Einführung in die Programmierung mit Java

Dr. E.Gutknecht, Dr. W.Meier

Vorlesungs-Skriptum Wintersemester 2001/2002

Inhaltsverzeichnis

Einleitung

Was ist Java ?

Java ist eine objektorientierte Programmiersprache, die von Sun Microsystems entwickelt wurde (James Gosling und Mitarbeiter, 1990-1995). Der grosse Erfolg von Java beruht auf einer neuen Art von Programmen, die mit Java erstellt werden können: Java-Applets für das Internet.

Arten von Java-Programmen

- *Applikationen*

  Java-Applikationen sind konventionelle Programme (wie Word, oder Excel), die auf dem Computer, auf dem sie verwendet werden, installiert und als eigenständige Programme ausgeführt werden.

- *Applets*

  Ein Applet ist ein Java-Programm, welches in einem HTML-Dokument aufgerufen wird. Wenn das HTML-Dokument mit einem Internet-Browser (Netscape, Internet Explorer) geöffnet wird, wird das Applet geladen und läuft in einem Window innerhalb des Dokumentes ab.

  Dabei spielt es keine Rolle, ob das HTML-Dokument lokal oder via Internet von einem anderen Computer angefordert wird. Im letzteren Fall wird das Applet (wie das HTML-File) auf den lokalen Computer heruntergeladen und auf diesem innerhalb des Internet-Browsers ausgeführt.

  Beispiel:

  Darstellung einer laufenden Uhr auf der Homepage der Firma Mondaine (www.mondaine.ch).

- *Servlets*

  Servlets sind eine Alternative zu Applets für kommerzielle Internet-Anwendungen (E-Commerce). Sie werden ebenfalls von einem HTML-Dokument aus aufgerufen, sie laufen jedoch im Gegensatz zu Applets direkt auf dem Web-Server, auf dem sie gespeichert sind.

  Input-Daten:

  Die Input-Daten für ein Servlet werden auf einem HTML-Formular auf dem Client eingegeben und an das Servlet gesendet.

  Output:

  Das Servlet stellt die Antwortdaten im HTML-Format zusammen und sendet diese als HTML-Dokument an den Client zurück, wo es vom Internet-Browser dargestellt wird.

  Der Browser auf dem Client muss bei diesem Konzept lediglich HTML verarbeiten können, kein Java. Weiter bringen Servlets gegenüber Applets signifikante Performance-Verbesserungen, da sie nicht auf den Client heruntergeladen werden, sondern auf dem Server laufen und dort die Requests der Clients verarbeiten.

  Beispiel:

  Die Abfrage des Postleitzahlen-Verzeichnisses der Schweiz.
  (www.post.ch → Briefpost → PLZ)

- Javascript

  Javascript wurde von Netscape eingeführt. Dabei werden Java-Statements direkt in ein HTML-Dokument eingefügt und beim Öffnen des Dokumentes vom Browser interpretiert. Dabei steht nur eine Untermenge von Java zur Verfügung.

  Einsatz-Beispiel:

  Für zusätzliche Effekte auf einer HTML-Page, z.B. für Navigations-Spalten: Beim Überstreichen eines Textes mit der Maus erscheint ein Menu. (Beispiel: www.unizh.ch).

Die Java Virtual Machine (JVM)

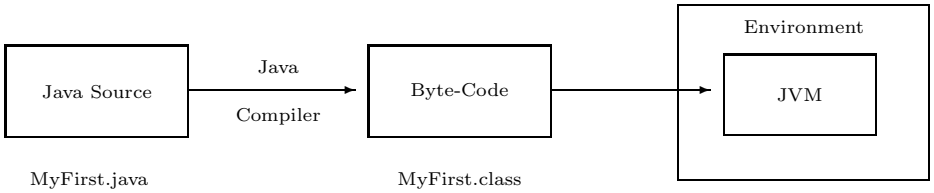
Bei traditionellen Programmiersprachen wie C, C++, Ada oder Pascal werden Programme vom Compiler (= Übersetzer) in die Maschinensprache für ein bestimmtes System, z.B. Windows, übersetzt.

Das übersetzte Programm ist folglich nur auf diesem System lauffähig und damit nicht geeignet für Internet-Anwendungen.

Aus diesem Grund hat Java einen anderen Weg gewählt:

- Der Java Compiler übersetzt ein Programm in einen *maschinenunabhängigen* Zwischencode, den sogenannten *Byte-Code* des Programmes. Byte-Code
- Bei der Ausführung des Programmes wird der Byte-Code vom Java Interpreter, der sogenannten *Java Virtual Machine*, interpretiert, d.h. in den Maschinen-Code des betreffenden Systems umgesetzt und ausgeführt.

Für die Ausführung des Byte-Codes eines Java Programmes ist also eine Java Virtual Machine (JVM) erforderlich, welche Bestandteil eines Internet-Browsers oder des Betriebssystems sein kann kann.



Zusammenfassung und Ausblick

Bestandteile von Java

Technisch gesehen umfasst der Begriff 'Java' die folgenden drei Bestandteile:

- Die objektorientierte *Programmiersprache* Java.
- Die *Java Virtual Machine* (JVM), eine Runtime Umgebung, in der Java Programme laufen.
- Die Java *Library*, eine umfassende, standardisierte Bibliothek mit Hilfsmitteln (Packages) für die Programm-Entwicklung:

Merkmale der Sprache Java

Bevor wir mit der Einführung der Programmiersprache Java beginnen, notieren wir einige charakteristische Eigenschaften der Sprache, für Leser, die schon andere Programmiersprachen kennen:

- Kontroll-Anweisungen gemäss ANSI C-Syntax:

  'for'-, 'while'- und 'do'-Schleifen, 'if'- und 'switch'-Statements

- Konsequente Verwendung der objektorientierten Denkweise

- Arrays und Strings sind Objekte. Dadurch sind sie selbstbeschreibend, z.B. kann ein Unterprogramm (im Gegensatz zu C/C++) jederzeit abfragen, wieviele Elemente ein übergebener Array oder String hat.

- Keine Pointer-Variablen und Pointer-Arithmetik. Dies erhöht die Sicherheit von Programmen enorm, zusammen mit dem Garbage Collector (nächster Punkt).

- Der Garbage Collector, der unsichtbar im Hintergrund läuft, gibt Objekte, die nicht mehr in Gebrauch sind, automatisch frei. Der Programmierer muss sich nicht um dieses Problem kümmern.

- Fehlerhandling mittels Exceptions (wie C++, aber konsequenter verwendet)

- Hilfsmittel zur Erzeugung und Verwaltung von Threads

Kapitel 1

Grundlagen

'Where shall I begin, please your Majesty?' 'Begin at the beginning,' the king said, gravely, 'and go on till you come to the end: then stop.'

– Alice in Wonderland

Wir führen in diesem Kapitel die Grundkonzepte der Programmierung ein: Variablen, Operationen, Input/Output, Ablaufsteuerung, Unterprogramme. Diese Konzepte bilden den Schlüssel für den Einstieg in die Welt der Programmierung.

In den folgenden Kapiteln werden die Konzepte vertieft.

1.1 Das erste Programm

Das folgende Listing zeigt ein möglichst einfaches Java Programm 'MyFirst'. Das Programm gibt zwei Begrüssungs-Zeilen auf den Bildschirm aus.

```
public class MyFirst
{
  static public void main( String[ ] args )
  {
    System.out.println("Hallo");
    System.out.println("Dies ist mein erstes Programm");
  }

}
```

Dies ist der *Quellen-Code* (Source-Code) des Programmes. Er wird mit einem Text-Editor eingegeben und als Text-File abgespeichert. Anschliessend wird das Programm mit dem *Java Compiler* übersetzt (compiliert) und dann mit dem *Java Interpreter* ausgeführt, Details dazu siehe unten.

Applikationen

In Java gibt es zwei Arten von Programmen: *Applikationen* und *Applets*. Applikationen sind konventionelle Programme, die als eigenständige Programme ausgeführt werden. Das Programm 'MyFirst' ist eine Applikation.

Applets

Applets sind eine Besonderheit von Java. Sie werden in einem HTML-Dokument aufgerufen, wenn dieses mit einem Internet-Browser (Netscape oder Internet-Explorer) lokal oder über Internet geöffnet wird.

Programm-Aufbau

Jedes Java-Programm wird im Rahmen einer 'class'-Anweisung geschrieben, in welcher der Name des Programmes festgelegt wird:

```
public class Programm-Name
{ ...
}
```

Dabei ist 'Programm-Name' ein beliebig wählbarer Name für das Programm. Die Angabe 'public' (öffentlich) ist nötig, damit das Programm vom Java Interpreter ausgeführt werden kann.

Mit dem Konzept der Klassen werden wir uns später intensiv beschäftigen, im Moment nehmen wir nur zur Kenntnis, dass Java-Programme innerhalb eines 'class'-Statements geschrieben werden.

Die Methode 'main'

Die Anweisungen des Programmes stehen in der Methode 'main':

```
static public void main( String[ ] args )
{
  System.out.println("Hallo");
  System.out.println("Dies ist mein erstes Programm");
}
```

Methoden

Eine *Methode* ist eine Programmeinheit, die aufgerufen werden kann zur Erfüllung einer bestimmten Aufgabe.

Jede Java Applikation hat eine Methode 'main', die Hauptmethode. Sie wird bei der Ausführung des Programmes vom Java Interpreter aufgerufen. Die Kopfzeile der Methode ist für alle Programme gleich, die Bedeutungen der verschiedenen Angaben wird später eingeführt.

In den geschweiften Klammern der Methode stehen die Anweisungen des Programmes, die beiden 'println'-Befehle. Mit 'println' wird eine Textzeile auf den Bildschirm ausgegeben, wie es der Name (print line) andeutet. Der auszugebende Text muss in doppelten Anführungszeichen geschrieben werden, als Sequenz von Characters (Character-String).

println

Neben 'println' steht der Befehl 'print' zur Verfügung, der sich von 'println' nur dadurch unterscheidet, dass nach der Ausgabe des Textes auf dem Bildschirm *kein* Zeilenvorschub erfolgt. Dies wird benötigt, wenn nachfolgende Ausgaben auf der gleichen Bildschirmzeile erfolgen sollen.

print

Notations-Konventionen

In Java müssen die folgenden Notations-Konventionen eingehalten werden.

- Gross-/Kleinschrift

  Gross-/Kleinschrift muss in Java beachtet werden, sie wird vom Compiler unterschieden. Dies gilt auch für alle Java-Bezeichnungen und -Befehle:

  `class`, `System.out.println`, `if`, `else`, `while`, `for` usw.

- Leerzeichen und Zeilenstruktur

  Zur Unterteilung von Wörtern (Befehle, Variablen-Namen usw.) werden Leerzeichen verwendet. Mehrere Leerzeichen sind für den Compiler gleichbedeutend wie ein einziges.

  Zur Erhöhung der Übersicht können Leerzeichen (oder Tabulatoren) nach Belieben eingefügt werden, z.B. für Einrückungen am Anfang der Zeilen (siehe obiges Beispiel).

  Die Aufteilung des Programmes auf verschiedene Zeilen ist ebenfalls dem Programmierer überlassen. Ein Zeilensprung (newline) ist für den Compiler gleichwertig wie ein Leerzeichen.

  Theoretisch kann das ganze Programm auf einer Zeile geschrieben werden. Zur Erhöhung der Lesbarkeit eines Programmes sollten jedoch Einrückungen verwendet werden, wie im oberen Beispiel.

- Strichpunkte

  Jedes Java-Statement wird mit einem Strichpunkt abgeschlossen:

  ```
  System.out.println("Hello");
  ```

  Nach einer geschweiften Schlussklammer '}' steht hingegen kein Strichpunkt (z.B. am Ende des Programmes).

- Kommentare

  Kommentare sind Erklärungen in einem Programm für den Leser, ohne Einfluss auf das Programm. Ein Kommentar wird eingeleitet mit '//' (zwei aufeinanderfolgende '/') und erstreckt sich bis zum Ende der betreffenden Zeile.

  ```
  System.out.println("Hello");          // Bildschirm−Ausgabe
  ```

Uebersetzung und Ausführung des Programmes

Das Programm wird mit einem Text-Editor eingegeben und dann in ein File 'MyFirst.java' abgespeichert. Der vordere Teil des File-Namens, 'MyFirst', muss mit dem im Programm angegebenen Namen übereinstimmen (inkl. Gross-/Kleinschrift).

Compiler Anschliessend wird das Programm mit dem *Java Compiler* 'javac' übersetzt (compiliert):

```
javac MyFirst.java
```

(Gross-/Kleinschrift beachten!) Der Compiler übersetzt das Programm
Byte-Code und erzeugt das File 'MyFirst.class', welches den sog. *Byte-Code* des Programmes enthält. Dies ist ein maschinenunabhängiger Code, der auf verschiedenen Betriebssystemen (Windows, Unix, MacOS) ausgeführt werden kann.

Die Ausführung des Programmes erfolgt schliesslich mit dem Befehl:

```
java MyFirst
```

Interpreter Dadurch wird der Java Interpreter 'java' gestartet, welcher das File 'MyFirst.class' mit dem Byte-Code ausführt.

1.2 Variablen, Konstanten und Ausdrücke

Variablen sind die Grundbausteine von Programmen. Eine Variable ist ein Speicherplatz zur Abspeicherung eines Wertes. Jede Variable hat einen Namen und einen *Datentyp*. Der Datentyp legt die möglichen Werte der Variablen fest.

Grundlegende Datentypen:

Datentyp	Wertebereich
int	ganze Zahlen $0, \pm 1, \pm 2, \ldots$
double	reelle Zahlen mit Dezimalstellen (doppelte Genauigkeit), z.B. 3.141592653589793
char	Characters, Ziffern und Sonderzeichen. Character-Werte werden in Apostrophen angegeben: 'A', 'B', 'a', 'b', '0', '1', ' ' (Leerzeichen), usw.
boolean	false und true (für logische Variablen)

Definition von Variablen:

Jede Variable muss definiert werden. Dabei wird der Datentyp angegeben, gefolgt von einem beliebigen Namen für die Variable:

```
int  i;
double  xMittel;
```

Dadurch wird der Speicherplatz für die Variablen reserviert. Anfangswerte (z.B. 0) werden *keine* automatisch eingesetzt, der Programmierer muss selber Werte eintragen, bevor die Variablen verwendet werden können (Ausnahmen siehe später, bei Arrays und Objekten).

Man beachte den Strichpunkt, der nach jedem Java-Statement erforderlich ist (siehe oben, Notations-Konventionen). Weiter erinnern wir daran, dass Klein- und Gross-Buchstaben unterschieden werden: *a* und *A* sind verschiedene Variablen-Namen!

Mehrere Variablen desselben Typs können in *einem* Statement definiert werden:

```
int n, m;
```

Der Name einer Variablen ist eine beliebige Zeichenkette aus Klein- und Grossbuchstaben, Ziffern und den Sonderzeichen '_', '$'. Das erste Zeichen darf keine Ziffer sein.

Namenskonventionen für Variablen

In Java hat sich die Konvention durchgesetzt, dass Namen von Variablen mit einem Kleinbuchstaben beginnen, und bei Beginn eines neuen Wortes wird der erste Buchstabe gross geschrieben:

anfangsPunkt, endPunkt, xMittel, usw.

Wertzuweisungen

Zuweisungs-Operator

Mit dem Operator '=' wird einer Variablen ein Wert zugewiesen. Man nennt diesen Operator daher den *Zuweisungs-Operator*. Er hat die allgemeine Form:

Variable = *Wert*

Dabei wird der auf der rechten Seite angegebene Wert in der Variablen der linken Seite abgespeichert. Beispiele von Zuweisungen:

```
i = 4;                          // i erhält den Wert 4
xMittel = −3.45;
```

Eine Wertzuweisung kann auch direkt bei der Definition einer Variablen erfolgen:

```
int p = 4;                      // Definition mit Initialisierung
```

Der Zuweisungs-Operator von Java ist also '=' und nicht der von Pascal verwendete Operator ':='. Damit stimmt Java mit C/C++ überein.

Auf der rechten Seite darf auch ein zusammengesetzter Ausdruck stehen:

```
xMittel = 0.5 * (x1 + x2);
```

Der Ausdruck auf der rechten Seite wird ausgewertet, anschliessend wird der Wert in die Variable auf der linken Seite übertragen. Daher darf die Variable auf der linken Seite auch im Ausdruck der rechten Seite vorkommen:

```
i = i + 1;
i = i − 1;
```

Wirkung: Der Inhalt der Variablen i wird um 1 erhöht bzw. vermindert.

Inkrement-Operator

Für diese beiden Operationen stehen weiter die von C/C++ bekannten *Inkrement-* und *Dekrement-Operatoren* '++' und '−−' zur Verfügung:

```
i++;                            // Wert um 1 erhoehen
i−−;                            // Wert um 1 vermindern
```

Ausdrücke

Algebraische Ausdrücke werden wie in der Mathematik geschrieben, mit den unten aufgeführten Operatoren und (wenn nötig) mit runden Klammern zur Beeinflussung der Auswertungsreihenfolge. Ohne Klammern erfolgt die Auswertung nach den üblichen Regeln der Mathematik (Multiplikation und Division vor Addition und Subtraktion).

Operator	Bedeutung
+, −	Addition , Subtraktion
*, /	Multiplikation, Division

Ein Potenzoperator ist nicht vorhanden. Ersatz ist die mathematische Funktion 'Math.pow', siehe unten, Seite 18.

Beispiele von Ausdrücken:

```
d = b*b − 4*a*c;                    // Diskriminante
x = b / (2*a);
```

Man beachte, dass die Klammern im zweiten Ausdruck wichtig sind, wenn man den mathematischen Ausdruck $\frac{b}{2a}$ berechnen will. Ohne Klammern stellt $b / 2 * a$ den Wert $\frac{b}{2} \cdot a$ dar, da Multiplikation und Division Operationen gleicher Stufe sind, also ohne Klammern von links nach rechts ausgewertet werden.

Achtung:

Wenn bei einer Division beide Operanden den Datentyp 'int' haben, wird die Operation ganzzahlig, d.h. *ohne* Dezimalstellen ausgeführt. Details dazu werden später diskutiert (Seite 52).

Konstanten

Konstanten werden wie Variablen definiert, mit dem Zusatz 'final' und mit einer Wertzuweisung:

```
final double g = 9.81;              // Erdbeschleunigung
final double m = 0.2;               // Masse
final double gewicht = m * g;
final int max = 100;
```

Der Wert einer Konstanten kann nicht verändert werden. Konstanten erhöhen die Lesbarkeit und die Wartbarkeit von Programmen!

Bildschirmausgabe

Der Inhalt einer Variablen wird wie ein konstanter Text mit den Befehlen 'System.out.print' oder 'println' ausgegeben. Dabei wird anstelle eines Textes "xxx" der Name der auszugebenden Variablen ohne Apostrophe angegeben:

```
int n = 12;
double x = −3.42;
System.out.print(n);
System.out.println(x);
```

Der auszugebende Wert kann auch als zusammengesetzer Ausdruck angegeben werden:

```
System.out.println(−3.5*x + 2*x*x);
```

1.3 Mathematische Funktionen

Die mathematischen Funktionen werden in Java mit dem Präfix 'Math.' aufgerufen:

```
y = Math.sin(x);                    // Aufruf der Sinus−Funktion
z = x + Math.sin(x);                // Aufruf in einem Ausdruck
double u = Math.sqrt(2);            // Aufruf der Wurzelfunktion in Initialisierung
```

Das Argument kann auch ein zusammengesetzter Ausdruck sein:

```
y = Math.sin(2*x−0.2);
```

Funktionen:

abs(x)	absoluter Betrag
sin(x)	Sinus-Funktion (x in Radians)
cos(x)	Cosinus-Funktion
tan(x)	Tangens-Funktion
asin(x)	Arcus-Sinus (Umkehrfunktion von sin)
acos(x)	Arcus-Cosinus (Umkehrfunktion von cos)
atan(x)	Arcus-Tangens (Umkehrfunktion von tan)
exp(x)	e^x
pow(x,y)	x^y ($x > 0$ oder y ganzzahlig)

sqrt(x)	Qudratwurzel ($x \geq 0$)
log(x)	natürlicher Logarithmus ($x > 0$)
random()	Zufallszahl im Intervall $[0.0, 1.0]$
round(x)	Umwandlung in ganze Zahl mit Runden (Anwendung siehe Seite 54)

Konstanten:

Math.PI	Zahl $\pi = 3.141592653589793$
Math.E	Eulersche Zahl $e = 2.718281828459045$

Bemerkungen:

1. Bogenmass

   Bei den trigonometrischen Funktionen sin, cos, tan muss der Winkel im Bogenmass (Radians) angegeben werden. Umrechnung eines Winkels in das Bogenmass:

   ```
   winkelRad = winkelGrad * Math.PI /180
   ```

2. Funktionen ohne Argumente

   Beim Aufruf einer Funktion wie 'random', die kein Input-Argument hat, müssen leere Klammern angegeben werden, sonst erkennt der Compiler nicht dass es ein Funktionsaufruf ist.

   ```
   double zufallszahl = Math.random();
   ```

1.4 Einführung von Schleifen

Die echte Leistungsfähigkeit von Programmen kommt erst durch Schleifen zustande. Bei einer Schleife wird eine Verarbeitung wiederholt durchgeführt, solange ('while') eine bestimmte Bedingung erfüllt ist.

Es gibt verschiedene Statements zur Realisierung von Schleifen. Im Moment führen wir nur das wichtigste ein, das 'while'-Statement.

Beispiel: Ausgabe von 100 Zufallszahlen

```
public class ZufallsZahlen
{
  static public void main( String[ ] args )
  { int i = 1;                                  // Zaehler
    while ( i <= 100 )
    { System.out.println(Math.random());        // Zufallszahl ausgeben
      i++;                                      // Zaehler erhoehen
    }
  }
}
```

Erklärung:

Die beiden Anweisungen in den geschweiften Klammern des 'while'-Statements werden wiederholt ausgeführt, solange die Bedingung $i <= 100$ erfüllt ist. Da i am Anfang auf 1 gesetzt und bei jedem Durchlauf um 1 erhöht wird, erfolgen 100 Durchläufe.

Allgemeines Format des 'while'-Statements:

```
while ( Bedingung )
{ Anweisungen
}
```

Die Bedingung wirkt als Fortsetzungsbedingung: Die Anweisungen werden solange ausgeführt, wie die Bedingung erfüllt ist.

Die Bedingung wird bei jedem Durchlauf *vor* der Verarbeitung geprüft. Wenn sie falsch ist, wird die Verarbeitung nicht mehr durchgeführt, und das Programm wird mit der Anweisung nach der Schlussklammer weitergeführt (wenn vorhanden).

Wenn die Bedingung eines 'while'-Statements schon bei Beginn der Schleife falsch ist, wird die Verarbeitung infolgedessen keinmal ausgeführt.

Bedingungen

Boolescher Ausdruck

Die Bedingung in einem 'while'-Statement ist ein Ausdruck, der wahr oder falsch sein kann, ein sog. *Boolescher Ausdruck*. Im Moment verwenden wir nur solche Boolesche Ausdrücke, die aus einer Vergleichs-Operation mit einem der folgenden Operatoren bestehen:

Operator	Bedeutung
<	kleiner
<=	kleiner oder gleich
==	gleich
!=	ungleich
>=	grösser oder gleich
>	grösser

▶ *Merke:*

Der Vergleichsoperator für Gleichheit ist '==', da das Zeichen '=' schon für den Zuweisungsoperator vergeben ist.

Beispiel: Quadratzahlen

Das folgende Programm gibt alle Quadratzahlen (1, 4, 9, ...), welche kleiner als 1000 sind, auf den Bildschirm aus:

```
// ———— Quadratzahlen ————————————
public class QuadratZahlen
{
  static public void main( String[ ] args )
  { int i = 1;
    while ( i*i < 1000 )
    { System.out.print("Quadrat von ");
      System.out.print(i);
      System.out.print(": ");
      System.out.println(i*i);
      i++;
    }
  }
}
```

Bemerkung: Seitenweise Ausgabe

Bei Bildschirmausgaben die mehr als eine Seite umfassen, kann man das 'more'-Utility von DOS/Windows bzw. Unix zu Hilfe nehmen, um die Daten seitenweise anzuzeigen. Zu diesem Zweck wird das Java Programm folgendermassen gestartet:

```
java Quadratzahlen | more
```

Dadurch wird die Ausgabe des Programmes Quadratzahlen nicht direkt auf den Bildschirm ausgegeben, sondern an das 'more'-Utility weiterge- *'more'-Utility*

leitet, welches die Daten seitenweise ausgibt.

Das 'more'-Utility wartet nach jeder ausgegebenen Seite auf eine Bestätigung des Benutzers zum Weiterfahren (*Enter*-Taste unter DOS/Windows, bzw. *Space*-Taste unter Unix).

Das Zeichen '|' ist auf der schweizerdeutschen Tastatur AltGr 7.

△ **Uebung:** Lösen Sie die Aufgabe 1.8.1.

1.5 Einführung von Fallunterscheidungen

Bei einer Fallunterscheidung oder Selektion wird aufgrund einer Bedingung einer von zwei Fällen ausgewählt. Fallunterscheidungen werden mit dem 'if'-Statement durchgeführt:

if .. else

```
if ( n >= 0 )
  System.out.print("positiv oder 0");
else
  System.out.print("negativ");
```

Funktionsweise:

Wenn die Bedingung `n >= 0` erfüllt ist, wird die erste Anweisung durchgeführt, sonst die nach dem Schlüsselwort 'else'.
Die Bedingung eines 'if'-Statements hat dasselbe Format wie beim while-Statement.

Achtung:

Wenn in einem 'if'-Statement bei einem Fall mehrere Anweisungen durchzuführen sind, müssen sie wie beim while-Statement in geschweifte Klammern eingeschlossen werden:

```
if ( n >= 0 )
{ System.out.println("positiv oder 0");
  System.out.print("Betrag: ");
  System.out.println(n);
}
else
{ System.out.println("negativ");
  System.out.print("Betrag: ");
  System.out.println(−n);
}
```

'if' ohne 'else'

Wenn bei einem 'if'-Statement im 'else'-Fall nichts zu tun ist, kann dieser weggelassen werden. Das Statement hat dann nur eine Wirkung, wenn die Bedingung erfüllt ist.

```
if ( n < 0 )
   n = −n;
```

Kombination von Schleifen und Selektionen

Schleifen und Selektionen können beliebig kombiniert werden. Das folgende Programm simuliert 100 Münzenwürfe und zählt, wie oft 'Kopf' bzw. 'Zahl' aufgetreten ist.

Zur Simulation eines Wurfes der Münze wird mit der mathematischen Funktion 'Math.random()' eine Zufallszahl im Intervall $[0, 1]$ produziert.

Wenn der erhaltene Wert kleiner oder gleich 0.5 ist, interpretieren wir dies als 'Kopf', die übrigen Werte als 'Zahl'.

Die Programmstruktur ist eine Schleife, in der die 100 Würfe simuliert werden. Bei jedem Wurf mit Resultat 'Kopf' wird der Zähler 'nKopf' um 1 erhöht.

```
// ———— Muenzenwuerfe ————————————————————————————
public class Muenze
{
  static public void main(String[ ] args)
  { final int nWuerfe = 100;                    // Anzahl Wuerfe
    int i = 0;
    int nKopf = 0;                              // Zaehler fuer 'Kopf'
    double x;
    while ( i < nWuerfe )
    { x = Math.random();                        // Zufallszahl
      if ( x <= 0.5 )
        nKopf++;
      i++;
    }
    System.out.print("Anzahl Wuerfe mit 'Kopf': ");
    System.out.println(nKopf);
    System.out.print("Anzahl Wuerfe mit 'Zahl': ");
    System.out.println(nWuerfe − nKopf);
  }
}
```

△ **Uebung:** Lösen Sie die Aufgaben 1.8.2 und 1.8.3

1.6 Einführung von Methoden

Das Geheimnis der Kunst der Programmentwicklung ist die Aufteilung einer Aufgabe in Teilaufgaben, die von *Unterprogrammen* ausgeführt werden. Unterprogramme heissen in Java *Methoden*, die beiden Begriffe sind Synonyme.

Auch die Methode 'main' einer Applikation kann als Unterprogramm aufgefasst werden, das vom Java Interpreter aufgerufen wird.

Bei einer geeigneten Unterteilung können die Teilaufgaben unabhängig voneinander gelöst werden. Man kann sich dann bei der Erarbeitung einer Hilfsmethode auf ein kleines, überblickbares Problem konzentrieren und das ganze Umfeld vergessen. Dies ist das *Lokalitätsprinzip*.

Wir kennen schon einige Methoden der Java Library:

```
y = Math.sin(x);
System.out.println("Hello");
```

Dies sind Methodenaufrufe. Mit dem ersten Befehl wird die Methode 'Math.sin' aufgerufen, mit dem zweiten 'System.out.println'. In den runden Klammern werden Parameter (Argumente) an die Methoden übergeben.

Man unterscheidet zwei Arten von Methoden:

- *Funktionen*

  Eine Funktion ist eine Methode, welche einen Wert als Resultat zurückgibt, wie die Sinus-Funktion 'Math.sin'. Das Resultat wird direkt mit dem Namen der Methode angesprochen und kann z.B. einer Variablen zugewiesen werden:

  ```
  y = Math.sin(x);
  ```

- *Prozeduren*

  Eine Prozedur ist eine Methode, die eine bestimmte Aufgabe erfüllt, z.B. eine Bildschirm-Ausgabe, welche kein eigentliches Resultat in der Form eines Wertes erzeugt:

  ```
  System.out.println("Hello");
  ```

Definition und Verwendung von Methoden

In einem Programm können neben der 'main'-Methode weitere Hilfsmethoden definert werden, die in der 'main'-Methode (oder in einer anderen Methode) aufgerufen werden.

Diese Hilfsmethoden werden analog zur 'main'-Methode vor oder nach dieser definiert.

Beispiel:

Wir führen im obigen Münzenwurf-Programm eine Hilfsmethode 'werfeMuenze' ein. Es ist eine Funktion, die einen Münzenwurf simuliert und als Resultat 0 (für Kopf) oder 1 (Zahl) zurückgibt.
Die eingerahmten Nummern verweisen auf nachfolgende Erklärungen.

```
public class Muenze
{
  static int werfeMuenze()                                          ①
  { double x = Math.random();
    if ( x <= 0.5 )
      return 0;                                                     ②
    else
      return 1;
  }

  static public void main(String[ ] args)
  { final int nWuerfe = 100;
    int i = 0;
    int nKopf = 0;
    while ( i < nWuerfe )
    { if ( werfeMuenze() == 0 )                                     ③
        nKopf++;
      i++;
    }
    System.out.print("Anzahl Wuerfe mit 'Kopf': ");
    System.out.println(nKopf);
    System.out.print("Anzahl Wuerfe mit 'Zahl': ");
    System.out.println(nWuerfe − nKopf);
  }
}
```

Erklärungen:

1. Definition der Methode

   Der Kopf der Funktion beginnt mit dem Attribut 'static', welches bei jeder Methode in einer Applikation (im Gegensatz zu einem Applet) angegeben werden muss. Die Bedeutung des Attributes wird später eingeführt.

Anschliessend folgen der *Datentyp des Resultates* ('int') und ein beliebig wählbarer *Name* der Funktion ('werfeMuenze').

Die leeren runden Klammern nach dem Namen sind für die Definition von Parametern vorgesehen (s. unten).

In den geschweiften Klammern stehen die Anweisungen der Funktion.

2. Return

return

Der Befehl 'return', gefolgt von einem Wert, beendet die Funktion mit dem angegebenen Wert als Resultat. Der Wert kann auch ein zusammengesetzter Ausdruck sein. Er wird mit einer internen Zuweisung dem Resultat zugewiesen, dann wird die Funktion beendet.

3. Aufruf der Methode

Die Funktion 'werfeMuenze' wird in der Schleife der 'main'- Methode aufgerufen. Dazu wird ihr Name, gefolgt von leeren runden Klammern angegeben. Die Klammern sind für Parameter (Argumente) vorgesehen.

Lokale Variablen

Die in einer Methode definierten Variablen sind nur innerhalb der betreffenden Methode bekannt. Man nennt sie daher *lokale Variablen*. Sie werden jedesmal, wenn die Methode aufgerufen wird, im Speicher angelegt, und wenn die Methode geendet hat wieder gelöscht. Der Inhalt einer lokalen Variable geht also beim Ende einer Methode verloren!

Für die lokalen Variablen wird ein spezieller Bereich des Speichers

Stack

verwendet, der sogenannte *Stack* (Stapel).

Parameter

Mit Parametern können einer Methode beim Aufruf Werte übergeben werden. Ein Parameter ist eine lokale Variable der Methode, die beim Aufruf der Methode vom aufrufenden Programm einen Wert erhält.

Die Parameter einer Methode werden in der Parameterliste nach dem Namen der Methode definiert (wie Variablen, unterteilt durch Kommas).

Beispiel:

Die folgende Funktion berechnet das arithmetische Mittel von zwei 'double'-Werten und gibt dieses als Resultat zurück:

```
static double mittel(double a, double b)
{
    return 0.5 * (a + b);
}
```

Aufruf der Funktion:

Beim Aufruf muss für jeden Parameter ein zugehöriger Wert übergeben werden (wie bei den mathematischen Funktionen):

```
double x = 10;
double y = -5;
double m = mittel(x, y);
double r = mittel(x+y, 2*x-y);          // Ausdruecke als Parameter
```

Die übergebenen Werte nennt man *aktuelle Parameter*. Es können zusammengesetzte Ausdrücke mit Variablen und Konstanten sein. Sie werden der Reihe nach ausgewertet und in die zugehörigen lokalen Variablen auf den Stack kopiert.

Dabei ist die Reihenfolge massgebend. Der erste aktuelle Parameter wird in den ersten Paremeter der Parameterliste kopiert usw. Die Namen der Parameter (a und b) spielen beim Aufruf keine Rolle.

- Innerhalb der Methode werden die Parameter immer mit den in der Parameterliste definierten Namen angesprochen. Sie können wie die übrigen lokalen Variablen gelesen und verändert werden.

  Wie jede lokale Variable geht jedoch der Wert am Ende der Methode verloren.

- Man beachte, dass Parameter in der Parameterliste mit Kommas unterteilt werden (wie aktuelle Parameter). Ferner muss für *jeden* Parameter der Datentyp angegeben werden (auch bei mehreren Parametern mit dem gleichen Datentyp):

  (Datentyp Parameter, Datentyp Parameter, ...)

- Methoden ohne Parameter haben eine leere Parameterliste, bestehend aus leeren runden Klammern. Beim Aufruf einer Methode ohne Parameter müssen leere Klammern geschrieben werden, sonst erkennt der Compiler nicht, dass es ein Methoden-Aufruf ist:

  ```
  x = Math.random();                    // Funktion ohne Parameter
  ```

△ **Uebung:** Lösen Sie die Aufgabe 1.8.4

Prozeduren

void

Bei der Definition einer Prozedur, wird als Resultat-Typ das Schlüsselwort 'void' (leerer Datentyp) angegeben, da eine Prozedur kein Resultat hat.

Ein 'return'-Statement ist in einer Prozedur nicht nötig. Die Prozedur endet, wenn der Programmablauf bei der Schlussklammer angelangt ist.

Beispiel:

Mit der folgenden Methode kann ein Character wiederholt ausgegeben werden, z.B. das Unterstreichungszeichen, zur Erzeugung einer Trennlinie:

```
static void linie(char ch, int len)
{ int i=0;
  while ( i < len )
  { System.out.print(ch);
    i++;
  }
  System.out.println("");                    // Zeile abschliessen
}
```

Aufruf der Methode:

```
linie('*', 60);                              // Linie aus 60 '*' ausgeben
```

1.7 Eigene Input/Output-Methoden

InOut

Die Standard-Methoden der Java Library für Input/Output sind leider ungenügend. Wir verwenden daher einige selber geschriebene Methoden, welche im File 'InOut.java' definiert sind (Klasse 'InOut'). Wir sehen bei dieser Gelegenheit, wie man Mängel der Java-Library leicht selber beheben kann.

Das File 'InOut.java' kann mittels anonymous FTP von der untenstehenden Adresse[1] kopiert werden. Es muss in das Directory der eigenen Java-Programme kopiert werden. Dann können die Methoden wie Standard-Methoden von Java verwendet werden.

[1] ftp://loki.cs.fh-aargau.ch/pub/java/InOut.java

Bei der ersten Compilation eines Programmes, welches eine Methode von 'InOut' aufruft, wird das File 'InOut.java' automatisch compiliert.

Input

Java stellt standardmässig nur eine Methode zum Einlesen eines einzelnen Zeichens zur Verfügung:

```
int code;
code = System.in.read();
```

Es ist eine Funktion ohne Parameter, die als Resultat den numerischen Code des eingelesenen Zeichens liefert. Da wir meistens Zahlen und nicht einzelne Zeichen einlesen wollen, dient uns diese Funktion wenig.

Eigene Einlese-Methoden für numerische Werte:

```
int n;
double x;

n = InOut.getInt();                    // ganze Zahl einlesen
x = InOut.getDouble();                 // reelle Zahl
```

Bei der Ausführung einer dieser Funktionen wartet das Programm, bis der Benutzer eine Eingabe gemacht hat, abgeschlossen mit der Return-Taste.

Anschliessend wird der Eingabewert in die betreffende Variable übertragen.

Formatierte Bildschirmausgaben

Leider besitzen die Standard-Methoden 'System.out.print' und 'println' von Java keine Parameter zur Festlegung des Ausgabeformates (z.B. Anzahl Dezimalstellen). Bei 'double'-Werten werden i.a. alle 15 Dezimalstellen ausgegeben, was nicht immer erwünscht ist.

Wir verwenden daher für diese Zwecke wieder selber geschriebene Methoden, die ebenfalls im File 'InOut.java' enthalten sind:

1. Ganzzahlige Werte

   Für Ausgaben von ganzzahligen Werten in tabellarischer Form stehen Methoden 'InOut.print' und 'InOut.println' zur Verfügung, welche einen zweiten Parameter haben, mit dem die Anzahl Stellen auf dem Bildschirm spezifiziert wird:

```
int n=10;
InOut.print(n, 8);     // rechtsbuendig mit 8 Stellen
```

Ausgabe: ____10 (‘_’ = Blank)

Wenn die angegebene Anzahl Stellen zu klein ist, wird sie automatisch erhöht.

2. Werte mit Dezimalstellen

   Für formatierte Ausgaben von reellen Zahlen stehen Methoden ‘print’ und ‘println’ zur Verfügung, welche zwei zusätzliche Parameter haben:

   – Anzahl Stellen vor dem Dezimalpunkt
   – Anzahl Dezimalstellen

```
double x = Math.PI;
InOut.print(x, 8, 4);     // 8 Stellen vor Dezimalpunkt, 4 Dez.stellen
```

Ausgabe: _____3.1416

Wenn die angegebene Anzahl Stellen vor dem Dezimalpunkt zu klein ist, wird sie automatisch erhöht, man kann also im Normalfall 1 angeben. Die spezifizierte Anzahl Dezimalstellen ist fix, die letzte ausgegebene Stelle ist gerundet.

Input/Output-Beispiel:

Das folgende Programm berechnet die Fallgeschwindigkeit beim freien Fall (ohne Luftwiderstand) von einer Anfangshöhe h, nach der Formel

$$v = \sqrt{2gh}$$

```
public class FreeFall
{
  static public void main(String[ ] args)
  { double h, v;
    final double g = 9.81;                    // Erdbeschleunigung
    InOut.print("Hoehe [m] : ");
    h = InOut.getDouble();                    // Hoehe einlesen
    v = Math.sqrt(2 * g * h);
    v = 3.6 * v;                              // Umrechnung in km/h
    InOut.print("Fallgeschwind. : ");
    InOut.print( v, 1, 2 );                   // 2 Dezimalstellen
    InOut.println( " km/h" );
  }
}
```

△ **Uebung:** Lösen Sie die Ausfgaben 1.8.5 und 1.8.6

1.8 Übungen

1.8.1 Zweierpotenzen

Erstellen Sie eine Applikation 'Potenzen', welche die Potenzen von 2 für Exponenten von 1 bis 20 ausgibt:

Exponent	Zweierpotenz
1	2
2	4
3	8
4	16
...	...

Hinweis:

Führen Sie eine 'int'-Variable 'potenz' ein, die bei jedem Schritt verdoppelt wird. Kontrollwerte:

$1\ K = 2^{10} = 1024$	1 Kilo
$1\ M = 2^{20} = 1\ K \cdot 1\ K = 1\,048\,576$	1 Mega
$1\ G = 2^{30} = 1\ K \cdot 1\ M$	1 Giga

Ein K ist also näherungsweise gleich 1000, $1\,M$ näherungsweise gleich einer Million und $1\,G$ ungefähr 1000 Mega.

1.8.2 Zufallszahlen

Erstellen Sie eine Applikation, welche 100 Zufallszahlen erzeugt und dabei die kleinste und die grösste bestimmt.

Führen Sie dazu zwei Variablen *min* und *max* ein, die während der Schleife laufend die bisher kleinsten bzw. grössten Werte enthalten.

1.8.3 Rekorde

Erstellen Sie eine Applikation 'Rekorde', welche $n = 100$ Zufallszahlen im Intervall $[0, 1]$ erzeugt und dabei zählt, wieviele Rekorde auftreten.

Eine Zahl x der Folge heisst *Rekord*, wenn sie grösser als alle vorangehenden ist.

Bemerkung:

Man kann mit den Methoden der Wahrscheinlichkeitsrechnung zeigen, dass die erwartete Anzahl Rekorde um den Wert 5 schwankt.

Die Situation kann veranschaulicht werden, wenn man die Zufallszahlen als Jahres-Temperaturen (Jahres-Mittel) interpretiert: In einem Jahrhundert sind also etwa 5 Rekordjahre zu erwarten, wenn keine globale Erwärmung vorhanden ist.

Lösungshinweis:

Führen Sie (neben weiteren Variablen) eine 'double'-Variable 'max' ein, welche immer das Maximum der bisherigen Zufallszahlen enthält.

In einer Schleife werden die Zufallszahlen erzeugt. Dabei wird für jede erzeugte Zufallszahl getestet, ob sie grösser als 'max' ist. Ist dies der Fall, so liegt ein neuer Rekord vor.

1.8.4 Bremsweg

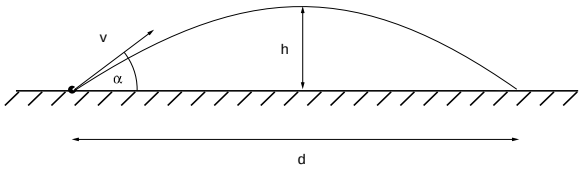
Der Bremsweg eines Autos (Richtwert auf trockener Strasse) kann mit der folgenden Formel berechnet werden:

$$s = 0.01 * v^2$$

Dabei ist v die Geschwindigkeit des Autos beim Einleiten des Bremsmanövers in km/h, und s ist der Bremsweg in m.

Erstellen Sie eine Funktion 'bremsweg' zur Berechnung des Bremsweges und verwenden Sie die Funktion zur Ausgabe einer Tabelle der Bremswege für die Geschwindigkeiten 50, 60, ... bis 150 km/h.

1.8.5 Wurfparabel



Erstellen Sie eine Applikation 'Wurf', die für einen schiefen Wurf auf der Erdoberfläche die Wurfweite d und die maximale Höhe h berechnet.

Input-Daten:	v	Abschuss-Geschwindigkeit in m/s.
	α	Abschuss-Winkel in Grad
Output:	d	Wurfweite in m
	h	Wurfhöhe

Berechnungsformeln:

$$d = \frac{v^2 \cdot \sin(2\alpha)}{g} \qquad h = \frac{v^2 \cdot \sin^2(\alpha)}{2 \cdot g}$$

Dabei ist $g = 9.81\, m/s^2$ die Erdbeschleunigung. Der Luftwiderstand ist in den Formeln nicht berücksichtigt. Für $v = 20.0\, m/s$ und $\alpha = 30^o$ müssen $d = 35.31\, m$ und $h = 5.10\, m$ herauskommen.

Führen Sie geeignete Funktionen mit Parametern ein.

1.8.6 GGT

Erstellen Sie eine Applikation 'GGT', die den grössten gemeinsamen Teiler von zwei positiven ganzen Zahlen a und b berechnet.

Algorithmus:

1. Wenn a gleich b ist, sind wir fertig, das Resultat ist gleich a.
2. Wenn a > b ist, ersetze a durch a-b, sonst b durch b-a.
3. Gehe zu Schritt 1.

Beispiel:

a	b
24	16
8	16
8	8

Bemerkung:

Die Begründung des Algorithmus ist einfach. Er beruht darauf, dass zwei Zahlen a und b den gleichen GGT haben wie a-b und b.

Dies folgt sofort daraus, dass zwei Zahlen a und b dieselben *gemeinsamen* Teiler wie a-b und b haben, wie man leicht einsieht.

Verpacken Sie den Algorithmus in eine Funktion ‘ggt’, die in der Methode ‘main’ aufgerufen wird.

Kapitel 2

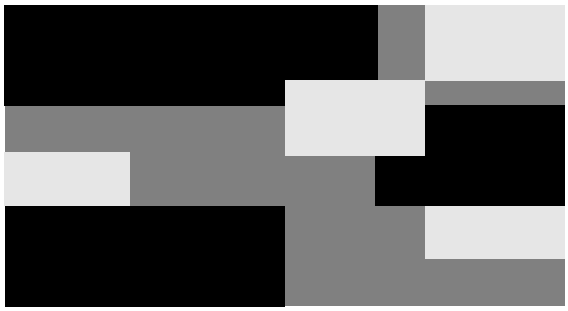
Applets

2.1 Einführung von Applets

Applets sind Java Programme, die in einem HTML-Dokument aufgerufen werden. Wenn das HTML-Dokument mit einem Internet-Browser geöffnet wird, wird das Applet geladen und läuft in einem eigenen Fenster innerhalb des HTML-Dokumentes ab.

Dies funktioniert gleichermassen auch via Internet. Dabei wird das Applet wie das zugehörige HTML-File auf den lokalen Computer hinuntergeladen und läuft in der lokalen Java Virtual Machine des Browsers.

Wir betrachten ein einfaches Beispiel eines Applets, welches das folgende Rechteckmuster zeichnet[1]:



[1] Nach P. Schweri, 'Artwork Code 86'

Applets haben automatisch ausgedehnte Graphik-Möglichkeiten zur Verfügung (in Applikationen sind für Graphik-Operationen einige Vorbereitungen erforderlich).

Wir verwenden daher im weiteren Verlauf des Kurses Applets für Anwendungen mit *Graphik*. Für reine Text-Programme sind hingegen Applikationen besser geeignet.

Listing des Applets:

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

public class Rechtecke extends Applet
{
  public void init()                          // Methode 'init'
  { setBackground(Color.blue);
  }

  public void paint(Graphics g)               // Methode 'paint'
  { g.setColor(Color.gray);
    g.fillRect(100, 100, 420, 240);
    g.setColor(Color.black);
    g.fillRect(100, 100, 280, 80);
    g.fillRect(100, 260, 210, 80);
    g.fillRect(378, 180, 142, 80);
    g.setColor(Color.red);
    g.fillRect(100, 217, 94, 43);
    g.fillRect(415, 100, 105, 60);
    g.fillRect(310, 160, 105, 60);
    g.fillRect(415, 260, 105, 42);
  }
}
```

Die Bestandteile des Applets werden in den folgenden Abschnitten erläutert.

Übersetzung des Applets

Applets werden wie Applikationen mit dem Java Compiler übersetzt:

```
javac Rechtecke.java
```

Dabei entsteht das File 'Rechtecke.class', welches wie bei Applikationen den Byte-Code des Programmes enthält.

Ausführung des Applets

Da Applets in einem HTML-Dokument ablaufen, muss für die Ausführung ein minimales HTML-Dokument erstellt werden:

```
<html>
<head>
<title> Java Applet </title>
</head>
<body>
<applet code="Rechtecke.class" width=640 height=480>
</applet>
</body>
</html>
```

Bemerkungen:

1. Das 'applet'-Statement

   Im 'applet'-Statement werden der Name des Applets und die Grösse des Windows, in dem es abläuft, angegeben.

2. Das HTML-Dokument wird mit einem beliebigen Namen abgespeichert, z.B. 'Rechtecke.html' und mit einem Browser (Netscape ab Version 3.01, Internet Explorer ab Version 3.0) geöffnet. Dabei wird der Byte-Code, d.h. das '.class'-File des Applets geladen und ausgeführt. (Quellencode wird dabei nicht benötigt).

   Über Internet wird der Byte-Code auf den lokalen Computer heruntergeladen und auf diesem ausgeführt.

3. Der Appletviewer

   Appletviewer

   Zum Testen von Applets kann auch der *Appletviewer* von Sun anstelle eines Browsers verwendet werden:

   ```
   appletviewer Rechtecke.html
   ```

   Der Appletviewer verarbeitet jedoch nur die 'applet'-Anweisungen, ev. vorhandenen weiteren Text des HTML-Dokumentes zeigt er nicht an.

   Für unsere Übungsbeispiele ist es sinnvoll, ein festes HTML-Dokument zu verwenden und für Tests jeweils den Applet-Namen zu modifizieren.

2.2 Aufbau eines Applets

Ein Applet ist in der Grundform eine Klasse mit zwei Methoden, *init* und *paint*:

```
import java.awt.*;
import java.applet.*;
public class Rechtecke extends Applet
{
  public void init()                    // Methode 'init'
  { ... }
  public void paint(Graphics g)         // Methode 'paint'
  { ... }
}
```

Applets sind *ereignisorientierte Programme*. Darunter versteht man Programme, die aus Methoden bestehen, welche bei bestimmten Ereignissen aufgerufen werden:

- Gleich nach dem Laden des Applets ruft der Browser die Methode *init* aufgerufen. Sie kann die nötigen Initialisierungen vornehmen.
- Anschliessend, wenn das Bild des Applets gezeichnet werden soll, wird die Methode *paint* aufgerufen.
- Später werden wir weitere Methoden einführen, die bei Ereignissen wie Maus-Klicks, Tasten-Drucken usw. aufgerufen werden.

Die aufgerufene Methode verarbeitet das Ereignis und gibt dann die Kontrolle wieder zurück an das aufrufende Programm (Browser).

Das Konzept der ereignisorientierten Programme eignet sich sehr gut für interaktive Programme, die vom Benutzer mittels Aktionen gesteuert werden.

Eine *main*-Methode ist in einem Applet *nicht* vorhanden.

Das Basis-Applet 'Applet'

Jedes Applet ist eine Erweiterung eines Basis-Applets 'Applet'. Diese Eigenschaft wird im 'class'-Statement mittels 'extends' spezifiziert:

```
public class Rechtecke extends Applet
```

Das Basis-Applet ist ein minimales Applet, welches ein leeres Window ausgibt. Es enthält dazu alle benötigten Methoden, u.a. Standard-Versionen von 'init' und 'paint'.

Durch die 'extends'-Angabe *erbt* das Applet 'Rechtecke' alle Methoden des Basis-Applets. Wenn eine vererbte Methode im Applet 'Rechtecke' mit gleichem Namen und gleichen Parametern neu definiert wird,

ersetzt sie diejenige des Basis-Applets. Man sagt sie *überschreibe* die des Basis-Applets.

Dies ist das Konzept der *Klassen-Erweiterungen*, mit dem wir uns später ausführlicher beschäftigen werden.

Die 'import'-Statements

Java Applets benötigen zwei 'import'-Statements, die vor dem 'class'-Statement stehen müssen:

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

Mit 'import'-Statements werden *Packages* der Java Library angegeben, die das Programm benötigt. Details zu Packages siehe unten.

Jedes Applet benötigt das Package 'java.applet' und das *Abstract Window Toolkit* 'java.awt' (für Graphik und Benutzeroberflächen).

2.3 Die Methoden 'init' und 'paint'

In einem Applet werden alle Methoden (auch eigene Hilfsmethoden) *ohne* das Attribut 'static' definiert. Der Grund wird später ersichtlich.

Die Methode 'init'

Die Methode 'init' wird als erste aufgerufen, vor der Eröffnung des Windows. Sie kann allgemeine Programm-Initialisierungen durchführen, z.B. die Festsetzung der Hintergrund-Farbe des Windows:

```
public void init()
{ setBackground(Color.blue);
}
```

Die Methode 'setBackground' wird unten, bei den Graphik-Methoden, eingeführt. Bildschirmausgaben sind in der 'init'-Methode noch nicht möglich.

Die Methode 'paint'

Die Methode 'paint' hat die Aufgabe, das Bild des Applets zu zeichnen.

```
public void paint(Graphics g)
{ g.setColor(Color.gray);
  g.fillRect(100, 100, 420, 240);
  g.setColor(Color.black);
  g.fillRect(100, 100, 280, 80);
  g.fillRect(100, 260, 210, 80);
  g.fillRect(378, 180, 142, 80);
  g.setColor(Color.red);
  g.fillRect(100, 217, 94, 43);
  g.fillRect(415, 100, 105, 60);
  g.fillRect(310, 160, 105, 60);
  g.fillRect(415, 260, 105, 42);
}
```

▶ *Merke:*

Die Methode 'paint' muss jederzeit in der Lage sein, das Bild zu reproduzieren. Sie wird zum ersten Mal nach 'init' aufgerufen. Später wird sie eventuell erneut aufgerufen, wenn das Bild neu gezeichnet werden muss, weil es durch Benutzer-Aktionen vorübergehend überdeckt oder minimiert worden ist.

Das Graphics-Objekt g

Die Methode 'paint' erhält einen Parameter 'g'. Dies ist keine normale Variable, sondern ein *Objekt*. Wir lernen hier das erste Java Objekt kennen.

Mit Objekten werden wir uns später intensiv beschäftigen. Im Moment benötigen wir nur die folgenden Grundeigenschaften von Objekten:

Was ist ein Objekt?

> Ein *Objekt* ist eine Zusammenfassung von Daten (Variablen) und Methoden zu einer Einheit, die mit einem Namen angesprochen werden kann.
>
> Die *Daten* beschreiben den Zustand des Objektes, und die *Methoden* führen Aktionen aus, die das Objekt betreffen und eventuell seinen Zustand verändern.

Dieses Modell entspricht weitgehend unserer Vorstellung eines realen Objektes, z.B. einer Stop-Uhr: Eine Stop-Uhr hat einen Zustand (gestartet/gestoppt, momentane Laufzeit) und Knöpfe, mit denen bestimmte Aktionen ausgelöst werden können wie Starten, Stoppen. Die Knöpfe entsprechen den Methoden eines Software-Objektes.

Das Objekt 'g' repräsentiert das Bildschirm-Fenster des Applets. Die *Daten* von 'g' enthalten momentane Einstellungen wie

- aktuelle Zeichenfarbe
- aktuelle Schrift-Attribute.

Die *Methoden* von 'g' dienen den folgenden Zwecken:

- Veränderung der aktuellen Einstellungen
- Ausgaben von Text und Graphik

Die Methoden eines Objektes werden mit dem Namen des Objektes und dem Punkt-Operator aufgerufen:

```
g.setColor(Color.black);                    // Zeichenfarbe setzen
g.fillRect(100, 100, 280, 80);              // Rechteck zeichnen
```

Die Details der Graphik-Methoden werden im nächsten Paragraphen eingeführt.

Bemerkungen:

1. In Java gehört jedes Objekt zu einer Klasse. Das Objekt 'g' gehört zur Klasse 'Graphics', die zum Package 'java.awt' gehört.

2. Da die Methode 'paint' das Graphics-Objekt als Parameter erhält, ist (wie bei allen Parametern) der Name 'g' willkürlich, er kann bei Bedarf abgeändert werden.

3. Das Graphics-Objekt kann als Parameter an weitere Hilfsmethoden übergeben werden, welche Ausgaben in das Bildschirm-Window machen. Eine solche Methode wird folgendermassen definiert:

   ```
   void xxxx(Graphics g, ...)
   { ... }
   ```

2.4 Grundlegende Graphik-Methoden

- Zeichenfarbe festlegen

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

Mit dieser Methode wird die Farbe für nachfolgende Graphik-Ausgaben festgesetzt.

Standard-Farben:

Color.black	Color.green	Color.red
Color.blue	Color.lightGray	Color.white
Color.cyan	Color.magenta	Color.yellow
Color.darkGray	Color.orange	
Color.gray	Color.pink	

- Hintergrund- und Default-Vordergrundfarbe festlegen:

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

Die Methode 'setBackground' legt die Hintergrundfarbe des Windows fest. Dies ist die Farbe, die bei Lösch-Operationen verwendet wird.

Die Methode 'setForeground' bestimmt die Default-Vordergrundfarbe des Windows. Diese Farbe wird bei Zeichenoperationen verwendet, bis mit der Methode 'g.setColor' eine andere gewählt wird.

Beide Methoden gehören nicht zum Objekt 'g', sondern zum Basis-Applet 'Applet' und werden daher *ohne* 'g.' aufgerufen.

Die Methoden 'setBackground' und 'setForeground' sollten nur in der 'init'-Methode (nicht in 'paint') aufgerufen werden.

- Rechtecke:

```
g.fillRect(left, top, width, height);
g.drawRect(left, top, width, height);
g.clearRect(left, top, width, height);
```

Die erste Methode zeichnet ein gefülltes Rechteck, die zweite eines aus Randlinien. Dabei wird die mit 'setColor' vorgewählte Farbe verwendet. Die Methode 'clearRect' zeichnet wie die erste ein gefülltes Rechteck, jedoch in der Hintergrundfarbe des Windows (für Löschoperationen).

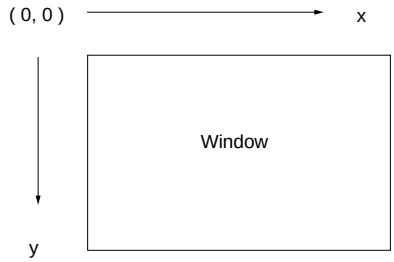
Die Parameter 'width' und 'height' legen die Breite und die Höhe des gezeichneten Rechtecks fest. Es sind ganzzahlige Werte in Pixel-Einheiten (Pixel = Picture Element, d.h. Bildpunkt).

Die Position des Rechteckes wird festgelegt durch die Koordinaten (left, top) der linken oberen Ecke.

Bildschirm-Koordinaten:

Koordinaten (x, y) von Punkten auf dem Bildschirm sind ganzzahlige Werte, bestehend aus Spalten- und Zeilen-Nummer in Bezug auf das Window des Applets.

Der Punkt (0, 0) liegt links oben im Window:



- Ellipsen und Kreise

```
g.fillOval(left, top, width, height);
g.drawOval(left, top, width, height);
```

Die Methoden zeichnen eine Ellipse, mit Umgebungsrechteck gemäss den angegebenen Parametern. Die Ellipse wird bei 'fillOval' ausgemalt, bei 'drawOval' wird nur die Randlinie gezeichnet.

Im Fall `width = height` wird ein Kreis gezeichnet.

- Punkte

Punkte werden als Rechtecke mit Breite und Höhe gleich 1 gezeichnet:

```
g.fillRect(x, y, 1, 1);
```

- Strecken

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

Strecke von (x1, y1) nach (x2, y2).

- Text-Ausgabe

```
g.drawString("Hello", x, y);
```

Dabei ist (x,y) der linke Fusspunkt für den ersten Buchstaben.

- Abfrage der Window-Grösse

Die Abmessungen (Breite und Höhe) des Windows eines Applets können mit der Methode 'getSize' wie folgt bestimmt werden:

```
Dimension dim = getSize();
int breite = dim.width;
int hoehe = dim.height;
```

Hier lernen wir ein weiteres Java Objekt kennen: Die Methode 'getSize' liefert als Resultat ein Objekt der Klasse 'Dimension'. Dies ist ein einfaches Objekt, welches nur zwei Variablen 'width' und 'height' enthält, welche mit dem Objekt-Namen und dem Punkt-Operator angesprochen werden können:

```
dim.width     (ohne Klammern, da es kein Methodenaufruf ist)
dim.height
```

Der gewählte Name 'dim' für das Objekt ist beliebig.

Man beachte, dass die Methode 'getSize' *ohne* das Objekt 'g' aufgerufen wird, da sie nicht zu diesem Objekt gehört, sondern zum Basis-Applet (wie setBackground).

2.5 Die Java Library

Die Java Library ist zweistufig organisiert:

- Klassen

  Zusammengehörige Methoden sind zu *Klassen* zusammengefasst, z.B. sind die mathematischen Funktionen Bestandteile der Klasse 'Math'.

- Packages

  Zusammengehörige Klassen sind zu *Packages* zusammengefasst. Ein Package ist also eine Zusammenfassung von zusammengehörigen Klassen.

Wichtige Packages:

Package	Inhalt
java.lang	allgemeine Klassen der Sprache, z.B. 'System', 'Math' (mathematische Funktionen), usw.
java.applet	Klassen für Applets, z.B. das Basis-Applet 'Applet'
java.awt	Abstract Window Toolkit mit Klassen für Graphik und Benutzeroberflächen
java.io	File Input/Output

'import'-Statements

Wenn ein Java Programm Klassen eines Packages benötigt, muss dieses in einem 'import'-Statement angegeben werden, damit der Compiler die betreffenden Klassen findet:

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

Ausnahme: Das Package 'java.lang' steht automatisch jedem Programm zur Verfügung. Das '*'-Zeichen bedeutet, dass dem Programm alle Klassen des Packages zur Verfügung stehen. (Anstelle von '*' könnte eine einzelne Klasse angegeben werden).

Dokumentation der Java Library

Für die Java Library steht eine umfassende Dokumentation im HTML-Format zur Verfügung, welche mit einem Browser konsultiert werden kann:

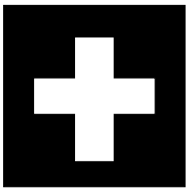
1. Dokument 'jdk\docs\api\packages.html' öffnen.

   Dabei ist 'jdk' das oberste Verzeichnis des Java Development Kits, welches je nach Installation ev. einen anderen Namen hat. Definieren Sie in Ihrem Browser eine Bookmark für das obige HTML-File.
2. Gewünschtes Package anklicken, z.B. 'java.awt'.
3. Gewünschte Klasse (z.B. 'Graphics') anklicken und zu den Methoden vorwärts blättern.

2.6 Übungen

2.6.1 Schweizer-Flagge

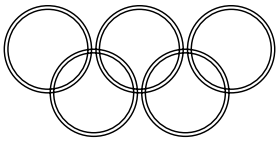
Erstellen Sie ein Applet 'Swiss', welches die Schweizer-Flagge zeichnet.



Balken : width = 100, height = 30.

2.6.2 Olympiade-Signet

Erstellen Sie ein Applet, welches das Signet der Olympiade, bestehend aus 5 Kreisringen, zeichnet.



Kapitel 3

Elementare Datentypen

Es gibt keinen Weg zum Z, der nicht am A vorbeiführte.

— F. Hebbel

Ein *Datentyp* ist eine Menge von Werten zusammen mit einem Satz von Operatoren (z.B. algebraische Operatoren für numerische Datentypen).

Übersicht über elementare Datentypen

- Numerische Typen
  - Ganzzahlige Typen (Integer Types)
  - Reelle Typen (Floating-Point Types)
- Der Datentyp 'boolean'

Die reellen Typen sind für reelle Zahlen, d.h. Werte mit Dezimalstellen. Da im Computer nur eine endliche Anzahl Dezimalstellen gespeichert werden kann, treten bei algebraischen Operationen mit reellen Typen immer Rundungsfehler auf.

Der Datentyp 'boolean' ist für logische Variablen, die nur die Werte *false* und *true* annehmen können.

Neben diesen elementaren Datentypen gibt es in Java die sogenannten *Referenztypen* für Variablen, die Objekte referenzieren. In diesem Kapitel beschäftigen wir uns nur mit den elementaren Datentypen.

3.1 Ganzzahlige Typen (Integer Types)

Die verschiedenen ganzzahligen Typen unterscheiden sich in der Anzahl Bits, mit denen die einzelnen Werte abgespeichert werden:

Datentyp	Anzahl Bits	Wertebereich
byte	8	-128 .. 127
short	16	-32768 .. 32767
int	32	-2^{31} .. $2^{31}-1$
long	64	-2^{63} .. $2^{63}-1$
char	16	0 .. 65535

Die Anzahl Bits bestimmt die Anzahl Werte des Typs. Mit 8 Bits ergeben sich 256 Werte, da jedes Bit zwei mögliche Werte annehmen kann:

$$2 \cdot 2 \cdot \ldots \cdot 2 = 2^8 = 256$$

Bei der Aufteilung in negative und positive Werte resultieren wegen der 0 asymmetrische Wertebereiche, z.B. -128 .. 127.

Integer-Operatoren

m + n	Addition
m − n	Subtraktion
−m	Multiplikation mit -1
m * n	Multiplikation
m / n	ganzzahlige Division
m % n	Rest bei Division durch n (modulo Operator)
m++	Erhöhung um 1 (Inkrement)
m−−	Verminderung um 1 (Dekrement)

Beispiele:

```
int m = 14;
int n = 3;
int q, r;
q = m / n;                    // ganzzahlige Division, Resultat: 4
r = m % n;                    // Rest modulo n: 2
```

Integer Overflow

Wenn bei einer algebraischen Operation der Wertebereich des betreffenden Datentyps überschritten wird, erfolgt *keine* Fehlermeldung, sondern das Programm läuft mit falschen Werten weiter! Diese Gefahr ist bei Verwendung des Datentyps 'short' aktuell.

Integer-Literals

Literals sind Symbole für konkrete Werte. Ganzzahlige Werte können dezimal, hexadezimal oder oktal angegeben werden:

Literal	Interpretation
234	Dezimalwert
0X3A4F	Hexadezimalwert (Kennzeichen: Präfix '0X')
0345	Oktalwert (Kennzeichen: erste Ziffer '0')

Diese Literals erhalten den Datentyp 'int'. Bei Bedarf kann der Datentyp auf 'long' geändert werden mit einem Suffix 'L' (oder 'ℓ') :

234L, 0X3AFFL

Der Integer-Typ 'char'

Der Datentyp 'char' ist vorgesehen für die Abspeicherung von Characters (Buchstaben, Ziffern und Sonderzeichen). Dazu wird jedem Character eine ganze Zahl zugeordnet wird, der sog. *Unicode* des Characters. *Unicode*
Beispiele von Unicodes:

Character :	' '	..	'0'	'1'	..	'A'	'B'	..	'a'	'b'	..
Unicode:	32	..	48	49	..	65	66	..	97	98	..

Die Unicode-Zuordnung ist eine Erweiterung der klassischen ASCII-Zuordnung (ASCII-Codes). Sie wurde eingeführt zur Berücksichtigung der Zeichen verschiedener Sprachen (Japanisch, Chinesisch, usw.).

Character-Literals

Character-Literals für Buchstaben, Ziffern und Sonderzeichen werden in einfache Hochkommas eingeschlossen:

```
'a'  'b'  'A'  'B'  ' '
```

Dies sind einfach Symbole für die zugehörigen Unicodes.

Escape-Sequenzen

Escape-Sequenzen sind Literals für spezielle Character-Werte:

`'\n'`	Linefeed
`'\r'`	Carriage return
`'\f'`	Form feed
`'\t'`	Tab
`'\' '`	'
`'\"'`	"
`'\\'`	\
`'\uxxxx'`	Character mit Unicode xxxx. Dabei ist xxxx eine 4-stellige Hexadezimal-Zahl. Beispiel: \u001F

△ **Uebung:** Lösen Sie die Aufgabe 3.7.1

3.2 Reelle Datentypen (Float Types)

Datentyp	Anzahl Bits	Anzahl signifikante Dezimalstellen
float	32	6 - 7
double	64	15

Die interne Darstellung der Werte erfolgt in der *Floating-Point* Darstellung (Beispiel $0.523842 \cdot 10^4$, aber mit Basis 16 statt 10). Man nennt diese Typen 'Floating-Point Types' oder einfach 'Float-Types'.

Float Operatoren

`x + y`	Addition
`x − y`	Subtraktion
`−x`	Multiplikation mit -1.0
`x * y`	Multiplikation
`x / y`	Division
`x++`	Erhöhung um 1 (Inkrement)
`x−−`	Verminderung um 1 (Dekrement)

Ein Potenzoperator steht nicht zur Verfügung. Ersatz ist die mathematische Funktion 'Math.pow'. Die mathematischen Funktionen wurden auf Seite 18 eingeführt.

Literals für Float-Types

Literal	Interpretation
2.371	Dezimalwert
7.63E2	$7.63 \cdot 10^2$
7.63E-2	$7.63 \cdot 10^{-2}$

Diese Literals erhalten den Datentyp 'double'. Bei Bedarf kann der Datentyp auf 'float' geändert werden mit einem Suffix 'F' (oder 'f') :

2.371f, 7.63E-2f

3.3 Der Datentyp 'boolean'

Der Datentyp 'boolean' ist für logische Variablen, die nur zwei mögliche Werte, *false* und *true*, annehmen können.

Datentyp	Anzahl Bits	Werte
boolean	8	false, true

Der Datentyp 'boolean' ist ein nicht numerischer Typ, die Werte müssen mit den Schlüsselwörtern *false* und *true* (nicht mit 0 und 1) angesprochen werden:

```
boolean fehler = false;
```

Zum Datentyp 'boolean' gehören die logischen Operatoren die bei Kontroll-Anweisungen benötigt werden (s. unten).

3.4 Auswertung von Ausdrücken

Ein *Ausdruck (Expression)* ist eine beliebige Kombination von Variablen, Konstanten, Literals, Operatoren ('+', '-', usw.) und Funktions-Aufrufen:

```
3.2 * Math.sin(x) - x / y                    // algebraischer Ausdruck
```

Auswertungsreihenfolge

Die Auswertung von Ausdrücken erfolgt nach den üblichen Regeln der Mathematik:

- Operationen zweiter Stufe (Multiplikation und Division) kommen vor den Operationen erster Stufe (Addition und Subtraktion).
- Operationen gleicher Stufe werden von links nach rechts ausgeführt.
- Mit runden Klammern kann die Auswertungsreihenfolge beeinflusst werden.

Beispiele:

Java Ausdruck	mathematische Bedeutung
`(a + b) / 2.4`	$\frac{a+b}{2.4}$
`a + b / 2.4`	$a + \frac{b}{2.4}$
`d / (2.4 * a)`	$\frac{d}{2.4 \cdot a}$
`d / 2.4 * a`	$\frac{d}{2.4} \cdot a$

Gemischte Ausdrücke

In Ausdrücken und Zuweisungen dürfen verschiedene numerische Datentypen (z.B. 'int' und 'double') gemischt auftreten, Java führt erforderliche Typen-Konversionen automatisch durch. Dabei gelten die folgenden Regeln:

- Wenn bei einer Operation '+', '−', '∗' oder '/' *beide* Operanden ganzzahlig sind, wird die Operation ganzzahlig, also *ohne Dezimalstellen* durchgeführt, auch dann, wenn das Resultat einer 'double'-Variablen zugewiesen wird. Dies kann bei Divisionen Probleme ergeben:

```
int m = 14, n = 5;
double x=2.3, y;
y = m / n;                    // Integer−Division, Resultat 2
```

  Wenn mindestens einer der beiden Operanden ein Float-Typ ist, wird der andere, wenn nötig, umgewandelt, und die Operation wird *mit* Dezimalstellen ausgeführt.

```
y = m / 4.0;                  // Double−Division, Resultat 3.5
y = 3.4 * n − 5 * x;          // Double−Operationen
y = 2.0 + m / 4;              // ganzzahlige Division, dann
                              // Double−Addition
y = Math.pow(x, 1.0 / 3);     // 3. Wurzel von x
```

Im zweitletzten Beispiel wird die ganzzahlige Division gewählt, da der Quotient *vor* der Double-Addition ausgewertet wird.

Das letzte Beispiel enthält eine gefährliche Fehlerquelle: Schreibt man den zweiten Parameter als 1/3 (statt 1.0/3), so wird er 0, weil er dann ohne Dezimalstellen berechnet wird.

Zur Vermeidung von ungewollten Effekten bei Divisionen stehen die expliziten Konversions-Operatoren zur Verfügung (siehe unten).

- Einschränkung für Konversionen bei Zuweisungen

  Java führt bei Zuweisungen erforderliche Typen-Konversionen automatisch durch.

  Bedingung:

  Eine automatische Konversion von einem Datentyp in einen anderen erfolgt nur dann, wenn dabei keine Informationen verloren gehen können, d.h. wenn der Zieltyp alle Werte des Ausgangstyp umfasst: z.B. von 'int' in 'double', aber nicht umgekehrt.

```
int n = 4;
double y;
y = n;                          // automatische Konversion
y = Math.sqrt(2);               // die 2 wird automatisch in double
                                // konvertiert
```

  Automatische Typen-Konversionen erfolgen auch bei Parameter-Übergaben und bei 'return'-Statements in einer Funktion.

Explizite Typen-Konversionen (Type-Casting)

Zur Vermeidung der Probleme bei Divisionen, sowie für Umwandlungen, bei denen die Bedingung für die automatische Konversion nicht erfüllt sind, stehen explizite *Typen-Konversionen* zur Verfügung:

Dabei wird der neue Typ in runden Klammern vor den umzuwandelnden Wert geschrieben:

(Zieltyp) Wert

Explizite Typen-Konversionen sind zwischen allen numerischen Daten-Typen möglich.

Beispiele:

1. Integer-Konversion

   Bei Integer-Konversionen werden die *Dezimalstellen abgeschnitten*, ohne Runden:

```
int n;
double x = 3.8;
n = (int) x;                    // Integer−Konversion, n wird 3
x = −3.8;
n = (int) x;                    // n erhaelt den Wert −3
```

   Für Umwandlungen in ganze Zahlen mit Runden steht die mathematische Funktion 'Math.round(x)' zur Verfügung. Das Resultat der Funktion hat den Datentyp 'long' und muss gegebenenfalls konvertiert werden:

```
int m = (int) Math.round(x);
```

2. Explizite Float-Konversion bei Divisionen

```
int m = 12;
int n = 10;
double x = 12.5;
double y;
y = (double) m / n;             // Float−Division, Resultat 1.2
y = x + (double) m / n;         // Float−Division, Resultat 13.7
```

   Dabei muss darauf geachtet werden, dass die Float-Konversion wirklich *vor* der Division erfolgt, sonst nützt sie nichts:

```
y = (double) (m / n);           // Integer−Division, Resultat 1
```

3.5 Erweiterte Zuweisungs-Operatoren

Häufig hat man algebraische Ausdrücke der Form:

```
a = a + b;
a = a − b;
a = a * b;
a = a / b;
```

Für solche Operationen stehen die folgenden abgekürzten Operatoren zur Verfügung:

```
a += b;
a -= b;
a *= b;
a /= b;
```

Man nennt diese Operatoren erweiterte Zuweisungsoperatoren. Wenn auf der rechten Seite ein zusammengesetzter Ausdruck ist, wird dieser zuerst ausgewertet, bevor die gewünschte Operation mit a durchgeführt wird:

```
a *= b + 2;                    // aequivalent zu a = a * (b + 2)
```

3.6 String-Konkatinierung mit '+'

Häufig hat man bei Ausgaben den Fall, dass man Text mit nachfolgenden numerischen Daten ausgeben möchte. Dies kann elegant mit *einem* Aufruf von 'print' bzw. 'println' geschehen, unter Verwendung des Operators '+' für Character-Strings:

```
double x=4.3;
System.out.println("Resultat: " + x);
```

Erklärungen:

Wenn bei einer '+' Operation einer der beiden Operanden ein Character-String ist, so wird der andere Operand in einen Character-String umgewandelt, und die beiden Strings werden zusammengefügt (konkatiniert).

Die Operation kann auch iteriert angewendet werden (wie der algebraische Operator '+') :

```
double x1, x2;
System.out.println("Loesung1: " + x1 + " Loesung2: " + x2);
```

Dabei wird zuerst der Wert von 'x1' in einen Character-String umgewandelt und an den ersten angehängt, usw.

△ **Uebung:** Lösen Sie die Übung 3.7.2

3.7 Übungen

3.7.1 Wann ist Ostern?

Das Osterdatum eines Jahres ist astronomisch definiert: Ostern ist der Sonntag nach dem ersten Vollmond, der am 21. März oder später eintritt.

Mit dem folgenden Algorithmus von T.H. O'Beirne kann das Osterdatum für jedes Jahr j von 1900 bis 2099 mit ganzzahligen Divisionen mit Rest berechnet werden.

Erstellen Sie eine Applikation 'Ostern', welche ein Jahr j einliest (mit 'InOut.getInt') und das zugehörige Osterdatum ausgibt.

Algorithmus:

Das Osterdatum eines Jahres j im Bereich

$$1900 \leq j \leq 2099$$

kann folgendermassen berechnet werden:

1. Setze $n = j - 1900$
2. Führe die folgenden 5 ganzzahligen Divisionen mit Rest durch:

Zähler	Nenner	Resultat	Rest
n	19	nicht benötigt	a
$7a + 1$	19	b	nicht benötigt
$11a + 4 - b$	29	nicht benötigt	m
n	4	q	nicht benötigt
$n + q + 31 - m$	7	nicht benötigt	w

3. Setze $k = 25 - m - w$
4. Wenn k positiv ist, ist Ostern am k-ten April, andernfalls ist sie im März, am Tag $31 + k$.

Kontroll-Daten: 1997: 30. März, 1998: 12. April

3.7.2 Umlaufzeit eines Satelliten

Erstellen Sie eine Applikation 'SatPer', welche die Umlaufzeit eines Satelliten auf einer Kreisbahn um die Erde berechnet, nach der Formel

$$T = \frac{2\pi}{R_E} \cdot \sqrt{\frac{r^3}{g}}$$

Sie ergibt die Umlaufzeit T in Sekunden, wenn die Grössen auf der rechten Seite in den Grundeinheiten Meter und Sekunde eingesetzt werden. Bedeutung der Terme:

$R_E = 6371\,km = 6.371 \cdot 10^6\,m$	Erdradius (Mittel)
$g = 9.81\,m/s^2$	Erdbeschleunigung auf der Erdoberfläche
r	Radius der Kreisbahn des Satelliten

Bemerkung:

In der Formel ist die Abnahme der Erdbeschleunigung mit wachsendem Abstand r berücksichtigt, obwohl nur das g auf der Erdoberfläche vorkommt.

Programm-Input:

Höhe h des Satelliten über der Erdoberfläche in km.

Der Bahnradius r ergibt sich daraus gemäss $r = R_E + h$ (R_E und h in Metern einsetzen).

Output:

Umlaufszeit T in Stunden umgerechnet.

Test:

Für $h = 36000\,km$ ergibt sich

$$T \;=\; 24.12\,Stunden \;\approx\; 1\,Tag$$

(geostationäre Satelliten).

Kapitel 4

Kontroll-Anweisungen

4.1 Strukturierte Programmierung

Der Methodiker Jackson hat erkannt, dass jedes Programm mit drei Grundstrukturen realisiert werden kann:

- Sequenz

  Bei einer Sequenz werden die Anweisungen nacheinander, der Reihe nach, durchgeführt.

- Fallunterscheidung (Selektion)

  Bei einer Fallunterscheidung wird aufgrund einer Bedingung einer von zwei Fällen ausgewählt.

- Schleife

  Bei einer Schleife wird eine Verarbeitung wiederholt durchgeführt, solange eine Bedingung erfüllt ist.

Programme, die aus diesen Grundstrukturen aufgebaut sind, nennt man *strukturierte Programme*.

Der entscheidende Fortschritt bei der Einführung der strukturierten Programmierung war der Verzicht auf den Sprungbefehl 'goto', welcher eine zentrale Ursache für undurchsichtige Programme war.

Zum Programmentwurf nach der Methode der strukturierten Programmierung gehört weiter das Prinzip der *schrittweisen Verfeinerung* eines Programmes:

Man beginnt mit einem Grobansatz und verfeinert diesen schrittweise. Dabei wird die Gesamtaufgabe in Teilaufgaben aufgeteilt, die unabhängig voneinander entwickelt werden können (Modularisierung).

Top Down Design

Die Methode der schrittweisen Verfeinerung heisst auch 'Top Down Design'.

In diesem Kapitel werden die drei Grundstrukturen für Programme eingeführt. Die Modularisierung ist Thema von nachfolgenden Kapiteln.

4.2 Die Sequenz

Eine Sequenz ist eine Folge von Anweisungen, die nacheinander ausgeführt werden:

```
statement_1;
statement_2;
...
statement_n;
```

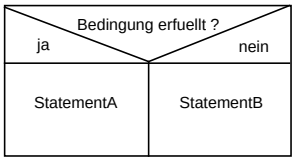
Eine Sequenz von Anweisungen kann in geschweifte Klammern eingeschlossen werden. Dadurch wird die ganze Sequenz für den Compiler als *ein* Statement betrachtet (zusammengesetztes Statement).

Dies ist wichtig, wenn eine Sequenz in ein anderes Statement (if, while usw.) eingefügt wird, siehe unten.

4.3 Die Selektion (Fallunterscheidung)

Die Grundform einer Fallunterscheidung kennen wir schon in der Form eines 'if'-Statements mit einer Bedingung:

```
if  ( Bedingung )
    StatementA;
else
    StatementB;
```



Struktogramm

Die graphische Darstellung ist das sogenannte *Struktogramm* der Selektion. Struktogramme sind Diagramme zur graphischen Darstellung von strukturierten Programmen.

Der 'else' Teil eines if-Statements kann wegfallen, wenn für diesen Fall keine Aktionen erforderlich sind.

Wenn im if- oder im else-Zweig mehrere Anweisungen durchzuführen sind, müssen diese mit geschweiften Klammern als zusammengesetztes Statement geschrieben werden:

```
if ( Bedingung )
{  Statement;
   Statement;
}
```

Aufbau von Bedingungen (Boolesche Ausdrücke)

Die in einer Fallunterscheidung auftretende Bedingung ist ein sog. *Boolescher Ausdruck*, d.h. ein Ausdruck mit möglichen Werten *true* oder *false*.

Boolesche Ausdrücke sind aus den folgenden Bestandteilen aufgebaut:

- Vergleichsoperatoren

<	kleiner
<=	kleiner oder gleich
>	grösser
>=	grösser oder gleich
==	gleich
!=	ungleich

  Achtung:

  Der Vergleichsoperator für Gleichheit ist '==', da das einfache Gleichheitszeichen schon für den Zuweisungsoperator vergeben ist.

- Logische Verknüpfungen

&&	Und-Verknüpfung
||	Oder-Verknüpfung
!	Negation (Verneinung)

  Der Oder-Operator besteht aus zwei Zeichen '|'. Auf der Schweizer-Tastatur wird dieses mit AltGr 7 erzeugt (Code 124 dezimal).

- Klammern: ( )

Typen-Konversionen bei Vergleichen

Bei Vergleichen werden wie in algebraischen Ausdrücken automatisch erforderliche Typen-Konversionen durchgeführt, sodass 'int' mit 'float' usw. verglichen werden kann.

Beispiele zu Booleschen Ausdrücken

1. ```
   if ( n >= 10   &&   n <= 100 )
       System.out.println("Wert im Bereich 10 .. 100");
   ```

2. ```
   if ( (n == 4   ||  n == 5 )   &&   ( x < 4.5) )    ...
   ```

3. Boolean-Vergleiche

   ```
   boolean fehler;
   if ( fehler == true )    ...
   ```

   Ein solcher Vergleich einer boolean-Variable mit *true* kann abgekürzt geschrieben werden:

   ```
   if ( fehler )    ...
   ```

   Analog kann eine Bedingung der Form

   ```
   if ( fehler == false )    ...
   ```

   mit dem Verneinungs-Operator geschrieben werden:

   ```
   if ( ! fehler )    ...
   ```

4. Character-Vergleiche

   ```
   char ch;
   if ( ch >= 'A'   &&   ch <= 'Z' )
     System.out.println("Gross-Buchstabe");
   ```

   Der Vergleich von Characters erfolgt aufgrund der Unicode-Werte.

Verneinungen

Die Umformung von Verneinungen ist eine häufige Fehlerquelle, wenn 'und' bzw. 'oder' Verknüpfungen dazu kommen.

1. Negation einer 'und'-Verknüpfung

   Eine 'und'-Verknüpfung ist genau dann erfüllt, wenn *beide* Bedingungen erfüllt sind. Folglich ist eine Negation

   ```
   ! ( a == 0   &&   b == 0 )
   ```

   genau dann erfüllt, wenn mindestens eine der beiden Bedingungen falsch ist, d.h. a *oder* b verschieden von 0 ist. Damit ist die Negation äquivalent zu

   ```
   a != 0   ||   b != 0
   ```
   'und' ersetzt durch 'oder' !

2. Negation einer 'oder'-Verknüpfung

   Eine 'oder'-Verknüpfung ist genau dann erfüllt, wenn mindestens eine der beiden Bedingungen erfüllt ist. Folglich ist eine Negation:

   ```
   ! ( a == 0   ||   b == 0 )
   ```

   genau dann erfüllt, wenn beide Bedingungen falsch sind, d.h. a *und* b verschieden von 0 sind. Damit ist die Bedingung äquivalent zu

   ```
   a != 0   &&   b != 0
   ```
   'oder' ersetzt durch 'und'

Allgemeine Umformungsregeln der Logik:

not (B and C)	ist äquivalent zu	(not B) or (not C)
not (B or C)	ist äquivalent zu	(not B) and (not C)

Vorzeitiger Abbruch bei 'Und'- bzw. 'Oder'-Auswertungen

Die Operatoren '&&' und '||' sind *Kurzversionen*, d.h. der Ausdruck auf der rechten Seite des Operators wird *nicht* ausgewertet, wenn das Resultat nach der Auswertung des linken Ausdrucks schon feststeht. Dies ist in gewissen Situationen wichtig.

Beispiel:

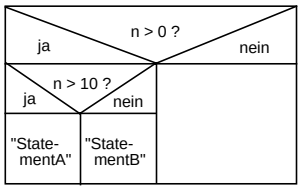
```
x >= 0.0   &&   Math.sqrt(x) < max
```

Wenn `x < 0` ist, wird der rechte Ausdruck nicht mehr ausgewertet, was wichtig ist, da `sqrt(x)` für negative `x` nicht definiert ist.

Verschachtelte if-Statements

In verschachtelten if-Statements bezieht sich ein *else* immer auf das nächst höhere if:

```
if (n > 0)
  if (n > 10)
    StatementA;
  else
    StatementB;
```



△ **Uebung:** Lösen Sie die Aufgabe 4.7.1

Mehrfach-Fallunterscheidungen

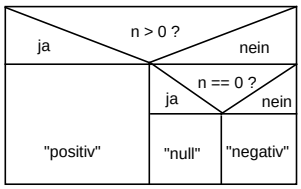
Wenn bei einer Fallunterscheidung mehr als zwei Fälle vorgegeben sind, aus welchen einer auszuwählen ist, so spricht man von einer *Mehrfach-Fallunterscheidung*.

Eine Mehrfach-Fallunterscheidung kann mit verschachtelten if-Statements auf Zweifach-Fallunterscheidungen zurückgeführt werden.

Beispiele:

1. Dreifach-Fallunterscheidung

```
int n;
if ( n > 0 )
  System.out.println("positiv");
else
  if ( n == 0 )
    System.out.println("null");
  else
    System.out.println("negativ");
```



Übersichtlichere Schreibweise:

```
if ( n > 0 )
  System.out.println("positiv");
else if ( n == 0 )
  System.out.println("null");
else
  System.out.println("negativ");
```

2. Umsetzung einer Wochentag-Nummer in Text

```
int wochentag;                    // 0=So, 1=Mo usw.

if ( wochentag == 0 )  System.out.println("Sonntag");
else if ( wochentag == 1 )  System.out.println("Montag");
else if ( wochentag == 2 )  System.out.println("Dienstag");
...
else if ( wochentag == 6 )  System.out.println("Samstag");
else  System.out.println("ungueltiger Tag");
```

Das switch-Statement

Für Mehrfach-Fallunterscheidungen, welche aufgrund eines ganzzahligen Wertes erfolgen (wie die oberen beiden Beispiele), stellt Java ein speziell übersichtliches Statement zur Verfügung, das 'switch'-Statement:

```
int  wochentag;

switch (wochentag)
{ case 0 :   System.out.println("Sonntag");
             break;
  case 1 :   System.out.println("Montag");
             break;
  case 2 :   System.out.println("Dientag");
             break;
    ...
  case 6 :   System.out.println("Samstag");
             break;
  default:   System.out.println("ungueltiger Tag");
}
```

Wirkungsweise:

Das 'switch'-Statement ist eigentlich eine Sprung-Tabelle:

Der bei 'switch' angegebene Wert (wochentag) bestimmt, wohin gesprungen wird, man nennt diesen Wert daher den *Selektor* der Fallunterscheidung. Er wird mit den bei den 'case'-Statements angegebenen Werten verglichen. *Selektor*

Der Programm-Ablauf verzweigt (mit einem 'goto') zum ersten Fall, dessen angegebener Wert gleich dem Selektor ist.

Von dort geht es *sequentiell* weiter, bis eine 'break'-Anweisung angetroffen wird. Eine solche bewirkt die Beendigung des Statements, d.h. eine Verzweigung zum nächsten Befehl nach dem 'switch'-Statement.

Wenn kein Fall zutrifft, erfolgt der Sprung zum 'default'-Zweig, sofern dieser vorhanden ist, sonst zum Ende des 'switch'-Statements.

Bemerkungen

break

1. Die 'break'-Anweisungen in einem 'switch'-Statement sind unbedingt erforderlich. Vergisst man sie, so werden ab dem ersten Fall der zutrifft, *alle* nachfolgenden Fälle ebenfalls ausgeführt!

2. Der Selektor eines 'switch'-Statements muss ein Integer-Wert sein. Er darf auch als zusammengesetzter Ausdruck angegeben werden.

3. Die bei den 'case'-Statements angegebenen Werte müssen *konstante* Integer-Ausdrücke sein (nur Literals und Konstanten erlaubt). Jeder Wert darf in der 'case'-Liste nur *einmal* erscheinen.

4. Der 'default'-Zweig kann auch weggelassen werden.

5. Die Statements der einzelnen Fälle müssen nicht in Klammern eingeschlossen werden, da sie sequentiell durchlaufen werden, bis ein 'break' kommt:

```
char operator;
double a, b, x;

switch ( operator )
{   case '+' :   System.out.println("Summe");
                 x = a + b;
                 break;
    case '−' :   System.out.println("Differenz")
                 x = a − b;
                 break;
}
```

6. Wenn für mehrere Werte die gleiche Verarbeitung durchzuführen ist, können mehrere case-Klauseln vor die betreffende Verarbeitung geschrieben werden:

```
int monat;
switch ( monat )
{    case 1 :
     case 2 :
     case 12 :  System.out.println("Winter");
                break;
```

```
        case 3 :
        case 4 :
        case 5 :    System.out.println("Fruehling");
                    break;
        case 6 :
        case 7 :
        case 8 :    System.out.println("Sommer");
                    break;
        case 9 :
        case 10 :
        case 11 :   System.out.println("Herbst");
                    break;
    }
```

4.4 Schleifen

Bei einer Schleife wird eine bestimmte Aktion wiederholt durchgeführt, solange eine bestimmte *Fortsetzungsbedingung* erfüllt ist. Neben der 'while'-Schleife, die wir schon kennen gibt es in Java noch zwei weitere Arten von Schleifen.

Übersicht:

- 'while'-Schleifen
- Zählschleifen ('for'-Schleifen)

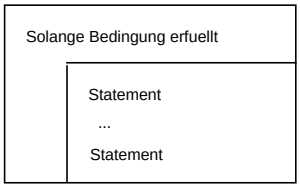
  Bei einer Zählschleife wird ein Zähler mitgeführt, der einen bestimmten Bereich *von .. bis* durchläuft.
- 'do-while'-Schleifen

  Bei 'do-while'-Schleifen wird die Fortsetzungsbedingung (im Gegensatz zum 'while'-Statement) *nach* der Verarbeitung geprüft.

While-Schleifen

Das normale 'while'-Statement kennen wir schon:

```
while ( Bedingung )
{  Statement;
   ...
   Statement;
}
```



Das Struktogramm veranschaulicht die Wirkung: Die Statements in den geschweiften Klammern werden wiederholt ausgeführt, solange die in den runden Klammern angegebene Bedingung erfüllt ist.

- Die Bedingung ist ein beliebiger Boolescher Ausdruck, wie bei 'if'-Statements. Sie wird, wie im Diagramm dargestellt, *vor* der Durchführung der Anweisungen geprüft. Wenn sie *false* ist, werden die Statements nicht mehr ausgeführt.
- Wenn die Bedingung schon beim ersten Mal falsch ist, wird die Aktion *keinmal* durchgeführt, das while-Statement hat dann keine Wirkung.
- Wenn in der Schleife nur *ein* Statement durchzuführen ist, können die geschweiften Klammern weggelassen werden.

Beispiele:

1. Die Siebner-Reihe

   Ausgabe aller Vielfachen von 7, die kleiner als 1000 sind:

   ```
   int n = 7;
   while ( n < 1000 )
   { System.out.println(n);
     n = n + 7;
   }
   ```

2. Die Harmonische Reihe

   Wir wollen untersuchen, ob es möglich ist, mit genügend vielen Gliedern der Reihe

   $$1 + 1/2 + 1/3 + ...$$

den Wert 10.0 zu übertreffen. Die Reihe heisst Harmonische Reihe. Obwohl ihre Glieder gegen null streben, divergiert die Reihe, d.h. die Summe wächst über alle Grenzen, wenn man genügend viele Glieder summiert.

Zur Lösung der Aufgabe führen wir eine double-Variable 'summe' ein, in welcher wir die Glieder in einer Schleife sukzessive aufsummieren. Weiter sei 'i' eine int-Variable, die immer angibt, wieviele Glieder schon summiert wurden.

```
int i = 1;
double summe = 1;                    // erste Summe
while ( summe <= 10.0 )
{ i++;
  summe += 1.0 / i;
}
System.out.println("Anz. Glieder: " + i);
System.out.println("Summe: " + summe);
```

△ **Uebung:** Lösen Sie die Aufgaben 4.7.2 und 4.7.3.

For-Schleifen (Zählschleifen)

Das for-Statement von Java ist ein abgekürzt geschriebenes while-Statement, welches sich insbesondere für Zählschleifen eignet.

Wir führen das Statement am Beispiel einer Zählschleife ein, in welcher die Laufvariable die Werte von 1 bis 100 durchläuft:

```
int i;
for ( i = 1; i <= 100; i++ )
{ System.out.print("Durchlauf: ");
  System.out.println(i);
}
```

Dieses for-Statement ist äquivalent zu dem folgenden while-Statement:

```
i = 1;
while ( i <= 100 )
{ System.out.print("Durchlauf: ");
  System.out.println(i);
  i++;
}
```

Erklärungen:

Bei einem for-Statement werden in runden Klammern drei Ausdrücke (getrennt durch Strichpunkte, nicht Kommas) angegeben:

- Eine Initialisierungs-Anweisung ('i = 1'), die vor der Schleife einmal durchgeführt wird.
- Eine Fortsetzungsbedingung ('i <= 100') für die Schleife, die bei jedem Durchlauf *vor* der Durchführung der Anweisungen der Schleife geprüft wird.
- Eine Schleifenanweisung ('i++'), die nach jeder Durchführung der Anweisungen der Schleife durchgeführt wird.

Allgemeines Format:

```
for ( InitAnweisung; Bedingung; SchleifenAnweisung )
{ ...
}
```

Bedeutung:

```
InitAnweisung;
while ( Bedingung )
{ ...
  SchleifenAnweisung;
}
```

Bemerkungen:

1. Falls die Schleifenverarbeitung eines for-Statements nur aus *einem* Statement besteht, können die geschweiften Klammern weggelassen werden.
2. In einem for-Statement müssen nicht alle drei Ausdrücke angegeben werden, es können auch Null-Statements, gekennzeichnet durch Strichpunkte verwendet werden.

   Beispiel:

   Die int-Variable 'zaehler' enthalte einen positiven Wert. Dann läuft die folgende Schleife bis der Wert auf 0 reduziert worden ist:

```
for ( ; zaehler > 0; zaehler--)          // ohne init_anweisung
{ ...
}
```

3. Lokale Laufvariablen

   Im Initialisierungs-Statement eines for-Statements kann eine Laufvariable direkt definiert werden, durch eine vorangestellte Typen-Angabe.

```
for ( int k = 1; k <= 100; k++ )
{ System.out.print("Durchlauf: ");
  System.out.println(k);
}
```

   Die Variable *k* ist nur innerhalb der Schleife verwendbar. Nach dem Ende der Schleife existiert sie nicht mehr. Man nennt sie daher eine *lokale Laufvariable* der Schleife.

   Der Name einer lokalen Laufvariablen darf nicht mit dem Namen einer bestehenden Variablen der Methode, in der das for-Statement steht, übereinstimmen.

Beispiel:

Berechnung der Summe der ersten 100000 Glieder der Harmonischen Reihe 1 + 1/2 + 1/3 + ...

```
double summe = 0;
for ( int i = 1; i <= 100000; i++ )
  summe += 1.0 / i;
System.out.println("Summe: " + summe);
```

Die Summe kann auch in absteigender Reihenfolge berechnet werden:

```
double summe = 0;
for ( int i = 100000; i > 0; i-- )
  summe += 1.0 / i;
System.out.println("Summe: " + summe);
```

Nach den Gesetzen der Mathematik müsste in beiden Fällen dasselbe Resultat herauskommen. Auf dem Computer kommt jedoch nicht exakt

dasselbe heraus, weil bei Float-Rechnungen immer Rundungsfehler ins Spiel kommen, die sich unterschiedlich auswirken, je nachdem ob eine kleine Zahl zu einer grossen oder einer kleinen addiert wird.

△ **Uebung:** Lösen Sie die Aufgabe 4.7.4

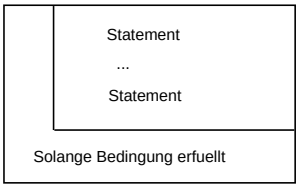
Do-while Schleifen

Die 'do-while'-Schleifen sind nur für spezielle Situationen. Sie unterscheiden sich von den 'while'-Schleifen darin, dass die Fortsetzungs-Bedingung *nach* der Ausführung der Schleifen-Statements geprüft wird.

```
do
   Statement;
while ( Bedingung );
```

oder, wenn in der Schleife mehrere Anweisungen durchzuführen sind:

```
do
{   Statement;
    ...
    Statement;
} while ( Bedingung );
```



Bei einer 'do-while'-Schleife wird die Verarbeitung also mindestens einmal ausgeführt. Diese Schleifen eignen sich für Situationen, in denen die Fortsetzungsbedingung innerhalb der Schleife entsteht. Im Normalfall sind 'while'-Schleifen vorzuziehen.

Beispiel:

Ausgabe der Ziffern einer nicht negativen, ganzen Zahl n, z.B. $n = 123$. Die Ziffern sind untereinander auf den Bildschirm auszugeben, beginnend mit der Einerziffer.

```
do
{ System.out.println(n % 10);          // letzte Ziffer von n ausgeben
  n = n / 10;                          // letzte Ziffer abschneiden
} while (n > 0);
```

Die Schleife funktioniert auch im Falle $n = 0$. Enthält nämlich n am Anfang den Wert 0, so wird die Aktion trotzdem einmal durchgeführt, sodass eine '0' ausgegeben wird, wie es sein muss.

Nach demselben Verfahren, können übrigens auch die Ziffern der *Dualdarstellung* der Zahl berechnet werden, es muss lediglich der Wert 10 im Algorithmus ersetzt werden durch 2. Die Ziffern sind dann natürlich alle '0' oder '1'.

△ **Uebung:** Lösen Sie die Aufgabe 4.7.5

Verschachtelte Schleifen

Schleifen können nach Belieben verschachtelt werden:

```
for ( int i = 1; i <= 10; i++ )
  for ( int j = 1; j <= 10; j++ )
    System.out.println( "i: " + i + "j: " + j );
```

In der äusseren Schleife läuft i von 1 bis 10. Für jeden Wert von i läuft die innere Schleife ab, bei welcher j von 1 bis 10 variiert.

Wie sieht die Ausgabe aus ?

Vorzeitiger Abbruch einer Schleife

Eine Schleife (while, for, do−while) kann jederzeit abgebrochen werden mit dem 'break'-Statement. *break*

Beispiel:

```
while ( ... )
{ ...
  if ( fehler )
    break;                                  // Schleife verlassen
  ...
}
```

Bei verschachtelten Schleifen wird bei einer 'break'-Anweisung nur die Schleife abgebrochen, in welcher die Anweisung steht.

4.5 Beenden einer Methode

Wie wir wissen, wird eine Funktion mit 'return', gefolgt von einem Ausdruck, dessen Wert dem Resultat zugewiesen wird, beendet.

return

In einer Prozedur kann das 'return'-Statement verwendet werden um die Methode an einer beliebigen Stelle vorzeitig zu beenden. Dazu wird das 'return'-Statement ohne nachfolgenden Wert geschrieben:

```
return;
```

4.6 Beenden eines Programmes

Mit dem Statement

exit

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

kann ein Programm an einer beliebigen Stelle beendet werden.

Dabei wird ein Return-Code angegeben (0 für normales Ende). Dieser Code kann bei Bedarf auf Betriebssystem-Ebene verwertet werden.
Der 'exit'-Befehl sollte in Applets nicht verwendet werden.

4.7 Übungen

4.7.1 Schaltjahre

Erstellen Sie eine Applikation 'Schaltjahre', welche für ein Jahr abklärt, ob es ein Schaltjahr ist oder nicht. Das Jahr wird mit der Methode 'InOut.getInt' eingelesen.

Schaltjahr-Definition nach Gregorianischem Kalender:

- Die Jahrhundert-Jahre (1900, 2000, 2100, ...) sind nur dann Schaltjahre, wenn sie durch 400 teilbar sind.
- Die Nicht-Jahrhundert-Jahre sind genau dann Schaltjahre, wenn sie durch 4 teilbar sind.

Zeichnen Sie ein Struktogramm, bevor Sie das Programm schreiben. Überprüfen Sie das Programm für die Jahre 1900, 1984, 1990, 2000.

4.7.2 Notendurchschnitt

Erstellen Sie eine Applikation zur Berechnung von Notendurchschnitten. Das Programm liest 'double'-Werte ein, bis ein negativer Wert vorliegt (=Ende Eingabe).

Anschliessend wird der Durchschnitt (Mittelwert) der Eingabewerte ohne dem letzten (negativen) Wert ausgegeben.

4.7.3 Potenzen

Erstellen Sie eine Funktion

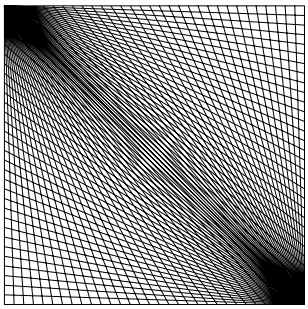
```
double potenz(double a, int n)
```

welche die Potenz a^n berechnet, wobei $n \geq 0$ vorausgesetzt wird. Dabei soll der am folgenden Beispiel dargestellte (schnelle) Algorithmus verwendet werden:

$$2^{10} = 4^5 = 4 \cdot 4^4 = 4 \cdot 16^2 = 4 \cdot 256^1 = 1024$$

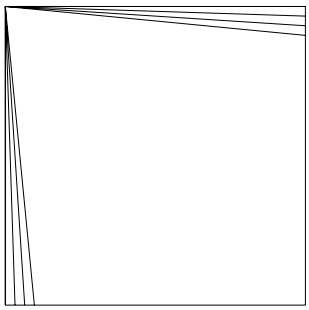
4.7.4 Moirée-Effekt

Erstellen Sie ein Applet 'Moiree', welches die folgende Figur zeichnet:



Die Figur besteht aus zwei Serien von Strecken, die erste von der linken oberen Ecke aus, die zweite von der rechten unteren. Das Phänomen, dass dabei zusätzliche Muster erscheinen, nennt man Moirée-Effekt.

Die erste Serie von Strecken entsteht folgendermassen:



Konstanten:

Anzahl Strecken mit Endpunkt auf einer Kante: $n = 32$
Abstand der Endpunkte zweier Strecken: $d = 8$ (Pixel)
Quadrat-Kante: $a = (n - 1) \cdot d$
linke obere Ecke: $left = 100$, $top = 40$

4.7.5 Primteiler

Erstellen Sie eine Applikation 'Prim', die für eine positive ganze Zahl n den kleinsten Primfaktor berechnet.

Algorithmus:

Durchlaufen Sie die Werte $p = 2, 3, 5, 7, \ldots$, bis n durch p teilbar ist, oder $p \cdot p > n$ ist. Im zweiten Fall ist n eine Primzahl.

Kapitel 5

Arrays

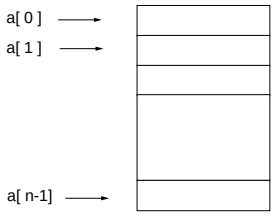
5.1 Einführung von Arrays

Ein *Array* ist eine Folge von Elementen eines bestimmten Datentyps. Die Elemente werden mit einem festen Namen, dem Namen des Arrays, und *Nummern* (Indizes) angesprochen.

Ist a ein Array von n Elementen, so werden seine Elemente mit

a[0], a[1], ... , a[n−1]

bezeichnet. Im Speicher liegen die Elemente direkt hintereinander:



Für die Indizes können beliebige ganzzahlige Ausdrücke mit Variablen, Konstanten usw. verwendet werden. Dies ermöglicht die Verarbeitung der Elemente mit *Schleifen*.

Der Datentyp der Elemente eines Arrays heisst *Elementtyp* des Arrays.

5.2 Definition eines Arrays

Die Definition eines Arrays erfolgt in Java in zwei Schritten. Im ersten Schritt wird eine Variable für den Array definiert:

```
int[ ]  a;                          // Variable für den Array
```

Dabei ist 'int' der Elementtyp des Arrays. Die leeren Klammern bedeuten, dass eine Variable für einen Array definiert wird. Der Name *a* der Variablen ist beliebig wählbar. Die Variable *a* ist nur eine sogenannte Referenzvariable

Die Elemente des Arrays werden in einem zweiten Schritt mit dem 'new'-Operator erzeugt:

```
a = new int[10];                    // 10 Elemente erzeugen
```

Dadurch werden die Elemente `a[0]`, `a[1]`, ... , `a[9]` definiert. Die Variable *a* enthält technisch gesehen die Speicheradresse des ersten Elementes. Genaueres dazu siehe unten.

Bemerkungen:

1. Die beiden Schritte zur Erzeugung eines Arrays können auch zu *einem* Statement zusammengefasst werden:

   ```
   int[ ]  a = new int[10];
   ```

   Manchmal ist es jedoch vorteilhaft, wenn die Erzeugung der Elemente später erfolgen kann, z.B. wenn die Anzahl Elemente von einem Input-Wert abhängt.

2. Bei der Erzeugung der Elemente eines Arrays wird die gewünschte *Anzahl Elemente* angegeben. Die Indizes der Elemente laufen immer von 0 bis *Anzahl Elemente - 1*.

   Die Anzahl Elemente wird mit einem beliebigen Integer-Ausdruck spezifiziert. Der Ausdruck wird an der Stelle der Definition ausgewertet. Nach der Definition der Elemente kann die Anzahl *nicht* mehr verändert werden.

3. Abfrage der Anzahl Elemente

   Die Anzahl Elemente eines Arrays 'a' kann jederzeit abgefragt werden mit dem Ausdruck:

a.length

   ```
   a.length
   ```

   Dies ist eine 'int'-Variable, die normal verwendet werden kann.

4. Die Elemente eines Arrays werden (im Gegensatz zu normalen Variablen) automatisch initialisiert: numerische Elemente mit 0, Boolesche Elemente mit *false*.

5.3 Verwendung von Arrays

Der entscheidende Punkt von Arrays ist die Möglichkeit, die Elemente mit *Schleifen* zu verarbeiten:

Beispiele:

1. Alle Elemente auf 0 setzen:

```
for ( int i=0; i < a.length; i++ )
  a[i] = 0;
```

2. Bestimmung des kleinsten Elementes:

```
int min;
min = a[0];                              // erster Kandidat fuer Minimum
for ( int i=1; i < a.length; i++ )
  if ( a[i] < min )
    min = a[i];                          // neues Minimum
```

Bei solchen Schleifen muss darauf geachtet werden, dass die Index-Werte mit denen Elemente angesprochen werden, im Bereich

```
0 .. a.length−1
```

liegen. Ist dies nicht der Fall, so wird eine sog. *Exception* (Ausnahmesituation) ausgelöst, welche das Programm mit einer entsprechenden Fehlermeldung abbricht: *Exception*

```
ArrayIndexOutOfBoundsException
```

Beispiel: Test von Zufallszahlen

Die folgende Applikation erzeugt 1000 ganze Zufallszahlen im Bereich 0 bis 9 und zählt die Vorkommen der einzelnen Werte.

Dazu wird ein Array a mit 10 Elementen verwendet. Die Elemente sind Zähler für die Werte 0 .. 9 der Zufallszahlen.

```
public class RndTest
{
  static int randomInt()                          // ganze Zufallszahl in 0 .. 9
  { double x = 10 * Math.random();
    int wert = (int) x;                           // Dezimalstellen abschneiden
    if ( wert == 10 )                             // Extremwert reduzieren
      wert = 9;
    return wert;
  }

  static public void main( String[ ] args )
  { final int nWerte = 1000;                      // Anzahl Zufallszahlen
    int i, k;
    int[ ] a = new int[10];                       // Array von Zaehlern
    for( k=0; k < nWerte; k++ )
    { i = randomInt();                            // ganze Zufallszahl in 0 .. 9
      a[i]++;                                     // zugehoerigen Zaehler erhoehen
    }
    for (i=0; i < a.length; i++)                  // Ausgabe
      System.out.println(a[i]);
  }
}
```

Die Funktion 'randomInt' erzeugt eine *ganze* Zufallszahl im Bereich 0 bis 9. Dazu berechnet sie zunächst eine *reelle* Zufallszahl x im Intervall $[\,0, 10\,]$ und schneidet dann die Dezimalstellen ab. Dies ergibt für $x < 10$ die folgenden Resultate:

x	Resultat
$0 \leq x < 1$	0
$1 \leq x < 2$	1
...	
$9 \leq x < 10$	9

Der Extremfall $x = 10.0$ wird speziell behandelt, er wird zum letzten Teilintervall hinzugenommen (Resultat 9). Dadurch scheint die 9 bevorzugt zu sein. Statistisch spielt dies jedoch *keine* Rolle, da ein exakter Wert wie 10.0 Wahrscheinlichkeit 0 hat.

△ **Uebung:** Lösen Sie die Aufgabe 5.7.1

5.4 Initialisierung mit Werteliste

Bei der Definition der Elemente eines Arrays kann anstelle des 'new'-Operators eine Liste von Werten für die Elemente in geschweiften Klammern angegeben werden:

Beispiel:

```
double[ ] p = { 1.2, 3.2, −5.3 };          // Raumpunkt
```

Dieses Statement ist äquivalent zur Sequenz:

```
double[ ] p = new double[3];
p[0] = 1.2;
p[1] = 3.2;
p[2] = −5.3;
```

Dabei können die Werte in den geschweiften Klammern beliebige Ausdrücke sein, unter Beachtung der Datentypen: Die angegebenen Werte müssen den Elementen des Arrays zugewiesen werden können (mit eventuellen automatischen Konversionen, wie bei Zuweisungen).

```
double z = 2.4;
double[ ] p = { 0, Math.sqrt(2), z * z };
```

▶ *Merke:*

Eine solche Zuweisung von Werten in geschweiften Klammern geht nur bei der *Definition* eines Arrays, später nicht mehr.

5.5 Array-Variablen und Speicheradressen

Die Bytes des Speichers eines Computers sind numeriert mit Nummern 0, 1, 2, ... Die Nummer eines Bytes nennt man seine *Adresse* (Speicheradresse).

Bei einem Zugriff auf das i-te Element eines Arrays wird die Adresse des Elementes aus der Anfangsadresse des Arrays berechnet:

```
Adresse Element i = Anfangsadresse + i * ElementLaenge.
```

Die Anfangsadresse des Arrays ist in der *Variablen* des Arrays gespeichert. Variablen, die Adressen gespeichert haben, nennt man in Java *Referenzvariablen.*

Referenzvariable

Referenzvariablen werden im nächsten Kapitel wichtig, bei Objekten. Arrays haben in Java viele Eigenschaften von Objekten.

▶ *Merke:* Eine Array-Variable enthält die Anfangsadresse des Arrays.

Diese Tatsache ist bei den folgenden Operationen wichtig.

1. Kopieren eines Arrays

   Beim Kopieren eines Arrays müssen die Elemente einzeln in einer Schleife übertragen werden:

   ```
   int[ ] a = new int[10];
   int[ ] b = new int[a.length];
   for ( int i=0; i < a.length; i++ )
     b[i] = a[i];
   ```

   Die Zuweisung

   ```
   b = a;
   ```

   ist zwar ebenfalls zulässig, sie überträgt jedoch nur den Inalt der Variablen a, d.h. die *Adresse* des Arrays. Nach der Zuweisung referenzieren folglich die Variablen a und b die gleichen Elemente im Speicher!

   Die ursprünglich für b erzeugten Elemente sind dadurch verloren, da sie nicht mehr angesprochen werden können.

2. Arrays als Parameter

   Arrays können als Parameter von Methoden verwendet werden. Die folgende Funktion 'max' bestimmt den grössten Wert eines 'int'-Arrays:

   ```
   int  max( int[ ] a )
   { int tmpMax = a[0];
     for (int i=1; i < a.length; i++)
       if ( a[i] > tmpMax )
         tmpMax = a[i];
     return tmpMax;
   }
   ```

   Aufruf der Methode:

   ```
   int[ ] werte = { 3, 10, 4, 20, 15, −2 };
   int m = max(werte);
   ```

▶ *Merke:*

Bei der Übergabe einer Array-Variablen als Parameter wird der übergebene Wert (wie üblich bei Parameter-Übergaben) auf den Stack kopiert.

Da eine Array-Variable die Adresse des Arrays enthält, wird folglich (wie bei einer Zuweisung) diese Adresse auf den Stack kopiert. Von den Elementen des Arrays werden *keine* Kopien erstellt.

Das bedeutet, dass die Methode direkt mit dem Array des aufrufenden Programmes arbeitet. Wenn sie Elemente verändert, beziehen sich die Änderungen auf den Array im aufrufenden Programm.

Folgerung:

Arrays die einer Methode als Parameter übergeben werden, können in der Methode verändert werden!

Dies steht im Gegensatz zu Änderungen von normalen Parametern, die ja am Ende der Methode verloren gehen, weil sie sich nur auf die lokalen Kopien der Parameter auf dem Stack beziehen.

Die folgende Methode setzt alle Elemente eines übergebenen 'int'-Arrays auf 0:

```
void clear(int[ ] a)
{ for (int i = 0; i < a.length; i++ )
    a[i] = 0;
}
```

△ **Uebung:** Lösen Sie die Aufgabe 5.7.2

Bemerkung:

Arrays können auch als Resultate von Funktionen verwendet werden, siehe Seite 96

5.6 Zweidimensionale Arrays

Zweidimensionale Arrays (Matrizen) sind in Java Arrays von Arrays. Sie werden folgendermassen definiert:

```
double[ ][ ]  a = new double[3][4];          // Matrix mit 3 Zeilen und 4 Spalten
```

oder direkt mit Wertzuweisungen:

```
double[ ][ ]  a = { { 13, 15, 19, 22 },
                    { 12, 18, 33, 66 },
                    { 48, 55, 33, 56 } };
```

In beiden Fällen wird ein Array aus 3 Elementen erzeugt. Jedes Element ist selber wieder ein Array von 4 Float-Elementen.
Das Ganze ist also eine 3×4 Matrix, d.h. eine Matrix mit 3 Zeilen und 4 Spalten:

13	15	19	22
12	18	33	66
48	55	33	56

Verwendung des Arrays:

Da a ein Array von Arrays ist, stehen die folgenden Ausdrücke zur Verfügung:

Ausdruck	Bedeutung
a[i]	i-te Zeile: Array von 4 Elementen (i=0, 1, 2)
a[i][j]	Element in Zeile i und Spalte j
a.length	Zeilenzahl
a[i].length	Spaltenzahl (Anz. Elemente der Zeile i)

Bildschirmausgabe:

```
for (int i=0; i < a.length; i++)
{ for (int j=0; j < a[i].length; j++)
     InOut.print(a[i][j], 6, 2);                  // 2 Dezimalstellen
  InOut.println("");                              // Zeilenumbruch
}
```

5.7 Übungen

5.7.1 Würfelsummen

Erstellen Sie eine Applikation 'Wuerfel', die alle möglichen Augenkombinationen beim Wurf von drei Würfeln durchläuft und dabei zählt, wie oft die möglichen Summen 3, 4, ... 18 vorkommen.

Ausgabe:

Summe	Vorkommen
3	1
4	3
...	...
18	1

Das Durchlaufen aller Augenkombinationen überlegt man sich am besten folgendermassen: Wenn nur *ein* Würfel vorliegt, gibt es einfach eine 'for'-Schleife von 1 .. 6.

Für zwei Würfel lässt man für jeden Wert des ersten Würfels eine zweite 'for'-Schleife von 1 .. 6 laufen. Für drei Würfel gibt es schliesslich drei verschachtelte 'for'-Schleifen.

Weiter wird für jede mögliche Summe von 3 .. 18 ein Zähler benötigt, der bei jedem Vorkommen der betreffenden Summe um 1 erhöht wird. Dazu eignet sich ein Array.

5.7.2 Polynomauswertung mit Horner-Schema

Erstellen Sie eine Applikation 'Horner', die den Wert y eines Polynoms

$$y = a_o + a_1 x + \ldots + a_n x^n$$

für ein x berechnet.

Programm-Input:

Grad n, Koeffizienten a_o bis a_n, Wert x.

Algorithmus:

Verwenden Sie das Horner-Schema, welches mit möglichst wenig Rechenoperationen auskommt: Der gesuchte Wert wird in einer Schleife in einer Hilfsvariablen h schrittweise aufgebaut:

Schritt	Wert von h
1	$h_1 = a_n$
2	$h_2 = a_{n-1} + h_1 \cdot x$
3	$h_3 = a_{n-2} + h_2 \cdot x$
...	...

Nach Schritt $n + 1$ hat man a_n also n-mal mit x multipliziert, a_{n-1} einmal weniger, usw. Damit enthält h am Ende den gesuchten Wert.

Bemerkung:

Die Koeffizienten des Polynoms werden in einem Array gespeichert, für die verschiedenen Werte von h genügt *eine* Variable.

Führen Sie eine Hilfsmethode (Funktion) ein, welche die Koeffizienten des Polynoms als Parameter erhält und den berechneten Wert als Resultat zurückgibt.

Kapitel 6

Klassen und Objekte

We should recognize that the art of programming is the art of organizing complexity.

– E.W.Dijkstra

6.1 Einleitung

Objekte

Ein Objekt (der Software-Welt) ist eine Zusammenfassung von *Daten* (Variablen) und *Methoden* zu einer Einheit im Speicher des Computers.

Die Daten beschreiben den *Zustand* des Objektes und die Methoden ermöglichen Aktionen, die den Zustand des Objektes verändern.

Mit dieser Definition lassen sich viele Objekte der realen Welt im Speicher des Computers nachbilden, sodass eine virtuelle Welt entsteht.

Beispiel: Stop-Uhren

- Daten: Status der Uhr (gestoppt, gestartet), momentane Laufzeit
- Methoden (Aktionen): Starten, Stoppen, Ablesen

Die Daten eines Objektes nennt man auch *Datenkomponenten*.

So lassen sich auch komplexe Objekte mit vielen Datenkomponenten und Aktionen in der Software-Welt nachbilden, z.B. ein MP3-Player,

mit dem MP3 Musik-Files abgespielt werden können. Dieses Beispiel zeigt, dass auch ganze Programme nichts anderes sind als Objekte im Speicher, mit Daten und Methoden.

Das Graphics-Objekt 'g'

Wir kennen schon ein wichtiges Objekt von Java: das Graphics-Objekt 'g', welches in einem Applet das Bildschirm-Window repräsentiert:

- Die Daten des Objektes enthalten die momentanen Einstellungen: Zeichenfarbe, Schrift-Attribute usw.
- Die Methoden ermöglichen Veränderungen der Einstellungen sowie Bildschirm-Ausgaben von Text und Graphik:

```
g.setColor(Color.red);
g.drawLine(10, 10, 100, 10);
```

Klassen und Instanzen

In Java gehört jedes Objekt zu einer *Klasse*. Bevor ein Objekt erzeugt werden kann, muss eine Klasse definiert werden.

Eine Klasse ist eine Vorlage ('Blueprint') für einen Typ von gleichartigen Objekten. In der Klasse sind die Datenkomponenten und Methoden der zugehörigen Objekte festgelegt.

M.a.W. definiert eine Klasse einen *Datentyp* für Objekte, so wie 'int' und 'double' Datentypen für Variablen repräsentieren.

Die Objekte einer Klasse nennt man auch *Instanzen* der Klasse.

Die Details zur Definition von Klassen und Objekten werden in den folgenden Paragraphen eingeführt.

6.2 Objekte als Datenverbunde

Records

Die einfachste Art von Objekten sind solche, die nur Datenkomponenten, jedoch keine Methoden enthalten. Diese nennt man *Datenverbunde* oder *Records*. Sie wurden schon in konventionellen Programmiersprachen (Pascal, C) eingeführt.

Im folgenden werden die erforderlichen Schritte für die Definition und Verwendung von Objekten an einem typischen Beispiel eingeführt:

1. Definition einer Klasse
2. Erzeugung von Objekten (Instanzen)
3. Verwendung der Objekte

Beispiel: Kreise auf dem Bildschirm

Ein Kreis ist festgelegt durch die Mittelpunkt-Koordinaten xm, ym (Bildschirm-Koordinaten) und den Radius r.

Definition der Klasse 'Kreis':

Die Datenkomponenten der Objekte werden in der Definition der Klasse festgelegt:

```
class Kreis
{ int xm, ym;                    // Mittelpunkt−Koordinaten
  int r;                         // Radius
}
```

Der gewählte Name 'Kreis' für die Klasse ist beliebig. Gemäss Konvention beginnen Klassennamen zur Unterscheidung von Variablen-Namen mit einem Grossbuchstaben.

Erzeugung von Objekten:

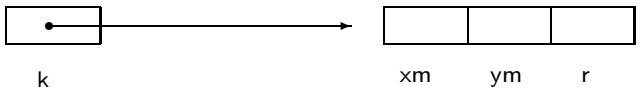
Die Erzeugung eines Objektes erfolgt (wie bei Arrays) mit dem 'new'-Operator:

```
Kreis k;                         // Variable fuer ein Kreis−Objekt
k = new Kreis();                 // 'new'−Operator
```

Im ersten Statement wird eine Variable für das Objekt definiert. Dabei wird der Name der Klasse als Datentyp angegeben. Diese Variable ist aber (wie bei Arrays) nur eine Referenzvariable, welche die Adresse eines Objektes aufnehmen kann.

Das Objekt selber wird mit dem Operator 'new' erzeugt. Dabei werden die Datenkomponenten des Objektes im Speicher angelegt, und die Anfangsadresse des Objektes wird in der Variablen k abgespeichert. Die runden Klammern sind vorgesehen für Parameter im Zusammenhang mit Konstruktoren (siehe unten).

Die Situation kann graphisch folgendermassen dargestellt werden:



Die beiden Statements zur Erzeugung eines Objektes können auch zusammengefasst werden:

```
Kreis k = new Kreis();
```

Verwendung der Datenkomponenten:

Die Datenkomponenten des Objektes werden mit der Variablen k und dem Punkt-Operator angesprochen:

```
k.xm = 200;
k.ym = 100;
k.r = 20;
```

Initialwerte von Datenkomponenten

Die Datenkomponenten eines Objektes werden (im Gegensatz zu lokalen Variablen von Methoden) bei der Erzeugung des Objektes automatisch initialisiert, numerische Daten auf 0, Boolesche Werte auf *false*.

Vorteile der Verwendung von Objekten

Die Zusammenfassung von Datenkomponenten zu einem Objekt bringt folgende Vorteile:

1. Erhöhung der Übersicht, v.a. wenn mehrere Objekte 'Kreise' vorhanden sind. Durch den Punktoperator ist die Eindeutigkeit von Komponenten gewährleistet:

   ```
   inkreis.xm = 100;
   umkreis.xm = 200;
   ```

   Zudem können dieselben Variablen-Namen auch in anderen Objekten oder als einzelne Variablen verwendet werden.

2. Vereinfachte Übergabe von Daten als Parameter an Methoden, z.B.

   ```
   void zeichneKreis(Graphics g, Kreis k)
   { g.drawOval(k.xm−k.r, k.ym−r, 2∗k.r, 2∗k.r);
   }
   ```

Das Applet 'LichtSignal'

Das folgende Applet zeichnet ein Lichtsignal mit drei Lampen (Kreise).
Dazu verwendet es die Klasse 'Kreis'.

```
// _____ Lichtsignal mit 3 Lampen (Kreise) _____
import java.applet.*;
import java.awt.*;

class Kreis                                          // Klasse fuer Kreise
{  int xm, ym;                                       // Mittelpunkt−Koordinaten
   int r;                                            // Radius
}

public class LichtSignal extends Applet
{
   void zeichneKreis(Graphics g, Kreis k)
   {  g.fillOval(k.xm−k.r, k.ym−k.r, 2*k.r, 2*k.r);
   }

   public void init()
   {  setBackground(Color.blue);                     // Farben setzen
   }

   public void paint(Graphics g)
   {  final int left = 100;
      final int top = 100;
      final int w = 100;                             // Breite (Gehaeuse)
      final int h = 300;                             // Hoehe
      final int r = w/2 − 10;                        // Lampen−Radius
      int d = 2*r + 15;                              // Abstand der Lampen
      Kreis k1 = new Kreis();                        // Lampen
      Kreis k2 = new Kreis();
      Kreis k3 = new Kreis();
      k1.r = r;  k2.r = r;  k3.r = r;
      k1.xm = left + w/2;
      k2.xm = k1.xm;  k3.xm = k1.xm;
      k2.ym = top + h/2;                             // mittlere Lampe
      k1.ym = k2.ym − d;  k3.ym = k2.ym + d;
      g.setColor(Color.black);
      g.fillRect(left, top, width, height);          // Gehaeuse
      g.setColor(Color.gray);
      zeichneKreis(g, k1);  zeichneKreis(g, k2);     // off
      g.setColor(Color.green);
      zeichneKreis(g, k3);                           // on
   }
}
```

Aufteilung von Klassen in Files

Im oberen Applet 'LichtSignal' ist die benötigte Klasse 'Kreis' direkt im File des Applets, vor der Klasse 'Olympia' definiert.

Besser ist es, wenn jede Klasse in einem eigenen File mit dem zugehörigen Namen 'xxxx.java' (xxxx=Klassen-Name) definiert und separat compiliert wird. Dann kann sie mit dem Attribut 'public' als öffentlich erklärt werden, was bedeutet, dass sie in jedem anderen Java-Programm verwendet werden kann:

```
public class Kreis                    // oeffentliche Klasse
{
}
```

Bemerkung:

Ein File xxxx.java kann mehrere Klassen enthalten, jedoch darf nur die Klasse mit dem Namen des Files ('xxxx') als public definiert werden. Bei der Compilation entsteht für jede Klasse des Files ein separates .class-File.

△ **Uebung:** Lösen Sie die Aufgabe 6.11.1

6.3 Referenzvariablen und Referenztypen

Wir führen in diesem Paragraphen einige technische Aspekte im Zusammenhang mit Objekten ein, die wichtig sind für bestimmte Operationen mit Objekten.

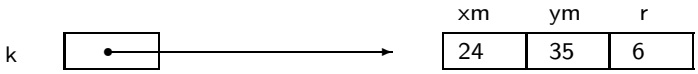
Wie wir wissen, ist eine Klasse, z.B. die Klasse *Kreis* von oben eine Vorlage für eine Menge von gleichartigen Objekten. Technisch gesehen, definiert eine Klasse einen *Datentyp* (wie 'int', 'double' usw.) der zur Definition von Variablen verwendet werden kann:

```
Kreis  k, inkreis, umkreis;
```

Dies sind aber noch keine Objekte, sondern nur sogenannte *Referenzvariablen*, welche die Adresse eines Objektes aufnehmen können. Objekte müssen mit dem 'new'-Operator erzeugt werden:

```
k = new Kreis();
k.xm = 24; k.ym = 35; k.r = 6;
```

Der Operator 'new' legt das Objekt im Speicher an und gibt die Adresse des Objektes als Resultat zurück. Die Adresse wird in der Variablen 'k' abgespeichert.



Wir kennen dieses Prinzip schon von den Array-Variablen. Es hat wichtige Konsequenzen bei Zuweisungen und Vergleichen von Referenzvariablen, sowie bei der Verwendung als Parameter.

Referenztypen

Die von Klassen definierten Datentypen für Referenzvariablen nennt man *Referenztypen*.

Die zugehörigen Referenzvariablen enthalten neben der Adresse eines Objektes weitere Informationen, die den Datentyp des Objektes, das sie referenzieren, betreffen. Diese Zusatzinformationen werden später, im Zusammenhang mit der Vererbung wichtig.

Operationen für Referenzvariablen

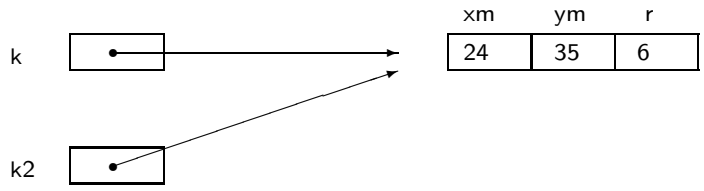
Für Referenzvariablen stehen die bekannten Operationen zur Verfügung: Zuweisungen, Vergleiche, Verwendung als Parameter von Methoden.

1. Zuweisungen

   Für Referenzvariablen desselben Datentyps steht der Zuweisungsoperator '=' zur Verfügung:

   ```
   Kreis  k2;
   k2 = k;
   ```

   Dabei wird jedoch nur die betreffende Adresse kopiert, nicht das Objekt selber. Folglich referenzieren nach dieser Zuweisung beide Variablen dasselbe Objekt:

   Dies kann in gewissen Situationen erwünscht sein, z.B. im Zusammenhang mit Arrays von Objekten, siehe unten.

   Eine *volle* Kopie des Kreises muss folgendermassen erzeugt werden:

```
k2 = new Kreis();
k2.xm = k.xm;
k2.ym = k.ym;
k2.r = k.r;
```

Jetzt referenzieren die Variablen k und k2 wirklich zwei verschiedene Objekte im Speicher (die im Moment die gleichen Werte enthalten).

2. Vergleiche

   Referenzvariablen desselben Datentyps können mit dem Vergleichsoperator '==' verglichen werden:

   ```
   if ( k == k2 ) ...
   ```

   Dabei werden jedoch nur die Adressen verglichen, nicht die Daten der Objekte. Für den Vergleich der Daten müssen Methoden geschrieben werden (siehe unten).

3. Der Wert 'null'

   Einer Referenzvariablen für ein Objekt kann der Wert 'null' zugewiesen werden, um anzugeben, dass sie kein Objekt referenziert:

   ```
   k = null;
   if ( k == null ) ...
   ```

4. Verwendung als Parameter

   In den oberen Beispielen haben wir bereits Referenzvariablen von Objekten als Parameter einer Methode verwendet:

   ```
   void zeichneKreis(Graphics g, Kreis k)
   {
   }
   ```

   Beim Aufruf der Methode werden entsprechende Referenzvariablen übergeben, welche (wie alle Parameter) auf den Stack kopiert werden. Von den Objekten selber werden *keine* Kopien gemacht, sodass die Methode direkt mit dem Objekt der aufrufenden Methode arbeitet.

   Dies ist ein wesentlicher Unterschied zu Parametern der elementaren Datentypen 'int', 'double', usw., bei welchen die Methode mit Kopien auf dem Stack arbeitet.

   ▶ *Merke:*

Wenn in einer Methode die Datenkomponenten eines übergebenen Objektes verändert werden, beziehen sich diese Aenderungen direkt auf das Objekt im aufrufenden Programm und bleiben damit nach dem Ende der Methode bestehen.

Diesen Sachverhalt haben wir schon bei Arrays kennen gelernt.

5. Objekte als Resultate

   Objekte können auch als Resultate von Funktionen in Erscheinung treten. Dabei wird das Objekt in der betreffenden Funktion erzeugt und am Ende der Funktion wird eine Referenzvariable des Objektes als Resultat zurückgegeben.

   Die folgende Funktion 'createCopy' erstellt eine Kopie eines Kreises. Als Resultat-Typ der Funktion wird die Klasse 'Kreis' angegeben:

```
Kreis createCopy(Kreis k);
{ Kreis tmp = new Kreis();
  tmp.xm = k.xm; tmp.ym = k.ym; tmp.r = k.r;
  return tmp;
}
```

   Aufruf der Funktion:

```
Kreis c;
c = createCopy(k);
```

   Dabei ist k das Kreis-Objekt von oben. Man beachte, dass bei der Definition von 'c' *kein* 'new'-Aufruf erforderlich ist, da dieser in der Funktion 'createCopy' erfolgt.

Der Stack und der Heap

Lokale Variablen und Parameter von Methoden werden in einem speziellen Bereich des Speichers angelegt, im sogenannten *Stack* (Stapel). Der Stack ist speziell für Methodenaufrufe organisiert: Bei jedem Aufruf einer Methode wird im Stack ein Bereich für die Methode angelegt und nach dem Ende der Methode wieder freigegeben.

Im Gegensatz dazu werden Datenkomponenten von Objekten in einem anderen Speicherbereich, dem sogenannten *Heap* (Haufen) angelegt. Die Daten auf dem Heap werden beim Ende einer Methode *nicht* freigegeben, d.h. sie bleiben erhalten.

Dadurch ist es möglich, in einer Methode ein Objekt zu erzeugen und dieses als Resultat zurückzugeben.

Arrays als Resultate von Funktionen

Wie normale Objekte können auch Arrays als Resultate von Funktionen vorkommen. Die folgende Funktion 'createArray' erzeugt einen 'int'-Array und gibt ihn als Resultat zurück. Die gewünschte Anzahl Elemente und der Initialwert für die Elemente werden als Parameter übergeben:

```
int[ ] createArray(int nElemente, int initWert)
{ int [ ] a = new int[nElemente];
  for ( int i=0; i < a.length; i++ )
    a[i] = initWert;
  return a;
}
```

Aufruf der Funktion:

```
int[ ] w;
w = createArray(10, 0);
```

Man beachte, dass im aufrufenden Programm keine Elemente für den Array 'w' erzeugt werden, da dies in der Funktion 'createArray' erfolgt.

Löschen von Objekten

Java bietet *keine* Möglichkeit, erzeugte Objekte wieder zu löschen, wenn sie nicht mehr gebraucht werden. Diese Aufgabe übernimmt der *Garbage Collector* von Java.

Garbage Collector

Der Garbage Collector läuft periodisch im Hintergrund und prüft, ob Objekte existieren, die nicht mehr verwendet werden können, da keine Referenzvariable ihre Adresse enthält. Wenn er solche Objekte findet, löscht er sie, sodass der Speicher wieder frei ist.

Wenn ein Objekt nicht mehr gebraucht wird, kann man folglich einfach alle Referenzvariablen, die das Objekt referenzieren, auf den Wert *null* setzen. Dann wird das Objekt beim nächsten Start des Garbage Collectors gelöscht.

6.4 Objekte mit Daten und Methoden

Der eigentliche Ursprung der objektorientierten Programmierung war die Einführung von Methoden, die an ein Objekt gebunden sind. Diese Methoden werden direkt in der zugehörigen Klasse definiert.

Sie werden mit einem Objekt der Klasse aufgerufen und operieren auf den Datenkomponenten dieses Objektes.

Beispiel: Punkte in der mathematischen xy-Ebene

Die Punkte werden durch 'double'-Koordinaten x und y in der unendlichen Ebene beschrieben, nicht durch Bildschirm-Koordinaten. Mit der Darstellung auf dem Bildschirm beschäftigen wir uns im Moment nicht.

```
public class Punkt2d
{    double x, y;                              // Datenkomponenten

     double distance()                         // Methoden
     { return Math.sqrt(x*x + y*y);
     }

     void translate(double dx, double dy)
     { x += dx;
       y += dy;
     }
}
```

Objekt-Methoden:

Die Methoden eines Objektes werden mit einem Objekt p der Klasse und dem Punktoperator aufgerufen (wie die Methoden des Graphics Objektes g):

```
double d = p.distance();                       // Methodenaufrufe
p.translate(4.2, 3.5);
```

Dabei ist p ein Objekt der Klasse, welches mit 'new' erzeugt wurde. Die Methoden können die Datenkomponenten x und y des Objektes, mit dem sie aufgerufen werden, ohne Punktoperator ansprechen (lesen, verändern):

- Die Methode *distance* berechnet den Abstand des Punktes vom Nullpunkt nach der Formel von Pythagoras $d = \sqrt{(x^2 + y^2)}$.
- Die Methode *translate* verschiebt den Punkt um eine Strecke dx in x-Richtung und um dy in y-Richtung.

Der Parameter 'this'

Technisch gesehen erhält jede Methode eines Objektes das Objekt, mit dem sie aufgerufen wird, als Parameter mit dem festen Namen 'this'. In der Methode können die Datenkomponenten dieses Objektes mit oder ohne 'this' angespochen werden:

```
this.x = ... ;                    // x−Koordinate des Objektes
```

oder äquivalent:

```
x = ... ;                         // x−Koordinate des Objektes
```

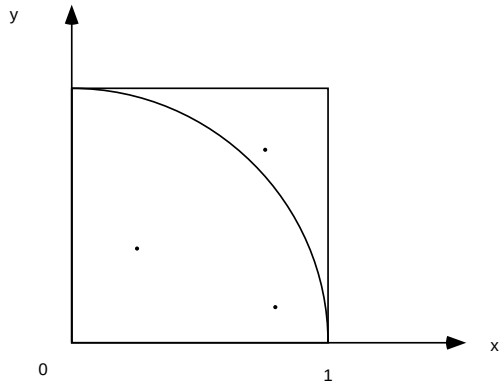
▶ *Merke:*

In jeder Methode eines Objektes ist 'this' eine Referenzvariable für das Objekt, mit dem die Methode aufgerufen wurde. In der Methode können die Datenkomponenten und Methoden des Objektes 'this' jedoch ohne 'this.' angesprochen werden.

Anwendung der Klasse 'Punkt'

Bestimmung der Zahl π mit Zufallsregen

Wir lassen n Regentropfen zufällig auf das Einheitsquadrat fallen. Dabei bestimmen wir die Anzahl n_1 der Tropfen, die im Viertelskreis liegen.



Wenn die Regentropfen gleichmässig über das Quadrat verteilt sind, entspricht das Verhältnis $n_1 : n$ (im Rahmen von statistischen Schwankungen) dem Verhältnis der Viertelskreisfläche zur Quadratfläche, also

$$n_1 : n \approx \frac{\pi}{4} : 1$$

Damit erhalten wir eine Abschätzung für π :

$$\pi \approx 4 \cdot \frac{n_1}{n}$$

Für die Regentropfen wird ein Objekt p der Klasse 'Punkt2d' erzeugt.

```
// ___________ Pi−Berechnung mit Zufallsregen ____________________________
public class Regen
{
  public static void main(String[ ] args)
  { int n, n1;
    Punkt2d p = new Punkt2d();                  // Regentropfen als Objekt
    InOut.print("Anzahl Regentropfen: ");
    n = InOut.getInt();
    n1 = 0;
    for (int i=0; i < n; i++)
    { p.x = Math.random();                      // Zufallskoordinaten
      p.y = Math.random();
      if ( p.distance() <= 1 )                  // Punkt im Viertelskreis ?
        n1++;
    }
    InOut.print("Schaetzung fuer Pi: " );
    InOut.println(4.0 * n1 / n, 1, 4);
  }
}
```

△ **Uebung:** Lösen Sie die Aufgabe 6.11.2

Lokale Variablen versus Datenkomponenten

Bei der Definition einer Klasse sind zwei Arten von Variablen zu unterscheiden:

- Datenkomponenten

  Die Variablen, die in der Klasse ausserhalb der Methoden definiert sind, sind Datenkomponenten für die Objekte der Klasse. Jedes Objekt erhält einen Satz dieser Variablen.

  Die Datenkomponenten werden bei der Erzeugung des Objektes angelegt und existieren solange wie das betreffende Objekt.

  Datenkomponenten eines Objektes werden bei der Erzeugung des Objektes automatisch initialisiert (siehe Seite 90).

- Lokale Variablen von Methoden

  Variablen, die in einer Methode definiert sind, sind *lokale* Variablen der Methode. Diese können nur innerhalb der betreffenden Methode verwendet werden. Sie werden bei jedem Aufruf der Methode in einem speziellen Speicherbereich, dem *Stack* erzeugt und am Ende wieder gelöscht.

  Lokale Variablen werden *nicht* automatisch initialisiert.

Lokale Überdeckungen von Variablen

Lokale Variablen und Parameter einer Methode dürfen dieselben Namen wie Datenkomponenten der Klasse aufweisen. In der Methode überdecken in diesem Fall die lokalen Variablen bzw. Parameter die schon definierten Variablen.

Dadurch können Methoden unabhängig vom Umfeld entwickelt werden (Lokalitätsprinzip).

Überlagerung von Methoden (Overloading)

Verschiedene Methoden einer Klasse können denselben Namen haben, wenn sie sich aufgrund der Parameter unterscheiden: verschiedene Anzahl Parameter, oder Parameter mit unterschiedlichen Datentypen. (Die Namen der Parameter, sowie der Datentyp des Resultates sind dabei *nicht* massgebend).

Die Bedingung ist die, dass der Compiler beim Aufruf der Methode aufgrund der übergebenen Parameter eindeutig entscheiden kann, welche Methode gewünscht wird.

In dieser Situation spricht man von überlagerten Methoden (overloaded methods).

Wir kennen dieses Prinzip schon von den Methoden ‘print’ und ‘println’:

```
System.out.print("Resultat: ");          // print fuer Strings
System.out.print(n);                     // print fuer int−Werte
```

Hier werden zwei verschiedene Methoden ‘print’ des Objektes ‘System.out’ aufgerufen, die sich durch die Parameter unterscheiden. ‘System.out’ ist ein Objekt der Klasse ‘PrintStream’, welches automatisch erzeugt wird und in jedem Programm zur Verfügung steht.

6.5 Konstruktoren

Ein Konstruktor einer Klasse ist eine Methode, die bei der Erzeugung eines Objektes (mittels 'new') automatisch aufgerufen wird. Zweck eines Konstruktors: Initialisierung des Objektes.

Beispiel:

Konstruktor für die Klasse 'Punkt2d', dem man Anfangswerte für die x- und y-Koordinaten des Punktes als Parameter mitgeben kann.

Definition eines Konstruktors

Ein Konstruktor einer Klasse wird wie eine normale Methode (mit oder ohne Parameter) definiert, mit den folgenden Abweichungen:

- Der Methoden-Name ist gleich dem Klassen-Namen.
- Die Methode hat keinen Resultat-Typ (keine Angabe, nicht 'void')

Als Beispiel betrachten wir den oben erwähnten Konstruktor für die Klasse 'Punkt2d':

```
public class Punkt2d
{ private double x, y;
  public Punkt2d(double xCoord,              // Konstruktor
                 double yCoord)
  { x = xCoord;                              // Datenkomponenten initialisieren
    y = yCoord;
  }
  ...
}
```

Der Konstruktor überträgt die Parameter *xCoord* und *yCoord* in die Datenkomponenten x und y des Objektes. Man beachte, dass die Parameter *xCoord* und *yCoord* (wie alle lokalen Variablen) am Ende der Methode gelöscht werden, die Datenkomponenten x und y des Objektes nicht.

Verwendung des Konstruktors

Bei der Erzeugung eines Objektes der Klasse müssen jetzt die Parameter des Konstruktors übergeben werden:

```
Punkt2d p = new Punkt2d(3.4, −2.8);
```

Bemerkungen:

1. *Parameter-Namen*

   Die Namen der Parameter müssen nicht unbedingt anders gewählt werden als die der Datenkomponenten. Bei gleichen Namen müssen im Konstruktor die Datenkomponenten des Objektes mit 'this' qualifiziert angesprochen werden um Eindeutigkeit zu erreichen:

```
public Punkt2d(double x, double y)      // Konstruktor
{ this.x = x;                           // Datenkomponenten initialisieren
  this.y = y;
}
```

2. *Klassen mit mehreren Konstruktoren*

   In einer Klasse können mehrere Konstruktoren definiert werden, wenn sie sich im Parameter-Profil unterscheiden, gemäss dem Overload-Prinzip.

3. *Leerer Konstruktor*

   Wenn in einer Klasse ein oder mehrere Konstruktoren definiert sind, ist der Standard-Konstruktor von Java (leerer Konstruktor ohne Parameter) ausser Kraft gesetzt.

   Im oberen Beispiel *müssen* folglich bei der Erzeugung eines Punktes die Koordinaten angegeben werden, ausser man definiert noch einen weiteren Konstruktor ohne Parameter, z.B. einen leeren Konstruktor:

```
public Punkt2d()                        // leerer Konstruktor
{  }
```

△ **Uebung:** Lösen Sie die Aufgabe 6.11.3

6.6 Dateneinkapselung

Ein wichtiges Prinzip der objektorientierten Programmierung ist die *Dateneinkapselung*. Darunter versteht man das Prinzip, dass die Datenkomponenten eines Objektes ausserhalb der betreffenden Klasse nicht direkt, sondern ausschliesslich mit Methoden des Objektes (Zugriffsmethoden) angesprochen werden.

Dies ermöglicht eine gesicherte *Datenintegrität*, indem die Zugriffsmethoden prüfen, dass nur gültige Werte abgespeichert werden.

Die Dateneinkapselung kann erzwungen werden, indem die Datenkomponenten mit dem Schlüsselwort 'private' versehen werden. *private*

Das Attribut 'private' bewirkt, dass die betreffende Datenkomponente nur von den Methoden dieser Klasse angesprochen werden kann.

Zum Setzen und Lesen der Komponente ausserhalb der Klasse müssen entsprechende Methoden zur Verfügung gestellt werden.

Beispiel: Tages-Zeiten

```
public class Tageszeit
{ private int hour, minute;                    // private Datenkomponenten
  private double seconds;

  // ———————— Zugriffsmethoden ————————————————
  boolean setVal(int h, int m, double s)       // Werte setzen
  { if ( h  < 0  || h  > 23   ||               // Werte pruefen
         m  < 0  || m  > 59   ||
         s  < 0  || s  >= 60 )
       return false;                           // Fehler
    hour = h; minute = m; seconds = s;         // Werte abspeichern
    return true;
  }
  int getHour() { return hour; }               // Werte lesen
  int getMinute() { return minute; }
  double getSeconds() { return seconds; }

  // ———————— weitere Methoden ————————————————
  ...
}
```

Attribute für Zugriffssteuerung

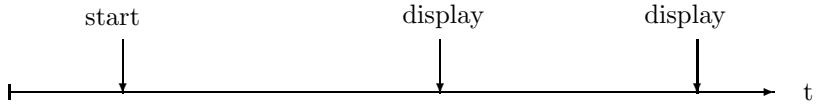
Zur Festlegung des Zugriffes auf Datenkomponenten und Methoden einer Klasse stehen die folgenden Attribute zur Verfügung:

Attribut	Bedeutung
private	Die Datenkomponente bzw. Methode kann nur von Methoden derselben Klasse verwendet werden.
public	Keine Einschränkung, Verwendung von allen Methoden beliebiger Klassen zugelassen
keine Angabe	Verwendung von allen Methoden desselben Packages zugelassen
protected	Verwendung von allen Methoden desselben Packages und von Methoden von Subklassen zugelassen.

Die letzten drei Attribute unterscheiden sich nur bei Verwendung von Packages oder Subklassen (Klassenerweiterungen). Diese beiden Konzepte werden erst später eingeführt. Wir verwenden daher bis auf weiteres die ersten beiden Attribute.

Beispiel zur Dateneinkapselung: Stop-Uhren

Für Zeitmessungen mit einem Computer führen wir Objekte Stop-Uhren ein, die man starten (Methode *start*) und dann zu beliebigen Zeitpunkten ablesen kann (Methode *display*):



Hilfsmittel: Abfrage der System-Zeit

Die Methode 'System.currentTimeMillis' liefert die Anzahl Millisekunden seit dem 1. Jan. 1970 (Datentyp 'long'):

```
long milliSek = System.currentTimeMillis();
```

Definition der Klasse 'Clocks':

```
// ———————— Stop−Uhren ————————————————————————————————————
public class Clocks
{
  private long startZeit;
  private boolean gestartet;                          // Status

  public void start()
  { if ( !gestartet )                                 // Uhr noch nicht gestartet ?
    { startZeit = System.currentTimeMillis();
      gestartet = true;
    }
  }

  public double display()                             // Anzeige in Sekunden
  { if ( gestartet )
      return ( System.currentTimeMillis() −
               startZeit) * 0.001;
    else
      return 0;
  }
}
```

Man beachte, dass die Datenkomponenten bei der Erzeugung eines Objektes automatisch auf 0 bzw. *false* initialisiert werden.

Anwendung der Klasse:

```
Clocks uhr = new Clocks();
uhr.start();
...
System.out.print("Laufzeit: ");
System.out.println(uhr.display());
```

△ **Uebung:** Lösen Sie die Aufgabe 6.11.4 (verbesserte Version der Klasse *Clocks*, für Uhren die man stoppen und wieder weiter laufen lassen kann).

6.7 Beispiele von Klassen von Java

Wir führen in diesem Paragraphen einige wichtige Klassen der Java-Library ein.

Die Klasse 'Color'

Eine Farbe wird beschrieben durch ihre Rot-, Grün- und Blau-Anteile, die sogenannten RGB-Werte. Dies sind ganze Zahlen im Bereich 0 .. 255.

Diese drei Werte werden zu einem Objekt der Klasse 'Color' zusammengefasst.

Bisher haben wir nur die vordefinierten Farben verwendet:

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

Color.blue ist ein vordefiniertes Objekt der Klasse 'Color'.

Für die Verwendung von Farben mit beliebigen RGB-Werten in einem Applet ist folgendermassen vorzugehen:

1. Erzeugung eines Objektes der Klasse 'Color'. Dabei werden dem Konstruktor die gewünschten RGB-Werte übergeben:

   ```
   Color dunkelRot = new Color(100, 0, 0);
   ```

2. Aktivierung der Farbe mit der Methode 'setColor' des Graphics Objektes 'g'. Dabei wird das 'Color'-Objekt als Parameter übergeben:

   ```
   g.setColor(dunkelRot);
   ```

Auf dem Bildschirm erscheint die Farbe je nach Einstellung des Video-Modes (256 Farben, High-Color, True-Color) nur näherungsweise mit den angegebenen RGB-Werten. Bei Bedarf wird auch *Dithering* verwendet (Erzeugung von Mischfarben mittels Mustern aus Bildpunkten verschiedener Farben).

Die Klasse 'Font'

Bei Ausgaben von Text in einem Applet mittels 'g.drawString' werden Default-Werte für die Schrift und deren Attribute verwendet. Diese Einstellungen können geändert werden. Dabei kommen (wie bei Farben) wieder Objekte zum Einsatz, welche die Attribute einer Schrift zusammenfassen:

– Name der Schrift

- Grösse in Points
- Stilangaben (normal, fett, kursiv)

Für die Verwendung von Schriften in einem Applet ist folgendermassen vorzugehen:

1. Erzeugung eines Objektes der Klasse 'Font'. Dabei werden dem Konstruktor die gewünschten Attribute der Schrift übergeben:

```
String name = "SansSerif";
final int size = 32;
int style = Font.PLAIN;
Font font = new Font(name, style, size);
```

2. Aktivierung der Schrift mit der Methode 'setFont' des Graphics Objektes 'g'. Dabei wird das 'Font'-Objekt als Parameter übergeben:

```
g.setFont(font);
```

Anschliessende Text-Ausgaben mittels 'g.drawString' verwenden die aktivierte Schrift und deren Attribute.

Zur Verfügung stehende Schriften:

```
"Dialog", "SansSerif", "Serif", "Monospaced", "DialogInput"
```

Styles:

```
Font.PLAIN, Font.BOLD, Font.ITALIC
```

Die Styles sind 'int'-Konstanten. Sie können mittels Addition kombiniert werden:

```
Font font = new Font("SansSerif", Font.BOLD + Font.ITALIC, 32);
```

Die Klasse 'Date'

Für Kalenderdaten steht die Klasse 'Date' der Java-Library zur Verfügung. Die Klasse gehört zum Package 'java.util', welches für die Verwendung der Klasse importiert werden muss:

```
import java.util.*;
```

Ein Objekt der Klasse 'Date' stellt ein vollständiges Kalenderdatum inklusive Tageszeit dar.

- Der Konstruktor (ohne Parameter) initialisiert die Datenkomponenten mit dem aktuellen Datum und der momentanen Zeit.

```
Date date1 = new Date();
```

- Die Datenkomponenten sind *private* und werden mit Methoden abgefragt:

```
int jahr      = date1.getYear();
int monat     = date1.getMonth();   // 0 = Jan, .. , 11 = Dez
int tag       = date1.getDate();    // 1 .. 31
int wochentag = date1.getDay();     // 0 = So, .. , 6 = Sa
```

Für Veränderungen der einzelnen Komponenten eines Datums stehen entsprechende 'set'-Methoden zur Verfügung, z.B.

```
date1.setMonth(5);                  // Monat auf 5 (Juni) setzen
```

Bemerkung:

Neben der Klasse 'Date' wurde unterdessen die Klasse, 'GregorianCalendar' eingeführt, welche erweiterte Möglichkeiten enthält (siehe Dokumentation der Java Library).

△ **Uebung:** Lösen Sie die Aufgabe 6.11.5

6.8 Arrays von Objekten

Bei der Definition eines Arrays von Objekten muss beachtet werden, dass Objekte durch Referenzvariablen repräsentiert werden. Infolgedessen ist ein Array von Objekten ein Array von Referenzvariablen.

Für die Definition eines Arrays von Objekten wird anstelle einer einzelnen Referenzvariablen

```
Punkt2d p;
```

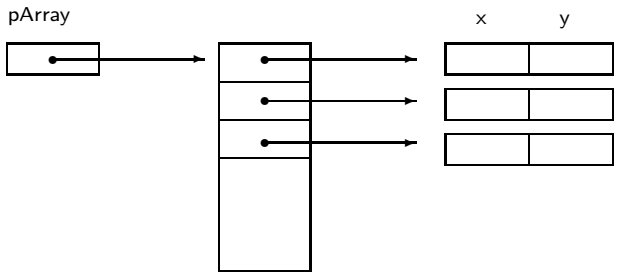
ein *Array von Referenzvariablen* definiert:

```
Punkt2d[ ] pArray = new Punkt2d[100];
```

Dies ergibt 100 Referenzvariablen für Objekte der Klasse 'Punkt2d'. Die Objekte müssen in einem zweiten Schritt in einer Schleife erzeugt werden:

```
for (int i=0; i < pArray.length; i++)
  pArray[i] = new Punkt2d();
```

Damit ist die folgende Datenstruktur erzeugt worden:



Verwendung der Objekte:

```
pArray[0].x = 10.2;
pArray[0].y = −3.9;
double abstand = pArray[0].distance();
```

Arrays von Farben

Arrays von 'Color'-Objekten kommen in Applets zum Einsatz. Das folgende Applet erzeugt einen Array mit $n = 100$ Rotstufen und zeichnet anschliessend n horizontale Strecken in den erzeugten Farben.

```
import java.applet.*;
import java.awt.*;
public class RotStufen extends Applet
{ int n = 100;                                  // Anzahl Elemente
  int rMax = 255;                               // maximale Stufe
  Color[ ] farben = new Color[n];               // Array von Referenzvariablen

  public void init()
  { for (int i=0; i < n; i++)                   // Color−Objekte erzeugen
    farben[i] = new Color( i * rMax / (n−1), 0, 0);
  }

  public void paint(Graphics g)
  { for (int i=0; i < farben.length; i++)
    { g.setColor(farben[i]);
      g.drawLine(0, i, 100, i);
    }
  }
}
```

Ein Array von Farben kann auch nach dem Prinzip erzeugt werden, dass bei der Definition eine Werteliste für die Elemente angegeben wird. Die Werte sind hier 'Color'-Objekte, die direkt mittels 'new Color(r, g, b)' erzeugt werden:

```
Color[ ] colors = {
        new Color(0, 0, 0),              // black
        new Color(0, 0, 128),            // blue
        new Color(0, 128, 0),            // green
        new Color(0, 128, 128),          // cyan
        new Color(128, 0, 0),            // red
        new Color(128, 0, 128),          // magenta
        new Color(128, 128, 0),          // brown
        new Color(192, 192, 192),        // white
        new Color(128, 128, 128),        // gray
        new Color(0, 0, 255),            // light_blue
        new Color(0, 255, 0),            // light_green
        new Color(0, 255, 255),          // light_cyan
        new Color(255, 0, 0),            // light_red
        new Color(255, 0, 255),          // light_magenta
        new Color(255, 255, 0),          // yellow
        new Color(255, 255, 255) };      // bright_white
```

Dies sind die altbekannten 16 Standard-Farben von DOS-VGA.

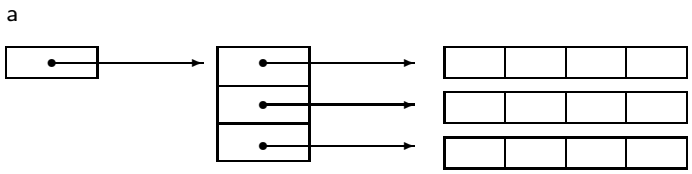
Arrays von Arrays

In Java sind Arrays bekanntlich Objekte. Damit sind die in Kap. 5 eingeführten Arrays von Arrays (Matrizen) ebenfalls Arrays von Objekten.

Beispiel: Matrix mit 3 Zeilen und 4 Spalten

```
double[ ][ ]  a = new double[3][4];          // 3 x 4 Matrix
```

Dies ist ein Array aus 3 Elementen a[0], a[1] und a[2]. Jedes Element a[i] ist selber wieder eine Referenzvariable für einen 'double'-Array. Folglich ist a ein Array von 3 Referenzvariablen:



Dieses Konzept hat den Vorteil, dass nicht alle Zeilen gleich lang sein müssen, z.B.:

```
double[ ][ ]  b = { { 13, 15 },
                    { 12, 18, 33 },
                    { 48, 55, 33, 56 } };
```

Weiter kann man die Elemente auch in mehreren Stufen erzeugen:

```
double[ ][ ] c;                    // Referenzvariable fuer einen Array von Arrays
c = new double[3][ ]               // Erzeugung des Arrays von 3 Referenzvariablen
for (int i=0; i < 3; i++)
  c[i] = new double[4];            // Elemente erzeugen
```

Dieser Array ist äquivalent zu dem oben definierten Array a.

6.9 Statische Elemente einer Klasse

Wir führen in diesem Paragraphen statische Methoden und Datenkomponenten einer Klasse ein. Diese sind nicht an ein Objekt der Klasse gebunden, sie existieren ohne Objekte, direkt in der Klasse.

Statische Methoden

Bei der Definition einer Methode einer Klasse kann das Attribut 'static' verwendet werden:

```
public static void xxx( ... )
{  ...  }
```

Dies bewirkt, dass die Methode nicht zu einem Objekt gehört, sondern *ohne* Objekt, direkt mit dem *Klassennamen* als Präfix aufgerufen wird. Infolgedessen hat sie jedoch *keine* Daten eines Objektes automatisch zur Verfügung (sondern nur via Parameter).

Beispiele:

1. Die mathematischen Funktionen 'Math.sqrt', 'Math.random' usw.
2. Statische Methoden sind auch sinnvoll, wenn sie zwei gleichberechtigte Objekte verarbeiten. Dies ist z.B. der Fall bei der folgenden Methode 'mittelPunkt' für die Klasse 'Punkt2d', welche den Mittelpunkt von zwei Punkten berechnet:

```
public static Punkt2d mittelPunkt(Punkt2d a, Punkt2d b)
{ Punkt2d mp = new Punkt2d();
  mp.x = 0.5 * (a.x + b.x);
  mp.y = 0.5 * (a.y + b.y);
  return mp;
}
```

Aufruf:

```
Punkt2d p1 = new Punkt2d(−2.4, 3.8);
Punkt2d p2 = new Punkt2d(1.5, 2.3);
Punkt2d p3 = Punkt2d.mittelPunkt(p1, p2);
```

Statische Variablen

Neben Methoden können auch Variablen und Konstanten, die in einer Klasse ausserhalb von Methoden definiert werden, mit dem Attribut 'static' versehen werden:

```
static int xxx;
```

Dies bedeutet, dass die Variable nicht als Datenkomponente für jedes Objekt separat angelegt wird, sondern nur einmal zu der Klasse.

Beispiele:

1. Die mathematischen Konstanten 'Math.PI' und 'Math.E'.

2. Zähler für die Anzahl Instanzen einer Klasse

```
public class A
{ private static int instanceCounter;      // Anzahl Instanzen
  public A()                               // Konstruktor
  { instanceCounter++;
  }
  public static int getInstanceCounter()
  { return instanceCounter;
  }
  ...
}
```

In der Datenkomponente *instanceCounter* wird die Anzahl erzeugter Instanzen (Objekte) der Klasse nachgeführt.

Da diese Variable statisch ist, existiert sie nur einmal, nicht zu jedem Objekt. Sie wird bei jedem Aufruf des Konstruktors um 1 erhöht. Zur Abfrage des Zählers dient die statische Methode *getInstanceCounter*.

Klassen- und Instanz-Methoden

Statische Methoden nennt man *Klassen*-Methoden im Gegensatz zu *Instanz*-Methoden, die zu Instanzen, d.h. Objekten gehören. Analog unterscheidet man *Klassen*-Variablen und *Instanz*-Variablen.

6.10 Applikationen und Applets

Applikationen und Applets werden ebenfalls als Klassen definiert. Daher können sämtliche Klassenkonzepte auch in Applikationen und Applets verwendet werden.

Eine Applikation ist einfach eine Klasse mit einer Methode 'main'.

▶ *Merke:*

In einer Applikation müssen (im Gegensatz zu einem Applet) alle Datenkomponenten und Methoden der Klasse mit dem Attribut 'static' definiert werden.

Grund:

Der Java Interpreter erzeugt kein Objekt der Klasse sondern ruft direkt die statische Methode 'main' auf.

```
public class X
{
  static int n1, n2;
  static void ausgabe()
  { ... }
  public static void main(String[ ] args)
  { ...
    ausgabe();
  }
}
```

Die Variablen und Konstanten, die ausserhalb der Methoden definiert werden, stehen allen Methoden der Klasse zur Verfügung, im Gegensatz zu den lokalen Variablen der Methoden.

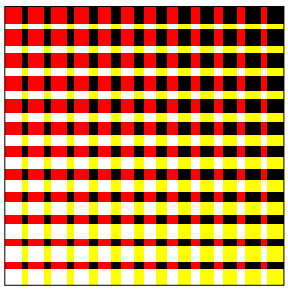
globale Variablen

Die Variablen und Konstanten, die ausserhalb von Methoden definiert sind nennt man aus diesem Grund auch *globale* Variablen/Konstanten. Daten, die nur lokal in einer Methode verwendet werden, sollten als lokale Variablen definiert werden.

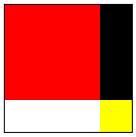
6.11 Uebungen

6.11.1 Farb-Muster nach J. Nänni

Erstellen Sie ein Applet 'FarbMuster', welches das folgende Muster zeichnet:



Das Muster besteht aus 12 x 12 Grundmustern der Form:



Das Grundmuster hat eine Kantenlänge von 24 Pixel.

Die horizontale Unterteilung des Grundmusters variiert in jeder Zeile von links nach rechts, von 18+6, 17+7 usw. Die vertikale Unterteilung variiert analog in jeder Spalte von oben nach unten.

Führen Sie eine Klasse für das Grundmuster ein.

6.11.2 Applet Lichtsignal

Führen Sie in der Klasse 'Kreis' die folgenden Methoden ein:

- Eine Methode zum Zeichnen des Kreises
- Eine Methode zum Verschieben des Kreises in x- und y-Richtung um Verchiebungsstrecken dx und dy.

Modifizieren Sie das Applet 'LichtSignal' von Seite 91, so dass es diese Methoden verwendet.

6.11.3 Die Klasse 'QuadGl'

Eine quadratische Gleichung hat die Form

$$ax^2 + bx + c = 0$$

Wir setzen voraus, dass $a \neq 0$ ist, da es sonst keine quadratische Gleichung ist. Die Lösungen x_1 und x_2 werden folgendermassen berechnet:

1. Berechne die Diskriminante

   $$D = b^2 - 4ac$$

2. Wenn $D < 0$ ist, gibt es keine reelle Lösung, andernfalls werden die Lösungen x_1 und x_2 folgendermassen berechnet:

   $$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$$

Im Falle $D = 0$ fallen die beiden Lösungen zusammen.

Erstellen Sie eine Klasse 'QuadGl' für quadratische Gleichungen.

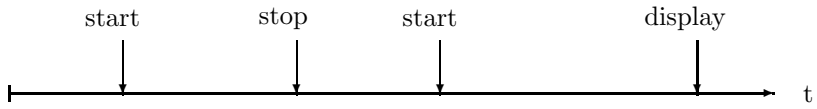
Führen Sie einen Konstruktor ein, dem die Koeffizienten der Gleichung als Parameter übergeben werden. Dieser berechnet gleich die Lösungen und speichert sie in Datenkomponenten ab. Zusätzlich wird die Anzahl Lösungen (0, 1 oder 2) festgehalten.

6.11.4 Erweiterte Stop-Uhren

Erweitern Sie die Klasse *Clocks* von Seite 105, so dass eine Uhr wie eine reale Stopuhr mehrfach gestartet und gestoppt werden kann. Dabei sollen die Laufzeiten von abgeschlossenen Start/Stop-Intervallen gespeichert und bei der Anzeige der Uhr berücksichtigt werden.

Weiter soll eine Uhr auch zurückgesetzt werden können (Methode 'reset').

Ablaufbeispiel:



6.11.5 Eine eigene Klasse für Kalender-Daten

In dieser Übung wird eine eigene Klasse 'Datum' für Kalender-Daten entwickelt, die zeigen soll, wie Datumrechnungen funktionieren. Es geht um die Berechnung des Wochentages eines Datums und der Anzahl Tage zwischen zwei Kalender-Daten.

Die beschriebenen Algorithmen zeigen, dass dies mit elementaren ganzzahligen Rechnungen möglich ist. Die Grundlage ist eine Hilfsmethode 'daysAbs', welche die Anzahl Tage eines Datums seit einem Referenzdatum berechnet. Als Referenzdatum wählen wir den 01.01.0001.

- Daten-Komponenten eines Datums:

  Jahr, Monat, Tag

- Methoden (Algorithmen siehe unten):

  (a) boolean schaltjahr()

  Die Methode bestimmt, ob das Jahr des Datums ein Schaltjahr ist.

  (b) int jahrestag()

  Die Methode berechnet den Jahrestag (1 .. 366) des Datums.

  (c) int daysAbs()

  Anzahl Tage seit 1.1.0001

(d) int wochentag()
Bestimmung des Wochentages, 0=So, 1=Mo, .. , 6=Sa

(e) int daysTo(Datum d)
Anzahl Tage bis zu einem Datum d

- Algorithmen:

(a) Schaltjahr-Definition (Gregorianischer Kalender):
Jahrhundert-Jahre (1900, 2000, ...) sind genau dann Schaltjahre, wenn sie durch 400 teilbar sind. Die übrigen Jahre sind genau dann Schaltjahre, wenn sie durch 4 teilbar sind.

(b) Führen Sie eine Tabelle mit den Tages-Zahlen der Monate ein (Schaltjahre berücksichtigen).

(c) Die gesuchte Anzahl Tage 'daysAbs' wird in 3 Schritten berechnet:

```
k = jahr−1;                          // Anzahl abgeschlossene Jahre
s = k/4 − k/100 + k/400;             // Anzahl Schaltjahre
daysAbs = k * 365 + s + jahrestag;   // gesuchte Anzahl Tage
```

(d) wochentag = Rest von daysAbs modulo 7
Da die Wochentage periodisch mit Periode 7 laufen, genügt eine Probe für ein einziges Datum zur Verifikation der Formel.

(e) Differenz der daysAbs plus 1 (Bsp: 1.1.1999 - 2.1.1999 gibt gemäss Definition 2 Tage)

- Test-Programm:

Input: Datum1, Datum2
Output: Wochentage der Daten und Anzahl Tage von Datum1 bis Datum2

Bemerkung:

Obwohl der Gregorianische Kalender mit der Schaltjahr-Definition erst *nach* unserem Referenzdatum eingeführt wurde (um 1500 herum), liefern die Algorithmen für Kalender-Daten nach dessen Einführung die richtigen Resultate.

Grund: Die Wochentag-Berechnung ist angepasst, und bei Differenzen fällt eine feste Abweichung heraus.

Kapitel 7

Strings

Ein String ist eine Sequenz (Aneinanderreichung) von Characters:

"5210 Windisch"

Strings sind in Java Objekte der Klasse 'String'.

7.1 Erzeugung von String-Objekten

Zur Erzeugung eines String-Objektes kann das für Objekte übliche Verfahren mit einem Konstruktor verwendet werden:

```
String ort = new String("5210 Windisch");
```

String-Literale als Objekte

Tatsächlich ist der 'new'-Operator im obigen Statement nicht nötig, da jedes String-Literal `"abcd"` selber schon ein String-Objekt darstellt, welches vom Compiler automatisch erzeugt wird. Daher kann das obige Statement abgekürzt geschrieben werden:

```
String ort = "5210 Windisch";
```

Aus diesem Grund können auch beim Aufruf von Methoden für String-Parameter direkt String-Literale übergeben werden:

```
System.out.println("5210 Windisch");
```

7.2 String Methoden

Im folgenden führen wir die wichtigsten String-Methoden auf. Für eine vollständige Liste verweisen wir auf die Dokumentation der Klasse 'String' (Package 'java.lang').

Für ein beliebiges String-Objekt 's' stehen die folgenden Methoden zur Verfügung:

charAt: Herauslesen eines Characters aus dem String

```
String s = "5210 Windisch";
char ch = s.charAt(5);
```

Die Methode liefert den Character im String an der angegebenen Position. Dabei hat der erste Character eines Strings die Positions-Nummer 0 (nicht 1), wie in einem Array. Die Variable 'ch' erhält also den Wert 'W'.

Merke:

Die Positions-Nummern der Characters in einem String beginnen bei 0, nicht bei 1.

length: Länge eines Strings bestimmen

```
String s = "5210 Windisch";
int len = s.length();
```

Die Methode liefert die Anzahl Characters des Strings (inkl. Leerzeichen). Die Variable 'len' erhält folglich den Wert 13.

Man beachte, dass 'length' bei Strings eine Methode ist, also mit runden Klammern aufgerufen wird, im Gegensatz zu Arrays, wo 'length' eine Datenkomponente des Objektes ist.

indexOf: Suche nach einem Teilstring

```
String s = "5210 Windisch";
int pos = s.indexOf("Windisch");
```

Die Methode liefert als Resultat die Position im String 's', wo der angegebene Teilstring beginnt. Wenn der Teilstring nicht vorkommt ist das Resultat gleich -1.

Im Beispiel ist das Resultat gleich 5, da die Positions-Nummern bei 0 beginnen.

Neben dieser Methode 'indexOf' steht noch eine zweite zur Verfügung, welche einen weiteren Parameter hat, mit dem man einen Startindex für die Suche im String 's' angeben kann:

```
pos = s.indexOf("Windisch", 4);          // Suche ab Index 4 in s
```

Diese Version ist nützlich für die Suche nach weiteren Vorkommen.

toUpperCase: Umsetzung in Gross-Schrift

```
String s = "5210 Windisch";
String s2 = s.toUpperCase();
```

Die Methode erzeugt einen neuen String mit dem in Gross-Schrift umgewandelten Inhalt von 's' und gibt diesen als Resultat zurück.

substring: Teilstring extrahieren

```
String s = "5210 Windisch";
int von=5;
int bis=8;
String s2 = s.substring(von, bis);       // Substring "Win"
String s3 = s.substring(von);            // bis zum Ende von s
```

Die Methode erzeugt einen neuen String, bestehend aus dem angegebenen Teilstring von s. Der Teilstring beginnt an der Position *von* und endet bei der ersten Version bei $bis - 1$, bei der zweiten beim Ende von s.

Der Character an der Endposition 'bis' gehört also *nicht* mehr zum Teilstring, der Substring s2 besteht in unserem Beispiel folglich aus den Characters "Win". (Die Länge des Teilstrings ist damit gleich der Differenz $bis - von = 3$).

equals: Vergleich zweier Strings auf Gleichheit

```
String s = "Java";
String s2 = "Java 2";
if ( s.equals(s2) ) ...
```

Die Methode vergleicht die Inhalte von 's' und 's2'. Wenn sie vollständig übereinstimmen (gleiche Länge, gleicher Inhalt), ist das Resultat *true* sonst *false*.

equalsIgnoreCase: Vergleich zweier Strings auf Gleichheit

Wie 'equals', aber ohne Berücksichtigung von Gross-/Klein-Schrift

compareTo: Lexikographischer Vergleich zweier Strings

Die Methode vergleicht zwei Strings lexikographisch, d.h. alphabetisch, aufgrund von Character-Vergleichen.

```
String s1 = "Meier";
String s2 = "Mueller";
if ( s1.compareTo(s2) < 0 ) ...
```

Bedeutung des Resultates der Methode:

Resultat	Bedeutung
negativ (d.h. < 0)	s1 < s2
0	s1 == s2
positiv (d.h. > 0)	s1 > s2

Für den Vergleich werden die Strings von links durchlaufen, bis zur ersten Position mit unterschiedlichen Characters. Diese beiden Characters (im Beispiel 'e' und 'u') sind massgebend für das Resultat: 'e' kommt vor 'u', also ist s1 < s2.

Wenn alle gemeinsamen Stellen übereinstimmen (z.B. "Meier" und "Meierhans"), sind die Längen massgebend.

Gleichheit ergibt sich nur bei gleicher Länge und gleichem Inhalt.

Dieses Prinzip führt zu der gewohnten Sortierung von Namen, Ortschaften usw.

△ **Uebung:** Lösen Sie die Aufgabe 7.8.1

7.3 String-Konkatinierung

Mit dem Konkatinierungs-Operator '+' können zwei Strings oder ein String und ein numerischer Wert zu einem neuen String konkatiniert werden.

```
int plz = 5210;
String ort = " Windisch ";
String kanton = "AG";
String s = plz + ort + kanton;
```

Die Operationen werden von links nach rechts ausgeführt. Die Bedingung für eine String-Konkatinierung '+' ist die, dass mindestens einer der beiden Operanden ein String ist. Der andere kann ein String, ein 'char', 'int'- oder 'double'-Wert usw. sein. Wenn nötig, wird einer der Operanden automatisch in String umgewandelt.

Der Konkatinierungsoperator ist v.a. nützlich bei Ausgaben mit 'System.out.println' (siehe Seite 55) oder 'g.drawString'.

7.4 Veränderungen von Strings

Der Inhalt eines Strings kann nach der Erzeugung nicht mehr verändert werden. Wie aus den oben aufgeführten String-Methoden hervorgeht (z.B. toUpperCase) ist die Philosophie die, dass bei Veränderungen neue Strings erstellt werden und als Resultat zurückgegeben werden.

Für Strings, deren Inhalt modifiziert werden kann, steht eine weitere Klasse 'StringBuffer' zur Verfügung.

Eine (meist flexiblere) Alternative ist die, dass man mit normalen 'char'-Arrays arbeitet, was im nächsten Paragraph weiter ausgeführt wird.

7.5 String-Konversionen

Umwandlung String → 'char'-Array

Zu diesem Zweck steht die String-Methode 'toCharArray' zur Verfügung:

```
String s = "Meier";
char[ ] charSequenz = s.toCharArray();
```

Die Methode erzeugt einen 'char'-Array mit gleicher Länge und gleichem Inhalt wie 's' und gibt ihn als Resultat zurück.

Umwandlung 'char'-Array → String

Für diese Konversion steht ein spezieller Konstruktor für Strings zur Verfügung:

```
char[ ] charSequenz = { 'M', 'e', 'i', 'e', 'r' };
String s = new String(charSequenz);
```

Umwandlung String → 'int'

Wenn ein String eine ganze Zahl in Character-Form enthält, kann diese aus dem String extrahiert und in 'int' konvertiert werden. Dabei kommt die Klasse 'Integer' zum Einsatz. Dies ist eine Klasse der Java Library mit Hilfsmethoden für den Datentyp 'int'. Die Umwandlung verwendet die (statische) Methode 'Integer.valueOf' :

Klasse 'Integer'

```
String s = " -335";
Integer wert = Integer.valueOf(s);          // Resultat als Integer-Objekt
int n = wert.intValue();                    // 'int'-Wert
```

valueOf

Das Resultat der Methode 'valueOf' ist (leider) nicht eine normale 'int'-Variable sondern ein Objekt der Klasse 'Integer'.

Ein Objekt dieser Klasse besteht aus einer Datenkomponente, in welcher ein 'int'-Wert abgespeichert ist, es stellt also einen ganzzahligen Wert im Gewand eines Objektes dar. Der 'int'-Wert des Objektes wird mit der Methode 'intValue' abgefragt.

Diese 'Integer'-Objekte sind erforderlich in Situationen, in welchen man aus technischen Gründen Objekte anstelle von elementaren Variablen benötigt. In der momentanen Situation könnte man darauf verzichten; die Methode 'valueOf' könnte genau so gut den Wert direkt als 'int'-Variable zurückgeben.

Wenn beim Aufruf der Methode 'valueOf' der übergebene String keine gültige ganze Zahl enthält, wird eine Exception (Fehler-Situation) ausgelöst, welche das Programm mit einer entsprechenden Fehlermeldung abbricht:

```
NumberFormatException
```

Wenn man im Fehlerfall keinen Programmabbruch haben möchte, kann man die Exception abfangen. Dies erfolgt mit einem 'try/catch'-Statement, auf welches wir im Moment nicht eingehen wollen.

Umwandlung String → 'double'

Die Extraktion einer reellen Zahl aus einem String erfolgt nach dem gleichen Schema wie bei einer ganzen Zahl, mit den entsprechenden Klassen 'Double' oder 'Float':

```
String s = " -335.421";
Double wert = Double.valueOf(s);            // Wert als Double-Objekt
double x = wert.doubleValue();              // 'double'-Wert
```

Man beachte, dass 'Double' die Klasse bezeichnet und 'double' den elementaren Datentyp.

Umwandlung 'int' / 'double' → String

Diese Umwandlung kann mit dem Konkatinierungs-Operator '+' für Strings oder mit der statischen Methode 'toString' der Klasse 'Integer' bzw. 'Double' erfolgen:

toString

```
int n = −335;
double x = −335.234;
String s1 = "" + n;                    // Konkatinierung zu Nullstring
String s2 = Integer.toString(n);
String s3 = "" + x;
String s4 = Double.toString(x);
```

Für Umwandlungen mit *Formatierungsangaben* enthält unsere eigene Klasse 'InOut' entsprechende Methoden 'toString', mit gleichen Parametern wie 'InOut.print', siehe Seite 29:

```
int n = −335;
double x = −335.234;
String s1 = InOut.toString(n, 8);          // mind. 8 Stellen
String s2 = InOut.toString(x, 8, 2);       // mind. 8 Stellen vor Dezimalpunkt,
                                           // 2 Dezimalstellen
```

7.6 Input/Output von Strings

Output:

Wir wissen wie man einen Strings 's' auf den Bildschirm ausgibt:

```
System.out.print(s);                   // ohne Zeilenvorschub
System.out.println(s);                 // mit Zeilenvorschub
```

Input:

Wir beschränken uns auf den praktisch wichtigsten Fall, das Einlesen einer ganzen Zeile in einen String.

Wir müssen auch hier wieder auf eine eigene Methode der Klasse 'InOut' zurückgreifen, da mit den Standard-Methoden von Java einige technische Vorbereitungen erforderlich sind, die die Programme aufblähen.

```
String s;
s = InOut.getLine();
```

Die Methode liest die ganze Eingabezeile ein und überträgt die Daten in einen String, der als Resultat zurückgegeben wird. Das Line-Terminator Zeichen von der 'Return'-Taste wird *nicht* in den String übertragen.

7.7 Arrays von Strings

Arrays von Strings gehören zum Thema ‘Arrays von Objekten’.

- *Erzeugung eines String-Arrays mittels Werte-Liste*

  ```
  String[ ] namen = { "Gutknecht", "Meier", "Mueller" };
  ```

  Verwendung der Strings:

  ```
  System.out.println(namen[0]);
  String s = namen[0].substring(0, 4);
  if ( namen[0].length() > 10 ) ...
  ```

- *Commandline-Parameter einer Applikation*

  Die ‘main’-Methode einer Applikation erhält einen String-Array als Parameter:

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

  Der Array enthält die beim Aufruf des Programmes angegebenen Parameter, die Commandline-Parameter:

  ```
  java  MyFirst  5210  Windisch
  ```

  Die angegebenen Parameter “5210” und “Windisch” stehen in der ‘main’-Methode als Strings

  ```
  args[0]  bzw.  args[1]
  ```

  zur Verfügung. Die Anzahl übergebener Commandline-Parameter kann mittels ‘args.length’ abgefragt werden.

7.8 Übungen

7.8.1 Lexikographischer Vergleich

Erstellen Sie eine eigene Vergleichsmethode

```
int compare(String a, String b)
```

die analog funktioniert wie die String-Methode ‘compareTo’ (lexikographischer Vergleich). Einzelne Characters können mit den normalen Operatoren $<$, $<=$ usw. verglichen werden.

Kapitel 8

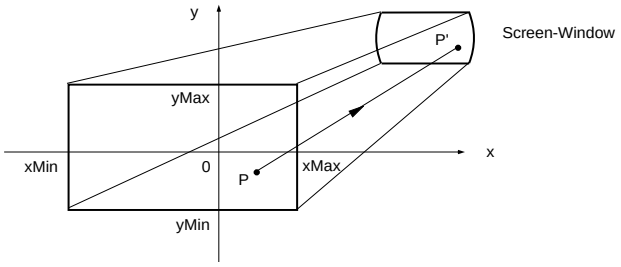
2D-Graphik

Graphik-Programme eignen sich sehr gut für einen praktischen Einsatz von Klassen und Objekten.

8.1 Welt- und Bildschirm-Koordinaten

Die Verwendung von ganzzahligen Bildschirm-Koordinaten *col* und *line* ist schwerfällig, sobald mathematische Berechnungen ins Spiel kommen. Besser geeignet sind mathematische xy-Koordinaten (float oder double) in der unendlichen Ebene.

Für die Übertragung der Punkte von der unendlichen xy-Ebene auf den Bildschirm, muss in der xy-Ebene ein Rechteck festgelegt werden, welches auf den Bildschirm transformiert wird:



Man nennt die mathematischen xy-Koordinaten in diesem Zusammenhang *Weltkoordinaten* im Gegensatz zu den Bildschirm-Koordinaten. Die Berechnung der Bildschirm-Kordinaten $(col, line)$ eines Punktes (x, y)

ist einfach:

$$\begin{aligned} col &= (x - xMin) \cdot width \ / \ (xMax - xMin) \\ line &= (yMax - y) \cdot height \ / \ (yMax - yMin) \end{aligned}$$

Dabei sind *width* und *height* die Abmessungen des Bildschirm-Windows (Anzahl Pixel). Die Formeln werden sofort verständlich, wenn man beachtet, dass $x - xMin$ der horizontale Abstand des Punktes (x, y) vom linken Rand des Rechteckes ist.

Dieser Abstand wird mit dem angegebenen Faktor so skaliert, dass für $x = xMax$ die maximale Kolonne *width* resultiert (rechter Bildschirmrand) .

Analog ist $ymax - y$ der vertikale Abstand vom oberen Rand des Rechteckes.

Zur Entlastung der Graphik-Programme sollte diese Umrechnung von Welt- in Bildschirm-Koordinaten aus den Programmen ausgelagert werden. Wir führen zu diesem Zweck eine Klasse 'Graph2d' mit geeigneten Methoden ein.

Die Klasse Graph2d

Die Klasse Graph2d ist eine selber geschriebene Klasse[1] zur Umrechnung von Welt-Koordinaten in Bildschirm-Koordinaten gemäss der oberen Konfiguration.

- Ein Objekt der Klasse enthält die Grenzen $xMin$, $xMax$, $yMin$, $yMax$ des Rechteckes in der xy-Ebene und die Abmessungen *width* und *height* des Bildschirm-Windows.
- Mit den Methoden 'pixCol' und 'pixLine' werden die Bildschirm-Koordinaten eines Punktes berechnet.
- Zusätzlich stellt die Klase zwei Methoden 'setPoint' und 'setLine' zum Zeichnen von Punkten und Strecken zur Verfügung. Bei diesen werden die Punkte in Weltkoordinaten angegeben und automatisch in Bildschirm-Koordinaten umgerechnet.

[1] erhältlich via anonymous FTP: loki.cs.fh-aargau.ch/pub/java/Graph2d.java

Methoden der Klasse 'Graph2d' :

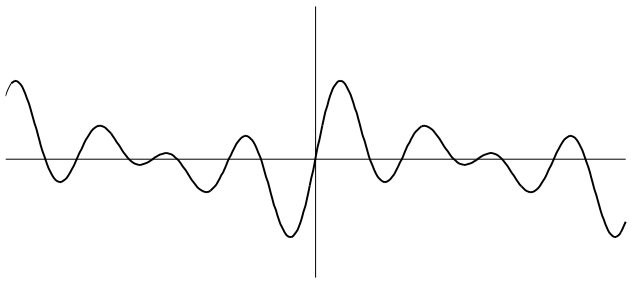
```
Graph2d(int width, int height)              // Konstruktor mit Angabe
                                            // der Window−Groesse
void setDisplayRange
        (double xMin, double xMax,          // Umgebungs−Rechteck
        double yMin, double yMax)
void setPoint(Graphics g,                   // Punkt zeichnen
        double x, double y)
void setLine(Graphics g,                    // Strecke zeichnen
        double x1, double y1)               // Strecke zeichnen
        double x2, double y2)
int pixCol(double x)                        // Pixel−Koord. berechnen
int pixLine(double y)
```

Beispiel:

Das folgende Applet zeichnet eine Funktion $y = f(x)$, gebildet mit 4 Sinus-Funktionen:

$$y = a \cdot \sin(x) + b \cdot \sin(2x) + c \cdot \sin(3x) + d \cdot \sin(4x)$$

Dabei sind a, b, c und d beliebige Konstanten.



```
import java.applet.Applet;
import java.awt.*;
public class Sinus extends Applet
{
  final double breite = 12.0;                    // x−Koordinatenbereich
  final double hoehe = 6.0;                      // y−Koordinatenbereich
  final double xMin = −0.5*breite;
  final double xMax = 0.5*breite;
  final double yMin = −0.5*hoehe;
  final double yMax = 0.5*hoehe;
  final int nPunkte = 400;                       // Anzahl Punkte der Kurve
  final double dx = breite / (nPunkte−1);        // Schrittweite
  double a=0.26, b=0.16, c=0.4, d=0.4;           // Koeffizienten

  Graph2d graph2d;

  // ———————— Methoden ————————————————————

  double f(double x)                             // y−Funktion
  { return a*Math.sin(x) +
           b*Math.sin(2*x) +
           c*Math.sin(3*x) +
           d*Math.sin(4*x);
  }

  public void init()
  { setBackground(Color.black);
    Dimension wnd = getSize();
    graph2d = new Graph2d(wnd.width, wnd.height);
    graph2d.setDisplayRange(xMin, xMax, yMin, yMax);
  }

  public void paint(Graphics g)
  { double x = xMin;
    double y = f(x);
    double xNext, yNext;
    g.setColor(Color.gray);
    graph2d.setLine(g, xMin, 0, xMax, 0);        // x−Achse
    graph2d.setLine(g, 0, yMin, 0, yMax);        // y−Achse
    g.setColor(Color.green);
    for(int i=1; i < nPunkte; i++)               // Funktion zeichnen
    { xNext = xMin + i*dx;
      yNext = f(xNext);
      graph2d.setLine(g, x, y, xNext, yNext);
      x = xNext; y = yNext;
    }
  }
}
```

Vermeidung von Verzerrungen

Bei der Festlegung des Rechteckes in der xy-Ebene sollte darauf geachtet werden, dass das Verhältnis Höhe zu Breite (in der xy-Ebene) gleich dem Verhältnis *height* zu *width* des Ausgabe-Windows ist:

$$hoehe = breite \cdot wnd.height/wnd.width$$

Ist dies nicht der Fall, so werden das Rechteck und damit alle Figuren bei der Transformation auf den Bildschirm gestaucht oder gedehnt, was zu Verzerrungen führt (Quadrate erscheinen auf dem Bildschirm als Rechtecke usw.)

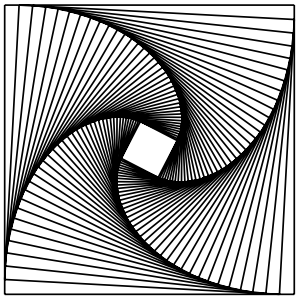
Die Klasse Punkt2d

Im folgenden verwenden wir öfters die Klasse 'Punkt2d', die im Kapitel 6 schrittweise eingeführt wurde. Am besten dient sie uns in der folgenden Form:

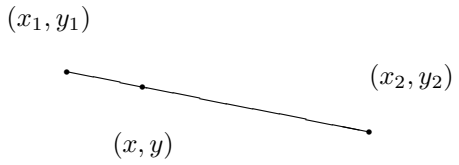
```
// ___________ Punkte in der Ebene ____________________________________
public class Punkt2d
{
  // ___________ Daten ____________________________
  public double x, y;                              // Koordinaten
  // ___________ Methoden _________________________
  public Punkt2d(){ }                              // leerer Konstruktor
  public Punkt2d(double x, double y)               // Konstruktor
  { this.x = x;
    this.y = y;
  }
  public void setVal(double x, double y)           // Koordinaten setzen
  { this.x = x;
    this.y = y;
  }
  public void setVal(Punkt2d from)                 // Koordinaten kopieren
  { x = from.x;
    y = from.y;
  }
  public double distance()                         // Abstand vom Nullpunkt
  { return Math.sqrt(x*x + y*y);
  }
  public void translate(double dx, double dy)      // Verschiebung
  { x += dx;
    y += dy;
  }
}
```

8.2 Folge von Quadraten

Wir zeichnen eine Folge von $n = 40$ verschachtelten Quadraten:



Das Bildungsgesetz der Folge ist einfach: Jedes Quadrat (ausser dem ersten) ist dem vorangehenden einbeschrieben. Die Eckpunkte eines einbeschriebenen Quadrates liegen auf den Kanten des vorangehenden:



Berechnung des Teilungspunktes (x, y) :

$$\begin{aligned} x &= p \cdot x_1 + q \cdot x_2 \\ y &= p \cdot y_1 + q \cdot y_2 \end{aligned}$$

Mit $p = 0.95$ und $q = 0.05$. (Für $p = 0.5$ und $q = 0.5$ wäre der Teilungspunkt der Mittelpunkt der Strecke, für $p = 0.95$ liegt der Teilungspunkt nahe bei (x_1, y_1)).

Das folgende Applet zeichnet das Ausgangsquadrat der Folge. Erweitern Sie das Applet, so dass es die ganze Folge der 40 Quadrate zeichnet.

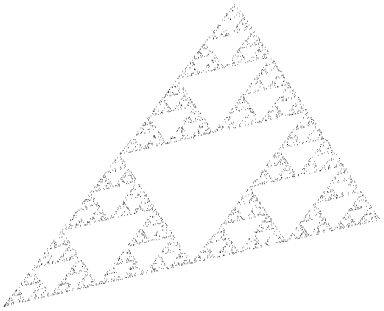
```
// ———— Rahmenprogramm fuer Quadrat−Folge ————————————————
import java.applet.*;
import java.awt.*;
public class Quadrate extends Applet
{
  final double a = 1.0;                          // halbe Quadratseite
  final double breite = 4.0 * a;                 // Koordinaten−Bereich
  final double p = 0.95;                         // Teilungskoeffizienten
  final double q = 0.05;
  Graph2d graph2d;

  void zeichneViereck(Graphics g, Graph2d graph2d,
              Punkt2d e1, Punkt2d e2,            // Ecken
              Punkt2d e3, Punkt2d e4)
  { graph2d.setLine(g, e1.x, e1.y, e2.x, e2.y);
    graph2d.setLine(g, e2.x, e2.y, e3.x, e3.y);
    graph2d.setLine(g, e3.x, e3.y, e4.x, e4.y);
    graph2d.setLine(g, e4.x, e4.y, e1.x, e1.y);
  }

  public void init()
  { setBackground(Color.blue);                   // Farben setzen
    setForeground(Color.white);
    Dimension wnd = getSize();                   // Window−Groesse
    graph2d = new Graph2d(wnd.width, wnd.height);
    double hoehe = breite * wnd.height / wnd.width;
    graph2d.setDisplayRange                      // Bild−Rechteck
              (−0.5*breite, 0.5*breite,
              −0.5*hoehe, 0.5*hoehe);
  }

  public void paint(Graphics g)
  { Punkt2d e1 = new Punkt2d(−a, a);             // Ecken
    Punkt2d e2 = new Punkt2d( a, a);
    Punkt2d e3 = new Punkt2d( a, −a);
    Punkt2d e4 = new Punkt2d(−a, −a);
    zeichneViereck(g, graph2d, e1, e2, e3, e4);
  }
}
```

Führen Sie für die Erweiterung eine Hilfsmethode zur Berechnung des Teilungspunktes auf einer Kante ein.

8.3 Das Sierpinski-Dreieck

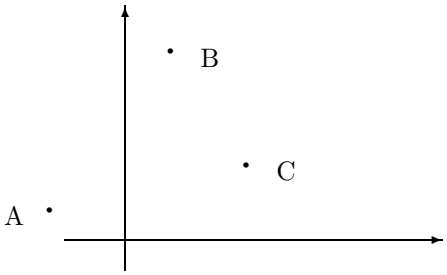
Das Sierpinski-Dreieck ist ein bekanntes *Fraktal*. Fraktale sind Figuren, die durch (meistens einfache) Algorithmen definiert sind, jedoch oft äusserst komplizierte Formen aufweisen.

Ein wichtiges Merkmal von Fraktalen ist die *Selbstähnlichkeit*, d.h. bei Vergrösserung von Ausschnitten erscheinen im Kleinen wieder die Formen des ganzen Fraktals. Das Sierpinski-Dreieck ist ein Dreieck mit einem Muster, welches die Eigenschaft der Selbstähnlichkeit zeigt:



Algorithmus für das Sierpinski-Dreieck

Gegeben sind drei Punkte A, B und C in der Ebene.



Der folgende Algorithmus zeichnet eine Folge von Punkten, die das Sierpinski-Dreieck darstellen. Dabei wird ein laufender Punkt P verwendet, der bei jedem Schritt transformiert wird:

1. Setze $P = A$ (oder gleich B oder C).
2. Zeichne den Punkt P.
3. Wähle zufällig einen der gegebenen Punkte A, B oder C und berechne den Mittelpunkt M von P und dem gewählten Punkt.
4. Setze P = M und gehe zu Schritt 2.

Bemerkung

Der gewählte Startpunkt ist von untergeordneter Bedeutung. Man kann auch irgend einen Punkt der Ebene als Startpunkt wählen. Auch dann entsteht das Sierpinski-Dreieck, mit einigen ausserhalb liegenden Punkten von den ersten Schritten.

Das Sierpinski Dreieck zieht also alle Punkte der Ebene an, man bezeichnet es daher auch als *Attraktor*.

Das folgende Applet zeichnet die 3 Eckpunkte A, B und C. Erweitern Sie es so, dass es $n = 20000$ Punkte nach dem obigen Algorithmus zeichnet.

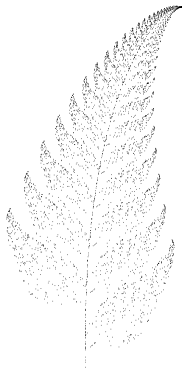
```
// ___________ Rahmenprogramm fuer Sierpinski−Dreieck ____________________
import java.applet.*;
import java.awt.*;
public class Sierp extends Applet
{
  final double breite = 4.0;                          // x−Koordinaten−Bereich
  final Punkt2d a = new Punkt2d(−1.0, 0.4);           // Eckpunkte
  final Punkt2d b = new Punkt2d(0.6, 2.5);
  final Punkt2d c = new Punkt2d(1.6, 1.0);
  Graph2d graph2d;

  public void init()
  { Dimension wnd = getSize();                        // Window−Groesse
    graph2d = new Graph2d(wnd.width, wnd.height);
    setBackground(Color.black);                       // Farben
    setForeground(Color.green);
    double hoehe = breite * wnd.height / wnd.width;
    graph2d.setDisplayRange
              (−0.5*breite, 0.5*breite, 0.0, hoehe);
  }

  public void paint(Graphics g)
  { graph2d.setPoint(g, a.x, a.y);
    graph2d.setPoint(g, b.x, b.y);
    graph2d.setPoint(g, c.x, c.y);
  }
}
```

Führen Sie einen laufenden Punkt P ein, der bei jedem Schritt durch den Mittelpunkt zu A, B oder C ersetzt wird.

Für die zufällige Wahl eines Eckpunktes wird eine Zufallszahl in $[0, 1]$ erzeugt. Wenn sie im ersten Drittel des Intervalles liegt wird A gewählt, im zweiten Drittel B, sonst C.

8.4 Das digitale Farn von M. Barnsley

Eines der genialsten Graphik-Programme ist das Programm von Michael Barnsley[2] zur Erzeugung des folgenden Farnes:



Das Farn entsteht nach einem analogen Algorithmus wie das Sierpinski Dreieck, geändert werden nur die Transformations-Gleichungen für die Berechnung des nächsten Punktes der Folge.

Anstelle des Prinzips mit den Mittelpunkten werden Transformations-Gleichungen der folgenden Form verwendet:

$$\begin{aligned} x' &= a \cdot x + b \cdot y + e \\ y' &= c \cdot x + d \cdot y + f \end{aligned}$$

Dabei sind a, b, c, d, e und f Konstanten und (x', y') sind die Koordinaten des transformierten Punktes. Man nennt solche Transformationen *affine Transformationen*. Sie umfassen eine grosse Klasse von Transformationen der Ebene wie Drehungen, Translationen, Streckungen usw.

[2] Michael Barnsley, 'Fractals Everywhere'

Die folgende Klasse eignet sich für die Transformation von Punkten mit affinen Transformationen:

```
public class ATransf                          // affine Transformationen
{ double a, b, c, d, e, f;                    // Koeffizienten

  public ATransf(double a, double b,          // Konstruktor
                 double c, double d,
                 double e, double f)
  { this.a = a; this.b = b;
    this.c = c; this.d = d;
    this.e = e; this.f = f;
  }

  public void transform(Punkt2d p)
  { double xTmp;
    xTmp = a * p.x + b * p.y + e;             // Hilfsvariable
    p.y = c * p.x + d * p.y + f;
    p.x = xTmp;
  }
}
```

Man beachte, dass die Hilfsvariable $xTmp$ erforderlich ist, weil in der Gleichung für $p.y$ das alte $p.x$ benötigt wird.

Das Farn wird mit den folgenden vier affinen Transformationen und dem nachfolgenden Algorithmus erzeugt:

Nr.	a	b	c	d	e	f
1	0.85	0.04	-0.04	0.85	0	1.6
2	0.2	-0.26	0.23	0.22	0	1.6
3	-0.15	0.28	0.26	0.24	0	0.44
4	0	0	0	0.16	0	0

Der Algorithmus für das Farn-Fraktal

Der folgende Algorithmus zeichnet eine Folge von Punkten, die das Farn darstellen. Wir verwenden dabei wieder einen laufenden Punkt P, der bei jedem Schritt transformiert wird.

1. Setze $P = (0, 0)$.
2. Zeichne den Punkt P.
3. Wähle zufällig eine Zeile der oberen Tabelle und transformiere den Punkt P mit dieser Transformation.
4. Gehe zu Schritt 2.

Zusatz zu Schritt 3:

Bei der zufälligen Wahl einer Transformation in Schritt 3, sind die folgenden Wahrscheinlichkeiten zu verwenden:

Nr.	Wahrscheinlichkeit
1	85 %
2	7 %
3	7 %
4	1 %

Dies verbessert die Qualität des Bildes wesentlich, indem das Farn gleichmässiger mit Punkten ausgefüllt wird.

Bemerkung:

Wie beim Sierpinski-Dreieck hat der Startpunkt keinen Einfluss auf das entstehende Bild, ausser dass je nach gewähltem Startpunkt die ersten Punkte nicht auf dem Farn liegen. Das Farn ist wie das Sierpinski-Dreieck ein *Attraktor*.

Erstellen Sie das Applet 'Farn' nach dem Schema des Sierpinski-Applets.

Anzahl Punkte: 40000

x-Koordinatenbereich: $xMin = -8, \quad xMax = 8$
y-Koordinatenbereich: $yMin = 0, \quad yMax = 12$

Koordinatenbereich für vergrösserten Ausschnitt des Farnes:

x-Koordinatenbereich: $xMin = 1, \quad xMax = 2$
y-Koordinatenbereich: $yMin = 1, \quad yMax = 2$

Bemerkung:

Das Applet hat einen technischen Mangel: Der Benutzer muss warten, bis alle Punkte gezeichnet sind, bevor er das Programm beenden kann. Dieser Mangel kann behoben werden unter Verwendung von Java Threads.

Weitere Fraktale nach dem Prinzip des Farnes

Mit dem Algorithmus für das Farn können auch andere Fraktale erzeugt werden, indem andere affine Transformationen verwendet werden. Barnsley hat die Methode weiterentwickelt zur *fraktalen Bildkompression*.

In den folgenden Tabellen sind Koeffizienten von affinen Transformationen mit zugehörigen Wahrscheinlichkeiten für andere Fraktale aufgeführt.

Ahorn-Blatt

a	b	c	d	e	f	p
0.4	-0.38	0.25	0.44	139.0	-41.0	0.21
0.59	0.0	0.02	0.6	126.0	104.0	0.28
0.4	0.35	-0.26	0.42	204.0	101.0	0.21
0.74	-0.03	0.02	0.74	76.0	-2.0	0.30

$xMin = 100,\ xMax = 500,\ yMin = 0,\ yMax = 300$

Spiralen

a	b	c	d	e	f	p
0.7879	-0.4242	0.2424	0.8598	1.7586	1.4081	0.90
-0.1212	0.2576	0.1515	0.0530	-6.7217	1.3772	0.05
0.1819	-0.1364	0.0909	0.1819	6.0861	1.5680	0.05

$xMin = -8,\ xMax = 8,\ yMin = 0,\ yMax = 12$

Drachen-Muster

a	b	c	d	e	f	p
0.5	-0.5	0.5	0.5	0	0	0.5
-0.5	-0.5	0.5	-0.5	-0.5	-0.5	0.5

$xMin = -0.9,\ xMax = 0.7,\ yMin = -0.9,\ yMax = 0.3$

8.5 Drehungen und Streckungen

Drehungen und Streckungen sind spezielle affine Punkt-Transformationen, die in Graphik-Programmen häufig zum Einsatz kommen.

Transformations-Gleichungen:

- Drehung um O mit Drehwinkel φ :

$$\begin{aligned} x' &= \cos(\varphi) \cdot x - \sin(\varphi) \cdot y \\ y' &= \sin(\varphi) \cdot x + \cos(\varphi) \cdot y \end{aligned}$$

  Für positive φ ergibt dies eine Drehung im Gegenuhrzeigersinn, für negative im Uhrzeigersinn.

- Streckung mit Streckungsfaktor q

$$\begin{aligned} x' &= q \cdot x \\ y' &= q \cdot y \end{aligned}$$

  Für $q > 1$ ergibt dies eine Vergrösserung, für $0 < q < 1$ eine Verkleinerung.

△ **Uebung:**

Erstellen Sie eine Klasse für Drehungen von Punkten der Klasse 'Punkt2d'. Der Konstruktor soll als Parameter den Drehwinkel φ erhalten. Die Klasse kommt im folgenden Paragraphen zum Einsatz.

8.6 Polygone und Streckenzüge

Wir führen noch drei weitere Methoden des Graphics-Objektes 'g' ein, die wir im zweiten Kapitel ausgelassen haben, da sie Arrays von Koordinaten als Parameter erhalten

drawPolygon: Polygon (Vieleck) zeichnen

Die Methode zeichnet die Kanten eines Polygons in der momentanen Farbe. Die Bildschirm-Koordinaten der Eckpunkte werden in zwei 'int'-Arrays übergeben:

```
int[ ]  xEcken = { 20, 40, 60 };
int[ ]  yEcken = { 20, 200, 30 };
g.drawPolygon(xEcken, yEcken, 3);       // Dreieck
```

Der dritte Parameter gibt die Anzahl Ecken an. Er ist nützlich, weil man häufig feste Arrays für die Koordinaten erzeugt, und diese für verschiedene Figuren mit unterschiedlicher Anzahl Ecken verwendet, wobei nicht immer alle Elemente der Arrays verwendet werden.

fillPolygon: Ausgefülltes Polygon zeichnen

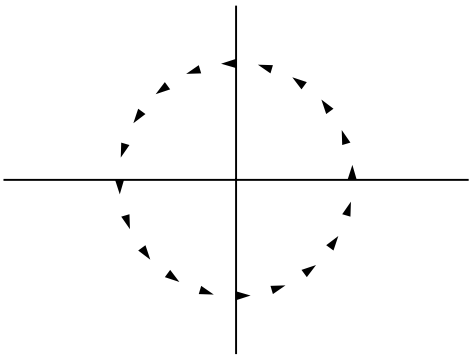
Die Methode funktioniert wie 'drawPolygon', sie malt jedoch das Innere des Polygons in der momentanen Farbe aus.

drawPolyline: Streckenzug zeichnen

Die Methode zeichnet einen Streckenzug in der momentanen Farbe. Sie hat dasselbe Format wie 'drawPolygon'. Der einzige Unterschied ist der, dass sie keine Verbindung vom letzten Punkt zum ersten zeichnet, sodass mit ihr offene Streckenzüge gezeichnet werden können.

Beispiel:

Das folgende Applet zeichnet das erste Dreieck der folgenden Figur auf der positiven x-Achse. Erweitern Sie das Applet, sodass es alle 20 Dreiecke mittels Drehung zeichnet.



```
import java.awt.*;
import java.applet.*;
public class Dreiecke extends Applet
{
  final double r = 1;                                  // Kreisradius
  final double h = 0.12;                               // Dreieckshoehe
  final double s = 0.3*h;                              // halbe Grundseite
  final double breite = 4.0;                           // x−Koordinatenbereich

  void zeichneDreieck(Graphics g, Graph2d graph2d,
                      Punkt2d a, Punkt2d b, Punkt2d c)
  {
    int[ ] x = { graph2d.pixCol(a.x),                  // Pixel−Koordinaten
                 graph2d.pixCol(b.x),
                 graph2d.pixCol(c.x) } ;
    int[ ] y = { graph2d.pixLine(a.y),
                 graph2d.pixLine(b.y),
                 graph2d.pixLine(c.y) } ;
    g.fillPolygon(x, y, 3);                            // ausgefuelltes Dreieck
  }

  public void init()
  { setBackground(Color.blue);
    setForeground(Color.yellow);
  }

  public void paint(Graphics g)
  { Dimension wnd = getSize();                         // Window−Groesse
    Graph2d graph2d = new Graph2d
                          (wnd.width, wnd.height);
    double hoehe = breite * wnd.height / wnd.width;
    graph2d.setDisplayRange(−0.5*breite, 0.5*breite,
                            −0.5*hoehe, 0.5*hoehe);
    Punkt2d a = new Punkt2d(r−s, 0);                   // linke Ecke
    Punkt2d b = new Punkt2d(r+s, 0);                   // rechte Ecke
    Punkt2d c = new Punkt2d(r, h);                     // Spitze
    zeichneDreieck(g, graph2d, a, b, c);
  }
}
```

Kapitel 9

Animationstechnik

Java eignet sich sehr gut zur Darstellung von Bewegungsabläufen, sogenannten *Animationen*. Bei einer Animation wird in einer Schleife eine Folge von Bildern gezeichnet, sodass die Wirkung eines Filmes entsteht.

Animationen, die ohne Flimmern und Flackern laufen sollen, benötigen einen sogenannten Offline-Screen.

9.1 Offline-Screens

Ein Offline-Screen ist ein Bereich im Speicher des Computers, in dem ein Bild der Animation vollständig vorbereitet wird, bevor es mit einem speziellen Befehl ('drawImage') als ganzes auf den Bildschirm ausgegeben wird.

Der 'drawImage'-Befehl synchronisiert die Ausgabe mit der Bildaufbereitung durch die Graphik-Karte, sodass kein Flackern entsteht.

Verwendung eines Offline-Screens

Die Erzeugung und Verwendung eines Offline-Screens erfolgt in den folgenden Schritten:

1. *Erzeugung eines Objektes für den Offline-Screen*

   Dazu wird die Methode 'createImage' verwendet. Diese gibt als Resultat ein Objekt der Klasse 'Image' zurückgibt:

   ```
   Image memScreen = createImage(width, height);       // Offline−Screen
   ```

   Die Methode gehört zum Basis-Applet und wird daher unqualifiziert aufgerufen. Mit den Parametern 'width' und 'height' wird die

gewünschte Grösse des Offline-Screens spezifiziert, welche normalerweise mit der Grösse des Ausgabe-Windows übereinstimmt.

2. *In den Offline-Screen zeichnen*

   Zu jedem Offline-Screen gehört ein *Graphics-Objekt*, welches mit der Methode 'getGraphics' geholt werden kann. Die Methode liefert als Resultat ein Objekt der Klasse 'Graphics' :

   ```
   Graphics memGr = memScreen.getGraphics();      // Graphics−Objekt
   ```

   Das Objekt 'memGr' stellt die vom Bildschirm gewohnten Methoden der Klasse 'Graphics' für Text und Graphik zur Verfügung, mit welchen ein Bild im Offline-Screen gezeichnet werden kann, z.B.

   ```
   memGr.setColor(Color.red);
   memGr.drawLine(0, 0, 100, 100);
   ```

3. *Übertragung auf den Bildschirm*

   Die Übertragung eines Offline-Screens auf den Bildschirm erfolgt mit der Methode 'drawImage' des Graphics-Objektes g des Bildschirmes:

   ```
   g.drawImage(memSreen, left, top, null);          // Ausgabe
   ```

   Mit den Parametern 'left' und 'top' wird die Ausgabe-Position der linken oberen Ecke des Offline-Screens im Window auf dem Bildschirm spezifiziert.

   Im vierten Parameter könnte ein sog. 'ImageObserver' angegeben werden, der in unserem Zusammenhang nicht aktuell ist.

   Da die Übertragungs-Zeit eines Offline-Screens auf den Bildschirm von der Grösse des Offline-Screens abhängt, sollte diese nicht zu gross sein, damit die Animation flüssig läuft (z.B. 480 x 360 Pixels).

9.2 Die Animations-Schleife

Die Animations-Schleife hat die Aufgabe, dafür zu sorgen, dass die 'paint'-Methode des Applets periodisch aufgerufen wird, zur Erzeugung des nächsten Bildes der Animation.

Dies muss so erfolgen, dass die Animation den Computer nicht blockiert, d.h. Benutzer-Aktionen (Maus-Klicks, Tasten) normal verarbeitet werden.

Dieses Problem wird in Java so gelöst, dass die Schleife in einem eigenen Thread läuft.

Ein *Thread* ist eine Umgebung (im Speicher des Computers), in der eine Methode parallel zu den anderen Methoden des Applets ablaufen kann. Der Garbage Collector läuft z.B. in einem eigenen Thread.

Thread

Das Applet 'Anim' [1]

Da die Animationsschleife für jede Animation dieselbe Struktur hat, kann sie in ein allgemeines Animations-Applet 'Anim' verpackt werden. Sie läuft nach den Regeln von Animationen in einem eigenen Thread und ruft periodisch die 'paint'- Methode auf.

Erweiterungen des Applets 'Anim'

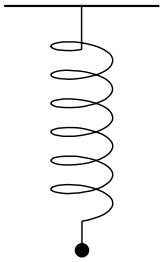
Mit dem Applet 'Anim' können leicht beliebige Animationen erstellt werden. Dabei kommt das Konzept der Klassenerweiterungen zum Einsatz: Die Animation wird als Erweiterung von 'Anim' definiert:

```
public class MyAnim extends Anim
```

Im Applet 'MyAnim' werden die gewünschten Methoden 'init' und 'paint' definiert, wodurch sie die (leeren) Standard-Versionen von 'Anim' ersetzen.

Beispiel: Feder-Pendel

Wir lassen eine Masse m an einer vertikalen Feder schwingen:



[1] erhältlich via anonymous FTP: loki.cs.fh-aargau.ch/pub/java/Anim.java

Die entstehende Bewegung ist eine *harmonische Schwingung*, im einfachsten Fall beschrieben durch die Funktion

$$y(t) = \cos(t) \ .$$

Das folgende Applet 'FederPendel' stellt die Bewegung der Pendelscheibe (ohne Feder) dar. Vor jeder Darstellung einer neuen Position wird die alte gelöscht durch nochmaligs Zeichnen mit der Hintergrundfarbe. Dieses Verfahren ist in dem Fall schneller als das Löschen des ganzen Offline-Bildspeichers.

```
// ---------- Feder-Pendel ----------------------------------------
import java.applet.*;
import java.awt.*;

public class FederPendel extends Anim              // Erweiterung des Applets 'Anim'
{
  final Color backColor = Color.darkGray;          // Hintergrund-Farbe
  final Color foreColor = Color.yellow;            // Vordergrund-Farbe
  final double breite = 4.0;                       // Koordinaten-Bereich
  final double dt = 0.05;                          // Zeitschritt
  final int radius = 10;                           // Scheibenradius
  double t;                                        // Zeit
  Graph2d graph2d;
  Image memScreen;                                 // Offline-Screen
  Graphics memGr;

  // ---------- Methoden ----------
  void zeichnePendel(Graphics g, Color color, double t)
  { double y = Math.cos(t);                        // Position
    int col = graph2d.pixCol(0);                   // Pixel-Koordinaten
    int line = graph2d.pixLine(y);
    g.setColor(color);
    g.fillOval(col-radius, line-radius,            // Pendelscheibe
               2*radius, 2*radius);
  }

  public void init()
  { setBackground(backColor);
    Dimension wnd = getSize();
    double hoehe = breite * wnd.height / wnd.width;
    graph2d = new Graph2d(wnd.width, wnd.height);
    graph2d.setDisplayRange(-0.5*breite, 0.5*breite,
                            -0.5*hoehe, 0.5*hoehe);
    memScreen = createImage                        // Offline Screen erzeugen
                (wnd.width, wnd.height);
    memGr = memScreen.getGraphics();
    t = 0.0;
    zeichnePendel(memGr, foreColor, t);            // erstes Bild im Offline-Screen
  }
```

```
public void paint(Graphics g)
  {
    g.drawImage(memScreen, 0, 0, null);        // Bildausgabe
    zeichnePendel(memGr, backColor, t);        // Bild loeschen (Offline−Screen)
    t += dt;
    zeichnePendel(memGr, foreColor, t);        // naechstes Bild (Offline−Screen)
  }
}
```

Veränderung der Bildfrequenz

Die Animations-Schleife im Applet 'Anim' wartet nach jedem Durchlauf eine kurze Zeit (10 Millisekunden). Diese Wartezeit kann jederzeit verändert werden, mit der Methode 'setPaintPeriod' :

```
setPaintPeriod(100);                           // 100 Millisekunden
```

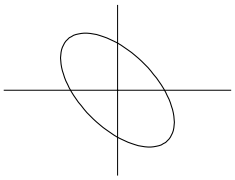
Da die Methode zum Applet 'Anim' gehört, wird sie unqualifiziert aufgerufen.

9.3 Übungen

9.3.1 Eine Ellipsen-Animation

Wir betrachten eine Bahnkurve eines Teilchens in der Ebene, beschrieben durch die Gleichungen:

$$\begin{aligned} x(t) &= \cos(t) \\ y(t) &= \sin(t - \varphi) \end{aligned}$$



Dabei ist φ eine beliebige Konstante, die sog. Phasenkonstante. Für $\varphi = 0$ entsteht eine Kreisbahn, die gleichförmig im Gegenuhrzeigersinn durchlaufen wird (t ist der Winkel zur x-Achse). Für $\varphi \neq 0$ ergibt sich eine Ellipse.

Wenn man die Phasenkonstante kontinuierlich verändert, entsteht eine schöne Animation, bei der sich die Ellipse langsam verformt. (Das Phänomen lässt sich auch auf einem Kathodenstrahl-Oszilloskop erzeugen.)

Erstellen Sie die Ellipsen-Animation gemäss den folgenden Angaben:

- Die Bahn wird als Streckenzug aus 100 Strecken gezeichnet, wobei t von 0 bis $2\,\pi$ läuft. Berechnen Sie die zugehörigen Punkte der Bahn und deren Pixel-Koordinaten (mit Graph2d).

  Die *Pixel*-Koordinaten werden in zwei Arrays *xArray* und *yArray* abgespeichert, anschliessend wird der Streckenzug mit der Methode 'drawPolygon' der Klasse Graphics gezeichnet:

  ```
  g.drawPolygon(xArray, yArray, nPunkte);
  ```

  Dabei ist 'g' das Graphics-Objekt des Offline-Screens. Die Methode 'drawPolygon' garantiert maximale Geschwindigkeit.

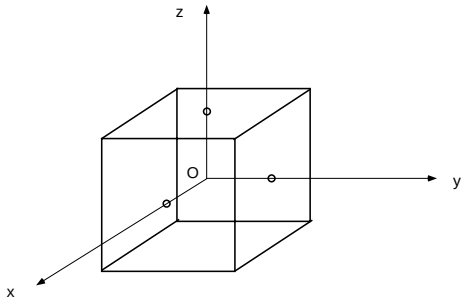
- Die Phasenkonstante φ ist für das erste Bild 0, dann wird sie pro Bild um $d\varphi = 0.02$ erhöht.
- Koordinaten-Bereich: breite = 6.0
- Farben: Hintergrund *Color.darkGray*, Vordergrund *Color.cyan*

9.3.2 Rotierender Würfel im Raum

In dieser Aufgabe wird ein Applet entwickelt, welches einen Würfel darstellt, der im Raum rotiert. Die Drehachse soll dabei ebenfalls variieren. Das Applet wird in 2 Schritten realisiert, im ersten Schritt stellt es einen ruhenden Würfel dar.

(a) Ruhender Würfel

1. Bestimmung der Eckpunkt-Koordinaten des Würfels in der folgenden Ausgangslage (Mittelpunkt im Koordinaten-Ursprung O):



   Die Koordinaten der Eckpunkte werden in einem 8 x 3 Array von 'double'-Elementen gespeichert:

```
double[ ][ ] e = { {a, −a, −a},
                   { a, a, −a }, ... }
```

Die Konstante a (halbe Kantenlänge) kann gleich 0.5 gewählt werden.

2. Drehung der Eckpunkte um die x-Achse mit einem beliebigen Drehwinkel α, z.B. $\alpha = 0.1$ (Formeln der Drehung siehe unten).
3. Drehung um die y-Achse mit einem beliebigen Drehwinkel β
4. Drehung um die z-Achse mit einem beliebigen Drehwinkel γ
5. Zeichnen der Kanten des Würfels mittels Normalprojektion auf die xy-Ebene (d.h. z-Koordinaten weglassen). Dabei werden die mathematischen Koordinaten wie gewohnt in Bildschirm-Koordinaten umgerechnet (Klasse 'Graph2d'). Koordinatenbereich:
   $xMin = -0.5,\ \ xMax = 0.5,$
   $yMin,\ \ yMax$ dem Window-Verhältnis entsprechend.

(b) Rotierender Würfel

Bei jedem Aufruf der 'paint'-Methode erfolgen die folgenden Aktionen:

1. Offline-Screen löschen durch Überzeichnen des Würfels in der Hintergrund-Farbe.
2. Drehwinkel um konstante Werte erhöhen, z.B.

   $$\begin{aligned} \alpha &\ += \ 0.01 \\ \beta &\ += \ 0.008 \\ \gamma &\ += \ 0.006 \end{aligned}$$

3. Würfel in der Ausgangsposition mit diesen neuen Winkeln drehen und wie in (a) im Offline-Screen darstellen.
4. Offline-Screen auf Bildschirm übertragen.

Drehungen um die Koordinatenachsen

Die Formeln für die Drehungen um die Koordiantenachsen ergeben sich leicht aus Drehungen in der Ebene. Z.B. werden bei einer Drehung um die x-Achse im Raum die yz-Koordinaten mit einer Drehung in der yz-Ebene transformiert, und x bleibt unverändert.

- Drehung um x-Achse mit Drehwinkel φ (Bogenmass)

$$\begin{aligned} x' &= x \\ y' &= \cos(\varphi) \cdot y + \sin(\varphi) \cdot z \\ z' &= -\sin(\varphi) \cdot y + \cos(\varphi) \cdot z \end{aligned}$$

  Die Drehung erfolgt bei positivem φ im Gegenuhrzeigersinn bei Blickrichtung in x-Achse.

- Drehung um y-Achse:

$$\begin{aligned} x' &= \cos(\varphi) \cdot x - \sin(\varphi) \cdot z \\ y' &= y \\ z' &= \sin(\varphi) \cdot x + \cos(\varphi) \cdot z \end{aligned}$$

- Drehung um z-Achse:

$$\begin{aligned} x' &= \cos(\varphi) \cdot x + \sin(\varphi) \cdot y \\ y' &= -\sin(\varphi) \cdot x + \cos(\varphi) \cdot y \\ z' &= z \end{aligned}$$

Geeignete Hilfsmethoden einführen !

Kapitel 10

Ereignisorientierte Programme

10.1 Ereignisse und Messages

Java Applets sind ereignisorientierte Programme: sie bestehen aus Methoden, die bei bestimmten Ereignissen (Events) aufgerufen werden:

Ereignis	aufgerufene Methode
Applet wurde geladen	`public void init()`
Bild muss (neu) gezeichnet werden	`public void paint(Graphics g)`
Maus-Taste wurde gedrückt	`public void mousePressed(MouseEvent e)`
Maus-Taste wurde losgelassen	`public void mouseReleased(MouseEvent e)`
Taste wurde gedrückt	`public void keyPressed(KeyEvent e)`
Taste wurde losgelassen	`public void keyReleased(KeyEvent e)`
Character-Taste wurde gedrückt	`public void keyTyped(KeyEvent e)`

weitere Ereignisse, siehe Seite 160

Vorteile von ereignisorientierten Programmen:

- Ereignisorientierte Programme erhalten die Kontrolle vom Betriebssystem nur dann, wenn sie ein Ereignis zu verarbeiten haben. Nach der Verarbeitung des Ereignisses, wenn die zugehörige Methode

endet, geht die Kontrolle an das Betriebssystem zurück. Dadurch können mehrere Programme aktiv sein (Multitasking).

- Das Konzept der ereignisorientierten Programmierung eignet sich für interaktive Programme, da der Benutzer den Ablauf des Programmes bestimmt, nicht das Programm.

Messages

Wenn bei einem Ereignis eine Methode eines Applets aufgerufen wird, sagt man auch, dass dem Applet eine *Message gesendet* werde. Die Daten des Ereignisses (z.B. Koordinaten des Mauszeigers bei einem Maus-Ereignis) werden der betreffenden Methode als Parameter übergeben.

Aktivierung von Messages

Die Methoden zur Verarbeitung von Maus- und Keyboard-Ereignissen werden bei den betreffenden Ereignissen *nicht* automatisch aufgerufen.

Zur Aktivierung der Messages sind im Applet spezielle Angaben erforderlich, mit denen sich das Applet als Empfänger von den gewünschten Messages anmeldet.

10.2 Maus-Messages

Das folgende Applet 'MouseTest' gibt bei jedem Maus-Klick die Koordinaten des Maus-Zeigers und die Anzahl Maus-Klicks auf den Bildschirm aus.

Die Nummern in den Kreisen verweisen auf nachfolgende Erklärungen.

```
// ———— Verarbeitung Maus−Klicks ————————————————————————
import java.applet.*;
import java.awt.*;
import java.awt.event.*;                                          ①
public class MouseTest extends Applet
             implements MouseListener                             ②
{
  int zaehler = 0;

  public void init()
  { addMouseListener(this);                                       ③
  }

  public void paint(Graphics g)
  { g.drawString("Bitte mit Maus klicken", 10, 20);
```

```
  }
  public void mousePressed(MouseEvent e)                                    ④
  {
    int xMouse = e.getX();                    // Maus−Koordinaten abfragen
    int yMouse = e.getY();
    zaehler++;
    Graphics g = getGraphics();               // Graphics Objekt holen
    g.clearRect(10, 20, 100, 60);             // Text−Bereich loeschen
    g.drawString("x−Koord: " + xMouse, 10, 40);
    g.drawString("y−Koord: " + yMouse,10, 60);
    g.drawString("Zaehler: " + zaehler, 10, 80);
  }
  public void mouseReleased(MouseEvent e) { }                               ⑤
  public void mouseClicked(MouseEvent e) { }
  public void mouseEntered(MouseEvent e) { }
  public void mouseExited(MouseEvent e) { }
}
```

Erklärungen:

1. Für die Verarbeitung von Ereignissen wird ein weiteres Package benötigt: 'java.awt.event' (ein Unterpackage von 'java.awt').

2. Mit der 'implements'-Angabe wird spezifiziert, dass das Applet das *Interface* 'MouseListener' implementiert.

   Was ist ein Interface?

   Ein *Interface* enthält eine oder mehrere Beschreibungen von Methoden, sogenannte *Spezifikationen* von Methoden. Eine Methoden-Spezifikation besteht aus dem Kopf der Methode (ohne den Code der Methode), sie beschreibt also vollständig das Aufruf-Format.

   MouseListener

   Das Interface 'MouseListener' besteht aus den folgenden Methoden-Spezifikationen:

   ```
   public void mousePressed(MouseEvent e);
   public void mouseReleased(MouseEvent e);
   public void mouseClicked(MouseEvent e);
   public void mouseEntered(MouseEvent e);
   public void mouseExited(MouseEvent e);
   ```

   Wenn eine Klasse (z.B. unser Applet) angibt, dass sie ein Interface implementiere ('implements'), verpflichtet sie sich, alle Methoden des Interfaces (als vollständige Methoden) zu definieren.

   Details zu Interfaces führen wir später ein.

3. Aktivierung der Maus-Messages

   Mit dem Aufruf der Methode 'addMouseListener' des Basis-Applets meldet sich das Applet ('this') beim Basis-Applet als Empfänger von Maus-Messages an. Dadurch werden bei Maus-Ereignissen die zugehörigen Methoden aufgerufen.

4. Verarbeitung der Maus-Messages

   Bei jeder Betätigung einer Maus-Taste wird die Methode 'mousePressed' aufgerufen.

MouseEvent

Die Methode erhält als Parameter ein Objekt *e* der Klasse 'MouseEvent'. Das Objekt enthält Daten des Ereignisses, u.a. die Koordinaten des Mauszeigers, welche mit 'getX', bzw. 'getY' aus dem Objekt herausgelesen werden können.

Anschliessend wir der globale Zähler der Maus-Klicks erhöht, bevor die Daten auf den Bildschirm ausgegeben werden.

getGraphics

Für die Ausgabe der Daten muss mit der Methode 'getGraphics' des Basis-Applets das Graphics Objekt des Bildschirm-Windows angefordert werden. Die Methoden zur Verarbeitung von Ereignissen erhalten das Graphics Objekt nicht wie 'paint' als Parameter.

5. Die weiteren Methoden für Maus-Ereignisse werden als leere Methoden definiert. Sie *müssen* im Applet definiert werden, da sie im Interface 'MouseListener' spezifiziert sind.

△ **Uebung:** Lösen Sie die Aufgabe 10.6.1

10.3 Keyboard-Messages

KeyListener

Keyboard-Messages werden nach dem gleichen Schema verarbeitet wie Maus-Messages. Anstelle des Interfaces 'MouseListener' kommt 'KeyListener' zum Einsatz, welches die folgenden Methoden-Spezifikationen enthält:

```
public void keyTyped(KeyEvent e);
public void keyPressed(KeyEvent e);
public void keyReleased(KeyEvent e);
```

Im Normalfall wird nur die Methode 'keyTyped' benötigt. Das folgende Applet gibt bei jedem Tastendruck den zugehörigen Character auf den Bildschirm aus.

```
// ———————— Keyboard−Messages ————————————————————————————
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class KBTest extends Applet
            implements KeyListener                  // Interface KeyListener
{
  public void init()
  { addKeyListener(this);                           // Keyboard−Messages aktivieren
    requestFocus();                                 // Input−Focus anfordern
  }

  public void paint(Graphics g)
  { g.drawString("Bitte eine Taste druecken", 10, 100);
  }

  public void keyTyped(KeyEvent e)
  { char inputChar = e.getKeyChar();                // Char. Wert der Taste
    Graphics g = getGraphics();
    g.clearRect(100, 130, 100, 20);                 // Text−Bereich loeschen
    g.drawString("Input−Character: " + inputChar, 100, 150);
  }

  public void keyPressed(KeyEvent e) { }
  public void keyReleased(KeyEvent e) { }
}
```

Bemerkungen:

- Zur Anmeldung des Applets als Empfänger von Keyboard-Messages wird die Methode 'addKeyListener' aufgerufen. Zusätzlich muss der Eingabe-Focus auf das Window des Applets gesetzt werden, mit der Methode 'requestFocus'. *requestFocus*

- Die Verarbeitung einer Taste erfolgt in der Methode 'keyTyped'. Der Character-Wert der Taste wird mittels e.getKeyChar() bestimmt. *getKeyChar*

  Die Methode 'keyTyped' wird nur bei normalen Tasten mit Character-Code aufgerufen, jedoch nicht bei Kontrolltasten (z.B. Pfeiltasten). Die Verarbeitung von Kontrolltasten erfolgt mit der Methode 'keyPressed', siehe unten.

Kontrolltasten

Bei jeder Betätigung einer Taste (normale Taste oder Kontrolltaste) werden die Methoden 'keyPressed' und 'keyReleased' aufgerufen. Zusätzlich wird bei normalen Tasten mit Character-Code die Methode 'keyTyped' aufgerufen.

getKeyCode

In den Methoden 'keyPressed' und 'keyReleased' kann der numerische Code der Taste, der sog. virtual Key-Code, abgefragt werden, mit der Methode 'getKeyCode':

```
int code = e.getKeyCode();
```

Für die Codes der Tasten stehen Konstanten zur Verfügung, z.B.

Konstante	Taste
KeyEvent.VK_UP	Pfeiltaste 'up'
KeyEvent.VK_DOWN	Pfeiltaste 'down'
KeyEvent.VK_LEFT	Pfeiltaste 'left'
KeyEvent.VK_RIGHT	Pfeiltaste 'right'
KeyEvent.VK_ENTER	Return-Taste

Die virtual Key-Codes enthalten keine Zusatzinformationen von der Shift-Taste (Gross/Klein), sie sind daher nur für Kontroll-Tasten geeignet.

△ **Uebung:** Lösen Sie die Aufgabe 10.6.2

Implementation mehrerer Interfaces

Wenn ein Applet Maus- *und* Keyboard-Ereignisse verarbeiten möchte, muss es die beiden Interfaces 'MouseListener' und 'KeyListener' implementieren. Diese müssen in einem einzigen 'implements'-Statement angegeben werden:

```
implements MouseListener, KeyListener
```

Für die Interfaces sind übrigens auch Beschreibungen in der Dokumentation der Java Library abrufbar (Package 'jawa.awt.event').

10.4 Die Methoden 'repaint' und 'update'

In Applets sollten grundsätzlich alle Ausgaben auf den Bildschirm in der 'paint'-Methode erfolgen.

Grund:

Wenn das Window des Applets durch Benutzeraktionen überdeckt worden ist, und dann wiederhergestellt werden muss, wird die 'paint'-Methode des Applets aufgerufen. Folglich werden Ausgaben, die nicht in der 'paint'-Methode erfolgen, dabei nicht reproduziert .

Für das obige Applet 'KBTest' bedeutet das, dass die letzte Ausgabe des Character-Wertes nach einer Überdeckung des Windows *nicht* mehr erscheint.

Dieses Problem tritt nicht auf, wenn alle Bildschirmausgaben in die 'paint'-Methode verlagert werden und zur Nachführung des Bildes eine 'paint'-Message erzeugt wird. Dabei kommt eine neue Methode 'repaint' zum Einsatz:

```
// ___________ Keyboard−Messages ____________________________________
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class KBTest extends Applet
             implements KeyListener
{  char inputChar = ' ';

   public void init()
   {  addKeyListener(this);
      requestFocus();
   }

   public void paint(Graphics g)
   {  g.drawString("Input−Character: " + inputChar, 100, 100);
   }

   public void keyTyped(KeyEvent e)
   {  inputChar = e.getKeyChar();                  // Char. Wert speichern
      repaint();                                   // 'paint'−Message erzeugen
   }

   public void keyPressed(KeyEvent e) { }
   public void keyReleased(KeyEvent e) { }
}
```

Die Methode 'keyTyped' speichert den Character-Wert der betätigten Taste in der globalen Variablen 'inputChar', dann ruft sie die Methode 'repaint' auf.

Die Methode 'repaint':

Aus technischen Gründen, die v.a. später, bei Verwendung von Kontroll-Elementen (Buttons, Checkboxes usw.) wichtig werden, sollte die 'paint'-Methode immer via 'repaint', nicht direkt aufgerufen werden.

Die Methode 'repaint' ist eine Methode des Basis-Applets ohne Parameter, welche die folgenden Aktionen ausführt:

1. Löschen des Bildschirm-Windows
2. Senden einer 'paint'-Methode, d.h. die 'paint'-Methode des Applets wird aufgerufen.

Bemerkung:

In gewissen Situationen möchte man bei einem 'repaint'-Aufruf das Löschen des Bildschirm-Windows vermeiden. Dies kann leicht erreicht werden. Die 'repaint'-Methode ruft nämlich eine Methode 'update' des Basis-Applets auf, welche die oben aufgeführten Aktionen 1. und 2. ausführt.

update

Die Methode 'update' kann überschrieben werden, z.B. so, dass sie ohne Löschen des Bildschirmes direkt 'paint' aufruft:

```
public void update(Graphics g)
{ paint(g);
}
```

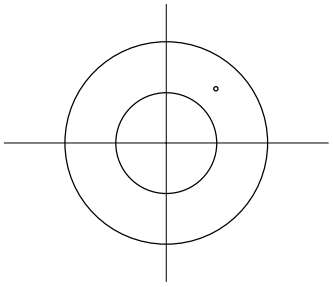
In der Methode 'update' können natürlich auch weitere Daten ausgegeben werden und weitere Aktionen ausgeführt werden..

Verwendung eines Offline-Screens

Die konsequenteste Methode zur Wiederherstellung des Bildschirm-Windows ist die Verwendung eines Offline-Screens (wie bei Animationen):

- Alle Ausgaben werden in den Offline-Screen geschrieben
- Die 'paint'-Methode überträgt den Offline-Screen auf den Bildschirm.
- Das automatische Löschen des Bildschirmes bei 'repaint' wird in der 'update'-Methode unterdrückt. Dadurch wird ein Flackern vermieden.

Das folgende Applet zeichnet eine Zielscheibe und bei jedem Mausklick eine Markierung an der Stelle des Maus-Cursors.



```
// ___________ Schuesse ____________________________________________________
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class Schuesse extends Applet
             implements MouseListener
{
  Image memScreen;                                  // Offline−Screen
  Graphics memGr;

  public void fadenKreuz(Graphics g, Dimension wnd)
  { int xm = wnd.width / 2;
    int ym = wnd.height / 2;
    int r1 = wnd.height / 4;
    int r2 = r1 / 2;
    g.drawLine(0, ym, wnd.width, ym);               // x−Achse
    g.drawLine(xm, 0, xm, wnd.height);              // y−Achse
    g.drawOval(xm−r1, ym−r1, 2*r1, 2*r1);           // grosser Kreis
    g.drawOval(xm−r2, ym−r2, 2*r2, 2*r2);           // kleiner Kreis
  }

  public void init()
  { setBackground(Color.blue);                      // Farben setzen
    setForeground(Color.white);
    Dimension wnd = getSize();                      // Window−Groesse
    memScreen = createImage                         // Offline−Screen
                (wnd.width, wnd.height);
    memGr = memScreen.getGraphics();
    fadenKreuz(memGr, wnd);
    addMouseListener(this);
  }

  public void paint(Graphics g)
  { g.drawImage(memScreen, 0, 0, null);
  }

  public void update(Graphics g)                    // Bildschirm nicht loeschen
  { paint(g);
  }

  public void mousePressed(MouseEvent e)
  { int xMouse = e.getX();
    int yMouse = e.getY();
    memGr.drawOval(xMouse−2, yMouse−2, 4, 4); // Markierung
    repaint();
  }

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

△ **Uebung:** Lösen Sie die Aufgabe 10.6.3

10.5 Standard-Messages für Applets

Neben 'init' und 'paint' gibt es weitere Methoden, die bei Standard-Ereignissen, die das Applet betreffen, automatisch aufgerufen werden:

Methode	Aufruf
void start()	jedesmal, wenn die HTML-Page aktiviert wird (erstmalig oder bei Rücksprung von anderer Page)
void stop()	die HTML-Page wird verlassen
void destroy()	Applet wird beendet (Browser wird beendet)

10.6 Übungen

10.6.1 Strecken zeichnen

Erstellen Sie ein Applet, mit welchem Strecken gezeichnet werden können. Die Strecken werden mit der Maus gezeichnet:

Der Maus-Cursor wird an den gewünschten Anfangspunkt der Strecke verschoben. Dann wird eine Maus-Taste gedrückt und der Maus-Cursor bei gedrückter Maus-Taste zum gewünschten Endpunkt der Strecke gezogen. Beim Loslassen der Maus-Taste wird die Strecke gezeichnet.

Das Applet verarbeitet die Ereignisse 'mousePressed' und 'mouseReleased'.

10.6.2 Kontrolltasten

Erstellen Sie ein Applet, welches einen roten Kreis (Lampe) zeichnet, dessen Rot-Komponente mit den Pfeiltasten 'up' und 'down' schrittweise verändert werden kann.

10.6.3 Repaint/Update

Stellen Sie das Applet der vorangehenden Aufgabe um, sodass alle Ausgaben in der 'paint'-Methode erfolgen. Unterdrücken Sie das automatische Löschen des Bildschirmes bei 'repaint', weil sonst ein Flackern auftritt.

Kapitel 11

Kontroll-Elemente

11.1 Einführung und Uebersicht

Kontroll-Elemente sind die Grund-Elemente für graphische Benutzeroberflächen (Graphical User Interfaces oder GUI). Die wichtigsten Kontroll-Elemente sind die folgenden:

- Button

- Checkbox

- Scrollbar

- TextField

  Ein TextField ist ein Eingabefeld für Text.

Diese Kontroll-Elemente werden in Java als Objekte der Klassen *Button, Checkbox, Scrollbar* bzw. *TextField* repräsentiert.

Jedes Kontroll-Element ist ein eigenes kleines Window, ein sogenanntes *Child-Window* des Haupt-Windows des Applets.

Beim Anklicken eines Kontroll-Elementes sendet dieses eine Message an eine zugehörige Methode des Applets, wenn sich das Applet als Empfänger der Messages angemeldet hat.

Die Erzeugung und Verwendung von Kontroll-Elementen verläuft nach einem Standard-Schema:

1. Erzeugung eines Objektes für das Kontroll-Element
2. Hinzufügung des Kontroll-Elementes zum Window des Applets, Festlegung von Grösse und Position
3. Aktivierung der Messages des Kontroll-Elementes
4. Verarbeitung der Messages des Kontroll-Elementes

Die Details führen wir an kommentierten Beispielen ein. Für weitere Informationen verweisen wir auf die offizielle Java-Dokumentation (Packages 'java.awt' und 'java.awt.event').

11.2 Buttons

In dem folgenden Applet kann eine Lampe (Kreis) mit einem Button ein- und ausgeschaltet werden.

```
// ___________ PushButton−Beispiel (Lampe ein− und ausschalten _____________
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class Lampe extends Applet
        implements ActionListener                                    (1)
{
  final int left = 20;
  final int top = 55;
```

```
  final int width = 60;
  final int height = 30;
  boolean lightOn;                          // Status der Lampe

  public void init()
  { setBackground(Color.gray);
    setForeground(Color.white);
    Button button = new Button("on/off");                          ②
    button.setBackground(Color.black);
    button.setForeground(Color.white);
    add(button);                                                   ③
    setLayout(null);
    button.setBounds(left, top, width, height);
    button.addActionListener(this);                                ④
  }

  public void paint(Graphics g)
  { if ( lightOn )
      g.setColor(Color.red);
    else
      g.setColor(Color.black);
    g.fillOval(120, 20, 100, 100);
  }

  // ___________ Button−Messages _________________
  public void actionPerformed(ActionEvent e)                       ⑤
  { lightOn = !lightOn;
    repaint();
  }
}
```

Erklärungen:

1. Für die Verarbeitung von Button-Messages muss das Interface 'ActionListener' implementiert werden. Dieses enthält nur eine einzige Methoden-Spezifikation: *ActionListener*

   ```
   public void actionPerformed(ActionEvent e);
   ```

2. Erzeugung eines Objektes der Klasse 'Button'. Dem Konstruktor wird der Beschriftungs-Text als Parameter übergeben.

3. Mit 'add(button)' wird das Objekt 'button' zum Applet-Window hinzugefügt. Anschliessend werden Grösse und Position des Buttons festgelegt (Methode 'setBounds'). Dies funktioniert nur, wenn der automatische *Layout-Manager* von Java inaktiviert wird:

   ```
   setLayout(null);
   ```

Es gibt verschiedene Layout-Manager, die Kontroll-Elemente automatisch positionieren, wir wollen auf diese nicht eingehen.

4. Aktivierung der Messages vom Button-Objekt. Das Applet ('this') meldet sich beim Button-Objekt mit der Methode 'addActionListener' als Empfänger von Messages an.

5. Zur Verarbeitung der Messages vom Push-Button wird die Methode 'actionPerformed' erstellt. Sie ändert den Status der Lampe und erzeugt mit 'repaint' eine 'paint'-Message.

△ **Uebung:** Lösen Sie die Aufgabe 11.6.1

Bestimmung der Herkunft einer Button-Message

Wenn in einem Applet mehrere Buttons verwendet werden, muss in der Methode 'actionPerformed' festgestellt werden können, von welchem Button die Message kommt.

getSource

Diese Information ist im Parameter 'e' der Methode 'actionPerformed' enthalten und kann mit der Methode 'getSource' abgefragt werden:

```
Button button = (Button)e.getSource();      // Quelle der Message
```

Klasse 'Object'

Die Methode 'getSource' liefert als Resultat eine Referenzvariable des Objektes, welches die Message ausgelöst hat. Das Resultat ist aus technischen Gründen, die später ersichtlich werden, eine Referenzvariable für allgemeine Objekte der Klasse 'Object'. Diese Klasse ist die Basis-Klasse aller Objekte in Java.

Das Resultat muss daher in eine Referenzvariable der Klasse 'Button' konvertiert werden, mit der von numerischen Konversionen bekannten Syntax:

```
(Button)e.getSource()                       // Typen−Konversion
```

Referenzvariablen für Objekte können so konvertiert werden. Dabei wird zur Laufzeit des Programmes geprüft, ob die Konversion zulässig ist.

Beispiel:

Im folgenden Applet 'Lampe2' kann die Farbe der Lampe mit drei Buttons gewählt werden. Für die Erzeugung der Buttons wird eine Hilfsmethode 'erzeugeButton' eingeführt.


```
// ___________ PushButton−Beispiel (Farbe waehlen mit Buttons) _____________
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class Lampe2 extends Applet
             implements ActionListener
{
  final int xm = 240;                          // Kreismittelpunkt
  final int ym = 150;
  final int radius = 60;                       // Radius
  Button rButton, gButton, bButton;            // PushButtons (Rot, Gruen, Blau)
  Color color = Color.black;                   // momentane Farbe

  Button erzeugeButton(String text,
               int left, int top,
               int width, int height)
  {
    Button button = new Button(text);          // Button−Objekt
    button.setBackground(Color.black);
    button.setForeground(Color.white);
    add(button);                               // Hinzufuegung zum Window
    button.setBounds(left, top, width, height); // Position und Groesse
    button.addActionListener(this);            // Button−Messages aktivieren
    return button;
  }

  public void init()
  { setBackground(Color.gray);
    setForeground(Color.white);
    setLayout(null);
    rButton = erzeugeButton("rot", 20, 100, 80, 30);
    gButton = erzeugeButton("gruen", 20, 140, 80, 30);
    bButton = erzeugeButton("blau", 20, 180, 80, 30);
  }

  public void paint(Graphics g)
  { g.setColor(color);
    g.fillOval(xm−radius, ym−radius, 2*radius, 2*radius);
  }

  // ___________ Button−Messages _________________
```

```
  public void actionPerformed(ActionEvent e)
  {
    Button button = (Button)e.getSource();      // Quelle
    if ( button == rButton )                    // Rot−Button
      color = Color.red;
    else if ( button == gButton )               // Gruen−Button
      color = Color.green;
    else if ( button == bButton )               // Blau−Button
      color = Color.blue;
    repaint();
  }
}
```

Der Eingabe-Focus

Wenn ein Pushbutton angeklickt wird, erhält er den Eingabe-Focus und behält ihn bei. Infolgedessen gehen alle nachfolgenden Keyboard-Messages ebenfalls an den Button und *nicht* an das Applet.

Wenn ein Applet Messages von Buttons und vom Keyboard verarbeiten möchte, muss es nach jedem Button-Klick den Eingabe-Focus zurück auf das Haupt-Window setzen. Dies erfolgt am besten nach jeder Verarbeitung einer Button-Message in der Methode 'actionPerformed' mittels:

```
requestFocus();
```

Übersicht:

Klasse 'Button'

Konstruktor:	`Button(String label)`

Messages

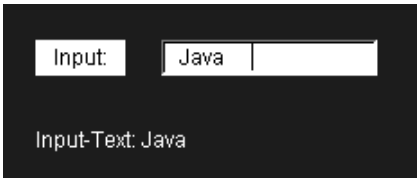
Interface:	`ActionListener`
Methode:	`void actionPerformed(ActionEvent e)`
Quelle bestimmen:	Methode 'getSource' von *e*

Aktivierung der Messages:

Methode	'addActionListener' des Buttons

11.3 TextFields

Ein TextField ist ein leeres Rechteck, in welchem Daten eingegeben werden können. Das folgende Applet gibt ein TextField aus und zeigt nach jeder Dateneingabe (abgeschlossen durch *Return*), den eingegebenen Text ('Java') unterhalb des TextFields an:



Das TextField ist mit einem vorangestellten *Label* versehen. Ein Label sieht aus wie ein Button, es erzeugt aber beim Anklicken keine Messages. Labels sind für Beschriftungen vorgesehen.

Label

Die Verarbeitung von Messages von einem TextField erfolgt wie bei Pushbuttons mit dem Interface 'ActionListener' (Methode 'actionPerformed').

```
// ———— Text−Field (Input−Daten) ————————————————————————
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class Input extends Applet
             implements ActionListener
{
  String s;                                          // Input−Text

  public void init()
  { setBackground(Color.blue);
    setForeground(Color.white);
    TextField tf = new TextField(10);                // TextField mit 10 Stellen
    tf.setBackground(Color.white);
    tf.setForeground(Color.black);
    Label lb = new Label("Input:" , Label.CENTER); // Label−Objekt
    lb.setBackground(Color.white);
    lb.setForeground(Color.black);
    setLayout(null);
    add(tf);                                         // Hinzufuegung zu Window
    add(lb);
    lb.setBounds(30, 40, 50, 30);
    tf.setBounds(100, 40, 120, 30);
    tf.addActionListener(this);                      // Text−Field Messages aktivieren
    tf.requestFocus();                               // Input−Focus anfordern
  }
```

```
  public void paint(Graphics g)
  { g.drawString("Input−Text: " + s, 30, 100);
  }

  // ___________ Text−Field−Messages _____________
  public void actionPerformed(ActionEvent e)
  { TextField tf = (TextField)e.getSource();
    s = tf.getText();                          // Input−Text speichern
    repaint();
  }
}
```

Bemerkungen:

1. Der Konstruktor für Labels hat einen zweiten Parameter, mit welchem die Ausrichtung des Textes im Label spezifiziert wird:

   ```
   Label.CENTER
   Label.LEFT
   Label.RIGHT
   ```

   Dies sind statische Konstanten der Klasse 'Label'.

2. Labels und TextFields stellen die Methoden 'setText' und 'getText' zur Verfügung, zum Setzen bzw. Lesen des Textes:

   ```
   String s = tf.getText();
   tf.setText("xxx");
   ```

   Zum *Löschen* des Textes eines TextFields wird er auf den Nullstring gesetzt:

   ```
   tf.setText("");                    // Text loeschen
   ```

△ **Uebung:** Lösen Sie die Aufgabe 11.6.2

Default-Grösse von Kontroll-Elementen

Die Wahl der Abmessungen von TextFields und Labels muss mit der Schriftgrösse abgestimmt werden. Dies kann auf unterschiedlichen Plattformen (Unix, Windows, MacOS) zu Unterschieden führen.

getPreferred-Size

Die diversen Kontroll-Elemente stellen eine Methode 'getPreferredSize' zur Verfügung, mit der eine passende Default-Grösse abgefragt werden kann:

```
setLayout(null);
TextField tf = new TextField(10);
add(tf);
Dimension tfDim = tf.getPreferredSize();
tf.setBounds(100, 40, tfDim.width, tfDim.height);
```

Achtung:

Die Methode 'getPreferredSize' muss *nach* 'add' aufgerufen werden, sonst liefert sie Abmessungen 0.

Der Operator 'instanceof'

Pushbuttons und TextFields verwenden für Messages beide das Interface 'ActionListener' mit der Methode 'actionPerformed'.

In einem Programm, welches mit beiden Arten von Kontroll-Elementen arbeitet, muss man folglich bei der Verarbeitung einer Message feststellen können, ob sie von einem Pushbutton oder von einem TextField stammt.

Dabei kommen die Methode 'getSource' und ein Operator von Java, der 'instanceof'-Operator, zum Einsatz:

```
public void actionPerformed(ActionEvent e)
{ Button button;
  TextField tf;
  Object quelle = e.getSource();          // Quelle der Message
  if ( quelle instanceof Button )
  { button = (Button)quelle;
    ...                                   // Button−Verarbeitung
  }
  else
  { tf = (TextField)quelle;
    ...                                   // TextField−Verarbeitung
  }
}
```

Wie oben (S. 164) erwähnt, liefert die Methode 'getSource' die Quelle der Message in der Form einer Referenz auf ein allgemeines Objekt der Klasse 'Object'.

Mit dem Operator 'instanceof' kann für eine beliebige Referenzvariable getestet werden, ob das Objekt, welches sie referenziert, von einer bestimmten Klasse (z.B. 'Button') ist.

Nach der Abklärung, was für ein Objekt die Message ausgelöst hat, kann das Resultat von 'getSource' entsprechend konvertiert und weiter verwendet werden.

Einsatzbeispiel:

Wenn ein Applet mehrere Eingabewerte benötigt, können mehrere TextFields verwendet werden, ergänzt durch einen Button, der angeklickt werden kann zur Verarbeitung der Eingabedaten, wenn alle eingegeben worden sind.

Übersicht Labels/TextFields:

Klasse 'Label'

Konstruktor:	Label(String text, int alignment)
Methoden:	String getText()
	void setText(String text)
Messages	keine

Klasse 'TextField'

Konstruktor:	TextField(int columns)
Methoden:	String getText()
	void setText(String text)

Messages

Interface:	ActionListener
Methode:	void actionPerformed(ActionEvent e)
Quelle bestimmen:	Methode 'getSource' von *e*

Aktivierung der Messages:

Methode	'addActionListener' des TextFields

11.4 Checkboxes

Eine Checkbox hat einen Status (markiert/demarkiert), der mittels Mausklick oder vom Programm aus geändert werden kann.

Übersicht:

Klasse 'Checkbox'

Konstruktor:	Checkbox(String label, boolean state)
Methoden:	boolean getState()
	void setState(boolean state)

Messages

Interface:	ItemListener
Methode:	void itemStateChanged(ItemEvent e)
Quelle bestimmen:	Methode 'getItem' von e

Aktivierung der Messages:

Methode	'addItemListener' der Checkbox

Das folgende Checkbox-Beispiel stellt den Text "Checkbox-Test" und zwei Checkboxes dar, mit welchen man Schriftattribute (fett, kursiv) des Textes verändern kann. Die Verwendung von Schriftattributen wurde auf Seite 106 eingeführt.


```
// ___________ Checkbox Beispiel (Schrift−Attribute waehlen) _________________
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class CBTest extends Applet implements ItemListener
{ String name = "SansSerif";
  final int size = 32;
  int style = Font.PLAIN;
  Font font = new Font(name, style, size);
  Checkbox fettCb;                              // Checkbox fuer Fett−Schrift
  Checkbox kursivCb;                            // Checkbox fuer Kursiv−Schrift

  Checkbox erzeugeCheckbox(String text,
              int left, int top,
              int width, int height)
  { Checkbox cb = new Checkbox(text, false);    // Checkbox−Objekt
    cb.setBackground(Color.black);
    cb.setForeground(Color.white);
    add(cb);                                    // Hinzufuegung zu Window
    cb.setBounds(left, top, width, height);     // Position und Groesse
    cb.addItemListener(this);                   // Messages aktivieren
    return cb;
  }
  public void init()
  { setBackground(Color.gray);
```

```
    setForeground(Color.black);
    setLayout(null);                              // ohne Layout−Manager
    fettCb = erzeugeCheckbox(" fett",
                80, 140, 80, 30);
    kursivCb = erzeugeCheckbox(" kursiv",
                200, 140, 80, 30);
  }
  public void paint(Graphics g)
  { g.setFont(font);
    g.drawString("Checkbox−Test", 80, 100);
  }
  // ___________ Checkbox−Messages _______________
  public void itemStateChanged(ItemEvent e)
  { style = Font.PLAIN;
    if ( fettCb.getState() )                      // Status abfragen
      style += Font.BOLD;
    if ( kursivCb.getState() )
      style += Font.ITALIC;
    font = new Font(name, style, size);
    repaint();                                    // Paint−Message ausloesen
  }
}
```

11.5 Scrollbars

Eine Scrollbar hat einen *Slider*, der mit der Maus verschoben werden kann. Jeder Position des Sliders entspricht ein Wert aus einem Bereich, der bei der Erzeugung der Scrollbar festgelegt wird:

```
Scrollbar sb = new Scrollbar(ausrichtung, anfangsWert,
                             sliderWidth, minVal, maxVal)
```

ausrichtung	Scrollbar.HORIZONTAL oder VERTICAL
anfangsWert	Anfangsposition des Sliders
sliderWidth	im Normalfall 1 (es ist nicht die absolute Grösse)
minVal, maxVal	Wertebereich der Sliderpositionen

Übersicht:

Klasse 'Scrollbar'

Konstruktor:	siehe oben
Methoden:	int getValue()
	void setValue(int newValue)

Messages

Interface:	AdjustmentListener
Methode:	void adjustmentValueChanged(AdjustmentEvent e)
Quelle bestimmen:	Methode 'getAdjustable' von e

Aktivierung der Messages:

Methode	'addAdjustmentListener' der Scrollbar

Das folgende Applet stellt eine horizontale Scrollbar dar und gibt laufend den Wert der Sliderposition aus.


```
// ___________ Scrollbar−Beispiel ____________________________________
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class SBTest extends Applet implements AdjustmentListener
{
  final int minVal = 0;                          // Parameter fuer Scrollbar
  final int maxVal = 100;
  final int slider = 1;                          // Slider−Size
  final int left = 50;
  final int top = 40;
  final int width = 140;
  final int height = 20;
  int wert = 50;                                 // momentaner Wert der Scrollbar

  public void init()
  {
    setBackground(Color.blue);
    setForeground(Color.white);
    setLayout(null);                             // kein Layout−Manager
    Scrollbar sb = new Scrollbar                 // Scrollbar−Objekt
                (Scrollbar.HORIZONTAL,
                wert, slider, minVal, maxVal);
    add(sb);                                     // Hinzufuegung zum Window
    sb.setBounds(left, top, width, height);      // Scrollbar−Pos./Groesse
    sb.addAdjustmentListener(this);              // Messages aktivieren
  }
```

```
  public void paint(Graphics g)
  { g.drawString("Wert: " + wert, 100, 100);
  }
  // ———————— Scrollbar−Messages ————————————
  public void adjustmentValueChanged(AdjustmentEvent e)
  {
    Scrollbar sb = (Scrollbar) e.getAdjustable();
    wert = sb.getValue();                          // momentaner Wert
    repaint();
  }
}
```

△ **Uebung:** Lösen Sie die Aufgabe 11.6.3

11.6 Übungen

11.6.1 Reset-Button

Erweitern Sie das Applet 'Schuesse' (S. 158) durch einen Button 'reset', mit dem man die Schüsse löschen kann.

11.6.2 Mittelwert

Erstellen Sie ein Applet zur Berechnung von Mittelwerten (z.B. Notendurchschnitte). Das Applet gibt ein TextField mit einem Label aus. Nach jeder Eingabe eines Wertes berechnet das Applet den momentanen Mittelwert und gibt ihn aus.

11.6.3 Scrollbars

Erweitern Sie das Applet 'Sinus' von Seite 129 durch 4 Scrollbars, mit denen man die Koeffizienten a, b, c und d im Bereich -1.0 bis 1.0 verändern kann.

Wählen Sie den Wertebereich der Scrollbars von 0 bis 100 und rechnen Sie die Werte in den Bereich der Koeffizienten um:

$$koeffizient = (wert - 50) \,/\, 50.0$$

Kapitel 12

Text-Files

12.1 Einführung und Uebersicht

Ein *Text-File* ist eine Sequenz (Aneinanderreihung) von Characters mit eingebetteten Line-Terminators. Ein Line-Terminator $\mathcal{L}$ ist ein Spezialzeichen, welches das Ende einer Zeile markiert.

Line-Terminator

Beispiel eines Text-Files mit 2 Zeilen:

F	H	-	A	a	r	g	a	u	$\mathcal{L}$	5	2	1	0	W	i	n	d	i	s	c	h	$\mathcal{L}$

$\mathcal{L} = Line - Terminator$

Als Line-Terminators kommen verschiedene Characters zum Einsatz, je nach Betriebssystem:

Character	Unicode dezimal	Unicode hexadezimal	Java Literal
Line-Feed (lf)	10	0A	'\n' (newline)
Carriage-Return (cr)	13	0D	'\r'

Die Namen 'Carriage Return' (Wagen-Rücklauf) und 'Line Feed' (Zeilen-Vorschub) stammen noch aus der Zeit der mechanischen Drucker.

- DOS/Windows:

  Unter DOS und Windows besteht ein Line-Terminator aus zwei Characters, einem Carriage-Return, gefolgt von einem Line-Feed.

- Unix:

  Unter Unix besteht ein Line-Terminator nur aus einem Zeichen, dem Linefeed-Character lf.

- Macintosh:

  Unter MacOS besteht ein Line-Terminator nur aus einem Zeichen, dem Carriage-Return cr.

Die Bearbeitung von Files mit Java-Programmen ist primär für Applikationen vorgesehen. In Applets sind Filezugriffe aus Sicherheitsgründen gesperrt. Ausnahme: Bei Verwendung des Appletviewers können die Sicherheitseinschränkungen aufgehoben werden.

Klassen für Bearbeitung von Text-Files

Files werden in einem Java-Programm mit Objekten der folgenden Klassen repräsentiert:

- FileReader, BufferedReader
- FileWriter, BufferedWriter, PrintWriter

12.2 Lesen eines Text-Files

Die Verarbeitung eines Input-Files mit einem Java-Programm erfolgt in 3 Schritten:

1. Erzeugung eines Objektes für das File

   Dies nennt man auch die Eröffnung des Files ('open').

2. Lesen von Characters oder ganzen Zeilen

   Dies erfolgt mit Methoden ('read', 'readLine') des erzeugten Objektes. Die Daten können nur sequentiell (der Reihe nach) gelesen werden, so wie sie im File gespeichert sind.

3. File schliessen ('close')

Die Details führen wir an dem folgenden Beispiel ein, welches ein File zeichenweise liest, und dabei zählt, wieviele Zeichen das File enthält (inklusive Line-Terminators).

```
// ___________ Characters in Text−File zaehlen ____________________________
import java.io.*;
public class CharCount
{
public static void main(String[ ] args)
            throws FileNotFoundException, IOException                    ①
{
  int ch;
  int zaehler = 0;
  FileReader f0 = new FileReader("test.txt");                            ②
  BufferedReader f = new BufferedReader(f0);
  ch = f.read();                                                         ③
  while ( ch >= 0 )
  { zaehler++;
    ch = f.read();
  }
  f.close();                                                             ④
  System.out.println("Anzahl Zeichen: " + zaehler);
  }
}
```

Erklärungen:

1. Exceptions

   Bei allen Fehlern, die bei File-Operationen auftreten können (ungültiger File-Name, Input/Output-Error, usw.) werden *Exceptions* (Ausnahme-Situationen) ausgelöst: *Exceptions*

   ```
   FileNotFoundException, IOException
   ```

   Wir kennen schon die Standard-Exceptions des Java Run Time Systems, z.B.

   ```
   ArrayOutOfBoundException,
   ```

   die ausgelöst wird, wenn ein Index für ein Array-Element einen ungültigen Wert enthält. Eine Exception führt (wenn kein Exception Handler vorhanden ist) zum Abbruch des Programmes mit einer entsprechenden Fehlermeldung.

   Die Exceptions des File-Handlings *müssen* im Gegensatz zu den Standard-Exceptions mit einem Exception-Handler behandelt oder an das aufrufende Programm weitergeleitet werden. Die Weiterleitung von Exceptions erfolgt mittels ‘throws’:

   ```
   throws FileNotFoundException, IOException
   ```

Dies muss im Kopf von *jeder* Methode, in welcher eine File-Exception auftreten kann, angegeben werden.

2. Eröffnung des Files

   Die Eröffnung eines Files erfolgt in Java durch Erzeugung von 2 Objekten:

   ```
   FileReader f0 = new FileReader("test.txt");
   BufferedReader f = new BufferedReader(f0);
   ```

   Das Objekt 'f0' der Klasse 'FileReader' ist das Basis-Objekt für das File. Es wird im zweiten Schritt zu einem 'BufferedReader' Objekt 'f' erweitert, welches die unten eingeführten Read-Methoden zur Verfügung stellt.

   File-Operationen sollten aus Performance-Gründen immer Buffers (Zwischenspeicher) verwenden, sodass nicht jeder Methoden-Aufruf wirklich zu einem physischen Zugriff auf den Disk führt. Die Klasse 'BufferedReader' verwendet solche Buffers.

   Der Grund für die zweistufige Erzeugung ist die allgemeine Ausrichtung der Klassen für File-Bearbeitung, für diverse Arten von Files, u.a. auch solche, die über Internet angesprochen werden.

3. Daten lesen

   Ein Objekt der Klasse 'BufferedReader' stellt die folgenden Read-Methoden zur Verfügung:

read

int read()

Die Methode liest das nächste Zeichen des Files. Das Resultat der Methode ist der Unicode des gelesenen Zeichens, bzw. der Wert -1, bei End-Of-File.

Der Datentyp des Resultates ist 'int' nicht 'char', wegen dem Wert -1, der nicht im 'char'-Bereich liegt. Der gelesene Wert kann mit einer Konversion *(char) Wert* in einen Characterwert konvertiert werden.

readLine

String readLine()

Die Methode liest eine ganze Zeile des Files. Das Resultat ist ein String mit den gelesenen Zeichen *ohne* Line-Terminator. Bei End-Of-File ist das Resultat *null* (Null-Referenz).

Die Methode kann auch dazu verwendet werden, den Rest einer Zeile, von der schon mit 'read' Zeichen gelesen wurden, einzulesen.

4. Abschliessung des Files: *close*

```
f.close();
```

Dadurch wird automatisch auch das Objekt 'f0' geschlossen.

Bei der Eröffnung des Files kann der angegebene Name auch ein vollständiger Pfadname sein. Dabei ist zu beachten, dass das Backslash-Zeichen '\' von DOS in einem Java-String als Escape-Sequenz '\\' geschrieben werden muss (siehe Seite 50):

```
FileReader f0 = new FileReader("c:\\java\\test.txt");
```

Read Ahead

Die Lese-Schleifen für Daten von Files haben die folgende Form, bei der das erste Element *vor* der Schleife gelesen wird ('read ahead'):

```
ch = f.read();                          // read ahead
while ( ch >= 0 )
{ ...                                   // Verarbeitung
  ch = f.read();                        // read next
}
```

So werden alle Zeichen des Files, auch das letzte, richtig verarbeitet. Bei leeren Files (keine Daten) wird keine Verarbeitung ausgeführt.

Die beiden Lese-Operationen können zu einer einzigen zusammengefasst werden, die direkt in der 'while'-Bedingung erscheint:

```
while ( (ch = f.read() ) >= 0 )
{ ...                                   // Verarbeitung
}
```

Erklärung:

In Java ist eine Zuweisung

```
a = b
```

ein Ausdruck mit einem Wert, nämlich dem Wert der linken Seite nach der Zuweisung. Der Wert kann direkt weiterverwendet werden:

```
if ( (a = b) >= 0 ) ...
```

Dabei wird der Ausdruck b ausgewertet, dann wird das Resultat der Variablen a zugewiesen und anschliessend mit 0 verglichen. Die runden Klammern um die Zuweisung sind erforderlich, weil sonst die Auswertungsreihenfolge nicht stimmt.

Für Lese-Schleifen mit 'read ahead' ist dies zu empfehlen, in anderen Situationen macht es den Code nur schwer verständlich.

△ **Uebung:** Lösen Sie die Aufgaben 12.6.1 und 12.6.2

Lesen von numerischen Werten

Zum Einlesen von numerischen Werten ('int', 'double') stehen leider keine Standard-Methoden zur Verfügung, es müssen String-Konversionen verwendet werden:

1. Characters der Zahl in einen String einlesen ('read' oder 'readLine')
2. Numerische Konversion mit 'valueOf' (siehe Seite 124).

12.3 Erstellung von Text-Files

Die Erstellung eines Text-Files verläuft analog, mit Objekten der Klassen 'FileWriter', 'BufferedWriter' und 'PrintWriter'.

1. *Objekt für das File erstellen (Eröffnung)*

   Wir verwenden als Standard-Namen für Output-Files 'g' um sie von Input-Files zu unterscheiden.

   ```
   FileWriter g0 = new FileWriter("test.txt");
   BufferedWriter g = new BufferedWriter(g0);
   ```

   Wenn schon ein File mit dem angegebenen Namen existiert, wird dieses überschrieben.

2. *Ausgabe von Daten*

   Das Objekt 'g' stellt die folgenden Methoden zur Ausgabe von Characters zur Verfügung:

write

void write(int ch)

Die Methode schreibt ein Character auf das File. Der Parameter hat den Datentyp 'int', was keine Komplikationen ergibt, da 'char'-Werte (Unicodes) automatisch in 'int' konvertiert werden. Beispiel:
`g.write('a');`

void newLine() *newLine*

Ausgabe eines Line-Terminators (ein oder zwei Characters, je nach Betriebssystem).

Für die Ausgabe von weiteren Datenelementen steht die Klasse 'PrintWriter' zur Verfügung, siehe unten.

3. *File schliessen*

   ```
   g.close();
   ```

   Der 'close'-Aufruf ist absolut erforderlich, sonst können Daten verlorengehen.

Beispiel:

Die folgende Applikation liest ein File zeilenweise und überträgt die Zeilen auf ein Output-File. Dabei werden Zeilen, die mehr als 80 Stellen enthalten (ohne Line-Terminator), im Output-File aufgeteilt.

Die File-Namen werden als Commandline-Parameter eingelesen (String-Array 'args').

```
// ___________ Aufteilung langer Zeilen ____________________________________
import java.io.*;
public class LineSplit
{
  static public void main(String[ ] args)
              throws FileNotFoundException, IOException
  { FileReader f0 = new FileReader(args[0]);
    BufferedReader f = new BufferedReader(f0);
    FileWriter g0 = new FileWriter(args[1]);
    BufferedWriter g = new BufferedWriter(g0);
    String zeile;
    int i, zaehler;
    while ( (zeile = f.readLine() ) != null )
    { zaehler = 0;
      for ( i=0; i < zeile.length(); i++)
      { if ( zaehler >= 80 )
        { g.newLine();                          // neue Zeile
          zaehler = 0;
        }
        g.write(zeile.charAt(i));               // Ausgabe
        zaehler++;
      }
      g.newLine();
    }
    f.close();  g.close();
    System.out.println("Verarbeitung abgeschlossen");
  }
}
```

12.4 Ausgabe von Datenelementen

Für Ausgaben von Datenelementen ('int', 'double', usw.) stehen die von Bildschirmausgaben bekannten Methoden 'print' und 'println' in der
PrintWriter Klasse 'PrintWriter' zur Verfügung.

Ein Objekt der Klasse 'PrintWriter' erhält man aus einem 'BufferedWriter'-Objekt :

```
FileWriter g0 = new FileReader("test.txt");
BufferedWriter g = new BufferedWriter(g0);
PrintWriter g1 = new PrintWriter(g);
```

print Das Objekt 'g1' stellt alle 'print'- und 'println'-Methoden, die wir von
println Bildschirm-Ausgaben kennen, zur Verfügung, z.B.

```
double x = 2 * Math.PI;
g1.println(x);
```

Formatierte Ausgaben

Bei der Ausgabe von numerischen Werten auf Files, kommt wieder das Problem, dass die 'print'- und 'println'-Methoden keine Zusatzparameter zur Formatierung (Anzahl ausgegebener Stellen) haben.

Daher kommen wieder die Methoden 'toString' unserer Klasse 'InOut' zum Einsatz (siehe Seite 125):

```
int n = 10;
double x = Math.PI;
g1.print("n=" + InOut.toString(n, 4));       // 4−stellig
g1.print("Pi=" + InOut.toString(x, 1, 4));   // 4 Dezimalstellen
```

△ **Uebung:** Lösen Sie die Aufgaben 12.6.3 und 12.6.4

12.5 Binäre Files

Binäre Files haben keine Zeilenstruktur, es sind einfach Sequenzen von
Streams Bytes, sogenannte Ströme (Streams). Beispiele sind Files mit den folgenden File-Typen

'.exe', '.jpg', '.gif', '.wav' usw.

Binäre Files werden nach dem gleichen Prinzip wie Text-Files bearbeitet, unter Verwendung der folgenden Klassen und Methoden:

Input-Files:

FileInputStream	
BufferedInputStream:	int read() void close()

Output-Files:

FileOutputStream	
BufferedOutputStream:	void write(int b) void close()

Die 'read'-Methode liefert den Wert des gelesenen Bytes (0 .. 255), bzw. -1 bei End-Of-File, 'write' schreibt den Wert eines Bytes auf das File.

Beispiel:

Die folgende Applikation kopiert ein beliebiges File.

```
import java.io.*;
public class FileCopy
{
  static public void main(String[ ] args)
              throws FileNotFoundException, IOException
  { int b;
    FileInputStream f0 = new FileInputStream(args[0]);
    BufferedInputStream f = new BufferedInputStream(f0);
    FileOutputStream g0 = new FileOutputStream(args[1]);
    BufferedOutputStream g = new BufferedOutputStream(g0);
    while ( ( b = f.read() ) >= 0 )
      g.write(b);                                    // Ausgabe
    f.close();  g.close();
    System.out.println("Verarbeitung abgeschlossen");
  }
}
```

Für Input/Output von Datenelementen ('int', 'double' usw.) in binärer Form stehen die Klassen 'DataInputStream' und 'DataOutputStream' zur Verfügung, siehe Dokumentation (Package 'java.io').

12.6 Übungen

12.6.1 Zeilenweises Lesen eines Files

Erstellen Sie eine Applikation, die ein Text-File zeilenweise liest und die Zeilen auf den Bildschirm ausgibt.

Das Programm soll den Filenamen als Commandline-Parameter erhalten (siehe Seite 126).

12.6.2 Ein Wordcount-Programm

Unix stellt ein Utility 'wc' (word count) zur Verfügung, welches die Anzahl Zeilen, Wörter und Zeichen eines Text-Files bestimmt:

wc *filename*

Schreiben Sie eine Java Applikation mit denselben Funktionen.

Hinweis:

Ein Wort ist eine Zeichenkette, beendet durch ein *Whitespace-Character:* ein Leerzeichen, Line-Terminator ('\n', '\r') oder Tabulatorzeichen ('\t').

Das File wird zeichenweise gelesen. Führen Sie eine Variable 'state' ein, die immer angibt, ob man sich innerhalb oder ausserhalb eines Wortes befindet. Beim Eintreten in ein Wort wird der Zähler für die Wörter um 1 erhöht.

12.6.3 Zeilen-Nummern

Erstellen Sie eine Applikation, die ein Text-File liest und die Zeilen mit vorangestellten 6-stelligen Zeilen-Nummern auf ein Output-File schreibt.

12.6.4 String-Replace

Erstellen Sie eine Applikation, die ein Text-File zeilenweise liest und die Zeilen auf ein Output-File überträgt. Dabei soll jedes Vorkommen eines Strings *search* ersetzt werden durch einen zweiten *replace*. Die Längen der Zeilen werden dadurch entsprechend verändert.

Die Strings *search* und *replace*, sowie die File-Namen werden dem Programm als Commandline-Parameter übergeben.

Anleitung:

Verändern Sie eine gelesene Zeile nicht im Speicher, sondern schreiben sie (die unveränderten und) die neuen Daten stückweise direkt in das Output-File.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S