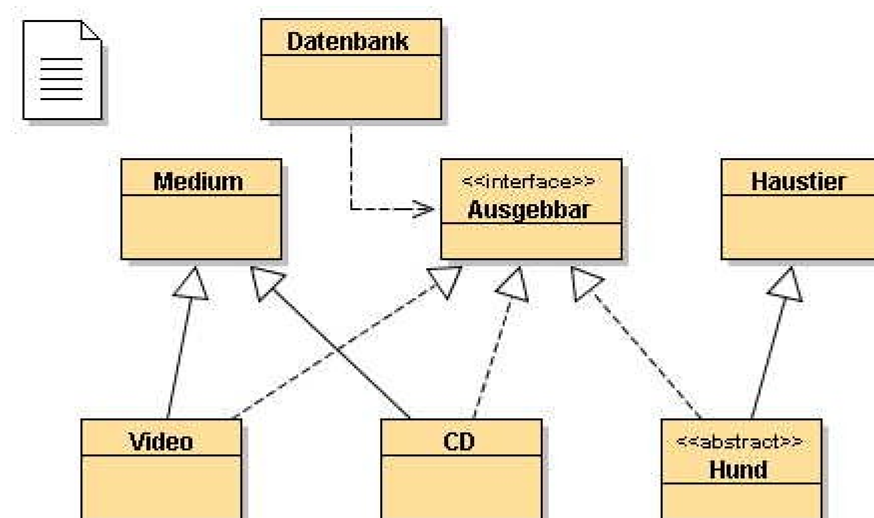
Syntax:

```
public interface Ausgebbar
{

public void ausgeben(); // Beachte: Semikolon statt
                                   Anweisungen

}
```

Ein Interface ähnelt einer Klassendefinition, die ausschließlich aus abstrakten Methoden besteht!
Alle Methodendeklarationen eines Interface sind implizit abstrakt.
Interfaces definieren also einen Namen für einen Teilaspekt einer Klasse.
Variablen vom Typ eines Interfaces dienen dazu, ein Objekt zu halten,
das diese Teilaspekte, also die entsprechenden Methoden, implementiert,
ganz gleich von welchem speziellen Klassentyp es ist.

Diese Vorgehensweise erlaubt dem Compiler zwei Überprüfungen:
1) Zur Compilezeit, ob die Methoden des Interface tatsächlich in der Klasse definiert werden.
2) Zur Laufzeit, ob ein Objekt die erforderlichen Methoden (der Schnittstelle) auch wirklich enthält.

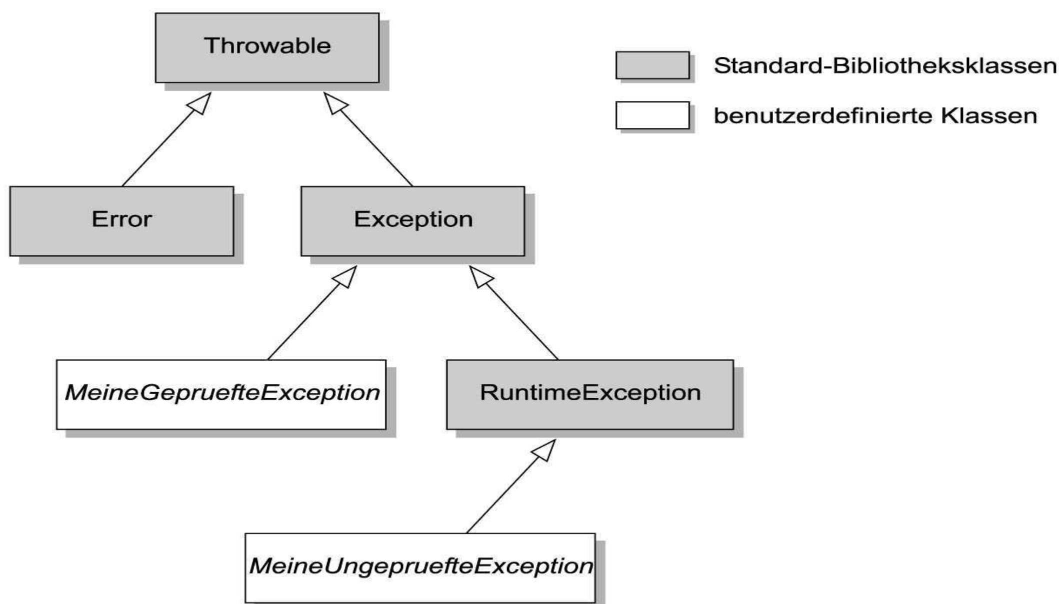
Das Konzept erlaubt es, auch völlig andere Objekte in die Datenbank einzutragen.
Jedes beliebige Objekt; das das Interface *Ausgebbar* implementiert, kann in die Datenbank eingetragen werden.



12 Fehlerbehandlung: Exceptions

Alle Laufzeitfehler in Java sind Unterklassen der Klasse `Throwable`. `Throwable` ist eine allgemeine Fehlerklasse, die im wesentlichen eine Klartext-Fehlermeldung speichern und einen Auszug des Laufzeit-Stacks ausgeben kann. Unterhalb von `Throwable` befinden sich zwei große Vererbungshierarchien:

- Die Klasse `Error` ist Superklasse aller *schwerwiegenden* Fehler. Diese werden hauptsächlich durch Probleme in der virtuellen Java-Maschine ausgelöst. Fehler der Klasse `Error` sollten in der Regel nicht abgefangen werden, sondern (durch den Standard-Fehlerhandler) nach einer entsprechenden Meldung zum Abbruch des Programms führen.
- Alle Fehler, die möglicherweise für die Anwendung selbst von Interesse sind, befinden sich in der Klasse `Exception` oder einer ihrer Unterklassen. Ein Fehler dieser Art signalisiert einen abnormen Zustand, der vom Programm abgefangen und behandelt werden kann.



12.1 Prinzip der Ausnahmebehandlung

Durch "**Werfen**" einer **Exception** kann der Dienstleister ein Problem signalisieren und eine Reaktion des Klienten erzwingen. **Exception**, engl. für Ausnahme(situation). Wenn der Klient eine aufgetretene Exception nicht behandelt, wird das Programm abgebochen.
Eine Exception wird durch das Schlüsselwort ***throw*** ausgelöst (geworfen).

Die Exception-Auslösung besteht aus zwei Schritten:

1) Instanziieren eines Exception-Objekts:	`ExceptionKlasse ex;`
	`ex = new`
	`ExceptionKlasse("Fehlertext");`

2) Werfen des Exception-Objekts:	**`throw ex`**

Beide Schritte können in eine Zeile geschrieben werden:

```
throw new ExceptionKlasse("Fehlertext");
```

Der Parameter des Konstruktors einer Exception-Klasse ist ein String, der als Fehlermeldung dient.
Jedes Exception-Objekt ist eine Instanz einer Klasse aus der folgenden Hierarchie:

Java unterscheidet zwei Kategorien von Exceptions:

Ungeprüfte Exceptions (unchecked exceptions): alle Subklassen von ***RuntimeException***

Geprüfte Exceptions (checked exceptions): alle anderen Subklassen von ***Exception***

Unterschied:
Ungeprüfte Exceptions
für Fälle, die im normalen Betrieb eines Programms gar nicht vorkommen können,
also Anwendung bei **Programmierfehlern**.
Ein harter Programmabbruch ist bei einem Programmierfehler angemessen.

Direkte Ableitungen:
ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, ConcurrentModificationException, DOMException, EmptyStackException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, ImagingOpException, IndexOutOfBoundsException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NullPointerException, ProfileDataException, ProviderException, RasterFormatException, SecurityException, SystemException, UndeclaredThrowableException, UnmodifiableSetException, UnsupportedOperationException

Geprüfte Exceptions
für Fälle, in denen eine Operation auch bei korrektem Programm **fehlschlagen** kann, z.B.
- Schreiben auf Festplatte und Platte voll,
- Einlesen einer Zahl und Tippfehler des Benutzers.

In diesen Fällen sollte ein Klient gezwungen werden, den Erfolg der Operation zu überprüfen.

Ein harter Programmabbruch ist keine adäquate Reaktion.

Auswirkung des Werfens einer Exception
Bei Ausführung einer ***throw***-Anweisung wird die Methode **beendet**.
Die restlichen Anweisungen werden nicht ausgeführt. Es wird kein Returnwert erzeugt.
Auch der Ablauf der Methode, die die werfende Methode aufgerufen hat, wird durch die Exception verändert.
Wie genau, hängt davon ab, ob sie die geworfene Exception **auffängt** (catch) oder nicht.

12.2 Behandlung von geprüften Exceptions

Eine Methode, die geprüfte Exceptions werfen könnte, muss diese in ihrer Methoden-Signatur hinter dem Schlüsselwort ***throws*** angeben.

```
public void speichereInDatei(String dateiname)
     throws IOException
 {
     ....
 )
```

Auch die ungeprüften Exceptions dürfen hinter *throws* angegeben werden, müssen aber nicht.

Eine Methode, die eine (andere) Methode mit throws in der Signatur aufruft, muss für den Fall, dass tatsächlich eine Exception geworfen wird, Vorkehrungen treffen. Sie hat zwei Möglichkeiten: Auffangen oder propagieren.

12.2.1 Auffangen der Exception

Dazu muss sie einen **Exception-Händler** bereitstellen, der die Exception auffängt.
Für das Beispiel eines Aufrufs von ***speichereDatei*** könnte das wie folgt aussehen:

```
String dateiname = null;
try {
    dateiname = ... ;
    adressbuch.speichereDatei(dateiname);
    System.out.println("Adressbuch gespeichert.");
}
catch(IOException e) {
    System.out.println("Speichern in " + dateiname + "
schlug fehl.");
}
```

Der ***try*-Block** enthält die "geschützten" Anweisungen, die auf das Auftreten einer Exception überprüft werden sollen.
Tritt die Exception dann tatsächlich auf, wird die Bearbeitung des *try*-Blocks abgebrochen.
Die Bearbeitung wird im zugehörigen ***catch*-Block** (= Exception-Handler) fortgesetzt.
Auf das geworfene Exception-Objekt kann man im zugehörigen *catch*-Block wie auf einen Methodenparameter zugreifen.

Mehrere Exceptions
Wenn in den Anweisungen des *try*-Blocks verschiedene Exception-Typen auftreten können, sucht Java den zugehörigen *catch*-Block als Exception-Handler .

<u>Beispiel:</u>
Gegeben sei eine Methode, die zwei Exceptions werfen könnte:

```
public void verarbeite()
   throws EOFException, FileNotFoundException
```

Diese könnte man wie folgt aufrufen:

```
try {
     ....
     ref.verarbeite()
     ....
}
catch(EOFException e){
     // Behandlung von Dateiende
     ....
}
catch(FileNotFoundException e){
     // Behandlung des Fehlens der Datei
     ....
}
```

"Zugehörig" bedeutet: Derjenige *catch*-Block, dessen Parameter zum geworfenen Exceptiontyp passt,
d.h. gleich ist oder Subtyp ist.

Die *finally*-Klausel
ist optional.
Sie enthält Anweisungen, die auf jeden Fall ausgeführt werden sollen, gleichgültig, ob eine Exception aufgetreten ist oder nicht.

```
try {
     // überwachte Anweisungen
}
catch(Exception e) {
     // Exception Handler
}
finally {
     // Anweisung, die auf jeden Fall ausgeführt werden sollen
}
```

Hinweis Haarspalterisch genau lässt sich auch ein Beispiel finden, in dem `finally` nicht ausgeführt wird.

```
try
{
  System.exit( 1 );
}
finally
{
  System.out.println( "Das wirst du nicht erleben!" );
}
```

12.2.2 Propagieren (weiterreichen) der Exception

Eine Methode x kann eine Exception an ihren Aufrufer weitereichen, indem sie sie entweder
- nicht auffängt oder
- nach dem Auffangen erneut wirft.

In diesem Fall muss die Methode x in ihrem Methodenkopf die weitergereichte Exception in einer ***throws***-Klausel deklarieren.

12.3 Behandlung von ungeprüften Exceptions

Ungeprüfte Exceptions sind Subklassen von *RuntimeException* .
Sie müssen weder gefangen werden noch in einer ***throws***-Klausel deklariert werden, dürfen aber.
Wenn sie nicht gefangen werden, führen sie zum Programmabbruch: akzeptabel nur bei Programmierfehlern.
Beispiel: ***NullPointerException***

13 Ein- und Ausgabe von Textdateien

Zwei Arten, Daten in Dateien zu speichern:

- **als Binärdateien**
  z.B. exe-Dateien, Bilddateien,
  Java-Klassen: Streams (Datenströme)
- **als Textdateien**
  Zeichenfolgen, zeilenorientiert, lesbar und schreibbar mit Editor, z.B. notepad
  Java-Klassen: Reader und Writer

Die Java-Klassen für die Ein/Ausgabe sind im Paket java.io enthalten.
Wir beschäftigen und hier nur mit Textdateien.

13.1 Textausgabe mit FileWriter

Die Ausgabe erfolgt immer in drei Schritten nach folgendem Grundmuster::

- **Datei öffnen** (open)
  durch Erzeugen eines *FileWriter*-Objekts
  `FileWriter writer = new FileWriter(Dateienpfad);`

- Solange Daten vorhanden:
  Daten schreiben
  durch die Methode *write* des *FileWriter*-Objekts
  `writer.write(nächsterText);`

- **Datei schließen** (close)
  durch die Methode *close* des *FileWriter*-Objekts
  `writer.close();`

Der Parameter *Datei* des Konstruktoraufrufs `FileWriter(Dateienpfad)` kann sein:

ein String, der den (absoluten) Dateinamen enthält oder

"E:/Daten/Thomas/Bilder/fhLogo.jpg"

- ein Objekt vom Typ ***File***, das die zu öffenende Datei repräsentiert.
  Es kann erzeugt werden durch seinen Konstuktoraufruf
  `File myFile = new File(Dateiname);`
  wobei *Dateiname* ein String ist, der den (absoluten) Dateinamen enthält.
  Vorteil: für Dateiobjekte gibt es nützliche Methoden, z.B.
  boolean exists() zur Überprüfung, ob die Datei existiert und
  boolean createNewFile() , um die datei neu anzulegen.

Der Parameter *nächsterText* des Aufrufs `writer.write(nächsterText)` ist ein String.
Eine Zeile der Datei wird abgeschlossen durch

```
writer.write("\n")
```
oder durch
```
writer.write('\n')
```

Jeder Schritt der Ausgabe kann fehlschlagen, auch wenn die Programme völlig korrekt sind. Gründe können z.B. sein: Festplatte voll, Diskette schreibgeschützt, Diskettenlaufwerk offen, Mit hilfe der Subklasse Bufferd

Folge: Man muss bei jedem Schritt mit einer Exception rechnen und dafür sorgen, dass sie aufgefangen wird.

13.1.1 Escape-Codes für Sonderzeichen

Achtung: Funktioniert nicht bei writer, es wird nur in der Datei vermerkt. Deswegen Zeilenumbrüche mit dem **Bufferwriter** realisieren.

Escape-Sequenz	Bedeutung
\n	Neue Zeile
\t	Tabulator (Tab)
\b	Rückschritt (Backspace)
\r	Wagenrücklauf (Carriage return)
\f	Seitenvorschub (Formfeed)
\\	Inverser Schrägstrich (Backslash)
\'	Einfache Anführungszeichen
\"	Doppelte Anführungszeichen
\d	Oktal
\xd	Hexadezimal
\du	Unicode-Zeichen

13.2 Texteingabe mit FileReader

Auch die Eingabe erfolgt in drei Schritten nach folgendem Grundmuster::

- **Datei öffnen** (open)
  durch Erzeugen eines *FileReader*-Objekts:
  ```
  FileReader fr = new FileReader(Datei);
  ```
  damit ein *BufferedReader*-Objekt erzegen für zeilenweise Eingabe:
  ```
  BufferedReader reader = new BufferedReader(fr);
  ```

- Solange Datei nicht am Ende:
  Daten lesen
  durch die Methode ***readLine*** des *BufferedReader*-Objekts, also:
  ```
  String zeile = reader.readLine();
      while (zeile != null) {
         // eingelesene Zeile verarbeiten:
  ```

```
    ....
   zeile = reader.readLine();
  }
```

- **Datei schließen** (close)
  durch die Methode *close* des *BufferedReader*-Objekts
  `reader.close();`

Auch bei der Eingabe kann jeder Schritt fehlschlagen, auch wenn die Programme völlig korrekt sind. Gründe können z.B. sein: Datei gelöscht, fehlendes Zugriffsrecht, Diskettenlaufwerk offen, ...

Folge: Man muss bei jedem Schritt mit einer Exception rechnen und dafür sorgen, dass sie aufgefangen wird.

13.3 Texteingabe von der Konsole

Das Prinzip ist das gleiche wie beim Einlesen von Dateien. Es wird dem BufferedReader beim Konstrukteraufruf lediglich ein anderer Reader (**`InputStreamReader(System.in)`**) übergeben. Das weitere Vorgehen ist Analog.

```
BufferedReader eingabe = new BufferedReader(new
                              InputStreamReader(System.in));
try {
       text = eingabe.readLine();
       abgebrochen = false;
    }
    catch ( java.io.IOException exc) {
       System.out.println( ""+ exc.getMessage());
    }
```

14 Grafik und Fenster im awk

14.1 Die Klasse Frame: Top-Level Grafikfenster

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Window
                    |
                    +--java.awt.Frame
```

Ein Java-Programm (eine Applikation) hat zunächst kein Grafikfenster und muß es sich erst erzeugen.
Geeignet als Top-Level Fenster ist ein Object der Frame-Klasse.
Ein Frame hat einen Rahmen mit Titelleiste und Buttons zum Minimieren, Verkleinern/Maximieren und (sehr wichtig) zum Schließen (Beenden).
Erzeugen eines *Frame*-Objekts durch den Konstruktoraufruf:

```
Frame f = new Frame("Titel");
```

Festlegen der Größe (Beispiel: Weite 400, Höhe: 200 Pixel)

```
f.setsize(400,200);
```

Sichtbarmachen des Frame f:

```
f.show()
```

Unsichtbarmachen des Frame f:

```
f.hide()
```

Löschen des Frame f:

```
f.dispose()
```

Beispiel:

```
import java.awt.*;

public class Fenster
{

    Frame f ;

    public Fenster()
    {

        f = new Frame("mein Titel");
        f.setSize(400,200);
        f.show();
    }
    public static void main(String[] args)
    {
        Fenster f = new Fenster();
    }
}
```

Jeder Frame sollte mindestens eine Methode für das Schließen (d.h. Beenden) des Frame haben.
Sie wird (von Java automatisch) aufgerufen, wenn der Schließen-Button geklickt wird.
Das Klicken eines Button nennt man ein **Ereignis / Event**,
die zugehörige Aktion wird von einer **Ereignismethode / Eventmethode** ausgeführt.
Die Methoden zur Reaktion auf Ereignisse von Frames sind im Interface ***WindowListener*** festgelegt.
Man muss deshalb beim *Frame*-Objekt ein *WindowListener*-Objekt anmelden,
d.h. ein Objekt einer beliebigen Klasse, die das *WindowListener*-Interface implementiert.

```
    f.addWindowListener(...)
```

Wichtig ist, dass zumindest die Eventmethode zum Schließen des Frame zur Verfügung gestellt wird.
Sie hat die Signatur

```
    public void windowClosing(WindowEvent e)
```

Beim Schließen des Hauptfensters sollte das Fenster gelöscht und das Programm beendet werden:

```
    f.dispose();
    System.exit(0);
```

Das *WindowListener*-Interface könnte man in einer eigenen Subklasse von *frame* implementieren:.

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

public class CloseableFrame extends Frame implements
WindowListener {


  public CloseableFrame() {
     this.addWindowListener(this);
  }

  public CloseableFrame(String title) {
     super(title);
     this.addWindowListener(this);
  }

   public void windowClosing(WindowEvent e) {
    this.dispose();
    System.exit(0);
  }
  // Die anderen sind implementiert, tun aber nichts:
  public void windowOpened(WindowEvent e) {}
  public void windowClosed(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
}
```

14.2 Die Klasse Panel: Ein Grafik-Container

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Panel
```

Ein *frame* kann ein oder mehrere grafische Teilfenster enthalten.
Einfachster Fall: ein ***Panel***-Objekt *p* wird durch die *add*-Methode dem Frame *f* hinzugefügt:

```
        f = new CloseableFrame("Titel");
        p = new Panel();
        f.setSize(600,300);
        f.add(p);
        f.show();
```

Ein ***Panel*** ist einfaches Fenster ohne Rahmen, ein Container, der weitere Grafikobjekte enthalten kann.
Man kann in einem Panel aber auch direkt elementare Graphikausgaben machen.
Dazu schreibt man eine eigene Subklasse von Panel, die die ***paint***-Methode von Panel überschreibt.
Die Anweisungen zum Erzeugen der Grafik stehen in der *paint*-Methode.
Beispiel:

```
import java.awt.*;

public class MyPanel extends Panel
{
    public MyPanel()
    {
        super();
    }

    public void paint(Graphics g)
    {
        g.drawString("Hallo Welt",10,10);
    }
}
```

Das *Frame*-Objekt ruft automatisch immer dann die *paint*-Methode unseres *Panel's* auf, wenn dessen Inhalt neu gezeichnet werden muss, z.B.
- beim Vorholen des *Frame* in den Vordergrund oder
- wenn die (durch andere Fenster) abgedeckte *Frame*-Fläche kleiner wird.

14.3 Die Klasse Graphics

```
java.lang.Object
  |
  +--java.awt.Graphics
```

Der *paint*-Methode unseres *Panel's* wird beim Aufruf ein Objekt der ***Graphics***-Klasse übergeben.
Ein ***Graphics***-Objekt enthält wichtige Informationen über die Zeichenfläche:
z.B. Größedes Fensters, Farbe des Hintergrunds, Zeichenfarbe, Schriftart für Texte.
Außerdem stellt es wichtige Methoden für die Grafikausgabe zur Verfügung,

Method Summary

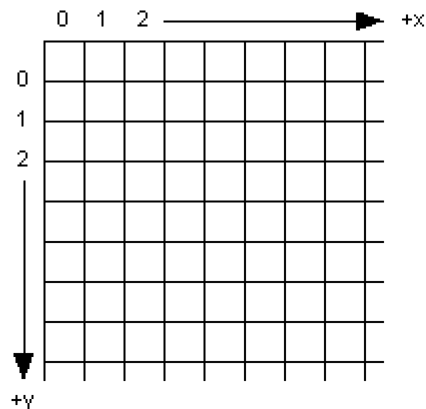
abstract void	**clearRect**(int x, int y, int width, int height) Clears the specified rectangle by filling it with the background color of the current drawing surface.
abstract void	**clipRect**(int x, int y, int width, int height) Intersects the current clip with the specified rectangle.
abstract void	**copyArea**(int x, int y, int width, int height, int dx, int dy) Copies an area of the component by a distance specified by `dx` and `dy`.
abstract Graphics	**create**() Creates a new `Graphics` object that is a copy of this `Graphics` object.
Graphics	**create**(int x, int y, int width, int height) Creates a new `Graphics` object based on this `Graphics` object, but with a new translation and clip area.
abstract void	**dispose**() Disposes of this graphics context and releases any system resources that it is using.
void	**draw3DRect**(int x, int y, int width, int height, boolean raised) Draws a 3-D highlighted outline of the specified rectangle.
abstract void	**drawArc**(int x, int y, int width, int height, int startAngle, int arcAngle) Draws the outline of a circular or elliptical arc covering the specified rectangle.
void	**drawBytes**(byte[] data, int offset, int length, int x, int y) Draws the text given by the specified byte array, using this graphics context's current font and color.
void	**drawChars**(char[] data, int offset, int length, int x, int y) Draws the text given by the specified character array, using this graphics context's current font and color.
abstract boolean	**drawImage**(Image img, int x, int y, Color bgcolor, ImageObserver observer) Draws as much of the specified image as is currently available.
abstract boolean	**drawImage**(Image img, int x, int y, ImageObserver observer) Draws as much of the specified image as is currently available.
abstract boolean	**drawImage**(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer) Draws as much of the specified image as has already been scaled to fit inside the specified rectangle.
abstract boolean	**drawImage**(Image img, int x, int y, int width, int height, ImageObserver observer) Draws as much of the specified image as has already been scaled to fit inside the specified rectangle.
abstract boolean	**drawImage**(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color bgcolor, ImageObserver observer) Draws as much of the specified area of the specified image as is currently available, scaling it on the fly to fit inside the specified area of the destination drawable surface.

abstract boolean	**drawImage**(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer) Draws as much of the specified area of the specified image as is currently available, scaling it on the fly to fit inside the specified area of the destination drawable surface.
abstract void	**drawLine**(int x1, int y1, int x2, int y2) Draws a line, using the current color, between the points (x1, y1) and (x2, y2) in this graphics context's coordinate system.
abstract void	**drawOval**(int x, int y, int width, int height) Draws the outline of an oval.
abstract void	**drawPolygon**(int[] xPoints, int[] yPoints, int nPoints) Draws a closed polygon defined by arrays of *x* and *y* coordinates.
void	**drawPolygon**(Polygon p) Draws the outline of a polygon defined by the specified Polygon object.
abstract void	**drawPolyline**(int[] xPoints, int[] yPoints, int nPoints) Draws a sequence of connected lines defined by arrays of *x* and *y* coordinates.
void	**drawRect**(int x, int y, int width, int height) Draws the outline of the specified rectangle.
abstract void	**drawRoundRect**(int x, int y, int width, int height, int arcWidth, int arcHeight) Draws an outlined round-cornered rectangle using this graphics context's current color.
abstract void	**drawString**(AttributedCharacterIterator iterator, int x, int y) Draws the text given by the specified iterator, using this graphics context's current color.
abstract void	**drawString**(String str, int x, int y) Draws the text given by the specified string, using this graphics context's current font and color.
void	**fill3DRect**(int x, int y, int width, int height, boolean raised) Paints a 3-D highlighted rectangle filled with the current color.
abstract void	**fillArc**(int x, int y, int width, int height, int startAngle, int arcAngle) Fills a circular or elliptical arc covering the specified rectangle.
abstract void	**fillOval**(int x, int y, int width, int height) Fills an oval bounded by the specified rectangle with the current color.
abstract void	**fillPolygon**(int[] xPoints, int[] yPoints, int nPoints) Fills a closed polygon defined by arrays of *x* and *y* coordinates.
void	**fillPolygon**(Polygon p) Fills the polygon defined by the specified Polygon object with the graphics context's current color.
abstract void	**fillRect**(int x, int y, int width, int height) Fills the specified rectangle.
abstract void	**fillRoundRect**(int x, int y, int width, int height, int arcWidth, int arcHeight)

	Fills the specified rounded corner rectangle with the current color.
void	**finalize**() Disposes of this graphics context once it is no longer referenced.
abstract Shape	**getClip**() Gets the current clipping area.
abstract Rectangle	**getClipBounds**() Returns the bounding rectangle of the current clipping area.
Rectangle	**getClipBounds**(Rectangle r) Returns the bounding rectangle of the current clipping area.
Rectangle	**getClipRect**() **Deprecated.** *As of JDK version 1.1, replaced by* `getClipBounds()`.
abstract Color	**getColor**() Gets this graphics context's current color.
abstract Font	**getFont**() Gets the current font.
FontMetrics	**getFontMetrics**() Gets the font metrics of the current font.
abstract FontMetrics	**getFontMetrics**(Font f) Gets the font metrics for the specified font.
boolean	**hitClip**(int x, int y, int width, int height) Returns true if the specified rectangular area might intersect the current clipping area.
abstract void	**setClip**(int x, int y, int width, int height) Sets the current clip to the rectangle specified by the given coordinates.
abstract void	**setClip**(Shape clip) Sets the current clipping area to an arbitrary clip shape.
abstract void	**setColor**(Color c) Sets this graphics context's current color to the specified color.
abstract void	**setFont**(Font font) Sets this graphics context's font to the specified font.
abstract void	**setPaintMode**() Sets the paint mode of this graphics context to overwrite the destination with this graphics context's current color.
abstract void	**setXORMode**(Color c1) Sets the paint mode of this graphics context to alternate between this graphics context's current color and the new specified color.
String	**toString**() Returns a `String` object representing this `Graphics` object's value.
abstract void	**translate**(int x, int y) Translates the origin of the graphics context to the point (*x*, *y*) in the current coordinate system.

14.3.1Das Koordinatensystem von Graphics

Alle Zeichenmethoden erwarten die x,y-Koordinaten für die jeweilige Aktion als Parameter. Manche erwarten mehr als ein Koordinaten-Paar: z.B. bei Linien, 1-mal x,y für den Anfangspunkt, ein anderes x,y-Koordinatenpaar für den Endpunkt.



Javas Koordinatensystem: Pixel als Maßeinheit (Integer). Der Koordinatenursprung (0,0) liegt in der oberen linken Ecke des Applet-Fensters. Die x-Werte wachsen nach rechts, ausgehend vom Ursprung, und die y-Werte wachsen nach unten.

14.3.2Zeichnen und Füllen

Es gibt zwei Arten von Zeichenmethoden
1) Methoden, deren Name mit ***draw*** beginnt. Zeichnen nur den Umriss
2) Methoden, deren Name mit ***fill*** beginnt. Füllen die jeweilige Figur mit der aktuellen Farbe.
Man kann auch Bitmap-Dateien (z.B. GIF- oder JPEG) ausgeben; siehe Klasse ***Image***.

14.3.3Linien

Die Methode ***drawLine()*** wird verwendet, um zwischen zwei Punkten eine Linie auszugeben. Die Methode erwartet vier Argumente: die x,y-Koordinaten des Startpunktes und die x,y-Koordinaten des Endpunktes:

```
drawLine(x1, y1, x2, y2);
```

zeichnet eine Linie von dem Punkt (x1, y1) zu dem Punkt (x2, y2). Breite der Linie: ein Pixel

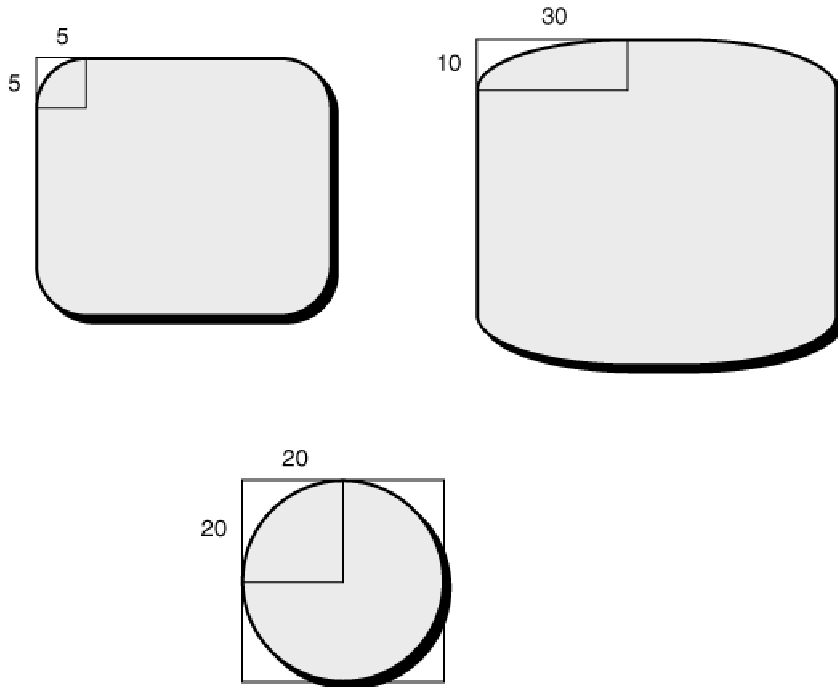
14.3.4Rechtecke

Graphics-Methoden für Rechtecke:
a) normale Rechtecke: ***drawRect()*** und ***fillRect()***

```
drawRect(x, y, breite, höhe);
```

b) mit abgerundeten Ecken: ***drawRoundedRect()*** und ***fillRoundedRect().***

```
drawRoundedRect(x, y, breite, höhe, rundx, rundy);
```



Rechtecke mit abgerundeten Ecken

14.3.5Polygone (Vielecke)

drawPolgon() und ***fillPolygon()***
Für jeden Punkt des Polygons die x,y-Koordinaten angeben. Eine Linie wird von einem Startpunkt zum Endpunkt gezeichnet. Dieser Endpunkt wird als Startpunkt für eine neue Linie verwendet und so weiter.
Koordinaten auf zweierlei Arten angeben:
- Als zwei Arrays mit Integern: x-Werte und y-Werte.
- Als *Polygon*-Objekt (mit Integer-Array für x und Integer-Array für y-Werte im Konstruktor)

Zweite Methode flexibler, einzelne Punkte können einem Polygon hinzugefügt werden. z.B.:

```
int x[] = { 10, 20, 30, 40, 50 };
int y[] = { 15, 25, 35, 45, 55 };
int points = x.length;
Polygon poly = new Polygon(x, y, points);
```

Punkte hinzufügen z.B.:

```
poly.addPoint(60, 65);
```

Polygon-Objekt zeichnen:

```
g.drawPolygon(poly);
```

Die Polygon-Klasse ist Bestandteil des Paketes *java.awt*. import java.awt.Polygon;

14.3.6Ovale und Bögen

drawOval() und ***fillOval()*** .
Diese Methoden erwarten vier Argumente (umschreibendes Rechteck):

```
drawOval(x, y, breite, höhe);
```

Ein Bogen ist ein Teil eines Ovals: ***drawArc()*** und ***fillArc()***
Diese Methoden erwarten sechs Argumente:

```
drawArc(x, y, breite, höhe, alpha, beta);
```

alpha: Winkel, bei dem der Bogen beginnt
beta: Winkel den der Bogen überstreicht
Gefüllte Bögen: Tortenstücke (Endpunkte mit dem Mittelpunkt des Ovals verbunden)

14.4 Grafische Textausgabe und Schrifttypen (Font)

Der verwendete Schrifttyp wird im *Graphics*-Objekt oder allgemein in **Componenten** (alle Subklassen) durch ein ***Font***-Objekt beschrieben.
Der Konstruktor hat drei Parameter: Schriftname, Schriftstil, Größe in Punkt
Schriftname ist ein String und kann sein: Dialog, DialogInput, Monospaced, Serif, SansSerif, or Symbol
Für die Schriftstile enthält die Klasse die drei Konstanten *Font.PLAIN*, *Font.BOLD* und *Font.ITALIC* .
Beispiel:

```
Font f = new Font("SansSerif", Font.BOLD, 24);
```

Die Methode *setFont* des *Graphics*-Objekts setzt den Schrifttyp für zukünftige *drawString*-Aufrufe.
Beispiel:

```
public void paint(Graphics g) {
    Font f = new Font("SansSerif", Font.ITALIC,72);
    g.setFont(f);
    g.drawString("Hallo Welt", 10, 100);
}
```

14.5 Die Farben der Grafikausgabe (Color)

```
java.lang.Object
  |
  +--java.awt.Color
```

Die Farben werden im *Graphics*-Objekt durch ***Color***-Objekte beschrieben.
Ein gängiger Konstruktor der Klasse ***Color*** ist *Color(int r, int g, int b)*
Die Parameter *r*, *g* und *b* definieren eine Mischfarbe durch die Anteile an rot, grün und blau.
r, *g* und *b* können Werte zwischen 0 und 255 annehmen.
Es gibt schon vordefinierte Standardfarben als Klassenvariablen der Klasse *Color*:

Farbe	Variable	RGB-Wert
Schwarz	black	(0,0,0)
Blau	blue	(0,0,255)

Cyan	cyan	(0,255,255)
Dunkelgrau	darkGray	(64,64,64)
Grau	gray	(128,128,128)
Grün	green	(0,255,0)
Hellgrau	lightGray	(192,192,192)
Magenta	magenta	(255,0,255)
Orange	orange	(255,200,0)
Rosa	pink	(255,175,175)
Rot	red	(255,0,0)
Weiß	white	(255,255,255)
Gelb	yellow	(255,255,0)

Beispiel:

```
g.setColor(Color.pink);
```

Mit eigenem Color-Objekt:

```
Color brush = new Color(255,204,102);
g.setColor(brush);
```

Nachdem die aktuelle Farbe gesetzt ist, erscheinen alle Zeichenoperationen in dieser Farbe.
Die Methode setBackground() legt die **Farbe des Hintergrundes** des *Panel*-Objekts fest, z.B.

```
setBackground(Color.white);
```

Aktuelle Farbe ermitteln: ***getColor()*** für ein *Graphics*-Objekt aufrufen oder ***getForeground()*** bzw. ***getBackground()*** für ein *Panel*-Objekt verwenden.

15 Events und Eventhandler

Die Eventverarbeitung beruht auf dem Konzept der **"event listener"**:

Grafische Programme reagieren auf Anwenderaktionen durch Events.
Events können u.a. sein:

Mausbewegungen im zugehörigen Fenster,
Mausklicks,
Tastaturanschläge,
Fenster wird verdeckt,
Fenster in den Vordergrund,
Fenster wird geschlossen,

Ein Programm muss auf solche Ereignisse reagieren.
Dabei ist wichtig: wo wurde geklickt, in welchem Teil eines Fensters, d.h. in welcher **Komponente**.
Komponenten sind die Bausteine einer Benutzeroberfläche (**GUI**).
Zugehörige GUI-Klassen sind im Packet java.awt und dessen Unterpaketen definiert.
Bisher von uns benutzt: java.awt.Panel
Stammbaum von Panel:

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Panel
```

Ein Panel ist eine Komponente,
und zwar eine, die andere Komponenten enthalten kann (Container).
Ereignisse sind jeweils einer Komponente zugeordnet,
Komponenten sind die Ereignis-Quellen (**Event sources**).
Programme wollen (sollen) auf Ereignisse reagieren, z.B. auf das Klicken eines Buttons.
Dazu muss die virtuelle Maschine (JVM) dem Programm mitteilen, dass ein Ereignis vorliegt.
Jede Komponente sollte Methoden enthalten, die bei einem *ihr* zuzuordnenden Ereignis aufgerufen werden:
Diese Methoden heißen **Eventhandler**.
Das Anwendungsprogramm muss die Komponente bitten, mitzuteilen wenn ein Event vorliegt.
Etwa so: "Liebe Komponente, bitte rufe meine Methode xyz auf, falls ein Maus-Ereignis auftritt."
Im Java-Jargon heißt das: Einen **Eventhandler** bei der Komponente **registrieren**.
xyz nennt man im Windows-Jargon auch eine **Call-back-Methode**.
Es gibt verschiedene Eventarten: spezielle Klassen, Subklassen von java.util.EventObject.
AWT-Events (nur die werden hier besprochen) sind alle Subklassen von java.awt.AWTEvent.

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--java.awt.AWTEvent
```

z.B. **die MouseEvent Klasse:**

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--java.awt.AWTEvent
              |
              +--java.awt.event.ComponentEvent
                    |
                    +--java.awt.event.InputEvent
                          |
                          +--java.awt.event.MouseEvent
```

Event-Objekte (Instanzen der Event-Subklassen), enthalten alle für die Auswertung nötigen Daten.
z.B. für ein MouseEvent
- den Typ: **getID()**, z.B. MOUSE_CLICKED, MOUSE_PRESSED, MOUSE_RELEASED etc.
- die Koordinaten: **getX(), getY(),** an denen der Mauszeiger beim Ereignis stand,
- die Klickanzahl: **getClickCount()**, Einfachklick, Doppelklick etc.

15.1 Eventhandler

Der Aufruf von Eventhandlern beruht auf dem Konzept der ***"event listener"***:
Ein Objekt, das vom Event erfahren will, ist ein ***event listener***.
Jede ***event source*** führt eine Liste der *listener*-Objekte, die bei einem Event zu verständigen sind.
Die *event source* benachrichtgt den *event listener* durch Aufruf einer Methode des listeners.
Aufrufparamer: ein Eventobject.

Damit die Eventsource die Methode des listeners aufrufen kann, muß diese implementiert sein:
Wird sichergestellt dadurch, daß der listener das zugehörige **listener-interface** implementiert.
z.B. müssen mouse listener das Interface **java.awt.event.MouseListener** implementieren, (und auch das Interface java.awt.event.MouseMotionListener) siehe Java-api-docu.

Das betreffende Listener-Object wird bei der Komponente registriert durch

```
add.xyzListener(listenerObject)
```

15.2 Listener Registration

15.2.1Klassenimplimentation

Die Verwendete Klasse Implementiert vollständig die Listener Interfaces.

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

public class MyPanel extends Panel
          implements MouseListener,MouseMotionListener

{


    public MyPanel()
    {
        super();
        this.addMouseListener(this);
        this.addMouseMotionListener(this);
    }
    public void paint(Graphics g)
    {
    }

    public void mousePressed(MouseEvent e) {
       getGraphics().drawLine(0,0,200,200);
    }

     public void mouseDragged(MouseEvent e) {
        getGraphics().drawLine(0,0,100,100);
     }

    // Die anderen, unbenutzten Methoden des
    //MouseListener Interface.

     public void mouseReleased(MouseEvent e) {;}
     public void mouseClicked(MouseEvent e) {;}
     public void mouseEntered(MouseEvent e) {;}
     public void mouseExited(MouseEvent e) {;}

    // Die anderen, unbenutzten Methoden des
    //MouseMotionListener Interface.

    public void mouseMoved(MouseEvent e) {;}
      }
```

Es gibt zu jedem Listener-Interface in der Java-Klassenbibliothek eine vordefinierte Klasse, die es implementiert, allerdings mit Methoden, die (noch) nichts tun: Man nennt sie **Adapterklassen.**
Die Methoden müssen **in Subklassen** der Adapterklassen **sinnvoll überschrieben** werden.
Der Adaptername ist aus dem Name des Listeners abgeleitet, *'Listener'* durch *'Adapter'* ersetzt, z.B.

Interface	**Adapter-Klasse**
mouseListener *mouseMotionListener*	*mouseAdapter* *mouseMotionAdapter*

Die Adaptersubklassen kann man z.B. auch **im Innern** der Panelklasse definieren.
Allgemeines Java.Konzept:

15.2.2 Innere Klassen

Klassen, die man in einem Block deklarieret, sind sichtbar nur in diesem Block.
Man spricht deshalb von **inneren Klassen**.
Sie können auf alle im Block sichtbaren Variablen zugreifen!

Beispiel: Projekt Scribble2 mit inneren Subklassen der Adapterklassen:

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

public class MyPanel extends Panel
{
 private static int n=0;
 private int start_x, start_y;

     public MyPanel()
     {
     super();
     start_x = start_y = 0;
     this.addMouseListener(new InnererMouseAdapter());
this.addMouseMotionListener(new InnererMouseMotionAdapter());
    }

     public void paint(Graphics g)
    {
        if (this.g==null) this.g=g;
        System.out.println("paint() "+ ++n);
        g.drawString("Objekt: "+this,0,10);
    }
```

```
     public class InnererMouseAdapter extends MouseAdapter
    {
         public InnererMouseAdapter()
         {
            super();
         }

        public void mousePressed(MouseEvent e) {
           // Anweisungen
        }
    }

public class InnererMouseMotionAdapter
     extend MouseMotionAdapter
    {

         public InnererMouseMotionAdapter()
         {
            super();
         }

        public void mouseDragged(MouseEvent e) {
           // Anweisungen
        }
    }
 }
```

15.2.3Anonyme innere Klassen

Klassen ohne expliziten Klassennamen heißen anonyme Klassem.
Sie sind werden als Subklassen einer anderen Klasse direkt in der new-Anweisung definiert.

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

public class MyPanel extends Panel
{

     public MyPanel()
     {
      super();
      this.addMouseListener(new MouseAdapter(){});
      this.addMouseMotionListener(new MouseMotionAdapter()
          {
             public void mouseDragged(MouseEvent e) {
                //Anweisungen
             }
          });
      }

}
```

15.3 Panel mit "Gedächtnis"

Am Beispiel des Grafikprogramms, zeichnen einer Linie.

Das bisherige Zeichenprogramm kann die Zeichnung nach dem Abdecken des Fensters nicht wiederherstellen.
Verantwortlich für die Widerherstellung ist die *paint*-Methode des *Panel*-Objekts.
Man muss sich deshalb die Linienelemente speichern, z.B als Objekte einer *Linie*-Klasse.

Die Linien-Objekte sollten sich selbst zeichen können.
Dadurch muss keine Methode, die Linien benutzt, auf deren private Koordinaten zugreifen.
Linien stellen dazu eine *draw*-Methode zur Verfügung, die z.B. die *paint*-Methode des Panels aufrufen kann.
In der folgenden erweiterten ***Panel*-Subklasse** wird
in der *mouseDragged*-Methode des *MouseMotionListeners* jeweils
- ein Linienobjekt erzeugt und
- in die **ArrayList *linien*** eingetragen.

Die *paint*-Methode kann dann über die *linien*-Objekte iterieren und sie auffordern, sich zu zeichnen:
Will man keine Freihandzeichnung sondern eine Zeichnung aus geraden Linienstücken, könnte man
- deren Anfang durch mousePresed und
- deren Endpunkt durch mouseReleased festlegen lassen.

Das Zeichnen könnte für den Anwender des Programms durch eine "Gummilinie" erleichtert

werden, die ständig zwischen dem Startpunkt und der aktuellen Mausposition gezeichnet wird. (Siehe Übung 15)

.

15.4 Event – Kurzübersicht

15.4.1 Focus-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	FocusEvent
Listener-Interface	FocusListener
Registrierungsmethode	addFocusListener
Mögliche Ereignisquellen	Component

Tabelle 28.1: Focus-Ereignisse

Ereignismethode	Bedeutung
focusGained	Eine Komponente erhält den Focus.
focusLost	Eine Komponente verliert den Focus.

Tabelle 28.2: Methoden für Focus-Ereignisse

15.4.2 Key-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	KeyEvent
Listener-Interface	KeyListener
Registrierungsmethode	addKeyListener
Mögliche Ereignisquellen	Component

Tabelle 28.3: Key-Ereignisse

Ereignismethode	Bedeutung
keyPressed	Eine Taste wurde gedrückt.
keyReleased	Eine Taste wurde losgelassen.
keyTyped	Eine Taste wurde gedrückt und wieder losgelassen.

Tabelle 28.4: Methoden für Key-Ereignisse

15.4.3 Mouse-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	MouseEvent
Listener-Interface	MouseListener

Registrierungsmethode	addMouseListener
Mögliche Ereignisquellen	Component

Tabelle 28.5: Mouse-Ereignisse

Ereignismethode	**Bedeutung**
mouseClicked	Eine Maustaste wurde gedrückt und wieder losgelassen.
mouseEntered	Der Mauszeiger betritt die Komponente.
mouseExited	Der Mauszeiger verläßt die Komponente.
mousePressed	Eine Maustaste wurde gedrückt.
mouseReleased	Eine Maustaste wurde losgelassen.

Tabelle 28.6: Methoden für Mouse-Ereignisse

15.4.4MouseMotion-Ereignisse

Eigenschaft	**Klasse, Interface oder Methode**
Ereignisklasse	MouseEvent
Listener-Interface	MouseMotionListener
Registrierungsmethode	addMouseMotionListener
Mögliche Ereignisquellen	Component

Tabelle 28.7: MouseMotion-Ereignisse

Ereignismethode	**Bedeutung**
mouseDragged	Die Maus wurde bei gedrückter Taste bewegt.
mouseMoved	Die Maus wurde bewegt, ohne daß eine Taste gedrückt wurde.

Tabelle 28.8: Methoden für MouseMotion-Ereignisse

15.4.5Component-Ereignisse

Eigenschaft	**Klasse, Interface oder Methode**
Ereignisklasse	ComponentEvent
Listener-Interface	ComponentListener
Registrierungsmethode	addComponentListener
Mögliche Ereignisquellen	Component

Tabelle 28.9: Komponenten-Ereignisse

Ereignismethode	**Bedeutung**
componentHidden	Eine Komponente wurde unsichtbar.
componentMoved	Eine Komponente wurde verschoben.

componentResized	Die Größe einer Komponente hat sich geändert.
componentShown	Eine Komponente wurde sichtbar.

Tabelle 28.10: Methoden für Komponenten-Ereignisse

15.4.6Container-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	ContainerEvent
Listener-Interface	ContainerListener
Registrierungsmethode	addContainerListener
Mögliche Ereignisquellen	Container

Tabelle 28.11: Container-Ereignisse

Ereignismethode	Bedeutung
componentAdded	Eine Komponente wurde hinzugefügt.
componentRemoved	Eine Komponente wurde entfernt.

Tabelle 28.12: Methoden für Container-Ereignisse

15.4.7Window-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	WindowEvent
Listener-Interface	WindowListener
Registrierungsmethode	addWindowListener
Mögliche Ereignisquellen	Dialog, Frame

Tabelle 28.13: Window-Ereignisse

Ereignismethode	Bedeutung
windowActivated	Das Fenster wurde aktiviert.
windowClosed	Das Fenster wurde geschlossen.
windowClosing	Das Fenster wird geschlossen.
windowDeactivated	Das Fenster wurde deaktiviert.
windowDeiconified	Das Fenster wurde wiederhergestellt.
windowIconified	Das Fenster wurde auf Symbolgröße verkleinert.
windowOpened	Das Fenster wurde geöffnet.

Tabelle 28.14: Methoden für Window-Ereignisse

15.4.8Action-Ereignisse

Eigenschaft	Klasse, Interface oder Methode

Ereignisklasse	ActionEvent
Listener-Interface	ActionListener
Registrierungsmethode	addActionListener
Mögliche Ereignisquellen	Button, List, MenuItem, TextField

Tabelle 28.15: Action-Ereignisse

Ereignismethode	**Bedeutung**
actionPerformed	Eine Aktion wurde ausgelöst.

Tabelle 28.16: Methoden für Action-Ereignisse

15.4.9Adjustment-Ereignisse

Eigenschaft	**Klasse, Interface oder Methode**
Ereignisklasse	AdjustmentEvent
Listener-Interface	AdjustmentListener
Registrierungsmethode	addAdjustmentListener
Mögliche Ereignisquellen	Scrollbar

Tabelle 28.17: Adjustment-Ereignisse

Ereignismethode	**Bedeutung**
adjustmentValueChanged	Der Wert wurde verändert.

Tabelle 28.18: Methoden für Adjustment-Ereignisse

15.4.10Item-Ereignisse

Eigenschaft	**Klasse, Interface oder Methode**
Ereignisklasse	ItemEvent
Listener-Interface	ItemListener
Registrierungsmethode	addItemListener
Mögliche Ereignisquellen	Checkbox, Choice, List, CheckboxMenuItem

Tabelle 28.19: Item-Ereignisse

Ereignismethode	**Bedeutung**
itemStateChanged	Der Zustand hat sich verändert.

Tabelle 28.20: Methoden für Item-Ereignisse

15.4.11Text-Ereignisse

Eigenschaft	**Klasse, Interface oder Methode**

Ereignisklasse	TextEvent
Listener-Interface	TextListener
Registrierungsmethode	addTextListener
Mögliche Ereignisquellen	TextField, TextArea

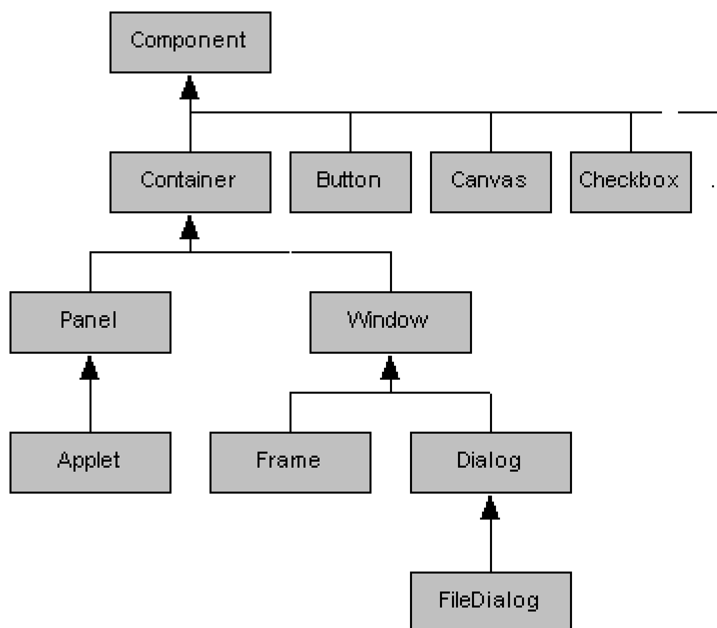
Tabelle 28.21: Text-Ereignisse

Ereignismethode	**Bedeutung**
textValueChanged	Der Text wurde verändert.

Tabelle 28.22: Methoden für Text-Ereignisse

16 Komponenten eines GUI

Die Elementare Grafik mit der Graphics-Klasse des ***awt*** (Abstract Windowing Toolkit) in einem *Frame*-Objekt mit *Panel*-Objekt wurde bereits behandelt in Kapitel 14, auch die Ereignisverarbeitung in Kapitel 15.
Damit sind die Hilfmittel für die Programmierung von GUIs im Prinzip alle vorhanden.
Aber: großer Programmieraufwand, deshalb besser die im *java.awt*-Package vordefinierten Klassen verwenden.
Diese grafischen Objekte nennt man in Java **Komponenten**.
(Unter MS-Windows heißen diese grafischen Objekte Controls.)
Sie sind die vorgefertigten **Bausteine für ein GUI**.
Fast alle diese Komponenten sind Subklassen von ***java.awt.Component.***
GUI-Komponenten werden in Fenstern oder Teilfenstern gruppiert, d.h. in **Container**.
Diese Container sind Subklassen von *java.awt.Container*. (z.B. auch *Panel*).
Jeder Container ist selbst wieder eine Komponente.



16.1 Die Komponenten des AWT

awt enthält eine Reihe vorgefertigter GUI-Komponenten, die im folgenden Applet vorgestellt werden:

Die Komponenten:

Button	Schaltfäche, kann geklickt werden
Canvas	"Leinwand", leerer Bereich, in den gezeichnet werden kann
Checkbox	kann angekreuzt werden (selektiert werden)

Choice	Drop-down-Liste, Optionsmenü
Component .	Superklasse aller Komponenten
FileDialog	Dialogbox zum Auswählen von Dateien
Label	einzeiliger Text, rein zur Beschriftung
List	Liste von Elementen, wovon jeweils eines ausgewählt
Scrollbar	"Aufzug", Verschiebebalken, horizontal, vertikal
TextArea	mehrzeiliger Text, editierbar
TextField	einzeiliger Text, editierbar

Die Menü-Komponenten:

CheckboxMenuItem	Umschaltknopf im Menü
Menu	PulldownMenü
MenuBar	Menü-Leiste: Komponente, die Pulldown-Menüs enthält
MenuComponent	Superklassse der Menü-Komponenten
MenuItem	Eine Menü-Schaltfälche
PopupMenu	Ein Menü, das sich unter der Maus öffnet

Die Container:

Container .	Superklassse aller Container
Dialog	Ein Fenster für eine Dialogbox
Frame	Top-Level-Fenster, mit Rahmen und u.U. mit Menü-Leiste
Panel	leerer Container, als Superklassse geeignet
ScrollPane	Ein Container, der seinen Inhalt scrollen kann
Window	Top-Level-Fenster ohne Rahmen, z.B. für Popup-Menü

16.2 Die vier Schritte für die Programmierung eines GUI:

16.2.1 Die benötigten Komponenten instanzieren

GUI Objekte durch Konstruktoraufrufe erzeugen,
z.B. die von Windows bekannten Buttons, Checkbox, Listbox, Scrollbar etc.
Eigenschaften der Objekte durch Parameter der Konstruktoraufrufe festlegen, z.B.

```
    Button quit = new Button("Beenden");
```

Geschieht üblicherweise in der Init-Methode eines Applets oder
im Konstruktor einer Applikation.

16.2.2 Die Komponenten in einen Container logisch einfügen.

Jede Komponente muß in einem Container sein, z.B. in einem Frame oder Panel oder Applet etc.

- Das Top-Level Fenster einer Applikation ist ein Object der Frame-Klasse.
- Das Applet Objekt ist das Top-Level Fenster eines Applets.

Da Container auch Komponenten sind, können Container auch andere Container enthalten.
Zum Einfügen von Komponenten haben *Container*-Klassen eine ***add*-Methode,**
z.B. `this.add(quit);`
fügt den oben definierten Button in ein *Panel* ein.

16.2.3 Die Komponenten im Container räumlich anordnen.

Die Anordnung dient der Schönheit: Ort und Größe der Komponenten festlegen.
<u>Möglich</u>: **absolute** Koordinaten.
<u>Besser</u>: nur **Layoutregeln** vorgeben.
So können sich Komponenten an die Containergröße automatisch anpassen.
Java-Hilfsmittel: ***LayoutManager,*** z.B.: `f.setLayout(new FlowLayout());`

16.2.4 Die Komponentenereignisse in Eventmethoden verarbeiten.

Statt der elementaren Maus-Ereignissse erzeugen Komponenten oft Events mit mehr "Sinngehalt", semantische Events höherer Ebene.
z.B. nicht nur *mousePressed* oder *mouseReleased*, sondern "Aktion ausgelöst" (*ActionEvent*).

16.3 Layout-Management

Die Anordnung der Komponenten in den Containern erfolgt durch Auswahl eines Layout-Managers:
- Nicht absoloute Positionierung, sondern Algorithmus für die Positionierung.
- Praktisch bei der Veränderung der Fenstergröße: automatische Anpassung.

Das Layout wird festgelegt, indem man die ***set.Layout_methode*** eines Containers aufruft.

16.3.1 FlowLayout

Einfachstes Layout: alle Komponenten werden in Containerfenster von links nach rechts angeordnet, bis Zeile voll. danach gehts in der nächsten Zeile weiter. (Wie bei Text in einem Editor-Fenster.)

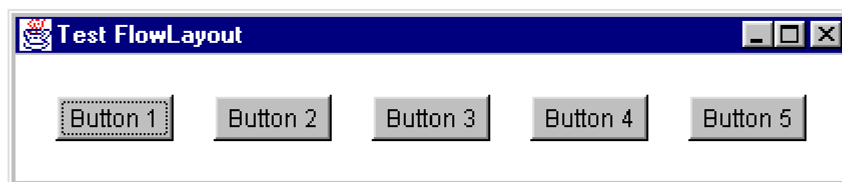
Beispiel:

```
            setLayout(new FlowLayout());
            setFont(new Font("Helvetica", Font.PLAIN,
14));
            add(new Button("Button 1"));
            add(new Button("Button 2"));
            add(new Button("Button 3"));
            add(new Button("Button 4"));
            add(new Button("Button 5"));
```

Die *FlowLayout*-Klasse hat 3 Konstruktoren:

```
            public FlowLayout()
            public FlowLayout(int alignment)
            public FlowLayout(int alignment,
                                   int horizontalGap, int
verticalGap)
```

Alignment: Konstanten in *FlowLayout: CENTER, LEFT, RIGHT,*
Gap: horizontaler und vertikaler Abstand
Standard: *CENTER,5,5*

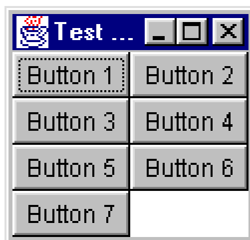


16.3.2 GridLayout

Alle Komponenten werden im Container im Raster von Zeilen und Spalten angeordnet.
Alle Komponenten gleich groß.
Der Konstruktor legt die Anzahl der Zeilen (rows) und Spalten (colums) fest.
rows=0 bedeutet, soviele Zeilen wie nötig.

Beispiel:

```
setLayout(new GridLayout(4,2));
    add(new Button("Button 1"));
    add(new Button("Button 2"));
    add(new Button("Button 3"));
    add(new Button("Button 4"));
    add(new Button("Button 5"));
    add(new Button("Button 6"));
    add(new Button("Button 7"));
    pack();
```



16.3.3 Die Größe der Komponenten

Wenn wir die Größe des Fensters nicht durch einen Aufruf von `pack`, sondern manuell festgelegt hätten, wäre an dieser Stelle ein wichtiger Unterschied zwischen den beiden bisher vorgestellten Layoutmanagern deutlich geworden. Abbildung 31.4 zeigt das veränderte bei dem die Größe des Fensters durch Aufruf von `setSize(500,200);` auf 500*200 Pixel festgelegt und der Aufruf von `pack` entfernt wurde:

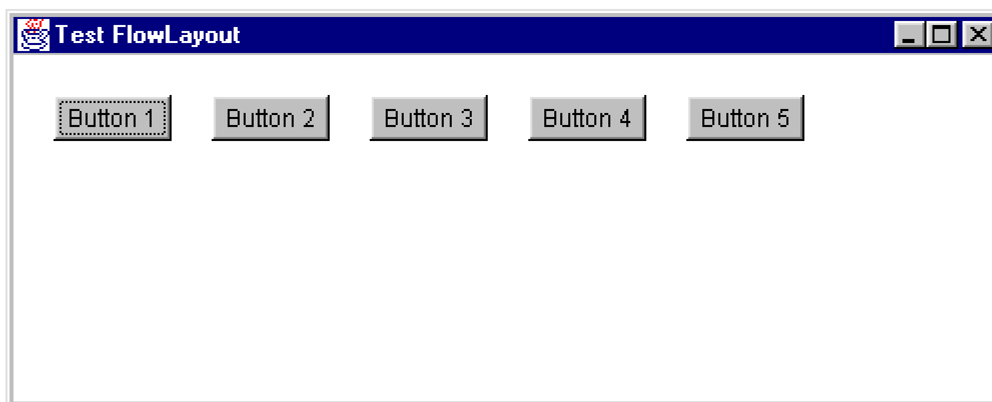


Abbildung 31.4: Das FlowLayout in einem größeren Fenster

Hier verhält sich das Programm noch so, wie wir es erwarten würden, denn die Buttons haben ihre Größe behalten, während das Fenster größer geworden ist. Anders sieht es dagegen aus, wenn wir die Fenstergröße ebenfalls auf 500*200 Pixel fixieren:

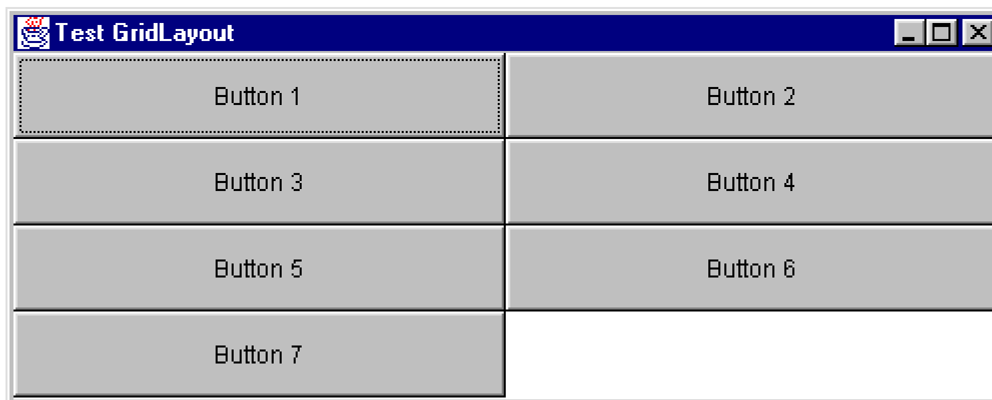


Abbildung 31.5: Das GridLayout in einem größeren Fenster

Nun werden die Buttons plötzlich sehr viel größer als ursprünglich angezeigt, obwohl sie eigentlich weniger Platz in Anspruch nehmen würden. Der Unterschied besteht darin, daß ein `FlowLayout` die *gewünschte* Größe eines Dialogelements verwendet, um seine Ausmaße zu bestimmen. Das `GridLayout` dagegen ignoriert die gewünschte Größe und dimensioniert die Dialogelemente fest in der Größe eines Gitterelements. Ein `LayoutManager` hat also offensichtlich die Freiheit zu entscheiden, ob und in welcher Weise er die Größen von Fenster und Dialogelementen den aktuellen Erfordernissen anpaßt.

16.3.4BorderLayout

Ordnet bis zu fünf Komponenten im Container an.
Je eine Komponente am unteren (South) und oberen (North) sowie
am linken (West) und rechten (East) Rand.
Eine Komponente in der Mitte (Center).
BorderLayout ist Default für Frame, Dialog und Window.

Beispiel:

```
setLayout(new BorderLayout());
add(new Button("North"),"North");
add(new Button("South"),"South");
add(new Button("East"),"East");
add(new Button("West"),"West");
add(new Button("Center"),"Center");
```

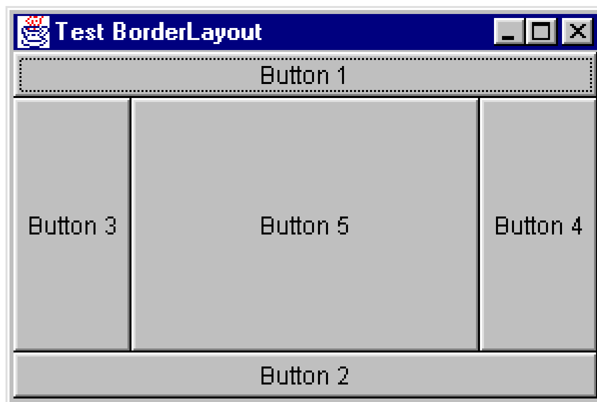


Abbildung 31.6: Verwendung der Klasse BorderLayout

Wichtig: wenn ein Container BorderLayout benutzt, müssen die Komponenten mit der add-Methode (des containers) mit zwei Parametern zugefügt werden:
Zweiter Parameter muß "North", "South", "East", "West", oder "Center" sein.
Es gibt zwei Konstruktoren:

```
public BorderLayout()    // Ohne Zwischenraum
public BorderLayout(int horizontalGap, int
   verticalGap)
```

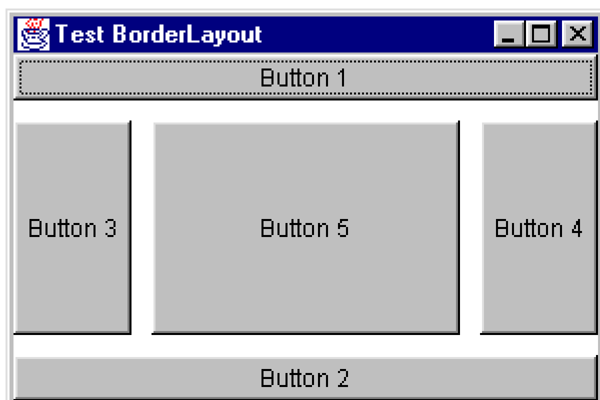


Abbildung 31.7: Ein BorderLayout mit Lücken

16.4 Dialoge

Ein Dialog ist ein Fenster mit eingeschränkter Funktionalität:
kann nicht verschoben werden,
kann nicht in Größe verändert werden.
Ein Dialog ist immer Unterfenster eines Frames,
d.h. er gehört zu einem ***parent***-Frame (ist also Kind dieser Eltern).
Ein Dialog kann
modal (blockiert den Rest des Programms bis zu seiner Beendigung)
oder
nicht modal (läßt das Programm weiterlaufen.)
sein.
In einer Dialogbox können Komponenten genauso wie im Frame verwendet werden,
außer Menuleisten.

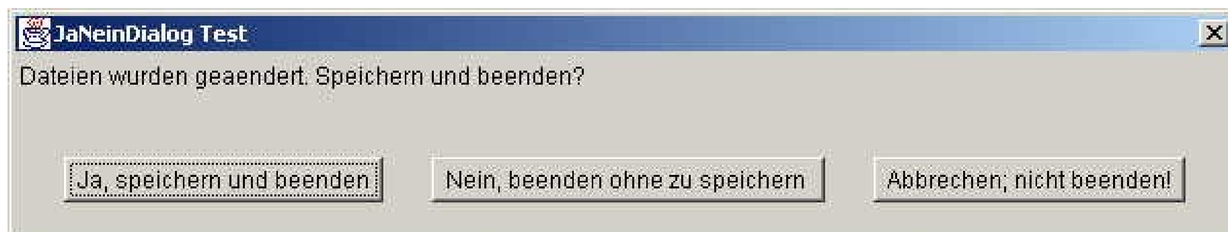
Beispiel: InfoDialog, ein einfacher Standard-Dialog zur Meldungsausgabe

Dieser Dialog ist modal, das Programm läuft erst weiter, wenn Okay geklickt wurde.

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

public class InfoDialog extends Dialog implements ActionListener
{
  protected Button okButton;
  protected Label label;

public InfoDialog(Frame parent, String title, String message,
  boolean modal)
  {
    super(parent, title, modal);
    this.setLayout(new BorderLayout(15, 15));
    label = new Label(message);
    this.add("Center", label);

    Panel p = new Panel();
    p.setLayout(new FlowLayout(FlowLayout.CENTER, 15, 15));
    okButton = new Button("Okay");
    okButton.addActionListener(this);

    p.add(okButton);
    this.add("South", p);
    this.pack();
```

```
  }
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() = = okButton)
      this.dispose();
  }

 }
```

Beispiel: Ein einfacher Standard-Dialog mit Benutzerantwort
Häufig benötigt: Dialoge, die Frage stellen und eine Antwort des Benutzers erwarten:
ja nein abbrechen
yes no cancel

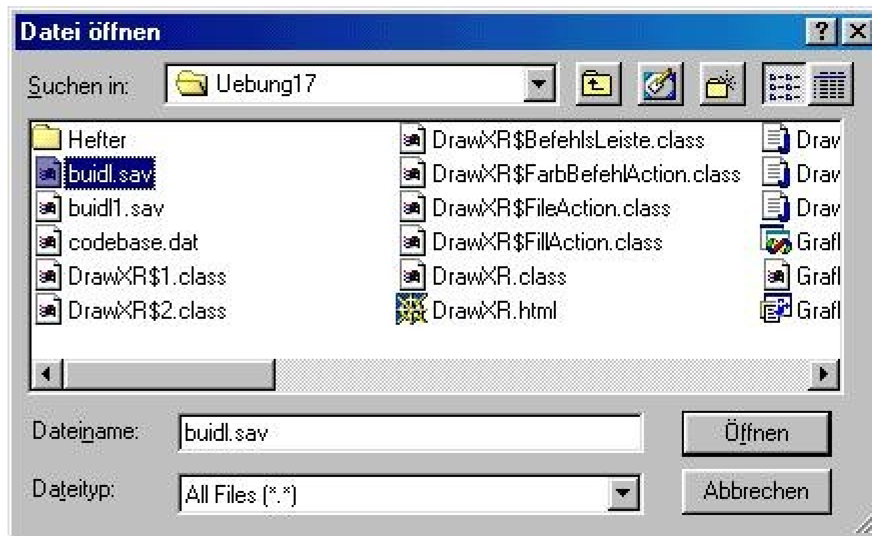



```
import java.awt.*;
import java.awt.event.*;

public class JaNeinDialog extends Dialog implements ActionListener
{


    public JaNeinDialog(Frame parent, String title, String message,
              String yes_label, String no_label, String cancel_label)
   {

    super(parent, title, true);
    this.setLayout(new BorderLayout(15, 15));
    this.add(new Label(message),"Center");
    Panel buttonbox = new Panel();
    buttonbox.setLayout(new FlowLayout(FlowLayout.CENTER, 25, 15));
    this.add(buttonbox,"South");

    if (yes_label != null) {
     Button yes = new Button(yes_label);
     yes.setActionCommand("ja");
     yes.addActionListener(this);
     buttonbox.add(yes);
    }

   if (no_label != null) {
      Button no = new Button(no_label);
      no.setActionCommand("nein");
      no.addActionListener(this);
      buttonbox.add(no);
    }
```

```
      if (cancel_label != null) {
       Button cancel = new Button(cancel_label);
       cancel.setActionCommand("abbrechen");
       cancel.addActionListener(this);
       buttonbox.add(cancel);
     }

    this.pack();
  }

protected ActionListener listeners = null;

public void addActionListener(ActionListener l) {
    listeners = AWTEventMulticaster.add(listeners, l);
  }

  public void removeActionListener(ActionListener l) {
    listeners = AWTEventMulticaster.remove(listeners, l);
  }


  public void actionPerformed(ActionEvent e) {
     this.dispose();
     if (listeners != null)
      listeners.actionPerformed(new ActionEvent(
           this, e.getID(),e.getActionCommand()));
       }

 }
```

16.5 Filedialoge

Das Paket java.io enthält eine sehr nützlichen Klasse für Dialoge zur Dateiauswahl.



Ein Filedialog wird wie folgt aufgerufen:

```
Frame ff = new Frame();
FileDialog dialog = new FileDialog(ff,
                       "Datei öffnen",FileDialog.LOAD);
dialog.show();
String pfadname = dialog.getDirectory();
String filename = dialog.getFile();
if (fileName!= null) {
String vollstaendigerDateiname = dialog.getDirectory()+
                                 dialog.getFile();
}
```

Die drei Parameter im Konstruktor: parentFrame, titelText, typ
die Klasse enthält Typkonstanten: LOAD und SAVE
(Unterschied: bei SAVE erscheint Unterdialog, wenn existierende Datei überschieben werden soll)
Ein Filedialog ist sinnvollerweise modal,
d.h. nach Aufruf der show()-Methode wartet das Programm auf die Beendigung des Dialogs.

17 Applets

17.1 Unterschiede zwischen Applets und Applikationen

Applikationen werden mit dem Java-Interpreter ausgeführt, der die Haupt-.class-Datei der Applikation lädt.
Java-Applets hingegen werden innerhalb eines Java-fähigen WWW-Browsers ausgeführt.

17.2 Sicherheitseinschränkungen von Applets

Folgende Dinge kann ein Applet nicht tun:
Applets können im Dateisystem des Benutzers weder lesen noch schreiben.

Applets können nur mit dem Internet-Server kommunizieren, von dem die Webseite stammt, die das Applet enthält.

Applets können keine Programme auf dem System des Benutzers ausführen.

Applets können keine Programme auf der lokalen Plattform laden, einschließlich gemeinsam genutzter Bibliotheken wie DLLs.
Für Java-Anwendungen gelten diese Beschränkungen nicht.
Obwohl es das Sicherheitsmodell von Java für bösartige Applets schwer macht, auf dem System des Anwenders Schaden anzurichten, ist dies keine 100prozentige Sicherheit.

17.3 Erstellen von Applets

Applets besitzen keine main()-Methode.
Statt dessen gibt es einige Methoden, die an verschiedenen Punkten der Ausführung eines Applets aufgerufen werden: 'milestone-Methoden'.

Alle Applets sind Subklassen der Klasse *java.applet.Applet*, die Subklasse von *java.awt.Panel* ist,
Vererbte Verhaltensweisen:
Verhaltensweisen, die das Applet als Teil des Browsers arbeiten lassen und Ereignisse wie das Neuladen der Seite im Browser behandeln.

Verhaltensweisen, um eine grafische Schnittstelle darstellen und Eingaben des Benutzers entgegennehmen zu können wie ein Panel.

Ein Applet hat die folgende Form:

```
public class myApplet extends java.applet.Applet {
    // Code des Applet
}
```

Applets müssen public sein. Hilfsklassen können entweder public oder private sein.

Der Browser erstellt automatisch eine Instanz der Klasse des Applets und
ruft Methoden von Applet auf, sobald bestimmte Ereignisse eintreten.
16.2 Wichtige Applet-Aktivitäten:
'milestone-Methoden': *Initialisieren*, *Starten* , *Stoppen, Zerstören* und *Anzeigen*.

17.3.1 Initialisieren

Der Browser veranlasst die Initialisierung eines Applets, wenn es vom Browser geladen wird.
Die init()-Methode eines Applet enthält Code, den andere Klassen im Konstruktur bereitstellen.

```
public void init() {
    //Code der Methode
}
```

17.3.2 Starten

Das Starten unterscheidet sich vom Initialisieren.
Während seines Lebenszyklus kann ein Applet mehrmals gestartet werden.
Die Initialisierung findet nur einmal statt.

```
public void start() {
    // Code der Methode
}
```

17.3.3 Stoppen

Ein Applet wird gestoppt, wenn der Benutzer im Browser die Seite mit dem Applet verläßt.

```
public void stop() {
    // Code der Methode
}
```

17.3.4 Zerstören

Durch das Zerstören kann ein Applet hinter sich aufräumen.
destroy()-Methode:

```
public void destroy() {
   // Code der Methode
}
```

destroy() ist vergleichbar mit *finalize()*:
destroy() ist nur für Applets anwendbar.
finalize() ist eine allgemeinere Art für ein Objekt eines beliebigen Typs, hinter sich aufzuräumen.

Java verfügt über eine automatische Speicherfreigabe, den Garbage Collector.
Aus diesem Grund muss man normalerweise Methoden wie *destroy()* und *finalize()* gar nicht verwenden

17.3.5Zeichnen (paint)

Die *paint()*-Methode ist wie beim *Panel* für die Bildschirm-Ausgabe eines *Applet* verantwortlich.

```
public void paint(Graphics g) {
    ... // Code der Methode
}
```

I

Ein einfaches Applet

```
import java.awt.*;
public class Palindrome extends java.applet.Applet {
    Font f = new Font("TimesRoman", Font.BOLD, 24);
    int count=0;
    public void init() {
        super.init();
        System.out.println("init "+this.getClass().getName());
    }
    public void start() {
        super.start();
        System.out.println("start "+this.getClass().getName());
    }
    public void paint(Graphics screen) {
        screen.setFont(f);
        screen.setColor(Color.red);
        System.out.println(++count + ". Paint ");
        screen.drawString("Dennis and Edna sinned.", 5, 30);
     }
    public void stop() {
        super.stop();
        System.out.println("stop "+this.getClass().getName());
    }
    public void destroy() {
        super.destroy();
        System.out.println("destroy "+this.getClass().getName());
    }
}
```

init(), start(), stop() oder *destroy()* hier nicht erforderlich, nur für Demo-Zwecke.

17.4 Applet in eine Webseite einfügen

Applets werden mit einer HTML-Anweisung, dem <APPLET>-Tag, in eine Seite eingefügt.

Das Tag <APPLET>

Um ein Applet auf einer Webseite einzufügen, verwenden Sie das Tag <APPLET>, das von allen Browsern unterstützt wird, die Java-Programme ausführen können.

Einfaches Beispiel für eine Webseite mit einem Applet.Palindrome.html

```
<HTML>
<HEAD>
<TITLE>The Palindrome Page</TITLE>
</HEAD>
<BODY>
My favorite palindrome is:
<BR>
<APPLET CODE="Palindrome.class" WIDTH=600 HEIGHT=100>
A secret if your browser does not support Java!
</APPLET>
</BODY>
</HTML>
```

Attribute des Tags <Applet> :
CODE.
WIDTH.
HEIGHT.

Java-Applets im Web bereitstellen

Die folgenden Dateien müssen auf den Server übertragen werden:
Die HTML-Seite, die das Applet beinhaltet.

Alle .class-Dateien, die von dem Applet verwendet werden und nicht Teil der Standard-Klassenbibliothek von Java sind.

CODE und CODEBASE
CODE dient zur Bezeichnung des Namens der .class-Datei, in der sich das Applet befindet.

```
    <APPLET CODE="Palindrome.class" HEIGHT=40 WIDTH=400>
    </APPLET>
```

Attribut CODEBASE: Suche nach dem Applet in einem anderen Ordner:

```
    <APPLET CODE="Palindrome.class" CODEBASE="j2-16-dateien\Palindrome"
HEIGHT=40 WIDTH=400>
    </APPLET>
```

17.5 Java-Archive

Ein Java-Archiv ist eine Sammlung von Java-Klassen oder anderen Dateien, die in eine einzige Datei gepackt werden (JAR-Dateien)

Der folgende Befehl packt alle Klassendateien und GIF-Bilddateien eines Verzeichnisses in ein einziges Java-Archiv mit dem Namen Animate.jar:

```
Jar cf Animate.jar *.class *.gif
```

c gibt an, daß eine Java-Archivdatei erstellt werden soll,
f legt fest, daß der Name der Archivdatei als nächstes Argument folgt.

Einzelne Dateien zu einem Java-Archiv hinzufügen:

```
    jar cf Smiley.jar ShowSmiley.class ShowSmiley.html spinhead.gif
```

jar ohne Argumente ausführen: Liste der verfügbaren Optionen.
Nachdem ein Java-Archiv erstellt ist, wird das Attribut ARCHIVES mit dem Tag <APPLET> verwendet, um anzugeben, wo sich das Archiv befindet. Sie können Java- Archive in einem <APPLET>-Tag wie folgt benutzen:

```
    <applet code="ShowSmiley.class" archives="Smiley.jar" width=45
height=42>
    </applet>
```

Weitere Archiv-Formate
Weitere Archiv-Formate sind möglich. aber Browser-abhängig
Beispiele:

```
    <APPLET CODE="MyApplet.class" ARCHIVE="appletstuff.zip" WIDTH=100
HEIGHT=100>
    </APPLET>

    <APPLET CODE="DanceFever.class" WIDTH=200 HEIGHT=450>
    <PARAM NAME="cabbase" VALUE="DanceFever.cab">
    </APPLET>
```

17.6 Parameter an Applets weitergeben

Elemente zur Handhabung von Parametern:
Ein spezielles Parameter-Tag in der HTML-Datei
Code im Applet, der diese Parameter analysiert

```
<APPLET CODE="QueenMab.class" WIDTH=100 HEIGHT=100>
<PARAM NAME="font" VALUE="TimesRoman">
<PARAM NAME="size" VALUE="24">
A Java applet appears here.
</APPLET>
```

Übernahme der Parameter im Applet mit der getParameter()-Methode:

```
String theFontName = getParameter("font");
```

Falls ein erwarteter Parameter nicht in der HTML-Datei angegeben wurde, gibt getParameter() Null zurück.
Deshalb muss man prüfen:

```
if (theFontName == null)
theFontName = "Courier"
```

Für numerische Werte Typwandlung erforderlich, da *getParameter()* nur String zurückliefert:

```
int theSize;
String s = getParameter("size");
if (s == null)
    theSize = 12;
else
    theSize = Integer.parseInt(s);
```