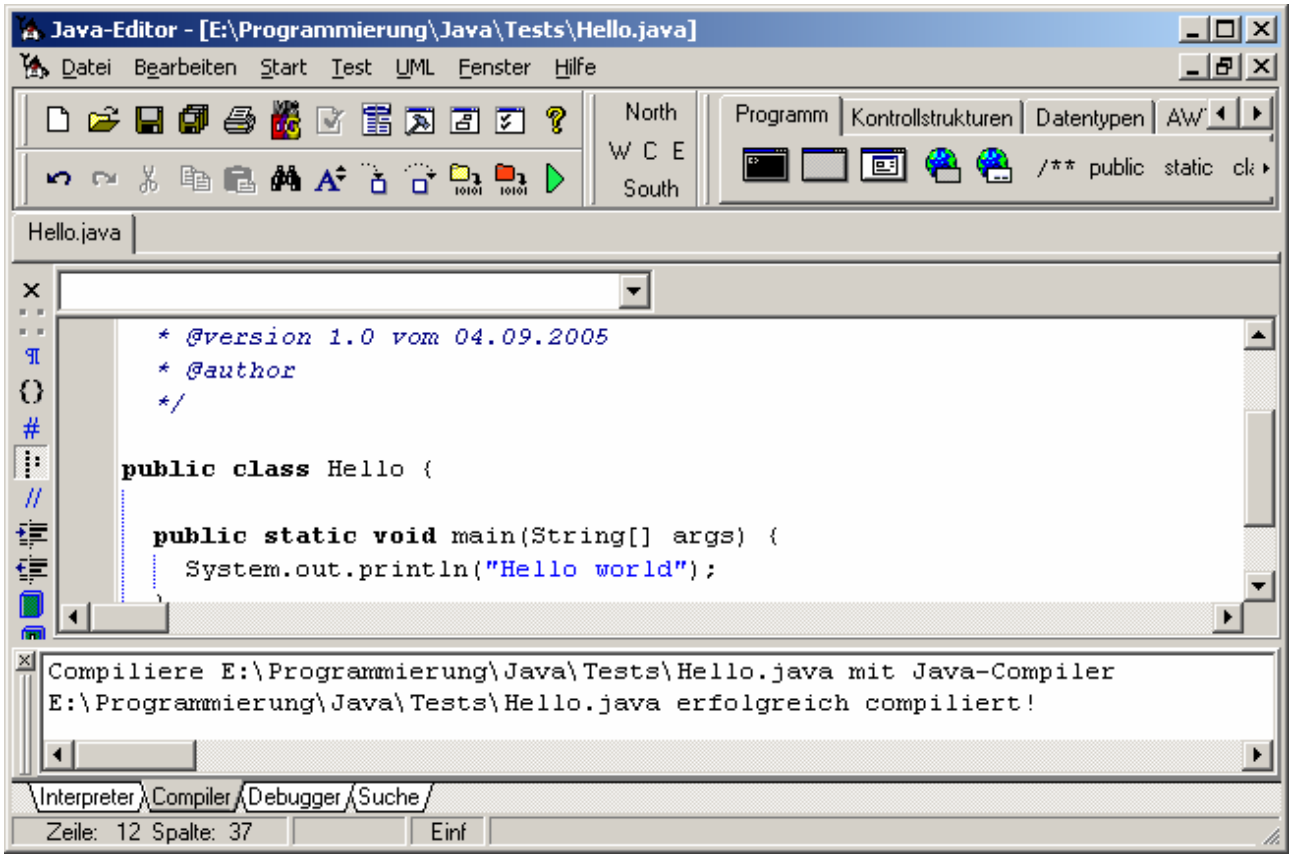
JAVA - Kursmaterial von Gerhard Röhner, Darmstadt
Algorithmen und Datenstrukturen

Wir bearbeiten den Themenkreis *Algorithmen und Datenstrukturen* mit der Programmiersprache Java. Zur Arbeit mit Java benötigt man das Java-Development-Kit (JDK) der Firma Sun. Es steht auf der CD als Windows-Version zur Verfügung und wird als erstes installiert, z. B. nach C:\Programme\Java. Die zugehörige Dokumentation wird in das Unterverzeichnis C:\Programme\Java\docs installiert. Das Tutorial WinHelpTutorial.zip kommt in das Unterverzeichnis C:\Programme\Java\tutorial. Als Alternative zum Java-Compiler javac.exe kopieren wir den schnelleren Compiler jikes.exe in den Ordner C:\Programme\Java\bin. Der CD-Ordner Bücher enthält einige frei verfügbare deutsche Java-Bücher.

Als Entwicklungsumgebung benutzen wir die Hausmarke JavaEditor und installieren dieses Programm in das Verzeichnis C:\Programme\JavaEditor.



Mit dem klassischen „Hello World“-Programm testen wir, ob der Java-Compiler und Interpreter korrekt konfiguriert und die Dokumentation, das JavaBuch und sonstige Hilfen korrekt in das Hilfe-Menü integriert sind.

Konsolenprogramme

Konsolenprogramme sind für die Arbeit auf der Konsole (MS-DOS-Eingabeaufforderung, Unix-Shell) gedacht. Sie verfügen über keine grafische Oberfläche und können deshalb auch außerhalb einer grafischen Umgebung wie Windows oder X11 bzw. KDE (unter Linux) eingesetzt werden. Insbesondere werden Konsolenanwendungen für die automatisierte Datenverarbeitung eingesetzt, bei der Benutzungsoberflächen für die interaktive Arbeit nicht nötig sind.

Die Grundstruktur einer Konsolenanwendung ist im Beispiel auf Seite 1 zu sehen. Innerhalb der Klasse *Hello* wird eine main-Methode deklariert, welche die Funktion des Hauptprogramms übernimmt. Die main-Methode gibt den Text *Hello World!* auf dem System-Ausgabekanal *out* aus. Die Kennzeichnung *public* macht Klassen und Methoden öffentlich zugänglich, *static* weist die main-

Methode als Klassenmethode aus, wodurch sich *main* auch ohne Objekt benutzen lässt. Mit *void* werden Methoden ohne Rückgabewerte gekennzeichnet, sie entsprechen den Prozeduren von Pascal.

Die main-Methode muss immer wie im Beispiel mit dem Parameter *args* deklariert werden. *Args* enthält die so genannten Aufruf-Parameter, mit denen das Programm von der Konsole aus gestartet wurde. Jeder Parameter wird als String übergeben, über einen Index kann auf den i-ten Parameter zugegriffen werden.

Mit dem folgenden Programm machen wir uns mit Aufruf-Parametern und ihrer Verarbeitung vertraut:

```
public class Argumente {
  public static void main (String[] args) {
     System.out.println("Anzahl der Argumente: " + args.length);
     for (int i=0; i < args.length; i++) {
       System.out.println(args[i]);
     }
   }
}
```

Dieses Programm kann mit dem Java-Editor geschrieben, compiliert und gestartet werden. Die Argumente gibt man im Start-Menü unter dem Befehl Parameter ein. Man kann aber auch über Startmenü/Ausführen/command bzw. cmd eine Konsole öffnen und von da aus das Programm als echte Konsolenanwendung starten. Allerdings muss man dabei die Verzeichnispfad für den Compiler, den Interpreter und das Programm angeben.

Der Java-Compiler übersetzt Quell-Programme (.java) in ausführbare Programme (.class-Dateien)

```
C:>C:\Programme\Java\bin\javac C:\Programme\JavaEditor\Argumente.java
```

Hierbei bedeuten

`C:>`	das Konsolenprompt
`C:\Programme\Java\bin\javac`	der Aufruf des Java-Compilers
`C:\Programme\JavaEditor\Argumente.java`	das zu übersetzende Programm

Der Java-Interpreter java.exe führt Java-Programme in .class-Dateien aus:

```
C:>C:\Programme\Java\bin\java -cp C:\Programme\JavaEditor Argumente Arg1 Arg2 Arg3
```

Hierbei bedeuten

`C:\Programme\Java\bin\java`	Aufruf des Java-Interpreters
`-cp C:\Programme\JavaEditor`	Angabe des Classpath, d. h. des Pfades (path) wo sich die ausführbaren Java-Programme (class-Dateien) befinden
`Argumente`	das auszuführende Programm
`Arg1 Arg2 Arg3`	einige Argumente, die vom Programm verarbeitet werden.

Aufgaben

1. Analysiere das Programm Argumente und beschreibe soweit möglich seine Funktionsweise.
2. Rufe den Index der Dokumentation auf und lasse dir die Dokumentation des Klasse *System* anzeigen. Welche weiteren Ein/Ausgabekanäle stehen zur Verfügung?
3. Schreibe eine Konsolenanwendung die zwei als Parameter übergebene Zahlen mittels Integer.parseInt in Integer-Zahlen konvertiert, die Summe berechnet, die Summe mittels Integer.toString in einen String konvertiert und das so erhaltene Ergebnis auf der Konsole ausgibt.
   Variablendeklaration: int Zahl1; int Zahl2; int Summe; String Ergebnis;
   Datenkonvertierung: Zahl1 = Integer.parseInt(args[0]); Schlage die Bedeutung von parseInt und toString in der Hilfe nach.
4. Erweitere das Programm aus 3 so, dass die Summe aller Parameter gebildet wird.

Felder

Der Themenbereich *Algorithmen und Datenstrukturen* gehört zu den klassischen Themen der Informatik. Wir untersuchen verschiedene typische Lösungsstrategien, Standardalgorithmen und abstrakte Datentypen und lernen dabei die Methoden und Konzepte kennen, die für die systematische und erfolgreiche Entwicklung von Softwaresystemen nötig sind. Wir legen los mit *Feldern*.

Ein Feld (engl. array) ist eine Reihe von Elementen gleichen Datentyps. Die einzelnen Elemente können über einen Index angesprochen werden. In Felder kann man eine begrenzte Anzahl gleichartiger Daten speichern. Im Gegensatz zu *einfachen* Datentypen wie int, float, char oder boolean hat ein Feld einen *strukturierten* Datentyp. Die Struktur eines Feldes wird im Bild beispielhaft deutlich.

Feld[0]	Feld[1]	Feld[2]	Feld[3]	...	Feld[i]	Feld[i+1]	...	Feld[n-1]	Feld[n]
17	64	4512	199		46				

Wir sehen ein Feld ganzer Zahlen. Jede Zahl befindet sich in einer Zelle des Feldes. Alle Zellen sind gleich groß und können genau eine ganze Zahl aufnehmen. Nicht alle Zellen des Feldes sind belegt, es können noch weitere Zahlen im Feld gespeichert werden. Das erste Feldelement ist 17 und hat den Index 0 (Merke: Computer zählen ab Null, Menschen zählen ab Eins), das zweite Element ist 64 und kann über den Index 1 angesprochen werden, das letzte Feldelement ist 46.

Deklariert werden Felder durch Kennzeichnung der Feldvariablen mit dem Klammernpaar []. In der Variablen *Elemente* merken wir uns, wie viele Elemente im Feld tatsächlich gespeichert sind. Diese Variable braucht man nicht, wenn das Feld immer komplett gefüllt ist, was häufig in Programmen vorkommt.

```
long[] Feld;
int Elemente;
```

Zum Anlegen eines Feldes benötigt man die *new*-Methode. Im Beispiel wird das Feld zur Aufnahme von maximal 100 Elementen angelegt.

```
Feld = new long[100];
```

Auf ein einzelnes Feldelement greift man über den Index zu:

```
Feld[7] = 15;
for (int i= 7; i < 20; i++) {
  Feld[i]= Math.round(50*Math.random());
  System.out.println(Feld[i]);
}
```

Die *Kapazität* eines Feldes ermittelt man mit der Methode *length:*

```
System.out.println("Maximales Fassungsvermögen: " + Feld.length);
```

Einfügen, Suchen und Löschen eines Elements, sowie Anlegen, Füllen und Sortieren eines Feldes sind Standardoperationen auf Feldern. Mit einem Java-Programm sollen diese Operationen realisiert werden.

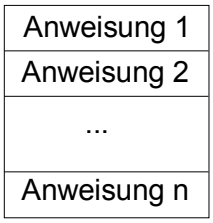
Aufgabe:

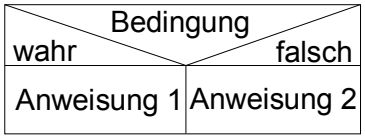
1. Schreibe ein Konsolenprogramm das ein Feld der Kapazität 20 mit 15 Zufallszahlen füllt und dann ausgibt: die größte Zahl, die kleinste Zahl, den Mittelwert der Zahlen, die Zahl, die dem Mittelwert am nächsten kommt.
2. Schreibe ein Konsolenprogramm das ein Feld mit 100 ganzzahligen Zufallszahlen von 1 und 9 füllt und dann die Häufigkeitsverteilung dieser Zahlen ermittelt und ausgibt.

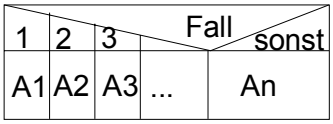
Kontrollstukturen und Struktogramme

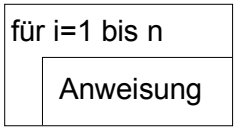
Algorithmen werden häufig im Pseudocode oder als Struktogramm dargestellt. Wir betrachten die Darstellung in *Struktogramm*-Form. Es gibt einige grundlegende Bausteine, die so genannte *Strukturblöcke,* aus denen komplexe Struktogramme durch Reihung (Sequenz) oder Schachtelung zusammengesetzt werden können. Jeder Strukturblock wird von der äußeren Form her als Rechteck dargestellt. Die zugehörige Anweisung wird in das Rechteck geschrieben. Eine *Sequenz* von Anweisungen wird durch Aneinanderreihung der Strukturblöcke dargestellt. Für Kontrollstrukturen und Prozeduraufrufe gibt es jeweils eigene Strukturblöcke.

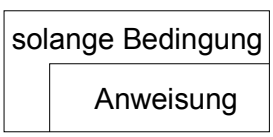
Strukturblock	Java-Struktur	Kommentar

 Sequenz	*Block* in geschweiften Klammern
	`{`
	`Anweisung1;`
	`Anweisung2;`
	`...`
	`AnweisungN;`
	`}`
		Eine Folge von Anweisungen, die alle der Reihe nach abgearbeitet werden, bezeichnet man als *Sequenz.*

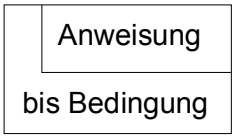
 Fallunterscheidung (bedingte Anweisung)	*if-Anweisung*
	`if (Bedingung) {`
	`  Anweisung1;`
	`}`
	`else {`
	`  Anweisung2;`
	`}`
		Mit einer Anweisung der Form
		Wenn Bedingung erfüllt
		dann führe Anweisung 1 aus
		sonst führe Anweisung 2 aus
		führt man eine *Fallunterscheidung* durch.

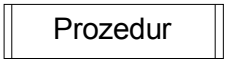
 Mehrfachauswahl	*switch-Anweisung*
	`switch (Ausdruck) {`
	`  case Wert1 :`
	`  Anweisung1;`
	`  break;`
	`  case Wert2:`
	`  Anweisung2;`
	`  break;`
	`  default:`
	`  AnweisungN;`
	`}`
		Mehrfachauswahl
		Der Ausdruck muss ganzzahlig sein. Das Programm wird an der case-Anweisung fortgesetzt, deren Wert dem Ausdruck entspricht. Falls Ausdruck keinem der Werte entspricht, geht es mit der default-Anweisung weiter.

 gezählte Schleife for-Schleife	*for-Schleife*
	`for (int i=1; i<=n; i++){`
	`  Anweisung;`
	`}`
		Eine Anweisung der Form
		für Zähler = Anfang bis Ende
		Anweisung
		heißt *gezählte Schleife*. Gezählt Schleifen werden dann benutzt, wenn man weiß, wie oft eine Schleife durchlaufen werden muss.

 while-Schleife	*while-Schleife*
	`while (Bedingung) {`
	`  Anweisung;`
	`}`
		Eine Anweisung der Form
		Solange Bedingung erfüllt
		führe Anweisung aus
		heißt *Schleife mit Eingangsbedingung*. Trifft die Bedingung anfangs nicht zu, so wird die Wiederholungsanweisung nicht

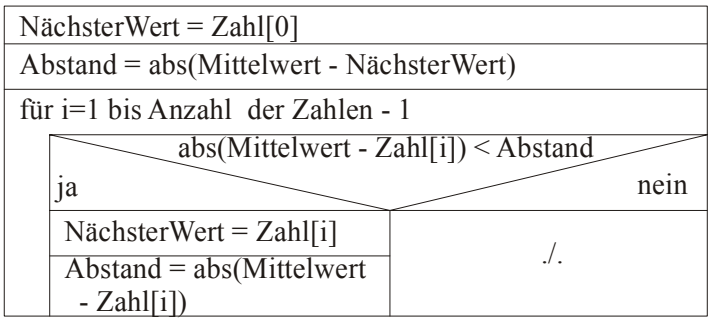
		ausgeführt.


repeat-Schleife
	do-while-(repeat)-Schleife
	
	`do {`
	`  Anweisung;`
	`} while (Bedingung);`
		Eine Anweisung der Form
		Wiederhole Anweisung
		solange Bedingung erfüllt
		heißt *Schleife mit Ausgangsbedingung*. Im Unterschied zur While-Schleife wird die zu wiederholende Anweisung mindestens einmal ausgeführt.

 Prozeduraufruf	*Prozeduraufruf* `Prozedurname(Arg1, Arg2,..., ArgN);`	Prozeduren werden über ihren Namen aufgerufen. In Klammern kann man Argumente übergeben.

Zum Ausfüllen der Strukturblöcke benutzt man nicht die Programmiersprache, in der der betreffende Algorithmus programmiert wird, denn Algorithmen sind zeitlos, Programmiersprachen hingegen dem raschen Wechsel unterworfen (z.B. Basic, Pascal, Modula, Oberon, Java im Bereich Schule).

Ein Algorithmus zur Berechnung der Zahl, die dem Mittelwert am nächsten kommt (vgl. Aufgabe 2, Felder) kann im Struktogramm wie folgt dargestellt werden:

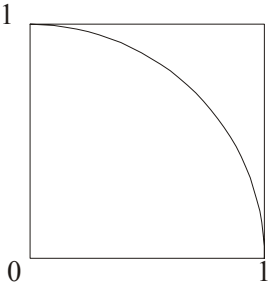


Aufgaben

1. Stelle Algorithmen zur Lösung der Aufgaben 1 und 2 auf Seite 3 als Struktogramm dar.
2. a) Zeichne mit einer Software deiner Wahl ein ordentliches Struktogramm.
   b) Bei Wikipedia.de findest du beim Stichwort „Struktogramme“ spezielle Struktogramm-Editoren. Erzeuge damit ein Struktogramm deiner Wahl.
3. Wandle die folgende gezählte Schleife in eine while-Schleife um:

```
for ( int i= 0 ; i < 15 ; i++ ) {
  Feld[i]= Math.round(10*Math.random());
  System.out.println(Feld[i]);
}
```

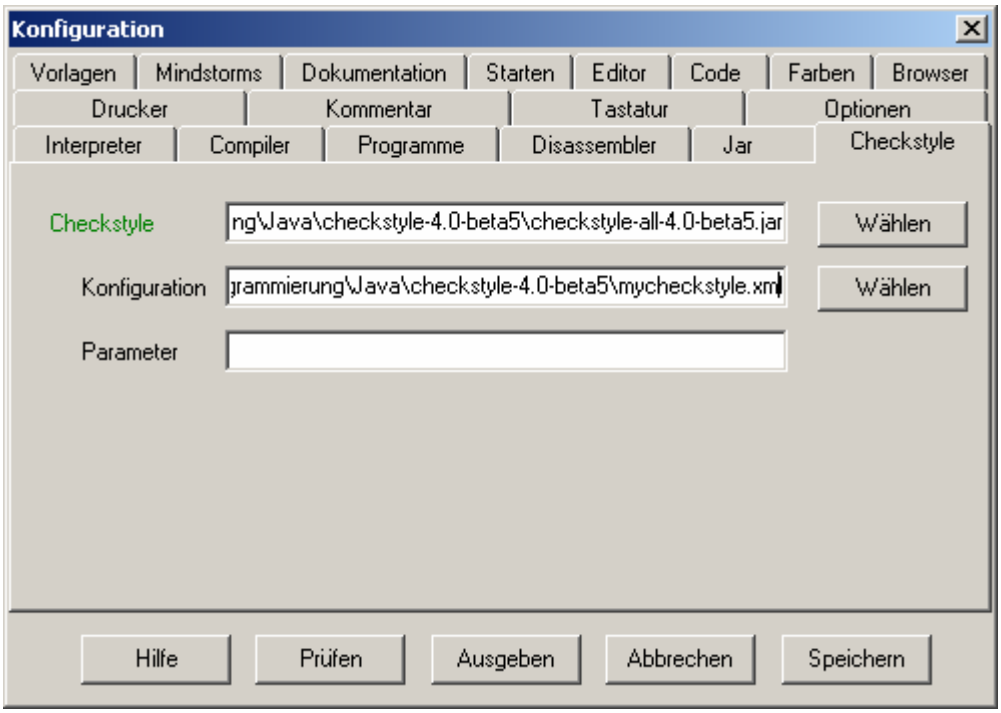
4. Warum ist der Begriff „*if-Schleife*“ ausgemachter Kokolores?
5. Wie kann man mit Hilfe von Math.random() ganze Zufallszahlen von 13 bis 19 erzeugen?
6. Die Monte-Carlo-Methode: Mit Zufallszahlen kann man die Zahl π näherungsweise berechnen. Man erzeugt per Zufall zwei Werte x und y die den Punkt P(x/y) repräsentieren. Liegt der Punkt im Viertelkreis (x^2 + y^2 <= 1) so wird er gezählt sonst nicht. Das Verhältnis der gezählten zu den insgesamt erzeugten Punkten entspricht dem Verhältnis des Viertelkreises zum Quadrat, also $r^2/4/r^2 = \pi/4$. Multipliziert man das ermittelte Verhältnis mit 4 ergibt sich ein Schätzwert für π. Probiere mit unterschiedlich vielen Zufallspunkten.



CheckStyle

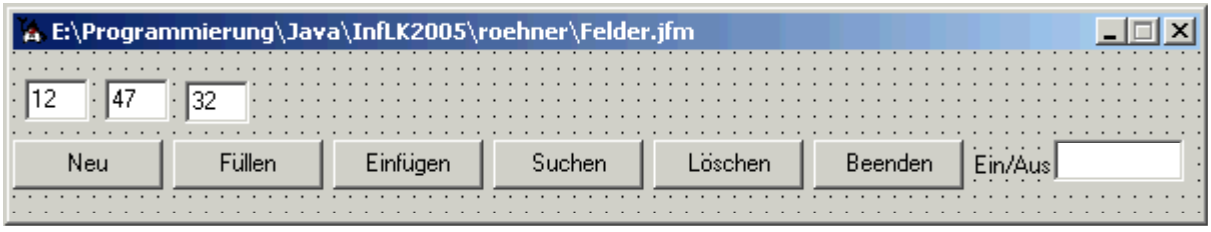
Ein einheitlicher Schreibstil beim Schreiben von Programmen erleichert das Lesen von Programmcode. Dies gilt insbesondere, wenn im Team programmiert wird und man auch fremde Programme lesen muss. Programme werden in der Regel einmal geschrieben, aber oft gelesen. Programmcode muss nämlich immer dann wieder gelesen werden, wenn eine Programmerweiterung oder –verbesserung vorgenommen werden soll. Um Programmcode leicht und schnell verstehen zu können, müssen *selbstsprechende* Bezeichner für die Variablen und Prozeduren benutzt werden und der Programmcode muss gut strukturiert sein. Insbesondere muss die Schreibstruktur der logischen Struktur entsprechen. Diese wird durch die verwendeten Kontrollstrukturen bestimmt. Bei der Seqeunz müssen die Anweisungen stets in der gleichen Spalte beginnen, bei Schleifen und Fallunterscheidungen müssen die inneren Anweisungen eingerückt werden.

Das in den Java-Editor integrierbare Programm *CheckStyle* nimmt eine Stilprüfung von Java-Programmen vor. Der Stil wird über die Datei *mycheckstyle.xml* konfiguriert. Auf der Java-CD befindet sich das ZIP-Paket von CheckStyle, das in den Java-Ordner entpackt werden muss. Auf der Registerkarte *Checkstyle* des Konfigurationsfensters ist die zu verwendende jar-Datei auszuwählen und die Stildatei mycheckstyle.xml anzugeben:



AWT-Programmierung

Als Konsolenprogramm lassen sich die Grundoperationen auf Feldern schlecht darstellen. Wir erstellen daher mit den AWT(Abstract Window Toolkit)-Komponenten eine GUI (Graphical User Interface)-Anwendung. Die grafische Oberfläche soll wie in folgendem Bild aussehen:



Im oberen Teil des Anwendungsfensters sind drei von 15 Textfelder dargestellt. Da es mühsam ist, 15 Textfelder einzeln zu platzieren und zu verwalten, werden wir die Textfelder per Programm erzeugen und im Fenster positionieren. Im unteren Teil befinden sich die zur Arbeit mit dem Feld benötigten Schalter. Rechts neben den Schaltern gibt es noch eine TextField-Komponente, um eine Zahl bequem ein- und ausgeben zu können.

Die einfachste GUI-Applikation wird durch die Frame-Komponente der Komponentenleiste *Programm* erzeugt und sieht wie folgt aus:

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

/**
  * @version 1.0 vom 13.09.2005
  * @author Gerhard Röhner
  */

public class FrameDemo extends Frame {
  // Anfang Variablen
  // Ende Variablen

  public FrameDemo(String title) {
    // Frame-Initialisierung
    super(title);
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent evt) { System.exit(0); }
    });
    int frameWidth = 300;
    int frameHeight = 300;
    setSize(frameWidth, frameHeight);
    Panel cp = new Panel(null);
    add(cp);

    // Anfang Komponenten
    // Ende Komponenten

    setVisible(true);
  }

  // Anfang Ereignisprozeduren
  // Ende Ereignisprozeduren

  public static void main(String[] args) {
    new FrameDemo("FrameDemo");
  }
}
```

Über die import-Direktiven haben wir direkten Zugriff auf die AWT-Komponenten. Die Klasse *FrameDemo* wird von der *Frame*-Klasse abgeleitet (**extends** Frame). Ein Frame ist ein Windows-Fenster mit Titelleiste und Systemschaltern zum Minimieren, Wiederherstellen und Schließen und entspricht einem Programmformular in Delphi. Die Kommentare `// Anfang Variablen` und `// Ende Variablen` **müssen** unangetastet bleiben, weil der Java-Editor diese Kommentare benutzt, um in diesen Bereich Variablendeklarationen für AWT-Komponenten unterzubringen. Analoges gilt für die Kommentare `//Anfang/Ende Komponenten` sowie `// Anfang/Ende Ereignisprozeduren.`

Das Hauptprogramm erzeugt mittels `new FrameDemo("FrameDemo")` das Programmformular. Der sogenannte Konstruktor `FrameDemo`, mit dem das Formular erzeugt wird, setzt mittels `super(titel)` den Fenstertitel, erstellt einen Ereignisprozedur für das Schließen des Fensters, setzt die Fenstergröße auf 300 x 300 und erstellt ein Panel für den Inhalt des Fensters. Ab `// Anfang Komponen-`

ten werden die Komponenten in das Programm eingefügt und die gewünschte Benutzungsoberfläche aufgebaut.

Mit dem GUI-Designer können die Schalter im unteren Bereich erzeugen. Setzt man einen Schalter auf ein Formular, so wird in den drei Abschnitten *Variablen*, *Komponenten* und *Ereignisprozeduren* entsprechender Quellcode erzeugt. In der Variablendeklaration wird der Schalter *Neu* vom Typ *Button* deklariert und mittels ***new*** *Button();* erzeugt.

```
// Anfang Variablen
private Button bNeu = new Button();
```

Im Abschnitt Komponenten werden die Größe und Beschriftung gesetzt, der Schalter dem Programm-Panel cp (content-panel) zugeordnet und eine Ereignisprozedur für das Anklicken des Schalters implementiert.

```
bNeu.setBounds(0, 44, 75, 25);
bNeu.setLabel("Neu");
cp.add(bNeu);
bNeu.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    bNeuActionPerformed(evt);
  }
});
```

Die Ereignisprozedur ruft die Prozedur *bNeuActionPerformed* auf, welche im Abschnitt *// Ereignisprozeduren* erzeugt wird. In dieser Prozedur wird programmiert, was beim Anklicken des Schalters *Neu* gemacht werden soll:

```
// Anfang Ereignisprozeduren
public void bNeuActionPerformed(ActionEvent evt) {

}
```

Es wäre mühsam, die fünfzehn TextField-Komponenten im oberen Teil des Formulars manuell anzulegen. Einfacher geht es mit einer for-Schleife, die die TextField-Komponenten erzeugt und auf dem Formular platziert.

```
// Anfang Komponenten
for (int i=0; i<15; i++) {
  TextField einTextFeld = new TextField("");
  einTextFeld.setBounds(5 + i*40 , 10, 40, 25);
  cp.add(einTextFeld);
}
```

Der Nachteil dieses Ansatzes ist, dass wir nach der Anzeige im GUI-Formular keinen Zugriff mehr auf die Textfelder haben. Die Variable *einTextFeld* ist nur in der Schleife deklariert und kann auch nur ein Textfeld bezeichnen. Zur Verwaltung von 15 Textfeldern brauchen wir ein Feld, wobei jedes Element des Feldes ein Textfeld ist:

```
// Anfang Variablen
private TextField[] textFeld = new TextField[15];
```

Die for-Schleife wird wie folgt ergänzt, um die einzelnen Textfelder im Feld für den späteren Zugriff zu speichern:

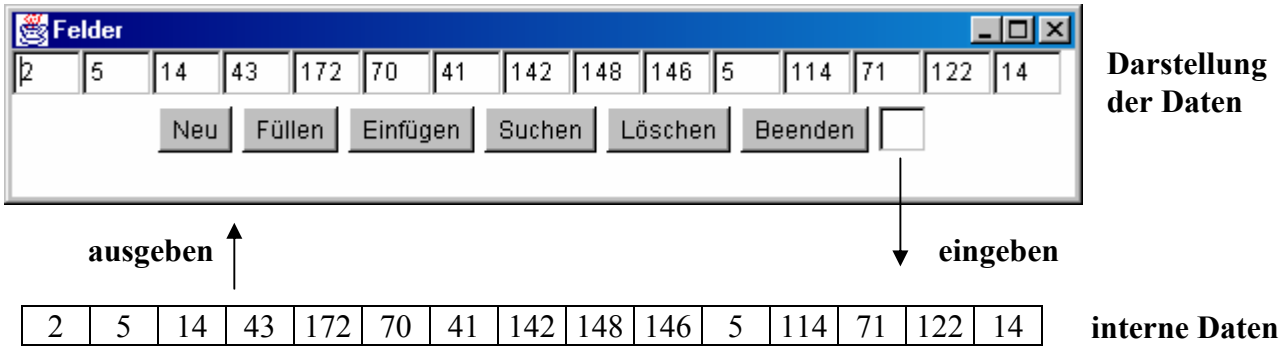
```
textFeld[i] = einTextFeld;
```

Aufgaben

1. Erstelle das Programmformular für das Felder-Programm.

Trennung von Daten und ihrer Darstellung

Die heutigen grafischen Benutzungsoberflächen sind darauf eingerichtet, die Daten eines Programms möglichst flexibel darstellen zu können. Es gibt hochspezialisierte Komponenten, zum visuellen Aufbereiten von Daten. Zum Beispiel lassen sich in Excel Zahlen mit oder ohne Tausenderpunkte, als Währung (€ oder DM) oder gar als Kalenderdatum darstellen. Wir benutzen TextField-Komponenten, um Zahlen eines Feldes darzustellen. Mit der eigentlichen Verarbeitung der Daten haben die visuellen Komponenten in der Regel nichts zu tun. Sie beschränken sich im Wesentlichen auf die Ein- und Ausgabe der Daten. Die Verarbeitung findet auf der internen Datenebene statt. Auf dieser Ebene befinden sich unsere Daten in einem int-Feld.



Für die bequeme Arbeit mit diesen beiden Ebenen ist es sinnvoll, Prozeduren zu schreiben, die für den Datenaustausch sorgen. Beispielsweise gibt die Ausgabe alle Zahlen des internen Feldes in den TextField-Komponenten aus. Die Eingabe aus dem Textfeld rechts neben dem Beenden-Schalter lässt sich mit folgender Methode realisieren:

```
public int eingeben() {
  // Gibt den Wert im Eingabefeld zurück
  return Integer.parseInt(tfEingabe.getText());
}
```

Es ist eine parameterlose Funktion, die einen int-Wert liefert, also einem Wert, der auf der Datenebene sofort verarbeitet werden kann. Bei der Arbeit mit dieser Funktion stellt man fest, dass sie empfindlich gegenüber Fehleingaben wie z.B. leeres Eingabefeld, unzulässiges Zahlenformat oder nicht numerische Zeichen in der Eingabe ist. Es erscheinen so genannte NumberFormatExceptions (engl. exception = Ausnahme). Zum Abfangen solcher Eingabefehler setzt man die *try*-Anweisung ein. Sie sorgt dafür, dass auftretende Fehler erkannt und sinnvoll behandelt werden können.

```
public int eingeben() {
  // Gibt den Wert im Eingabefeld zurück
 try {
   return Integer.parseInt(tfEingabe.getText());
 } catch (NumberFormatException e) {
   return -1;
 }
}
```

Aufgaben

1. Erweitere den Vaiablenbereich um ein Feld von int-Zahlen der Kapazität 15. Die Zahl der tatsächlich sich darin befindlichen Zahlen soll in der Variablen *Elemente* gespeichert werden.
2. Schreibe eine Prozedur Ausgabe, welche die im Feld gespeicherten Zahlen in den Textfeldern ausgibt.
3. Beim Füllen sollen soviele Zufallszahlen im Feld erzeugt und dann angezeigt werden, wie im Eingabe-Textfeld angegeben werden.

Lösungen

Aufgabe 1:

```
// Anfang Variablen
private int[] feld = new int [15];
private int elemente;
```

Aufgabe 2:

```
public void ausgeben() {
  for (int i = 0; i < elemente; i++) {
    textFeld[i].setText(Integer.toString(Feld[i]));
  }
  // Rest des Feldes löschen
  for (int i = elemente; i < 15; i++) {
    textFeld[i].setText("");
  }
}
```

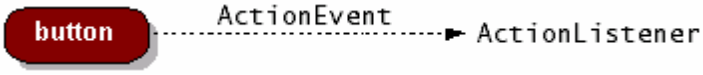
Aufgabe 3:

```
public void bFuellenActionPerformed(ActionEvent evt) {
  elemente = Eingabe();
  for (int i = 0; i < elemente; i++) {
    feld[i] = (int) Math.round(1 + 10 * Math.random());
  }
  ausgeben();
}
```

Schalter - Ereignis - Ereignismethode

Beim Anklicken eines Schalters wird ein Ereignis (engl. event) ausgelöst, das zur Unterscheidung von anderen Ereignissen *ActionEvent* genannt wird. Da der Anwender natürlich keinen realen Schalter klickt, sondern irgendwo auf den Bildschirm, auf dem verschiedene Fenster, möglicherweise sich verdeckend, angezeigt werden, muss Windows aus der geklickten Bildschirmposition das Programm ermitteln, das für den Bildschirmbereich zuständig ist. Es schickt dann das Klickereignis an das zuständige Programm.

Bei einem Java-Programm müssen sich die Objekte, die auf Ereignisse reagieren, anmelden. Ein Schalter tut dies mittels *addActionListener*. Der dabei angegeben ActionListener enthält eigentlich nur die Ereignismethode *actionPerformed*, welche beim Anklicken ausgeführt wird. Für einen leichterer Einstieg in die Java-Programmierung erzeugt der Java-Editor den benötigten Code automatisch:




```
buEinfuegen.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
     buEinfuegenActionPerformed(evt);}});
```

Die automatisch erzeugte Ereignismethode *actionPerformed* ruft eine Methode auf, im Bereich `// Ereignisprozeduren` deklariert wird. Beim Einfügen-Schalter sieht die Implementierung dieser Methode wie folgt aus:

```
public void buEinfuegenActionPerformed(ActionEvent evt) {
  if (elemente < feld.length ) {
    int zahl = eingabe();
    feld[elemente] = zahl;
    elemente = elemente + 1;
    ausgeben();
  }
  else
    tfEingabe.setText("Feld voll!");
}
```

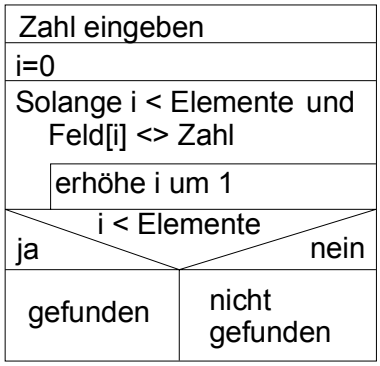
Aufgabe

Schreibe Prozeduren zum Suchen und Löschen einer Zahl.

Lösungen

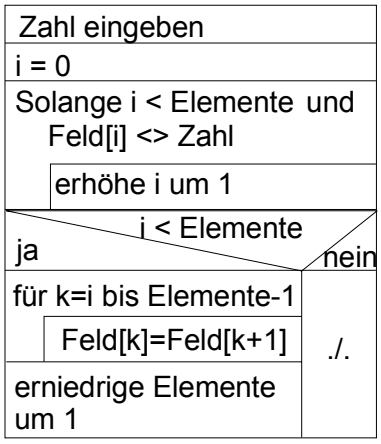
```
public void bSuchenActionPerformed(ActionEvent evt) {
  int zahl = eingabe();
  int i = 0;
  while (i < elemente && feld[i] != zahl) {
    i++;
  }
  if (i < elemente) {
    textFeld[i].setBackground(Color.RED);
  } else {
    tfEinAus.setText("nicht gefunden");
  }
}
```

Algorithmus *Suchen*




```
public void bLoeschenActionPerformed(ActionEvent evt) {
  int zahl = eingabe();
  int i = 0;
  while (i < elemente && feld[i] != zahl) {
    i++;
  }
  if (i < elemente) {
    for (int k = i; k < elemente - 1; k++) {
      feld[k] = feld[k + 1];
    }
    elemente = elemente - 1;
  }
  ausgeben();
}
```

Algorithmus *Löschen*



Zum Weiterdenken

1. Wieso lautet die Bedingung für erfolgreiches Suchen *i* < *elemente* und nicht *feld[i]* = *zahl*, was doch viel verständlicher wäre?
   a) Analysiere das in der Frage angesprochene Problem im Fall eines vollen Feldes.
   b) Im Feld wird wie angegeben eine Zahl gelöscht. Anschließend wird eine Zahl gesucht, die im Feld die Platznummer *Elemente* hat. Welche Konsequenz hat dies in Bezug auf die Eingangsfrage?

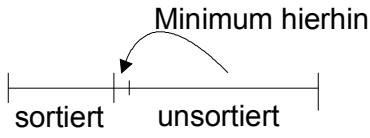
2. Höhere Logik
   a) Solange *i* < *elemente* ***und*** *feld[i]* != *zahl* wird in der Suchschleife i um eins erhöht. Die Schleifenbedingung ist also eine logische Und-Verknüpfung der Form *A* ***und*** *B*. Nach Abarbeitung der Schleife gilt ***nicht*** *(A* ***und*** *B)*. Was bedeutet das im Suchen-Beispiel eigentlich?
   b) Wieso garantiert die Bedingung *i* < *elemente* nach der Schleife, dass *feld[i]* = *zahl?*

3. Algorithmen lassen sich oft auf Kosten von Daten vereinfachen. Vor jedem Suchvorgang schreiben wird die gesuchte Zahl an die erste freie Position hinter dem Feld *feld[elemente]* = *zahl.*
   a) Wie lässt sich die Schleifenbedingung dadurch vereinfachen?
   b) Betrachte den Sonderfall des vollen Feldes!

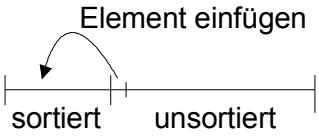
Sortieren

Stellen wir uns vor, das Telefonbuch wäre nicht nach Namen sortiert. Es wäre praktisch unmöglich, eine Telefonnummer zu finden! Sortieren sorgt offenbar dafür, dass man Daten schneller finden kann. Wir betrachten daher im folgenden das Sortieren von Daten eines Feldes, sowie das Suchen und Einfügen in ein sortiertes Feld. Zwei elementare Verfahren sind das Sortieren durch Auswählen bzw. durch Einfügen. Beide Verfahren gehen davon aus, dass zu jedem Zeitpunkt des Sortiervorgangs das zu sortierende Feld in einen schon sortierten und ein noch zu sortierenden Bereich zerlegt werden kann.

- beim *Sortieren durch Auswählen* wählt man bei jedem Sortierschritt das Minimum des unsortierten Bereichs aus und hängt es an den sortierten Bereich an.
- beim *Sortieren durch Einfügen* nimmt man das nächste Element des unsortierten Bereichs und fügt es an der richtigen Stelle in den sortierten Bereich ein.



Sortieren durch Auswählen



Sortieren durch Einfügen

Beispiel: Sortieren durch Auswählen für die Zahlen 38, 17, 25, 1, 7.

38	17	25	*1*	7
1	17	25	38	*7*
1	**7**	25	38	*17*
1	**7**	**17**	38	*25*
1	**7**	**17**	**25**	**38**

Der fett umrahmte Bereich ist jeweils sortiert, das kursiv gesetzte Element das Minimum des unsortierten Bereichs.

Algorithmus *Sortieren durch Auswählen*

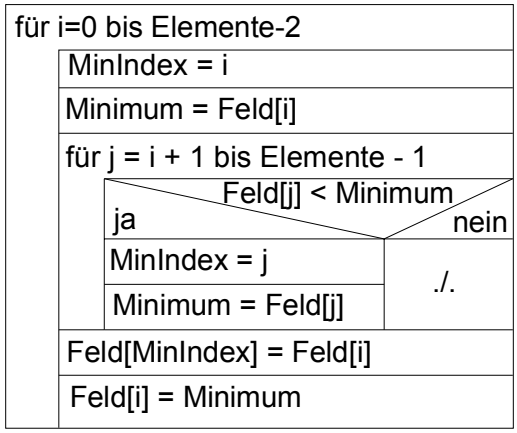
Die Grundidee des Sortierens durch Auswahl muss für die maschinelle Ausführung als Algorithmus ausgeführt werden, der dann in einer Programmiersprache implementiert wird. Kompliziertere Algorithmen kann man im Sinne des *Top-Down-Prinzips* durch schrittweise Verfeinerung entwickeln. Ein erster Ansatz ist:

Das Suchen eines Minimums und Tauschen zweier Feldelemente sind keine Anweisungen, die in einer Programmiersprache direkt zur Verfügung stehen. Also müssen diese Anweisungen weiter verfeinert werden, bis sie auf elementare Anweisungen zurückgeführt sind:

für i=0 bis Elemente-2	
	Suche Index j des Minimums
	Tausche Feld[i] mit Feld[j]

Jetzt kommen nur noch Wertzuweisungen, Schleifen und Fallunterscheidungen vor, also Anweisungen und Kontrollstrukturen, die in imperativen Programmiersprachen vorhanden sind. Die Programmierung ist daher nicht mehr schwer:

```
public void buAuswahlSortierenActionPerformed
              (ActionEvent evt) {
  for (int i = 0; i < elemente - 2 ; i++) {
    int minIndex = i;
    long minimum = feld[i];
    for (int j = i + 1; j < elemente ; j++)
      if (feld[j] < minimum) {
        minIndex = j;
        minimum = feld[j];
      }
    feld[minIndex] = feld[i];
    feld[i] = minimum;
  }
  anzeigen();
}
```



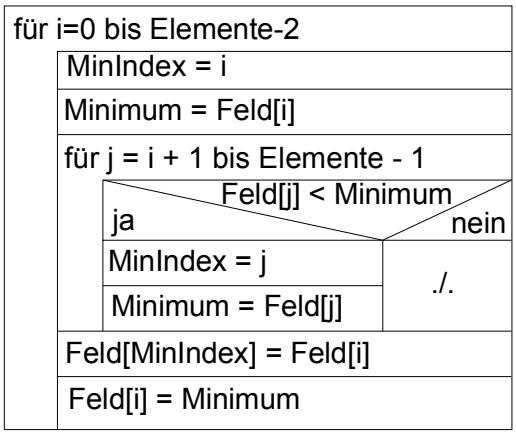
Komplexität

Unter der *Komplexität* eines Algorithmus versteht man den zu seiner Durchführung erforderlichen Aufwand an Betriebsmitteln wie Speicherplatz und Rechenzeit. Bei den Sortierverfahren interessiert insbesondere die Rechenzeit, denn der erforderliche Speicherplatz ist proportional zur Anzahl der Eingangsdaten.

Meistens geht es beim Sortieren darum, eine große Menge von *Datensätzen*, die mehrere verschiedene Informationen enthalten, nach einer dieser Informationen, dem *Schlüssel*, zu ordnen. In einem Warenkatalog hat jedes Kleidungsstück einen Namen, eine Größe, einen Preis und eine Bestellnummer. Als Schlüssel ist hier die Bestellnummer geeignet. Im Telefonbuch für einen Ort ist der Schlüssel der Name der Person, unter dem der Anschluss angemeldet ist.

Um das zeitliche Verhalten von Sortierverfahren zu beurteilen, untersucht man einerseits, wie viele *Vergleiche V* von Feldelementen (oder deren Schlüsseln) von dem jeweiligen Algorithmus benötigt werden und wie viele *Bewegungen B* von Feldelementen notwendig sind. Diese Unterscheidung wird deshalb vorgenommen, weil der Vergleich von Schlüsseln sehr viel schneller durchgeführt werden kann als die Bewegung eines Feldelements, wenn es sich dabei um relativ große Datensätze handelt.

Da die Anzahlen sehr von der Verteilung der zu sortierenden Elemente abhängen kann, gibt es Untersuchungen für den *besten Fall (best case), den Durchschnittsfall (average case) und den schlimmsten Fall (worst case).* Uns soll vor allem der worst case interessieren.



Wir bestimmen zuerst die Anzahl der Vergleiche. Zur Vereinfachung der Schreibweise nehmen wir an, dass die Anzahl der Elemente n beträgt. Beim ersten Durchlauf wird die innere Schleife und damit der Vergleich für j = 1 bis n-1, also n-1 mal ausgeführt. Beim zweiten Durchlauf werden nur noch für j = 2 bis n-1 Vergleiche ausgeführt, also n-2 Vergleiche. Beim letzten Durchlauf gilt für j = n-1 to n-1, also wird der Vergleich noch einmal ausgeführt. Wir ziehen Bilanz:

$$V(n) = n-1 + n-2 + n-3 + \ldots + 3 + 2 + 1$$

Zur Berechnung dieser Summe verdoppeln wir geschickt und halbieren abschließend das Ergebnis:

$$2 \cdot V(n) = n-1 + n-2 + n-3 + \ldots + 3 + 2 + 1$$
$$1 + 2 + 3 + \ldots + n-3 + n-2 + n-1$$

$$2 \cdot V(n) = (n-1) \cdot n$$

$$V(n) = \frac{(n-1) \cdot n}{2}$$

Die Anzahl der Vergleiche ist also quadratisch von n abhängig.

Betrachten wir nun die Anzahl der Bewegungen, das sind alle Wertzuweisungen mit Feldelementen. Solche Wertzuweisungen kommen an den vier mit (1), (2), (3) bzw. (4) gekennzeichneten Stellen vor. Im besten Fall ist das Feld schon sortiert, die Bedingung *Feld[j]<Minimum* ist dann niemals erfüllt. Pro Schleifendurchlauf werden im besten Fall also drei Bewegungen durchgeführt, insgesamt also 3(n-1) Bewegungen. Im schlimmsten Fall ist das Feld in umgekehrter Reihenfolge sortiert. Die Bedingung *Feld[j]<Minimum* ist bei jedem Schleifendurchlauf erfüllt. Es kommen somit durch (4) folgende Bewegungen hinzu: n-1 + n-2 + n-3 + ... + 3 + 2 + 1.

```
für i=0 bis Elemente-2
    MinIndex = i
    Minimum = Feld[i]                    (1)
    für j = i + 1 bis Elemente - 1
        Feld[j] < Minimum
        ja                    nein
        MinIndex = j          ./.
        Minimum = Feld[j] (4)
    Feld[MinIndex] = Feld[i]             (2)
    Feld[i] = Minimum                    (3)
```

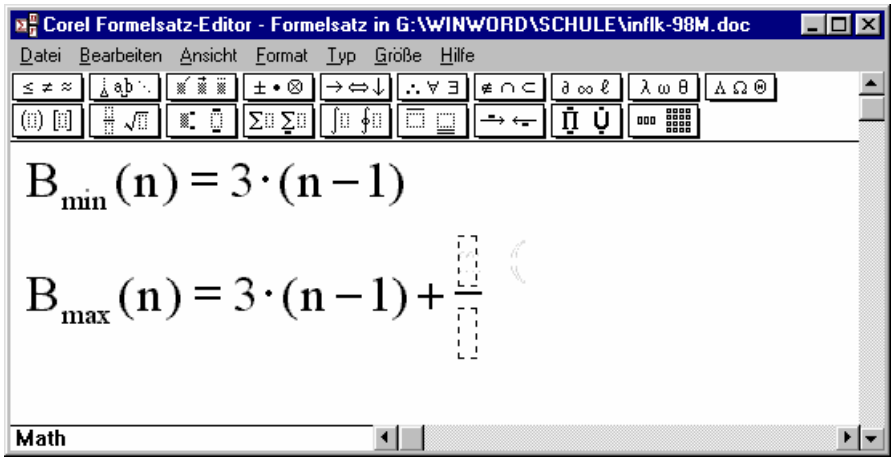
$$B_{min}(n) = 3 \cdot (n-1)$$

$$B_{max}(n) = 3 \cdot (n-1) + \frac{n \cdot (n-1)}{2}$$

Im schlechtesten Fall ist also auch die Anzahl der Bewegungen quadratisch von n abhängig.

Formelsatz

Die obigen Formeln sind mit einem mathematischen Formelsatz-Programm erstellt. Bei Word muss dieses Programm nachträglich mit dem Setup-Programm installiert werden. Über *Einfügen/Objekt* ruft man die Liste der einfügbaren Objekte auf und wählt den Formeleditor aus. Die erstellte Formel wird über *Datei/Beenden* in die Textverarbeitung als Formel übernommen. Per Doppelklick kann sie jederzeit zwecks Bearbeitung in den Formeleditor geladen werden (OLE-Technik = Object-Linking and Embedding)



Algorithmus *Sortieren durch Einfügen*

Das folgende Beispiel zeigt die Arbeitsweise des *Sortierens durch Einfügen*.

38	17	25	1	7
17	**38**	25	1	7
17	***25***	**38**	1	7
1	**17**	**25**	**38**	7
1	***7***	**17**	**25**	**38**

Der fett umrahmte Bereich ist jeweils sortiert, das kursiv gesetzte Element das zuletzt eingefügte.

Ein erster algorithmischer Ansatz liefert:

```
für i= 1 bis Elemente - 1
    Element = Feld[i]
    Füge Element am
    richtigen Platz im
    Bereich 0 bis i-1 ein
```

Das Einfügen kann aufgeteilt werden in das Suchen der Einfügestelle, dem Verschieben der Feldelemente, um an der Einfügestelle Platz zu haben und dem Einsetzen des Elements an der frei gewordenen Stelle. Einfacher ist es aber, Suchen und Verschieben zusammen auszuführen. Dann ergibt sich:

```
für i=1 bis Elemente-1
    Einfuegezahl = Feld[i]
    j = i
    solange (0<j) und (Einfuegezahl<Feld[j-1])
        Feld[j] = Feld[j-1]
        j = j- 1
    Feld[j] = Einfuegezahl
```

In die Komplexitätsbetrachtung gehen nur Vergleiche und Wertzuweisungen (Bewegungen) für Feldelemente ein, Vergleiche und Wertzuweisungen an Index-Variablen hingegen nicht. Index-Variablen können von optimierenden Compilern in schnellen Register des Mikroprozessors verwaltet werden, während Feldelemente beispielsweise lange Strings oder gar Verbunde (Records) sein können, bei denen diese Operationen vergleichsweise sehr aufwendig sind. Es ergibt sich:

$$V_{min}(n) = n - 1 \qquad B_{min}(n) = 2 \cdot (n-1)$$

$$V_{max}(n) = \frac{n \cdot (n-1)}{2} \qquad B_{max}(n) = 2 \cdot (n-1) + \frac{n \cdot (n-1)}{2}$$

Im schlechtesten Fall ist also auch beim Sortieren durch Einfügen die Zahl der Vergleiche und Bewegungen quadratisch von n abhängig.

Lineare Suche

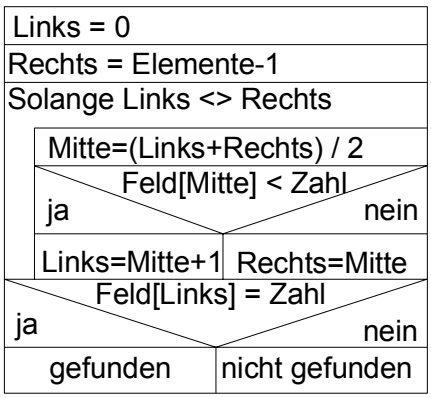
Im unsortierten Feld sucht man Elemente mittels *linearer Suche* (siehe Seite 11) Dieser Begriff rührt daher, dass die mittlere Laufzeit linear mit der Anzahl der Feldelemente wächst. Für die Anzahl der Vergleiche gilt:

$$V_{min}(n) = 1 \quad V_{mittel}(n) = \frac{n}{2} \quad V_{max}(n) = n$$

Binäre Suche

Im sortierten Feld kann man deutlich schneller suchen. Man vergleicht zunächst die zu suchende Zahl mit dem Element in der Mitte des Feldes. Ist die Zahl größer so setzt man die Suche im rechten Teilfeld, ansonsten im linken Teilfeld fort. Im ausgewählten Teilfeld wird die zu suchende Zahl wieder mit dem Element in der Mitte des Teilfeldes verglichen und wie vor beschrieben mit dem linken bzw. rechten Viertel die Suche fortgesetzt.

Algorithmus Suche(Zahl)



Diese grobe Beschreibung der binären Suche ist weder für einen Menschen, geschweige denn für eine Maschine ausreichend präzise. Lediglich für einen Menschen wird die prinzipielle Idee deutlich. Unklar ist beispielsweise, was genau mit dem linken Teilfeld, dem rechten Viertel oder der Mitte gemeint ist. Zur Präzisierung des Gemeinten beschreiben wir Teilfelder durch die zwei Indizes *Links* und *Rechts*, welche die Indexgrenzen des jeweils zu betrachtenden Teilfeldes präzise festlegen. Die Feldmitte kann dann durch den mathematischen Ausdruck *(Links + Rechts) / 2* berechnet werden.

So einfach dieser Algorithmus auch erscheinen mag, er hat es in sich. Haben wir da nicht potentiell eine Endlosschleife programmiert bzw. ist stets gesichert, dass nach einigen Schleifendurchläufen *Links = Rechts* gilt? Könnten wir nicht analog zu Links besser *Rechts = Mitte -1* nehmen?

Aufgabe

1. Ergänze im Formular eine Label-Komponente, welche jeweils die Anzahl der im Feld vorhandenen Elemente anzeigt.
2. Implementiere den Algorithmus zur binären Suche.

Komplexität der binären Suche

Wir haben die Anzahl der Vergleiche V in Abhängigkeit von der Anzahl n der im Feld befindlichen Elemente zu bestimmen, also V(n). Vergleiche treten innerhalb der While-Schleife auf, zum Schluss kommt noch ein weiterer Vergleich hinzu. Die Anzahl der Vergleiche innerhalb der Schleife stimmt mit der Anzahl der Schleifendurchläufe und damit mit der Anzahl der Halbierungen überein. Wir nennen diese Anzahl k. Die Schleife terminiert, wenn *Links = Rechts* gilt, wenn die Breite B des Suchbereichs also 1 ist. Wir stellen die Breite B des Suchbereichs als Funktion der Halbierungen k in Form einer Wertetabelle zusammen:

Anzahl k der Halbierungen	0	1	2	3	4	5	6	7	8
Breite B des Suchbereichs	n	n/2	n/4	n/8	n/16	n/32	n/64	n/128	n/256

Offenbar gilt $B(k) = \frac{k}{2^k}$ aus der Bedingung B(k) = 1 folgt: $\frac{n}{2^k} = 1$

Löst man diese Gleichung durch Logarithmieren nach der gesuchten Anzahl k auf, so ergibt sich: $k = \log_2(n)$. Zusammen mit dem Vergleich am Ende des Such-Algorithmus haben wir also

$$V(n) = \log_2(n) + 1$$

Ist n keine Zweier-Potenz, so liefert $\log_2(n)$ keine ganze Zahl. Beispielsweise erhalten wir für n=7, $\log_2 7 = 2{,}807354922058$. Wir brauchen die aufgerundete Zahl. Dafür schreibt man:

$$V(n) = \lceil \log_2(n) \rceil + 1$$

Einfügen in ein sortiertes Feld

Sucht man im sortierten Feld mit der binären Suche und ist die zu suchende Zahl x im Feld nicht enthalten, so ist zuletzt die nächst größere Zahl markiert. Das ist genau die Stelle p an der die neue Zahl eingefügt werden muss. Man verschiebt die größeren Zahlen um eine Position nach rechts und fügt x mit Feld[p]= x in das Feld ein. Was aber, wenn es keine nächst größere Zahl gibt, wenn also x größer als alle Feldelemente ist. Dann bleibt unser Algorithmus bei der letzten Zahl im Feld stehen. Dort darf die einzufügende Zahl x aber nicht eingefügt werden. Mit einer Fallunterscheidung lässt sich dieses Problem lösen, einfacher ist es aber, den Suchalgorithmus mit Rechts = Elemente zu beginnen. Wieso gibt das keine Probleme?

Schnellere Sortierverfahren wie Mergesort und Quicksort benutzen *Rekursion*. Rekursive Algorithmen lernen wir später am Beispiel rekursiver Grafiken kennen.

Aufgabe

1. Ergänze die Operation „Einfügen in ein sortiertes Feld“ in dein Feld-Programm.

Nachtrag zur Komplexitätsbetrachtung

Beim Sortieren durch Auswählen ist die durchschnittliche Anzahl der Bewegungen nicht gleich dem arithmetischen Mittel von $B_{min}(n)$ und $B_{max}(n)$. Um das prinzipiell zu verstehen betrachten wir das Beispiel n = 3. Hier gilt für die innere Schleife, dass wir minimal 0 und maximal 2 Bewegungen haben, das arithmetische Mittel ergibt also 1. Tatsächlich können die Feldelemente sechs verschiedene Reihenfolgen aufweisen.

Feld[1] < Feld[2] < Feld[3] B=0
Feld[1] < Feld[3] < Feld[2] B=0
Feld[2] < Feld[1] < Feld[3] B=1
Feld[2] < Feld[3] < Feld[1] B=1
Feld[3] < Feld[1] < Feld[2] B=1
Feld[3] < Feld[2] < Feld[1] B=2

Die B-Werte geben an, wie oft Bewegungen jeweils stattfinden. Im Schnitt haben wir also (0+0+1+1+1+2)/6 Bewegungen, also lediglich 5/6 statt 1. Mit einigem (universitären) mathematischen Aufwand erhält man das Ergebnis: $B_{avg}(n) = n \cdot (\ln n + g)$ wobei g die Eulersche Konstante g = 0,577216... ist. $B_{min}(n)$ ist also linear, $B_{max}(n)$ quadratisch und $B_{avg}(n)$ mit n*ln n von n abhängig.

Objektorientierte Programmierung

Die Entwicklung guter Software gelingt nur unter Beachtung von Qualitätsmerkmalen und Einsatz entsprechender Werkzeuge und Methoden. Die Bedeutung der Qualitätsmerkmale hat sich im Laufe der Zeit geändert. Programme, die auf den Rechnern der 1. und 2. Generation laufen sollten, mussten eine gute Speicher- und Zeiteffizienz aufweisen. Dies wurde oft durch trickreiche und kunstfertige Programmierung erreicht. Die Entwicklung besserer Hardware erlaubte es, komplexere Anwendungen anzugehen, welche nicht mehr von einer einzelnen Person bearbeitet werden konnten. Mit den benutzten Entwicklungsmethoden wurden die Programme immer fehlerhafter, was um 1965 zur *Softwarekrise* führte.

Die Qualitätsmerkmale *Zuverlässigkeit, Korrektheit* und *Robustheit* rückten ins Zentrum des Interesses. Erreicht werden sollen diese Qualitätsmerkmale durch die *strukturierte Programmierung* auf der Basis des Top-Down-Prinzips und der Methode der schrittweisen Verfeinerung.

Wenn Programme immer größer werden, lassen sich Fehler auch bei strukturierter Programmierung nicht vermeiden. Daher ist es wichtig, dass man auftretende Fehler rasch analysieren und beheben kann. Dies gelingt dann gut, wenn die Software das Qualitätsmerkmal der *Wartbarkeit* (ab 1970) erfüllt. Die Wartbarkeit wird durch das Modulkonzept, das *Geheimnisprinzip* und die *Kapselung* unterstützt.

Eine weitere defensive Maßnahme zur Erhöhung der Zuverlässigkeit besteht in der Wiederverwendung bewährter Module. Dies setzt allerdings voraus, dass Softwaremodule so entworfen werden, dass sie dem Qualitätsmerkmal der *Wiederverwendbarkeit* entsprechen. Wiederverwendbare Module müssen kombinierbar, verständlich, stetig erweiterbar und geschützt sein. Hier greift die *objektorientierte Programmierung*, die mit Klassen und Objekten, sowie Vererbung und Polymorphismus geeignete Konzepte und Methoden zur Realisierung der Wiederverwendbarkeit bereit stellt.

Der objektorientierte Ansatz verspricht für die heutige Softwareentwicklung ähnliche Bedeutung zu erlangen wie die strukturierte Programmierung in den vergangenen Jahrzehnten. Er setzt auf den bislang entwickelten Konzepten des Software-Engineerings auf und entwickelt sie konsequent weiter. Die strukturierte Programmierung bleibt dabei als wichtige Methode erhalten, sozusagen als Grundvoraussetzung für eine ordentliche Programmierung. Die objektorientierte Softwareentwicklung erlaubt es nunmehr, Software übersichtlicher zu strukturieren und sie aus wiederverwendbaren Bausteinen zu konstruieren.

Klassen und Objekte

Objektorientierung ist eigentlich etwas ganz Normales, was täglich benutzt wird, um mit der Komplexität der Umwelt zurechtzukommen. Wir betrachten die Welt als eine Menge von Objekten, die zueinander in Beziehung stehen und gegebenenfalls miteinander kommunizieren. Geht man beispielsweise im Supermarkt einkaufen, so wird man sich für die meisten Menschen nur insoweit interessieren, dass man sich mit den Einkaufswagen nicht gegenseitig behindert und in der Warteschlange vor einer Theke nicht vorgedrängelt wird. Man nimmt die Menschen nur als Ganzes - als Objekt - wahr und interessiert sich nicht für die Person. Die Marktleitung sieht in den Menschen im wesentlichen Kunden. Dabei wird ebenfalls von den Eigenheiten der einzelnen Individuen abstrahiert und auf das Wesentliche reduziert: ein Kunde ist ein Mensch der im Supermarkt einkauft. Der Begriff Kunde ist daher eine Abstraktion für einen einkaufenden Menschen. In objektorientierter Terminologie spricht man von der Klasse Kunde.

Die wichtigsten Bestandteile eines objektorientierten Programms sind *Objekte* und *Klassen*. *Objekte* sind Elemente, die in einem Anwendungsbereich von Bedeutung sind. In einem Textverarbeitungsprogramm beispielsweise könnte ein Objekt einen bestimmten Absatz repräsentieren. In einer Finanzbuchhaltung ist die Bilanz des Monats November ein mögliches Objekt und bei einem Auftragsverwaltungssystem wird man es mit Objekten wie Kunden und Aufträgen zu tun haben.

Aus der Sicht des Benutzers stellen Objekte bestimmte Dienstleistungen und Informationen zur Verfügung. Ein Aufzug beispielsweise besitzt die Funktionalität, auf- und abwärts zu fahren, sowie Informationen über die augenblickliche Position oder anzufahrende Stockwerke. Aus der Sicht des Programmierers sind Objekte Funktionseinheiten eines Programms, die zusammenarbeiten, um eine gewünschte Funktionalität zur Verfügung zu stellen.

Eine *Klasse* entsteht durch die Abstraktion von den Details gleichartiger Objekte und beschreibt die Eigenschaften und Struktur einer Menge nahezu gleicher Objekte. Gleichartige Objekte werden also durch eine gemeinsame Klasse zusammengefasst und beschrieben. Die Objekte sind *Exemplare* dieser gemeinsamen Klasse und werden manchmal auch Instanzen genannt. Die Möglichkeit, von einer Menge ähnlicher Objekte Gemeinsamkeiten zu abstrahieren und in einer Klasse allgemein zu beschreiben, ist ein markantes Merkmal objektorientierter Systeme.

Eine Klasse ist die Beschreibung gleichartiger Objekte.

Attribute und Methoden

Die gleichberechtigte Betrachtung von Daten und Funktionen innerhalb der objektorientierten Programmierung kommt durch den Aufbau von Objekten zur Geltung. Jedes Objekt besteht aus Attributen und Methoden.

Die Attribute stellen den Datenbereich eines Objekts dar. In der zugehörigen Klasse wird die Datenstruktur dieses Datenbereichs beschrieben. Als Attribute der Klasse *Kunde* könnte man beispielsweise Name, Adresse und Bonität verwenden. Objekte dieser Klasse werden dann durch bestimmte Wertbelegung charakterisiert. Diese Wertbelegung legt den Zustand von Objekten fest.

Methoden stellen die Operationen dar, mit denen Objekte manipuliert werden können. Sie kommen in Form von Prozeduren und Funktionen vor. Über Prozeduren kann der Zustand von Objekten, charakterisiert durch die Attributwerte, geändert werden. Mit Funktionen lässt sich Auskunft über den Zustand von Objekten einholen.

Klassen in Java

Als einfaches, überschaubares Beispiel nehmen wir eine Klasse für Rechtecke. Klassen werden mittels *class* deklariert und in eigenen Dateien gespeichert. Auf das Schlüsselwort *class* folgt der Name der Klasse, welcher gleichzeitig Dateiname sein muss.

Ein Rechteck kann durch zwei gegenüberliegende Punkte P1(x1, y1) und P2(x2, y2) festgelegt werden. Die Koordinaten werden als Attribute der Klasse vor den Methoden aufgeführt.

```
class Rechteck {
  protected int x1, y1;
  protected int x2, y2;

  public Rechteck(int _x1, int _y1, int _x2, int _y2) {
    x1 = _x1;
    y1 = _y1;
    x2 = _x2;
    y2 = _y2;
  }

  public void zeigen() {
    System.out.println("Von: " + x1 + " " + y1);
    System.out.println("Bis: " + x2 + " " + y2);
  }

  public double gibFlaeche() {
    return Math.abs((x2-x1)*(y2-y1));
  }
}
```

In dieser Ausbaustufe gibt es einen Konstruktor (Rechteck) zum Erzeugen eines Rechtecks und zwei Methoden (zeigen und gibFlaeche). Der Konstruktur hat den gleichen Namen wie die Klasse. Über die Parameter des Konstruktors kann ein neues Rechteck mit Anfangswerten versehen werden. Die Methode *zeigen* gibt auf der Konsole Anfangs- und Endpunkt des Rechtecks aus, es handelt sich um eine Prozedur. Die Methode *gibFlaeche* ist als Funktion programmiert, die in einem *double*-Wert die Fläche des Rechtecks zurück gibt. Zur Berechnung wird die Betragsfunktion benutzt. Auf die mathematischen Funktionen hat man über die Klasse *Math* Zugriff.

Über die Klasse werden allgemeine Eigenschaften von Rechtecken modelliert. (diagonal gegenüberliegende Punkte, Fläche). Es wird allerdings keine Aussage über irgendein konkretes Rechteck gemacht. Dazu brauchen wir das Konzept des *Objekts.*

Ein Objekt ist ein individuelles *Exemplar* einer Klasse.

Objekte in Java

Zum Herstellen eines individuellen Exemplars einer Klasse benutzt man *new* zusammen mit einem Konstruktor. In der Klasse *Rechteck* gibt es nur einen Konstruktor. Klassen können mehrere Konstruktoren haben, sie unterscheiden sich dann durch den Typ und die Anzahl der Parameter. Im folgenden Konsolenprogramm werden zwei Rechteck-Objekte erzeugt und dann benutzt:

```
public class AppRechteck {

  public static void main(String[] args[]) {
    Rechteck R1 = new Rechteck(  5, 10, 100, 200);
    Rechteck R2 = new Rechteck(100, 50, 200, 100);
    R1.zeigen();
    R2.zeigen();
    System.out.println("Fläche: " + R1.gibFlaeche());
    System.out.println("Fläche: " + R2.gibFlaeche());
  }
}
```

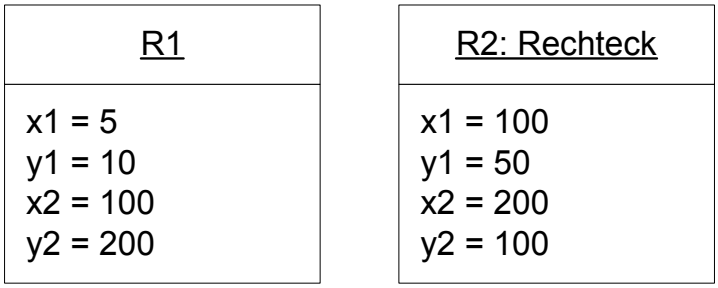
Alternativ wäre auch eine Trennung zwischen Deklaration der Strecken-Variablen und der Erzeugung der Strecken-Objekte möglich gewesen:

```
public class AppRechteck {

  public static void main(String[] args) {
    Rechteck R1;
    Rechteck R2;
    R1 = new Rechteck(  5, 10, 100, 200);
    R2 = new Rechteck(100, 50, 200, 100);
    R1.zeigen();
    R2.zeigen();
    System.out.println("Fläche: " + R1.gibFlaeche());
    System.out.println("Fläche: " + R2.gibFlaeche());
  }
}
```

Man sieht daran schön, dass eine Klasse einem Datentyp entspricht und man von diesem Datentyp Variablen deklarieren kann (`Rechteck R1;`). Da es sich dabei aber nicht um einfache Datentypen handelt, wie int, long oder double, müssen die Werte mittels *new* <*constructor*> erzeugt werden. Die so erzeugten Werte sind die Objekte.

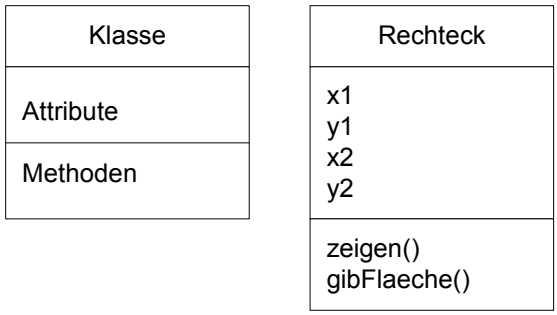
UML-Darstellung von Objekten

Objekte werden als Rechtecke dargestellt. Der Objektname steht im Singular und zentriert im Kopf des Rechtecks, bei Bedarf folgt mit Doppelpunkt abgetrennt der Klassenname. Im Unterschied zur Darstellung einer Klasse ist der Objektname unterstrichen. Bei Objekten gibt man nur die Attribute und zwar zusammen mit ihren Werten an. Die Methoden werden nicht angegeben, denn sie sind für aller Objekte einer Klasse gleich und ergeben sich aus der Klasse.



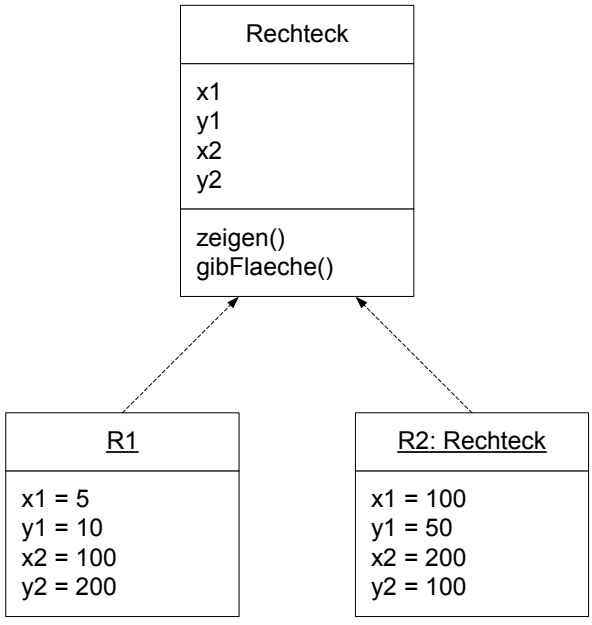
UML-Darstellung von Klassen

Klassen werden als Rechtecke dargestellt. Der Klassenname steht im Singular und zentriert im Kopf des Rechtecks. Darunter - durch eine Strecke abgeteilt - werden die Attribute linksbündig aufgeführt, ganz unten stehen die Methoden der Klasse.



Objekt-Klassen-Beziehung

Der Zusammenhang zwischen Objekten und Klassen kann in einem Objekt-Klassen-Diagramm dargestellt werden. Dabei werden die Objekte mit gestrichelten Pfeilen in Richtung der Klasse verbunden.



Aufgabe

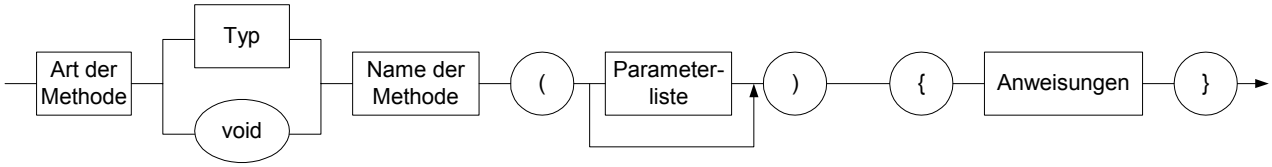
1. Ergänze die Klasse Rechteck um die Methoden
a) gibUmfang()
b) verschiebe(x, y)
c) erzeuge_zufällig()
d) schneidet(Rechteck r)

2. Definiere eine Klasse
a) Kreis (geometrisch)
b) Schüler
mit sinnvollen Attributen und Methoden.

Prozeduren und Funktionen

Aufgabe 1 von der vorherigen Seite beleuchtet nochmals den Unterschied zwischen Methoden ohne Ergebnis (Prozeduren) und Methoden mit Ergebnis (Funktionen). Die Funktion *gibUmfang()* liefert den Umfang eines Rechtecks als Ergebnis, die Funktion *schneidet()* gibt Auskunft, ob ein Rechteck von einem anderen geschnitten wird. Die Prozedur *verschiebe(x, y)* verschiebt ein Rechteck ohne ein bestimmtes Ergebnis zu liefern, die Prozedur *erzeuge_zufällig()* legt Anfangs- und Endpunkt zufällig fest.

Das folgende *Syntaxdiagramm* legt fest, wie Methoden geschrieben werden:

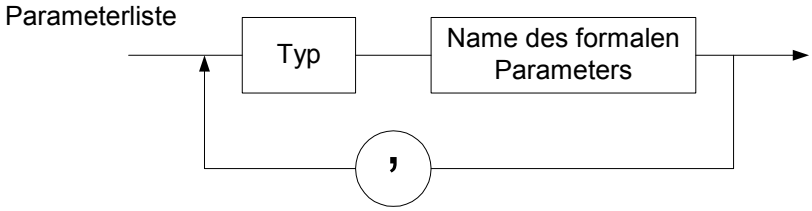


Als Art der Methode wird oft *public, private* oder *static* benutzt. Prozeduren werden anschließend mit *void* gekennzeichnet, bei Funktionen gibt man den Typ des Funktionswerts an. Es folgen der Name der Methode und in runden Klammern die Liste der Parameter, die auch leer sein kann. In den geschweiften Klammern stehen die Anweisungen, die in der Methode ausgeführt werden. Bei Funktionen muss mit einer *return*-Anweisung der Funktionswert zurückgegeben werden.

```
public double gibUmfang() {
  return 2 * (Math.abs(x2 - x1) + Math.abs(y2 - y1));
}
```

Parameter

Wenn die Methode zum Verschieben eines Rechtecks nur um einen festen Vektor verschieben könnte, wäre sie kaum sinnvoll zu gebrauchen. Flexibel nutzbar wird sie erst durch Einsatz von Parameter. Bei der Deklaration einer Methode benutzt man sogenannte *formale* Parameter. Die Parameterliste wird nach folgendem Syntaxdiagramm aufgebaut:




```
public void verschiebe(int x, int y) {
  x1 = x1 + x;
  y1 = y1 + y;
  x2 = x2 + x;
  y2 = y2 + y;
}
```

Beim Benutzen einer Methode werden anstelle der formalen Parameter aktuelle Parameterwerte eingesetzt. Dies können konstante Werte,Variablen oder auch komplexe Ausdrücke sein.

Beispiel: `R1.verschiebe(10, 20)`

Konstruktoren

Zum Erzeugen von Objekten benötigt man Konstruktoren. Beispielsweise wird mit

```
Rechteck R1 = new Rechteck(5, 10, 100, 200);
```

ein Rechteck R1 erzeugt. Ein Konstruktor ist eine besondere Methode, denn normalerweise kann man Methoden nur mit Objekten einsetzen, die schon vorhanden sind. Dies geschieht dann in der Punktschreibweise für den Methodenaufruf:

Beispiel: `R1.anzeigen();`
Allgemein: `Objekt.Methode(Parameterliste);`

Mit einem Konstruktor wird aber erst ein Objekt erzeugt, bei dem man später die Methoden nutzen kann. Daher kann auch die Punktschreibweise für den Aufruf eines Konstruktors nicht benutzt werden. Stattdessen wird eine Wertzuweisung an die Objektvariable benutzt, bei der der Wert über die Systemoperation *new* und Aufruf des Konstruktors mit aktuellen Parameterwerten bestimmt wird. Mit den aktuellen Parameterwerten werden die Attribute des Objekts initialisiert.

Java stellt für jede Klasse standardmäßig einen Konstruktor zur Verfügung. Er trägt wie jeder andere Konstruktor den Namen der Klasse und hat eine leere Parameterliste. Man könnte also auch wie folgt ein Rechteck erzeugen.

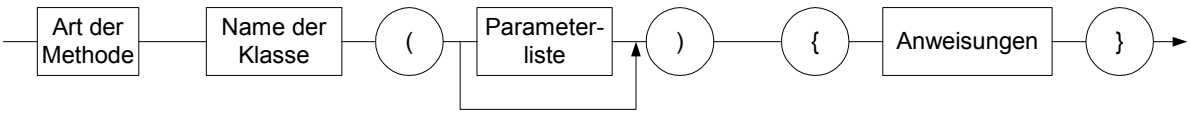
```
Rechteck R1 = new Rechteck();
```

allerdings wären dessen Koordinaten dann undefiniert. Konstruktoren sollten daher so programmiert werden, dass sie alle Attribute auf definierte Anfangswerte setzen. Dies geschieht im Rechteck-Beispiel durch:

```
public Rechteck(int _x1, int _y1, int _x2, int _y2) {
  x1 = _x1;
  y1 = _y1;
  x2 = _x2;
  y2 = _y2;
}
```

Der Unterschied zu normalen Methoden wird auch im Syntaxdiagramm deutlich.

Syntaxdiagramm für Konstruktor



Es fehlt die Typ-Angabe. Das ist aber verständlich, weil der von einem Konstruktor gelieferte Typ grundsätzlich festliegt, es ist natürlich der Typ der zugehörigen Klasse.

Aufgabe
a) Modelliere Autos als Klasse mit KFZ-Kennzeichen, Kilometerstand, Tankvolumen, Kraftstoffverbrauch, Kraftstoffmenge, tanken(Menge), fahren(Strecke), anzeigen().
b) Erzeuge mit einem Konstruktor zwei Autos und lasse sie fahren.
c) Modelliere entsprechend ein Elektroauto.

Eine Klasse kann mehrere Konstruktoren haben. Diese haben alle den gleichen Name, unterscheiden sich aber durch die Parameterliste. Beispielsweise kann in der Klasse Auto sinnvoll ein Konstruktor mit nur drei Parameter verwendet werden und zwar mit Kennzeichen, Tankvolumen und Kraftstoffverbrauch. Erzeugt man damit ein neues Auto, so soll automatisch die Kraftstoffmenge und der Kilometerstand auf 0 gesetzt werden, als ob das Auto frisch vom Band käme.

Aufgabe 1: Ergänze diesen weiteren Konstruktor.

GUI-Oberfläche und Fachkonzept

In vielen Methoden, die für die Schüler-, Auto- und Elektroauto-Klassen geschrieben wurden, sind Ausgabeanweisungen eingebaut worden. Dies erweist sich dann als wenig sinnvoll, wenn das Programm eine andere Benutzungsoberfläche erhalten soll. Die Auto-Klasse soll nur das modellieren und realisieren, was fachlich von Bedeutung ist. Wir sprechen daher von einer Fachklasse bzw. vom Fachkonzept, wenn mehrere Fachklassen vorkommen. Die Auto-Klasse hat aber nichts mit der Benutzungsoberfläche zu tun und darf daher auch dort nichts ausgeben.

Aufgabe 2: Entferne alle Ausgabeanweisungen aus der Fachklasse Auto.

Für die Benutzung der Fachklasse gestalten wir jetzt mit Hilfe einer GUI-Klasse eine grafische Benutzungsoberfläche. Zunächst kennt diese GUI-Klasse noch nicht die Auto-Fachklasse.

Im Variablen-Bereich der GUI-Klasse ergänzen wir dann eine Anweisung zum Erzeugen eines Auto-Objekts.

```
private Auto dasAuto = new Auto("DA-HT 644", 70, 7.6);
```

Dieses Auto kann man dann im GUI-Formular anzeigen, tanken und fahren. Die Anzeige kann wie folgt programmiert werden:

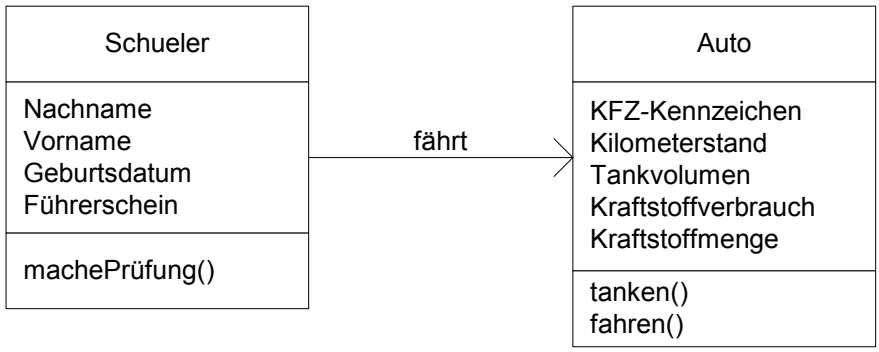
```
public void bAnzeigenActionPerformed(ActionEvent evt) {
  tfKennzeichen.setText(dasAuto.getKFZKennzeichen());
  tfKilometerstand.setText("" + dasAuto.getKilometerstand());
  tfTankanzeige.setText("" + dasAuto.getKraftstoffmenge());
}
```

Die Ereignismethode zum Anzeigen benutzt die get-Methoden für die Attribute, die man sich beim Anlegen einer neuen Klasse automatisiert erzeugen lassen kann. Hat man dies nicht gemacht, so kann man nachträglich diese Methoden schreiben. Einfacher wäre es die Sichtbarkeit der Attribute von private auf public zu ändern, doch das wäre ein klarer Verstoß gegenüber dem Geheimnisprinzip, nachdem die Objekte selbst für ihre Attribute verantwortlich sind und nur über Methoden auf diese Attribute zugegriffen werden kann.

Aufgabe 3:
a) Erstelle das GUI-Formular.
b) Beim Tanken sollen immer 30 Liter getankt, beim Fahren immer 100 km zurückgelegt werden.
c) Erstelle bei Bedarf eine Luxusversion des Programms.

Assoziation

Zwischen den Objekten von Klassen können konkrete Beziehungen bestehen. Beispielsweise kann der Schüler Fabian das Auto mit dem Kennzeichen DA-HT 644 fahren oder der Kunde Hans Meier kann bei seiner Bank zwei Konten mit den Nummern 4235 und 8283 haben. Objektbeziehungen werden durch das Konzept der Assoziation modelliert. So wie Objekte Exemplare von Klassen sind, sind Objektbeziehungen Exemplare einer Assoziation.



In der UML werden Assoziationen als Strecken zwischen den Klassen dargestellt. Die Strecke wird mit einem Beziehungsnamen beschriftet, der die Assoziation inhaltlich beschreibt. Meistens sind wie im Beispiel Assoziationen gerichtet. Sie sind dann nur in Richtung der offenen Pfeilspitze navigierbar.

Implementierung einer gerichteten Assoziation

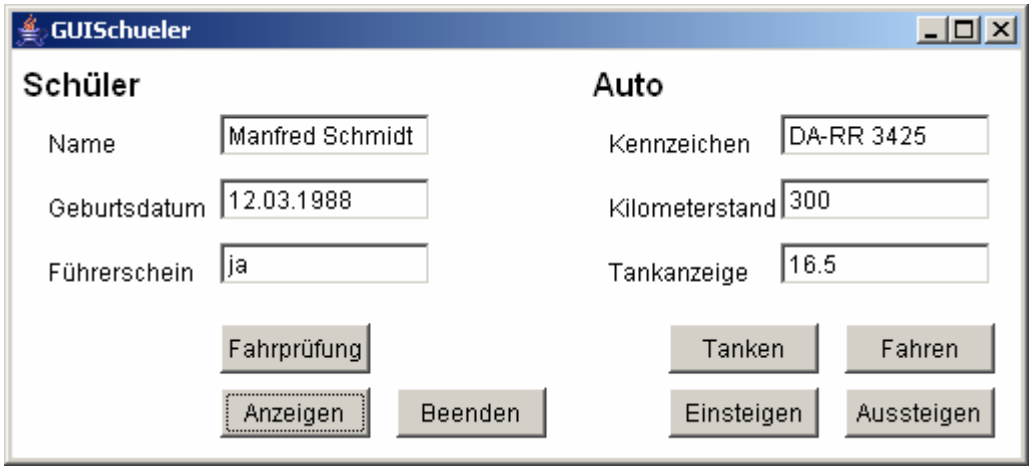
Die gerichtete Assoziation *Schüler fährt Auto* wird so implementiert, dass die Klasse Schüler um ein Attribut *dasAuto* ergänzt wird. Um die Anwendung einer Assoziation explizit zu machen, kann man folgende drei Operationen ausführen:

Operation	Beispiel	Allgemein
Herstellen einer Beziehung	setAuto(einAuto)	link
Abfragen einer Beziehung	getAuto(): Auto	getlink
Aufheben einer Beziehung	removeAuto()	unlink

```
public class Schueler {
  private Auto dasAuto;

  ...

  public void setAuto(Auto einAuto) {
    dasAuto = einAuto;
  }

  public Auto getAuto() {
    return dasAuto;
  }

  public void removeAuto() {
    dasAuto = null;
  }
}
```

Aufgabe 2

Nach Muster des folgenden GUI-Prototypen soll die Assoziation zwischen einem Schüler und einem Auto realisiert werden. Der Schüler darf erst nach der Fahrprüfung ein Auto alleine fahren. Über den Schalter *Einsteigen* wird die Assoziation hergestellt, über den Schalter *Aussteigen* getrennt.

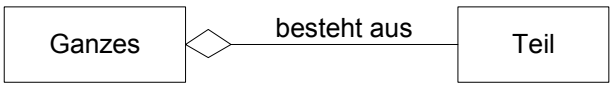


Die GUI-Klasse erhält nur ein Attribut Schüler, kein Attribut Auto. In der GUI-Klasse kann man aber mit den drei genannten Assoziationsbeziehungen arbeiten. Beispiele:

```
public void bEinsteigenActionPerformed(ActionEvent evt) {
  Auto Porsche = new Auto("DA-RR 3425", 80, 14.5);
  derSchueler.setAuto(Porsche);
}

public void bFahrenActionPerformed(ActionEvent evt) {
  Auto einAuto = derSchueler.getAuto();
  einAuto.fahren(100);
}
```

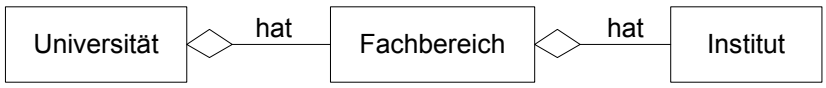
Aggregation – Die hat-Beziehung

Ein Spezialfall der Assoziation ist die *Aggregation*, bei der die beteiligten Klassen in einer Ganzes-Teile-Beziehung (hat-Beziehung) stehen. Eine Aggregation bildet also die Zusammensetzung eines Objekts aus einer Menge gleichartiger Einzelteile ab. Dabei übernimmt die Aggregatklasse stellvertretend Aufgaben für die untergeordnete Klasse.



In der UML-Darstellung wird die Aggregatklasse mit einer Raute versehen. Die Raute symbolisiert das Behälterobjekt, in dem die Teile gesammelt werden. Die Navigierbarkeit muss immer vom Ganzen zum Teil gegeben sein, auf eine entsprechende Pfeilspitze wird daher meist verzichtet.

Die Teil-Klasse kann selbst wieder eine Aggregatklasse sein:



Zur Implementierung einer Aggregation muss die Aggregatklasse um ein Attribut ergänzt werden, in dem mehrere Objekte gespeichert werden können. In Java benutzt man ein Feld oder einen Vektor.

Telefonbuch

In Handys oder PDAs (Personal Digital Assistant) findet man kleine Applikationen, die als Telefonbuch oder Merkzettel dienen. Diese stellen meist keinerlei Datenbankfunktionalität bereit - allerhöchstens sind sie in der Lage die Einträge zu sortieren. Man kann sie gut mit einem Karteikasten vergleichen, der eine bestimmte Sorte Karteikarten bereithält.

Betrachten wir ein vereinfachtes Telefonbuch eines Handys, bei dem die Einträge unsortiert gespeichert werden und ihre Anzahl auf 10 begrenzt sein soll.

a) Modelliere ein Telefonbuch, das Einträge mit den Feldern Nummer (eine interne automatisch erzeugte Nummerierung), Name, Telefonnummer sowie ein Ja/Nein-Feld Kurzwahl (Bedeutung: der Eintrag ist einer Kurzwahltaste zugeordnet) beinhaltet.

Folgende Operationen sollen möglich sein:
- eine neuer Eintrag wird angelegt, dabei wird er automatisch nummeriert und
- nach Angabe einer Nummer wird der zugehörige Eintrag ausgegeben.

Begründe deine Entscheidungen und stelle das Ergebnis in einem UML-Klassendiagramm dar.

b) Setzen deine Überlegungen durch Implementierung eines passenden Java-Programms mit einer grafischen Benutzungsoberfläche um und erweitere dementsprechend das UML-Klassendiagramm.

Turtle

Die Programmiersprache LOGO wurde speziell zum Einstieg ins Programmieren entworfen. Sie enthält eine spezielle Grafikkonzeption, die so genannte Turtle-Grafik. (engl. turtle = Schildkröte, dt. Igel-Grafik), mit der Prozeduren bei geometrischen Fragestellungen gut veranschaulicht werden können. Es gibt Befehle zur Steuerung der Turtle. Bei Bewegungen hinterlässt sie Spuren im "Sand", zeichnet dabei also auf ihre Zeichenfläche.

Attribute der Turtle

turtleX	x-Position der Turtle
turtleY	y-Position der Turtle
turtleW	Richtung der Turtle, 0^0 nach rechts, 90^0 nach oben
drawDynamic	Die Turtle zeichnet zum Zuschauen langsam.

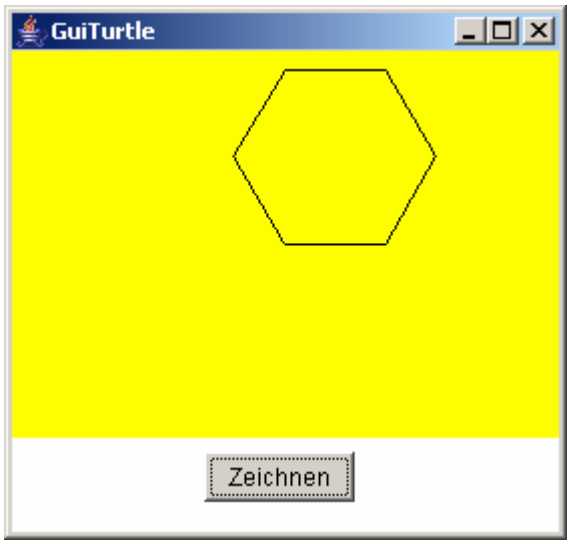
Anfangs befindet sich die Turtle in der Mitte ihrer Zeichenfläche und schaut nach rechts.

Methoden der Turtle

Turtle(int width, int height)	Erzeugt eine Turtle mit der einer Width x Height großen Zeichenfläche.
turn (double angle)	Ändert die Zeichenrichtung relativ um den Winkel angle.
turnto (double angle)	Legt die Zeichenrichtung absolut fest
draw (double length)	Die Turtle zeichnet in der aktuellen Richtung eine Strecke der Länge *length*.
drawto(double x, double y)	Die Turtle zeichnet von der aktuellen Position eine Strecke zum Punkt P(x/y).
move (double lenght)	Bewegt die Turtle in der aktuellen Zeichenrichtung um eine Strecke der Länge *length* ohne zu zeichnen.
moveto (double x, double y)	Setzt die Turtle auf den Punkt P(x/y)
setForeground(Color c)	Setzt die Farbe c als Zeichenfarbe
setBackground(Color c)	Setzt die Farbe c als Hintergrundfarbe
clear()	Löscht die Zeichenfläche.

Die Zeichenfläche der Turtle hat ihren Ursprung in der linken oberen Ecke. Entgegen den Gewohnheiten der Mathematik ist die y-Achse bei Computergrafiken nach unten gerichtet.

```
myTurtle = new Turtle(200, 200);
cp.add(myTurtle);

// Anfang Ereignisprozeduren
public void buZeichneActionPer-
formed(ActionEvent evt) {
  for (int i = 1; i <= 6; i++) {
    myTurtle.draw(50);
    myTurtle.turn(60);
  }
}
```



Eine detaillierte Beschreibung der Turtle in Form eines HTML-Dokuments erzeugt *JavaDoc*.

Rekursion mit Grafik

Schreibe eigenständige Methoden zum Zeichnen folgender Bilder, welche über Ereignismethoden innerhalb eines Java-Programms aufgerufen werden.

Bild 0

Bild 1

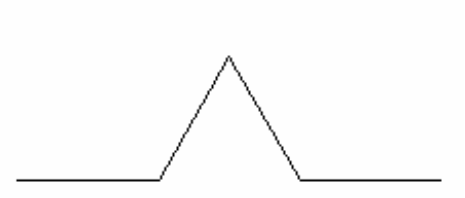


Bild 2

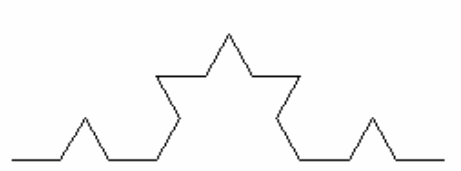
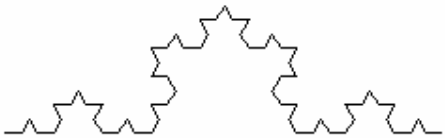


Bild 3



Initiator-Generator

Der Initiator der Kochschen Kurve ist eine Strecke:

```
public void initiator(double laenge) {
  myTurtle.draw(laenge);
}
```

Die Kochsche Kurve wird durch den Generator von Bild 1 erzeugt. Jede Strecke des Initiators wird durch den Generator ersetzt.

```
public void zeichneKurve1(double laenge) {
  myTurtle.draw(laenge/3);
  myTurtle.turn(60);
  myTurtle.draw(laenge/3);
  myTurtle.turn(-120);
  myTurtle.draw(laenge/3);
  myTurtle.turn(60);
  myTurtle.draw(laenge/3);
}
```

Bild 2 lässt sich mit der Zeichenmethode von Bild 1 wie folgt zeichnen:

```
public void zeichneKurve2(double laenge) {
  zeichneKurve1(laenge/3);
  myTurtle.turn(60);
  zeichneKurve1(laenge/3);
  myTurtle.turn(-120);
  zeichneKurve1(laenge/3);
  myTurtle.turn(60);
  zeichneKurve1(laenge/3);
}
```

Bild 3 lässt sich wiederum mit der Zeichenmethode von Bild 2 wie folgt zeichnen:

```
public void zeichneKurve3(double laenge) {
  zeichneKurve2(laenge/3);
  myTurtle.turn(60);
  zeichneKurve2(laenge/3);
  myTurtle.turn(-120);
  zeichneKurve2(laenge/3);
  myTurtle.turn(60);
  zeichneKurve2(laenge/3);
}
```

Gesucht ist nun eine Methode, mit welcher man das Bild mit der Nummer n erzeugen kann. Vergleicht man die Methoden miteinander, so fällt auf, dass sie sich nur letztlich nur in der Endung des Namens unterscheiden. Wir machen daher aus der Endung des Namens einen Parameter und erhalten folgende unvollständige Zwischenlösung:

```
public void zeichneKurve(int n, double laenge) {
  zeichneKurve(n-1, laenge/3);
  myTurtle.turn(60);
  zeichneKurve(n-1, laenge/3);
  myTurtle.turn(-120);
  zeichneKurve(n-1, laenge/3);
  myTurtle.turn(60);
  zeichneKurve(n-1, laenge/3);
}
```

Beginnen wir mit n = 4, so wird n der Reihe nach auf 3, 2, 1, 0, -1, -2, ... reduziert. Wir müssen also dafür sorgen, dass bei n = 0 die Turtle zeichnet. Daraus ergibt sich die Lösung:

```
public void zeichneKurve(int n, double laenge) {
  if ( n > 0 ) {
    zeichneKurve(n-1, laenge/3);          // Rekursionsfall
    myTurtle.turn(60);
    zeichneKurve(n-1, laenge/3);
    myTurtle.turn(-120);
    zeichneKurve(n-1, laenge/3);
    myTurtle.turn(60);
    zeichneKurve(n-1, laenge/3);
  }
  else
    myTurtle.draw(laenge);                // direkter Fall - Terminierung
}
```

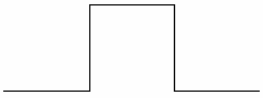
Das Charakteristische dieser Methode ist, dass sie sich in ihrem Anweisungsteil selbst aufruft!

Eine Methode heißt ***rekursiv***, wenn sie sich in ihrem Anweisungsteil selbst aufruft.

Aufgabe:

a) Benutze als Initiator ein gleichschenkliges Dreieck. Jede Strecke des Initiators wird durch den Koch-Generator ersetzt.

b) Der Initiator sei ein Quadrat, die nebenstehende Figur der Generator.

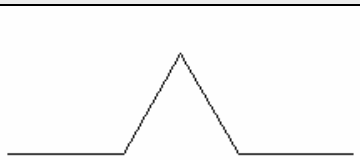
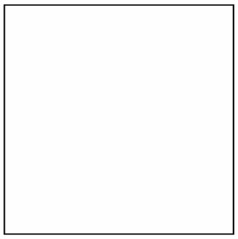
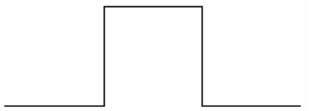
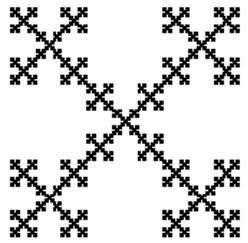


Selbstähnlichkeit

Eine rekursive Zeichenprozedur ruft sich selbst im Anweisungsteil auf. Das muss sich natürlich auf die erzeugte Grafik auswirken. Der Selbstaufruf führt zur Selbstähnlichkeit.

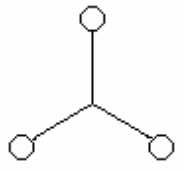
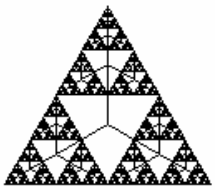
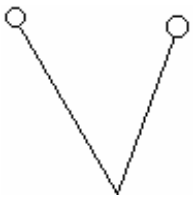
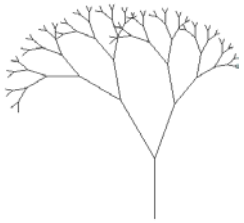
> Eine geometrische Figur heißt **selbstähnlich**, wenn sie sich in kongruente Teile zerlegen lässt, die ihr alle ähnlich sind. Vergrößern wir eine der Teilfiguren, so ergibt sich das Ganze.

Selbstähnliche Figuren lassen sich mit der *Initiator-Generator-Methode* erzeugen. Sie funktioniert wie folgt: In einem Streckenzug I, dem *Initiator*, wird jede Strecke durch eine Figur G, den *Generator*, ersetzt. Daraufhin wird jede Strecke der erzeugten Figur durch die Generatorfigur G ersetzt. Diese ***Streckenersetzung*** wird beliebig lange wiederholt.

Initiator	Generator	selbstähnliche Figur
		
		

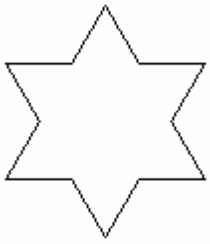
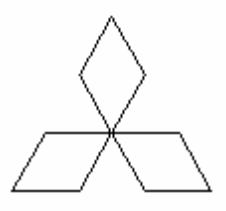
Eine Variante der *Initiator-Generator-Methode* besteht darin, statt der Strecken die Ecken durch den Generator zu ersetzen. Sie funktioniert wie folgt: Zeichne eine Anfangsfigur I und ersetze jede Ecken von I durch eine Figur G. In der so entstandenen Figur ersetze jede Ecke durch G. Diese ***Eckenersetzung*** wird beliebig lange wiederholt.

Bei der Streckenersetzung werden die Draw-Befehle des Generators durch rekursive Aufrufe ersetzt. Bei der Eckenersetzung werden an allen Ecken des Generators rekursive Aufrufe vorgenommen. Eine Variante hiervon besteht darin, nur einige ausgewählte Ecken zu ersetzen. Die ausgewählten Ecken sind nachfolgend durch Kreise markiert.

Initiator	Generator	selbstähnliche Figur
(Punkt)		
		

Orientierung

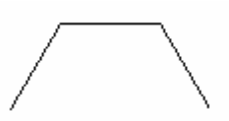
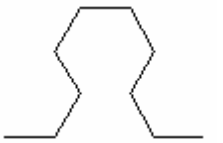
Nimmt man als Initiator ein gleichseitiges Dreieck und den Generator der Kochschen Kurve, so erhält man je nach Orientierung zwei unterschiedliche Grafiken.

Orientierung nach außen	Orientierung nach innen
	

Bis auf die Vorzeichen der Winkel in den Turn-Befehlen sind die beiden Zeichenprozeduren identisch. Die Unterschiede lassen sich durch Einführung eines dritten Parameters *Orientierung*, der den Wert +1 bzw. -1 hat, ausgleichen. Es gibt dann eine Zeichenprozedur, die beim Aufruf mit +1 die eine und beim Aufruf mit -1 die andere Figur zeichnet.

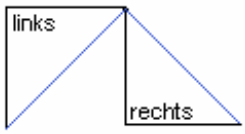
Pfeilspitzenkurve

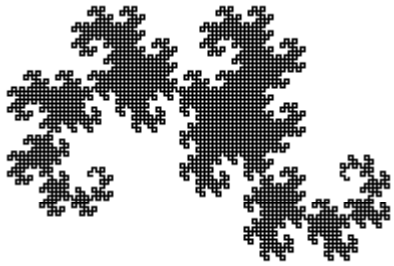
Die Pfeilspitzenkurve hat als Initiator eine Strecke und als Generator die linke Figur. Zusätzlich wird bei jeder Streckenersetzung die Orientierung geändert. Daher erhält man in der nächsten Stufe die rechte Figur.

n = 1		n = 2	
			

Drachenkurve

Nach dem gleichen Prinzip entsteht die *Drachenkurve*.

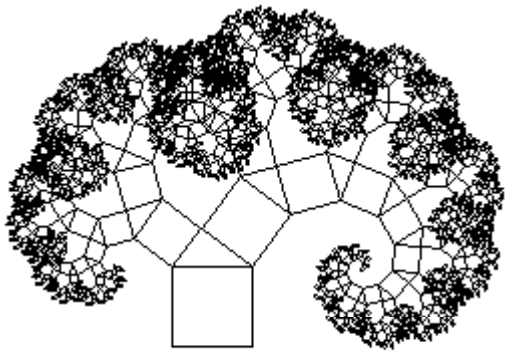




Bei der Streckenersetzung wird abwechselnd der Haken nach links bzw. nach rechts ausgeführt. Die Zeichenmethode muss dazu um einen Richtungsparameter ergänzt werden.

Baum des Pythagoras

Der Baum des Pythagoras war eine Aufgabe aus dem Bundeswettbewerb Informatik 1986. Die Seitenverhältnisse des Dreiecks sind 3:4:5. Die Rekursionstiefe wird zusätzlich durch die Seitenlänge bestimmt. Ist sie kleiner gleich 1 so ist ein rekursiver Aufruf nicht mehr sinnvoll. Dies gilt übrigens auch für alle bisher entwickelten Zeichenprozeduren.



Komplexität rekursiver Grafiken

Die Komplexität rekursiver Grafikalgorithmen kann nicht mit Feldvergleichen bzw. Feldbewegungen gemessen werden. Die Zeichendauer ist letztlich von der Zahl der Drehbewegungen und der gezeichneten Strecken abhängig. Eine Drehbewegung der Turtle kann der Computer mittels einer Real-Addition schnell ausrechnen. Zum Zeichnen einer Strecke muss er aber jeden zwischen Anfangs- und Endpunkt liegenden Pixelpunkt bestimmen. Die Komplexität wird deshalb durch die Anzahl der zu zeichnenden Strecken S in Abhängigkeit von der Rekursionstiefe n, also S(n) bestimmt. Mit einer kleinen Tabelle lässt sich der funktionale Zusammenhang ermitteln:

n	0	1	2	3	n
S(n)	1	4	4*4	4*4*4	4^n

Exponentielles Wachstum

Die Tabelle zeigt, dass exponentielles Wachstum vorliegt. Was exponentielles Wachstum bedeutet, zeigt die nachfolgende Kalkulationstabelle. In ihr kann die Zeit zum Zeichnen einer Strecke vorgegeben werden.

Komplexität der Kochschen-Kurve

Zeit zum Zeichnen einer Strecke: 0,001 Sekunden

n	Potenz(4;n)	Sekunden	Minuten	Stunden	Tage	Jahre
0	1	0,001	0	0	0	0
1	4	0,004	0	0	0	0
2	16	0,016	0	0	0	0
3	64	0,064	0	0	0	0
4	256	0,256	0	0	0	0
5	1024	1,024	0	0	0	0
6	4096	4,096	0	0	0	0
7	16384	16,384	0	0	0	0
8	65536	65,536	1	0	0	0
9	262144	262,144	4	0	0	0
10	1048576	1048,576	17	0	0	0
11	4194304	4194,304	70	1	0	0
12	16777216	16777,216	280	5	0	0
13	67108864	67108,864	1118	19	1	0
14	268435456	268435,456	4474	75	3	0
15	1073741824	1073741,824	17896	298	12	0
16	4294967296	4294967,296	71583	1193	50	0
17	17179869184	17179869,184	286331	4772	199	1
18	68719476736	68719476,736	1145325	19089	795	2
19	2,74878E+11	274877906,944	4581298	76355	3181	9
20	1,09951E+12	1099511627,776	18325194	305420	12726	35
21	4,39805E+12	4398046511,104	73300775	1221680	50903	139
22	1,75922E+13	17592186044,416	293203101	4886718	203613	558
23	7,03687E+13	70368744177,664	1172812403	19546873	814453	2231

Aufgabe

a) Untersuche die Komplexität der Kreuzstichkurve und des Dreieck-Teppichs.
b) Stelle Dein Ergebnis als Kalkulationstabelle dar.

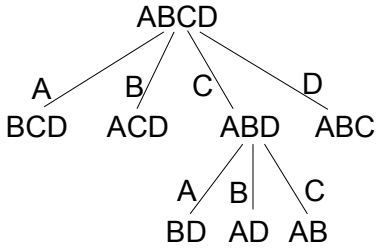
Permutationen

Unter einer Permutation versteht man die Umordnung einer Zeichenfolge. Permutationen der Zeichenfolge 'ABCD' sind beispielsweise 'CBAD' und 'DCBA'. Gesucht ist ein Algorithmus, mit dem man alle Permutationen einer vorgegebenen Zeichenfolge berechnen kann.

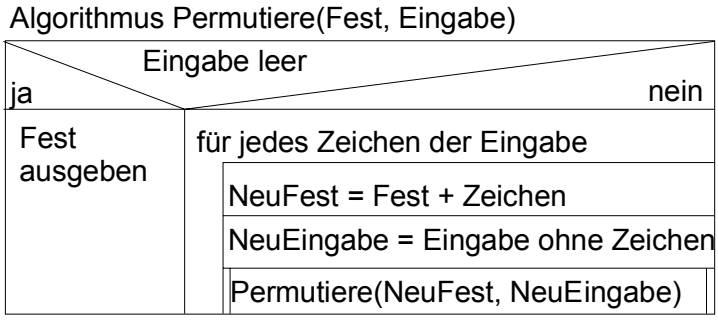
Problemlösung

Eine gute Problemlösefähigkeit erreicht man durch fortgesetzte Übung, welche Erfahrung vermittelt, durch Hartnäckigkeit, welche die eigene Leistungsfähigkeit bestätigt, aber auch durch Einsatz typischer Problemlösemethoden: was ist gegeben, was ist gesucht, mache eine Zeichnung, wie kann man das Problem vereinfachen.

Anhand einer Zeichnung erfasst man die möglichen Permutationen. Ausgehend von ABCD hält man zunächst A fest und bestimmt dann alle Permutationen von BCD. Den Vorgang wiederholt man für alle weiteren Buchstaben, die im Ausgangswort enthalten sind. Alle Permutationen von BCD erhält man nach dem gleichen Schema. Man hält B fest und bildet alle Permutationen von CD, dann hält man C fest und bildet alle Permutationen von BD, zuletzt hält man D fest und bildet alle Permutationen von BC.



Im Algorithmus muss also unterschieden werden, was festzuhalten und was noch zu permutieren ist. Insgesamt gesehen werden die Permutationen von vier Zeichen durch vier Permutationen zu je drei Zeichen bestimmt, wodurch die rekursive Struktur des Lösungsalgorithmus gegeben ist.



Zur Programmierung einige Hinweise: Die verfügbaren String-Methoden befinden sich in der Klasse *String* (siehe API). Nützlich sind charAt und substring sowie "+". Die Ausgabe erfolgt in ein mehrzeiliges Textfeld, also eine TextArea. Mit deren geerbter Methode *setText* kann die TextArea gelöscht werden. Eine neue Zeile kann man mit *TextAreaName.append(zeile + "\n")* hinzufügen.

Übung:

a) Programmiere den Algorithmus Permutiere.
b) Bestimme seine Komplexität.

Fibonacci-Zahlen

Die Zahlen der Folge 0, 1, 1, 2, 3, 5, 8, 13, 21,... bezeichnet man als Fibonacci-Zahlen. Das Bildungsgesetz lautet fib(n) = fib(n-1) + fib(n-2) mit den beiden Anfangswerten fib(0) = 0 und fib(1) = 1. Das Bildungsgesetz hat eine rekursive Struktur, weswegen man leicht eine rekursive Funktion zur Berechnung von Fibonacci-Zahlen programmieren kann.

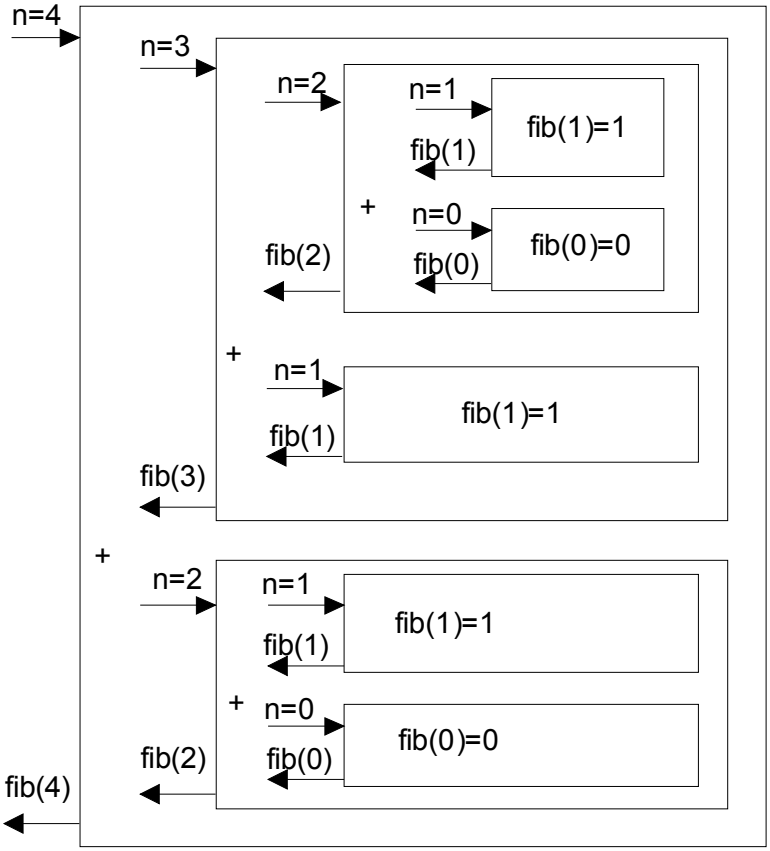
a) Bis zu welchem n berechnet die rekursive Funktion fib(n) in erträglicher Zeit?
b) Vergleiche mit einer iterativen Lösung.

Aufrufdiagramm

Stellt man jeden Aufruf der *fib*-Funktion als Rechteck dar, so erhält man das nebenstehende Aufrufdiagramm. Über den rechtsgerichteten Pfeilen steht jeweils das Argument, mit dem *fib* aufgerufen wird. Die linksgerichteten Pfeile zeigen die Ergebnisse an.

Das Aufrufdiagramm macht deutlich, dass Rekursion *Wiederholung durch Schachtelung* ist.

Die Komplexität des Berechnungsalgorithmus wird letztlich durch die rekursiven Aufrufe bestimmt. In der nachfolgenden Tabelle sind in Abhängigkeit von n die Fibonacci-Zahlen und die Zahl der zwecks Berechnung erforderlichen *fib*-Aufrufe dargestellt. Man sieht, dass die Zahl der Aufrufe noch stärker wächst als die Fibonacci-Zahlen.



N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Fib(n)	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610
Aufrufe(n)	1	1	3	5	9	15	25	41	67	109	177	287	465	753	1219	1973

Die Mathematik liefert für die Fibonacci-Zahlen die Formel: Sie macht deutlich, dass der Berechnungsalgorithmus exponentielle Komplexität hat.

$$\text{fib(n)} = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Lineare Zeitkomplexität erhält man durch die folgende iterative Lösung, konstante Zeitkomplexität durch Anwendung der Formel.

```
public int fib(int n) {
  int fib0=0, fib1=1, fibn=0;
  if (n==0) return fib0;
  if (n==1) return fib1;
  for ( int i=2 ;i <= n  ; i++ ) {
    fibn = fib0 + fib1;
    fib0 = fib1;
    fib1 = fibn;
  }
  return fibn;
}
```

Rekursion ist Wiederholung durch Schachtelung.
Iteration ist Wiederholung durch Aneinanderreihung.

Rekursion versus Iteration

Es stellt sich die Frage, wann man ein Problem iterativ oder rekursiv lösen sollte. Als Richtschnur kann dabei gelten, dass Rekursion angewendet werden sollte, wenn

- das zu lösende Problem "von sich aus" eine rekursive Struktur hat,
- der Zeit- und der Speicheraufwand einigermaßen überschaubar sind, und
- es keine gleichwertige, d.h. genauso einsehbare, oder effektivere iterative Lösung gibt.

Falls das zugrunde liegende Problem die entsprechende Struktur hat, sind rekursive Lösungen oft besser lesbar und eleganter, aber, wie gezeigt, nicht unbedingt effektiver. Eine rekursive Problemlösung sollte immer dann vermieden werden, wenn eine iterative Lösung offensichtlich ist.

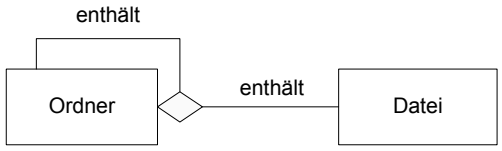
Übungen

1. Der größte gemeinsame Teiler (ggT) zweier ganzer nicht negativer Zahlen m und n kann nach dem griechischen Mathematiker Euklid wie folgt definiert werden:
   für m < n: ggT(m, n) = ggT(n, m) //n und m werden vertauscht
   für m > n: ggT(m, n) = ggT(n, m mod n)
   für m = n: ggT(m, n) = m
   Dabei ist *mod* der in Pascal vorhandene Modulo-Operator. m *mod* n ergibt den Rest bei der ganzzahligen Divison von m durch n.
   a) Schreibe gemäß Euklid eine rekursive Funktion zur Berechnung des ggT.
   b) Entwickle auch eine iterative Lösung.

2. Die Anzahl der Permutationen (vgl. Seite 34) von n Zeichen beträgt
   a) in rekursiver Formulierung: n! = n*(n-1)! mit 1! = 1 (n! = n-Fakultät)
   b) in iterativer Formulierung: n! = n*(n-1)*(n-2)...*3*2*1
   Gib jeweils eine rekursive und iterative Funktion zur Berechnung von n! an.

3. Die Potenz a^n kann wie folgt rekursiv definiert werden: $a^n = a * a^{n-1}$, $a^0 = 1$. Programmiere dementsprechend eine rekursive Funktion zur Berechnung von a^n.

4. Ein Ordner auf einer Festplatte enthält Dateien und Ordner. Es liegt eine rekursiv Struktur vor, die wie gezeigt als UML-Diagramm dargestellt werden kann. Zum Durchlaufen eines Ordners setzt man daher rekursive Algorithmen ein. Schreibe eine Prozedur, die die Namen aller Dateien in einem Ordner samt aller Unterordner ausgibt. Benutze die Klasse File.

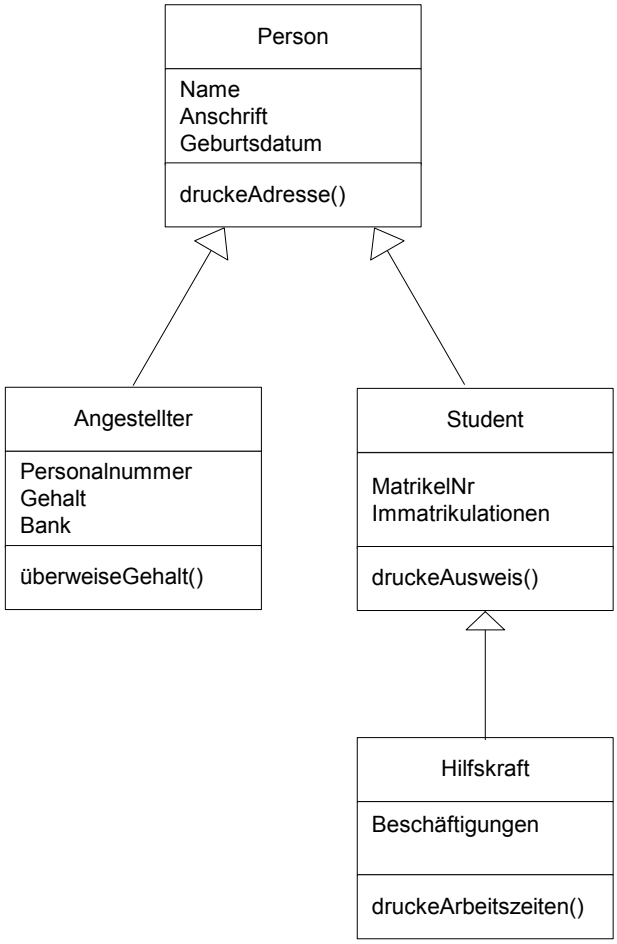


5. Der Turm von Hanoi
   Nach einer alten Legende befanden sich vor einem Tempel in Hanoi drei Säulen. Auf der ersten Säule befanden sich hundert Lochscheiben, deren Durchmesser von unten nach oben abnahm. Es bestand eine Weissagung, dass die Welt untergehen würde, wenn es jemandem gelänge, den Turm unter Beachtung der folgenden Regeln auf eine der anderen Säulen umzubauen:
   1. Man darf jeweils nur eine einzelne Scheibe umlegen.
   2. Nur die oberste Scheibe einer Säule darf bewegt werden.
   3. Es ist verboten, eine Lochscheibe mit größerem Durchmesser auf eine Lochscheibe mit kleinerem Durchmesser zu legen.

   a) Finde eine Algorithmusidee, indem du eine Säule aus zwei, drei, vier Scheiben selbst konkret umbaust, z.B. mit Münzen.
   b) Erstelle hierzu ein Simulationsprogramm, das ausgibt, von wo nach wo gerade eine Scheibe umgelegt wird.

Generalisierung – Vererbung – Die Ist-Beziehung

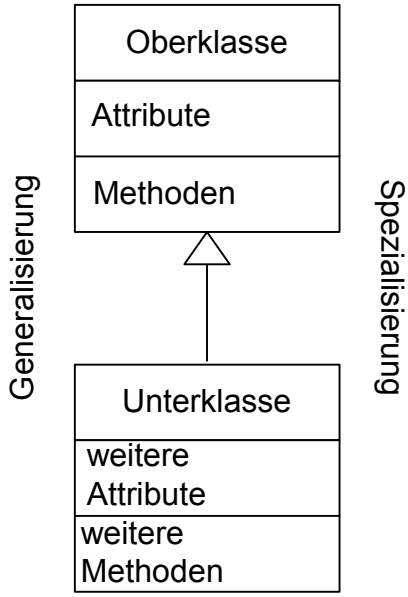
Die Generalisierung beschreibt eine Beziehung zwischen einer allgemeinen Klasse (Basisklasse) und einer speziellen Klasse. Die spezialisierte Klasse ist vollständig konsistent mit der Basisklasse, enthält aber zusätzliche Informationen (Attribute, Methoden, Assoziationen). Sie wird auch als abgeleitete Klasse bezeichnet. Ein Objekt der spezialisierten Klasse kann überall dort verwendet werden, wo ein Objekt der Basisklasse erlaubt ist. Wir sprechen von einer Klassenhierarchie. Die allgemeine Klasse wird auch als Oberklasse, die spezialisierte als Unterklasse bezeichnet.



Betrachten wir die drei Klassen Angestellter, Student und studentische Hilfskraft. Durch Modellierung einer Oberklasse Person, können die drei Klassen in eine Generalisierungsstruktur gebracht werden.

Die Generalisierung wird durch eine Strecke mit einem nicht ausgefüllten Dreieck bei der Basisklasse dargestellt.

Das Konzept der Generalisierung besitzt wesentliche Vorteile. Aufbauend auf existierenden Klassen können mit wenig Aufwand neue Klassen erstellt werden. Insbesondere werden GUI-Oberflächen mittels Generalisierung gestaltet. Jedes Programmformular ist von einer Fensterklasse abgeleitet, die die Entwicklungsumgebung bereit stellt.



Im Rahmen der objektorientierten Analyse, also bei der Erstellung des Fachkonzepts, benutzt man den Begriff Generalisierung. Im Kontext des objektorientierten Designs und einer objektorientierten Programmiersprache wird eher der Begriff der *Vererbung* benutzt. Hierbei gilt:

1. Besitzt die Oberklasse ein Attribut A, dann besitzen es auch alle Objekte der Unterklasse. Der Wert des Attributs wird jedoch nicht vererbt.
2. Alle Methoden, die auf Objekte der Oberklasse anwendbar sind, sind auch auf Objekte der Unterklasse anwendbar.

Aufgabe

a) Modelliere eine Klasse Konto mit Kontonummer, Kontostand, einzahlen(double Betrag) und auszahlen(double Betrag).

b) Modelliere mittels Vererbung eine Klasse Sparkonto. Ein Sparkonto wird jährlich zu einem festen Zinssatz verzinst. Es darf nicht überzogen werden.

c) Modelliere mittels Vererbung eine Klasse Girokonto. Ein Girokonto darf bis zu einem bestimmten Kredit überzogen werden. Überweisungen auf andere Konten sind möglich. überweisen(Konto einKonto, double Betrag), Überziehungszinsen werden monatlich fällig.

Konstuktoren und Vererbung

Konstuktoren werden nicht vererbt, daher müssen diese in Unterklassen neu definiert werden. Ein Konstruktor der Unterklasse muss zunächst einen Konstruktor der Oberklasse aufrufen. Normalerweise würde dies mit dem Namen der Oberklasse erfolgen. Da dieser Name aber schon für die Klasse und den Typ der davon gebildeten Objekte benutzt wird, muss man stattdessen einen Konstruktor der Oberklasse mit ***super(...)*** aufrufen. Über die Art und Anzahl der Parameter wird festgelegt, welcher Konstruktor der Oberklasse aufgerufen wird.

Erfolgt in einem Konstruktor kein Aufruf von super(...) so fügt der Compiler automatisch und damit implizit einen Aufruf des standardmäßigen Konstruktors super() ohne Argumente ein. Dies passiert z. B. bei allen Konstuktoren von Klassen an der Spitze der Klassenhierarchie, z. B. bei der Klasse Konto.

Überschreiben von Methoden

In einer Unterklasse können neue Attribute und Methoden definiert werden. Diese können den gleichen Namen wie die geerbten Attribute und Methoden haben. Bei Attributen sollte man dies wegen der doppelten Datenhaltung tunlichst vermeiden, bei Methoden wird diese Möglichkeit gerne genutzt und als ***Überschreiben von Methoden*** bezeichnet.

Aufgabe
Überschreibe die Auszahlen-Methode in den Klassen Girokonto und Sparkonto. Die neuen Methoden sollen das fachliche Konzept der Auszahlung bei einem Giro- bzw. Sparkonto umsetzen.

this und super

Mit *this* kann man explizit auf Attribute und Methoden der eigenen Klasse zugreifen. Das Schlüsselwort *super* ermöglicht in analoger Weise den Zugriff auf geerbte Attribute und Methoden, soweit das aufgrund einer Überschreibung erforderlich sein sollte. Mit *super.auszahlen(500)* kann z. B. die Methode auszahlen der Girokonto-Klasse auf die geerbte und überschriebene Methode auszahlen der Konto-Klasse zugreifen.

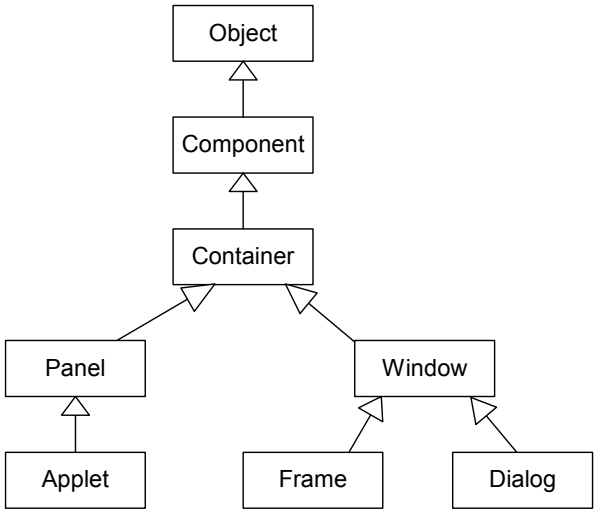
Die Klasse Object

Die Vererbungsbeziehung wird durch das Schlüsselwort ***extends*** ausgedrückt. Wird bei Oberklassen kein extends angegeben, so fügt der Compiler automatisch ein ***extends Object*** zur Klassendefinition hinzu. Damit wird erreicht, dass alle Klassen die Klasse Object als gemeinsame Oberklasse haben. Die Klasse Object hat keine Attribute aber einige nützliche Methoden, die damit in allen Klassen zur Verfügung stehen.

Klassenhierarchie

Java stellt eine sehr große Zahl von Klassen zur Verfügung. Alle Klassen können in einer Klassenhierarchie dargestellt werden.

Aufgabe 2
a) Wie wird Vererbung inder Java-API dargestellt?
b) Erweitere die Klassenhierarchie um die Klassen JFrame, System, Vector und Girokonto.



Videoverleih

a) Stelle das folgende Programm als UML-Diagramm dar und erläutere die auftretenden Klassen, Attribute, Methoden, Konstruktoren und Klassenbeziehungen.

b) Ausleiher können beim Videoverleih Videos ausleihen. Erweitere dementsprechend das Fachkonzept.

c) Die Video-Klasse enthält Basisinformation und wäre für Dokumentarfilme und Lehrfilme ok. Aber für Spielfilme werden mehr Informationen benötigt. Modelliere mittels Vererbung eine Klasse Spielfilm, die zusätzlich den Namen des Regisseurs und eine Altersfreigabe enthält. Ergänze dazu das UML-Diagramm und implementiere die Klasse Spielfilm.

d) Erweitere das Hauptprogramm so, dass der Ausleiher Max Meier nach Angabe seines Alters einen Spielfilm ausleiht.

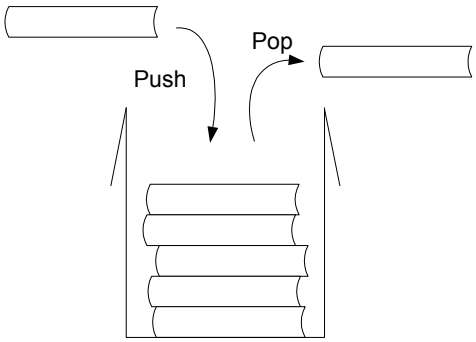
```
class Video {
  String   Titel;
  int      Laenge;       // in Minuten
  boolean  vorhanden;

  public Video(String Titel) {
    this.Titel = Titel;
    Laenge = 90;
    vorhanden = true;
  }

  public Video(String Titel, int _Laenge) {
    this.Titel = Titel;
    Laenge = _Laenge;
    vorhanden = true;
  }

  public void anzeigen() {
    System.out.println(Titel + ", " + Laenge + " min, verfügbar: " + vorhanden);
  }

}

class VideoVerleih {

  public static void main(String[] args) {
    Video[] alleVideos = new Video[20];
    int anzahlVideos = 2;
    alleVideos[0] = new Video("Jaws", 120);
    alleVideos[1] = new Video("Star Wars");

    for (int i = 0; i < anzahlVideos; i++) {
      alleVideos[i].anzeigen();
    }

  }
}
```

Höhere Datenstrukturen

Prinzip eines Keller

Ein Keller, auch Stapel oder Stack genannt, ist eine Datenstruktur bei der die Daten eine Folge bilden, die nur durch die Ein- und Ausfügeoperationen bestimmt wird. Das Prinzip eines Kellers besteht darin, dass stets das zuletzt eingefügte Element eines Kellers als erstes wieder entfernt werden muss. Die Kellerverwaltung arbeitet nach dem *LIFO-Prinzip*: Last In First Out. Wer zuletzt in den Keller kam, kommt zuerst aus dem Keller auch wieder raus.

Man kann sich einen Keller als eine Bücherkiste vorstellen, in die man einzeln Bücher legen und wieder herausnehmen kann



Üblicherweise stehen folgende fünf Kelleroperationen zur Verfügung:

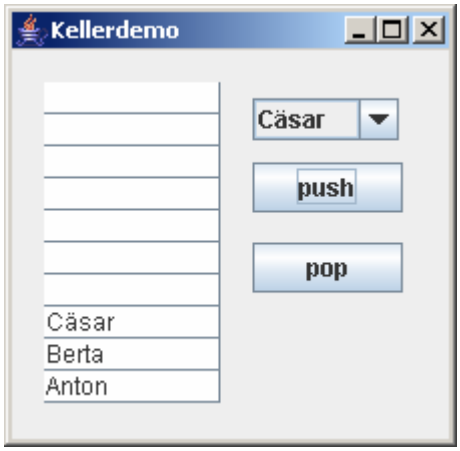
Stack:	Initialisiert einen neuen Keller.
push:	Legt ein Element auf dem Keller ab.
pop:	Holt ein Element aus dem Keller.
top:	Schaut nach, welches Element im Keller obenauf liegt.
empty:	Prüft, ob ein Keller leer ist.

Java bietet uns mit der Klasse *Stack* eine Implementierung dieser Datenstruktur an. Das folgende Demo-Programm zeigt eine einfache Art der Nutzung:

```
import java.util.Stack;
public class KellerDemo {

  public static void main (String args[]) {
    Stack meinKeller = new Stack();
    meinKeller.push("Anton");
    meinKeller.push("Berta");
    meinKeller.push("Cäsar");
    meinKeller.push("Die Letzten werden die Ersten sein!");
    while (! meinKeller.empty() )
      System.out.println(meinKeller.pop());
  }
}
```

Swing-Komponenten

Das Wirkprinzip eines Kellers soll nun in einem Programm dargestellt werden. Im Bild haben wir links eine JTable-, rechts eine JComboBox- und zwei JButton-Komponenten.



Der aktuelle Wert einer ComboBox kann mit meineComboBox..getSelectedItem() ermittelt werden.

Mit meineTabelle.`setValueAt("Anton", Position, 0);` kann der String „Anton“ in die Tabelle geschrieben werden.

Aufgabe:

Entwickle das Programm. Beachte dabei die nötigen Fallunterscheidungen.

Auswertung arithmetischer Ausdrücke

Mit Hilfe des Computers sollen arithmetische Ausdrücke ausgewertet werden. In den Ausdrücken dürfen neben natürlichen Zahlen als Operanden die Grundrechenarten als Operatoren und Klammern vorkommen. Beispiele:

6*(4+2)+(3-1)/4*5 -7
(1-3)/4+(1-7*4)/2

Zur korrekten Auswertung solcher Ausdrücke sind drei Regeln zu beachten: von links nach rechts, Klammern zuerst und Punkt- vor Strichrechnung. Die grundsätzliche Arbeitsweise eines Termauswerters beruht auf der Verwendung eines Kellerspeichers, auf dem er sich Zwischenergebnisse und Teilausdrücke solange merkt, bis sie weiter ausgewertet werden können. Um diese Grundidee besser herausstellen zu können, lassen wir nur so genannte *vollständig geklammerte* arithmetische Ausdrücke zu. Zu deren Auswertung ist nämlich nur noch eine Regel erforderlich: Klammern zuerst.

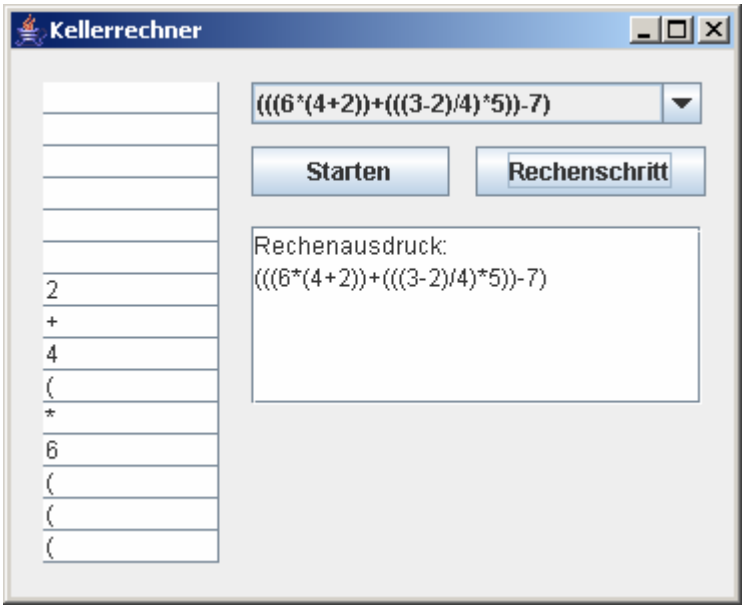
- Der Ausdruck 14 - 7 - 3 erfordert für die richtige Berechnung die Regel *von links nach rechts*, weil die Subtraktion nicht kommutativ ist. Vollständig geklammert ((14 - 7) - 3) ist diese Regel nicht mehr erforderlich.
- Der Ausdruck 5+3*4 erfordert für die richtige Berechnung die Regel Punkt- vor Strichrechnung. Vollständig geklammert (5+(3*4)) ist diese Regel nicht mehr erforderlich.

Mit welchem Algorithmus können nun vollständig geklammerte Ausdrücke ausgewertet werden?

(((6*(4+2))+(((3-2)/4)*5)) -7)
(((1-3)/4)+((1-(7*4))/2))

Eine öffnende Klammer beginnt einen Teilausdruck. Wir merken uns diese Klammer im Kellerspeicher, denn dazu muss es eine korrespondierende schließende Klammer geben. Ein natürliche Zahl als Operand sowie ein Operator werden gleichfalls auf dem Keller gespeichert. Erst wenn eine schließende Klammer kommt, haben wir etwas zu tun. Vom Keller abzuholen sind dann: der rechte Operand, der Operator, der linke Operand und die zugehörige öffnende Klammer. Das Ergebnis der Teilberechnung wird wieder im Kellerspeicher abgelegt. Sind alle Zeichen der Eingabe abgearbeitet wird das letzte Ergebnis als Resultat der Auswertung vom Keller geholt.

Im Bild sehen wir in der ComboBox den zu berechnenden Term. Es wurde gerade die Zahl 4 gelesen und auf den Keller gelegt. Wird per Schalter *Rechenschritt* die erste schließende Klammer gelesen, so werden die Elemente 2, + und 4 vom Keller geholt und durch 6 ersetzt.



Nutzung von API-Klassen - StringTokenizer

API ist das Akronym (Kurzwort aus Anfangsbuchstaben) für *Schnittstelle zur Anwendungsprogrammierung* (engl. application programming interface). Die Java API umfasst hunderte von Klassen, die man zur Programmierung benutzen kann. Was man dort findet, muss man nicht mehr selbst programmieren. Man muss allerdings in der Lage sein, die gefundenen Klassen im eigenen Programm zu benutzen. Dabei sind drei Dinge zu klären:

1. Wie erhalte ich Zugriff auf die API-Klassen? Einbinden über eine import-Anweisung
2. Wie erzeuge ich ein Objekt der API-Klasse? Auswahl eines geeigneten Konstruktors
3. Wie benutze ich das Objekt? Anwendung der geeigneten Methoden des Objekts

Für den Kellerrechner ist eine sinnvolle Erweiterung, dass er auch mehrstellige Zahlen verarbeiten kann. Dazu muss der Eingabestring mit dem Rechenausdruck gelesen und in seine Bestandteile zerlegt werden. Für diese Aufgabe kann man die Klasse StringTokenizer einsetzen. Ein StringTokenizer zerlegt Strings in ihre Bestandteile.

zu 1. In der Java-API wird angegeben, dass die Klasse StringTokenizer im Paket java.util enthalten ist. Das Anwendungsprogramm braucht daher die import-Anweisung *import java.util.*;*

zu 2. Von den drei zur Verfügung stehenden Konstruktoren ist der mit den drei Argumenten für unser Programm sinnvoll nutzbar. Als Trennzeichen (engl. delimiter) haben wir (, +, -, , / und) . Diese sollen auch als Bestandteile (Token) eines Rechenausdrucks erkannt werden. Der Konstruktor wird daher so benutzt: *einTokenizer = new StringTokenizer(Rechenausdruck, "(+-*/)", true);*

zu 3. Von den Methoden des StringTokenizers brauchen wir die boolesche Funktion hasMoreToken() zur Programmsteuerung und die String-Funktion nextToken() zum Lesen der Bestandteile eines Rechenausdrucks. Mit dem folgenden Konsolenprogramm kann man sich dann die Funktionsweise des StringTokenizers verdeutlichen.

```
import java.util.*;
public class TokenizerDemo {
  public static void main(String[] args) {
    StringTokenizer einTokenizer;
    String Rechenausdruck = "(((143.2-43)/24.87)+((123.9-(71.2*3.44))/72))";
    einTokenizer = new StringTokenizer(Rechenausdruck, "(+-*/)", true);
    while ( einTokenizer.hasMoreTokens() )
       System.out.println(einTokenizer.nextToken());
  }
}
```

Ausgabe:

```
(
(
(
143.2
-
43
)
/
24.87
)
...
```

Die API-Klasse Vector – Lineare Liste

Listen treten in vielfältiger Form auf, zum Beispiel Schülerlisten, Listenwahl, Anwesenheitsliste, Rangliste, Starterliste, Einkaufsliste. Zur Modellierung von Listen in Informatiksystemen wird man sich nicht mit speziellen Listen befassen, sondern einen allgemeinen Ansatz verfolgen, der später in einer beliebigen Anwendungssituation genutzt werden kann. Zu einem allgemeinen Ansatz kann man durch Analyse verschiedener Beispiele und Abstraktion der jeweiligen Details kommen. Es zeigt sich dann, dass eine Liste eine endliche Folge (Sequenz) von Elementen eines gegebenen Grundtyps ist. Im Unterschied zu Mengen ist auf Listen eine Ordnung definiert, es gibt also ein erstes, zweites, drittes, usw. Element und zu jedem Element - außer dem ersten und letzten - einen Vorgänger und einen Nachfolger. Weiterhin dürfen Listen Elemente auch mehrfach enthalten. Zur Arbeit mit Listen sind Grundoperationen nötig: erzeugen, einfügen, löschen, suchen, durchlaufen, ...

Die API-Klasse *Vector* stellt den abstrakten Datentyp (ADT) für lineare Liste in Java bereit. Man könnte zwar auch Felder zur Darstellung von linearen Listen verwenden, aber bei Feldern muss man im voraus die Kapazität festlegen. Hingegen passt sich die Länge eines Vectors dynamisch zur Laufzeit an die Anzahl der Listenelemente an.

Aufgaben

1. Informiere Dich in der Java-API über die Klasse Vector. Notiere Dir wichtige Methoden.

2. Für ein Skirennen soll eine Starterliste mit den Namen der Skiläufer erstellt werden. Schreibe ein einfaches Konsolenprogramm, das die Starterliste erstellt und ausgibt.

3. Die Skilläufer haben im Wettkampf eine Startnummer. Die Starterliste soll zusätzlich zum Namen die Startnummer jeden Rennläufers enthalten. Erstelle eine eigene Klasse Skiläufer. Stelle die Lösung von Aufgabe 2 auf die Verwendung von Skiläufer-Objekten um.

4. Die Klasse Skiläufer ist automatisch von der Klasse Object abgeleitet und kann somit die geerbte Methode toString benutzen. Verwende toString für die Ausgabe der Starterliste. Überschreibe die Methode toString in der Klasse Skiläufer durch ein eigenes Ausgabeformat (vgl. S. 38).

5. Sobald ein Läufer nach einem fehlerfreien Lauf die Ziellinie überquert, sollen seine aktuelle Platzierung und eine Liste der 10 besten Skiläufer ausgegeben werden. Verwende dazu eine zweite Liste und mache ein schönes GUI-Programm.

Warteschlange

In der Informatik ist die Warteschlange (Queue) eine Datenstruktur, mit der Daten zwischengespeichert werden können. Das Prinzip einer Warteschlange besteht darin, dass das zuerst eingefügte Element einer Warteschlange als erstes wieder ausgelesen werden muss. (FIFO-Prinzip, engl. First In First Out).

Die typischen Operationen auf einer Warteschlange sind:

Queue:	Initialisiert eine Warteschlange.
Empty:	Prüft, ob die Warteschlange leer ist.
Enter:	Ein neues Element wird am Ende der Warteschlange angefügt.
Front:	Liefert das erste Element der Warteschlange, ohne es aus der Warteschlange zu entfernen.
Remove:	Entfernt das erste Element von der Warteschlange und liefert es zurück.

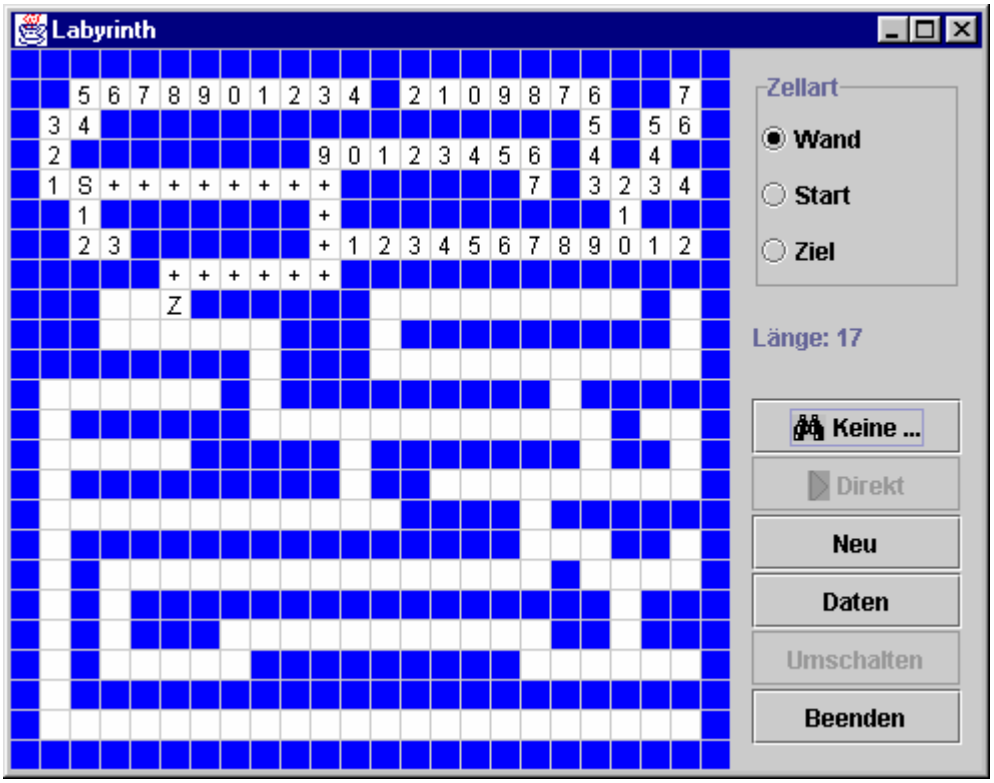
Aufgaben

1. Eine Klasse Warteschlange soll auf der Basis der Klasse Vector implementiert werden. Dazu erhält die Klasse Warteschlange mittels Assoziation ein Vector-Objekt. Die Elemente der Warteschlange werden in dem Vector gespeichert. Entwirf die Methoden der Klasse Warteschlange.
2. Implementiere diese Klasse.
3. Schreibe ein Demonstrationsprogramm zur Funktionsweise einer Warteschlange.

Graphenalgorithmen: Tiefen- und Breitensuche

Als Anwendung der Datentypen Keller und Schlange betrachten wir einen Algorithmus zur Lösung des Labyrinth-Problems, bei dem ein Weg vom Start zum Ziel gesucht ist. Klassische Lösungsansätze benutzen Tiefensuche, gehen also auf der Suche nach dem Ziel soweit wie möglich in das Labyrinth rein. Irgendwann kommt man in eine Sackgasse, bzw. nur noch an Stellen, die man schon passiert hat. Dann heißt es rückwärts gehen und die nächst Abbiegemöglichkeit benutzen. Das Wechselspiel zwischen weitergehen im Labyrinth und zurückgehen bis zur nächsten noch nicht benutzen Abzweigung wiederholt sich, bis das Ziel gefunden, oder keine Möglichkeiten mehr vorhanden sind. Der Tiefensuch-Algorithmus für Labyrinthe ist beliebt, weil er ganz einfach mit *Rekursion* programmiert werden kann.

Rekursion lässt sich grundsätzlich immer durch Iteration ersetzen. Dafür sehen wir hier ein Beispiel. Die jeweils noch zu untersuchenden Positionen im Labyrinth werden ausgehend vom Startfeld in einem Keller gespeichert. Gemäß dem LIFO-Prinzip wird immer mit der aktuellsten Position weitergearbeitet, dies führt zur Tiefensuche.



Ein alternative Methode ist die Breitensuche, bei dem die nächsten zu inspizierenden Felder eines Labyrinths in einer Schlange, also gemäß dem FIFO-Prinzip verwaltet werden. Im Bild sieht man, dass vom Startfeld aus drei Felder erreicht werden können. Diese drei Felder stellen sich der Reihe nach in einer Schlange auf. Man nimmt das erste Element der Schlange, ermittelt alle von diesem Feld erreichbaren freien Felder, welche gleichfalls an das Ende der Schlange angehängt werden. Dann nimmt man das nächste Element der Warteschlange dran und bearbeitet es nach dem gleichen Verfahren. Der Vorgang setzt sich fort, bis das Ziel erreicht ist, oder die Schlange leer wird. Im letzteren Fall gibt es offenbar keinen Weg vom Start zum Ziel.

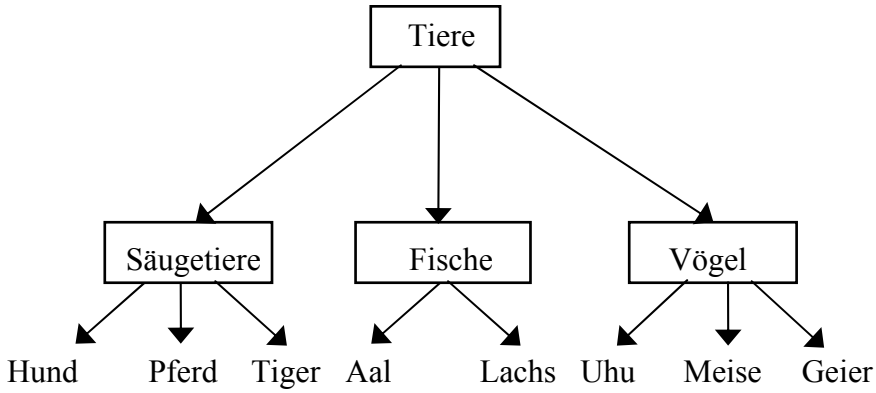
Aufgaben

1. Analysiere das Labyrinth-Programm.

a) Skizziere einen Suchbaum für die dargestellte Start- und Zielposition.
b) Stelle in einem Klassendiagramm die wesentlichen Datenstrukturen dar (Vererbung und Komposition)
c) Analysiere den Tiefen- und Breitensuch-Algorithmus.
d) Welche relevanten Datenstrukturen werden benutzt? Erläutere sie.
e) In welcher Beziehung stehen die Datenstrukturen zum Suchalgorithmus.
f) Wie kann man am Ziel angekommen, den Weg vom Start zum Ziel ermitteln
g) Vergleiche die Lösung der Tiefensuche mit der der Breitensuche.

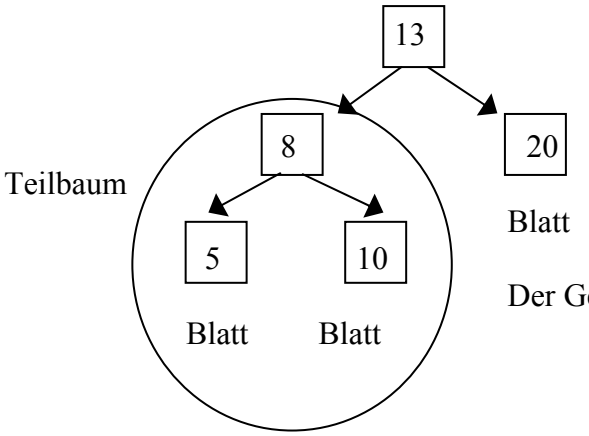
Bäume

Hierarchische Beziehungen zwischen Objekten lassen sich gut in *baumartig verzweigten Strukturen* darstellen, z. B. Gliederungen innerhalb des Tierreichs:



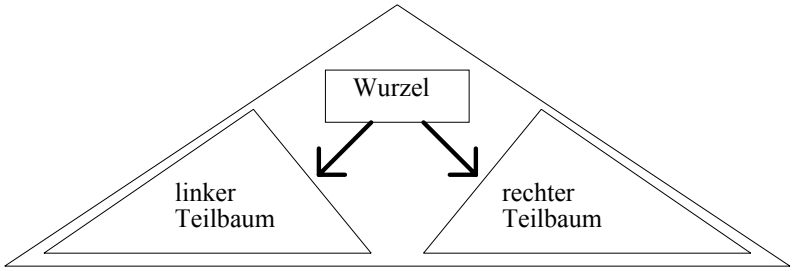
Gehen von jedem Knoten höchstens zwei Verzweigungen aus, so spricht man von einem Binärbaum. Die Verzweigungspunkte in einem Baum werden *Knoten* genannt. In den Knoten stehen die Informationen. Der Anfangsknoten heißt *Wurzel*, die Endknoten heißen *Blätter*. Die Wurzel ist der einzige Knoten, der keinen *Vorgänger* hat. Die Blätter sind die Knoten, die keine *Nachfolger* haben. Die Verbindungen heißen *Kanten*. Sie sind vom Vorgänger zum Nachfolger *gerichtet*.

Wählt man einen bestimmten Knoten eines Baumes als neue Wurzel, so bildet dieser Knoten mit allen dranhängenden Knoten einen *Teilbaum* des ursprünglichen Baumes. Einen Weg, der von der Wurzel eines Baumes bis zu einem Blatt führt, nennt man einen *Pfad*. Die maximale Anzahl der Knoten längs eines Pfades heißt *Tiefe* (oder Höhe) des Baumes.



Der Gesamtbaum hat die Tiefe 3.

Die Vorstellung, dass ein Binärbaum aus einer *Wurzel* besteht, an der ein *linker Teilbaum* und ein *rechter Teilbaum* hängen, die selbst wiederum von dieser Struktur sind, ist eine wichtige Grundvorstellung zur Beschreibung des abstrakten Datentyps und zur Implementation der Methoden:

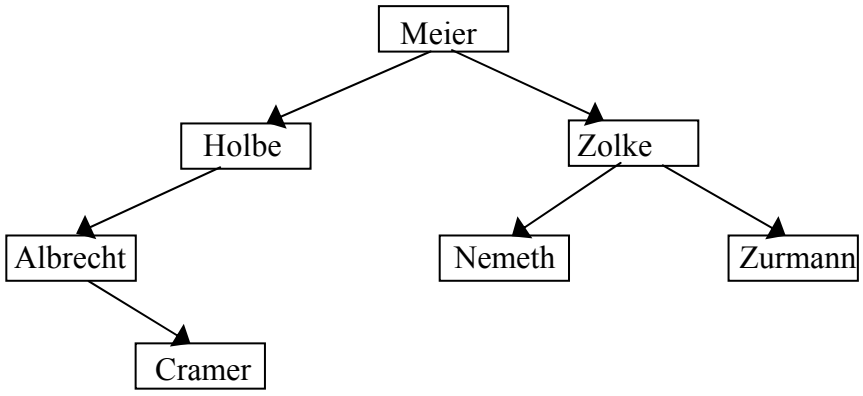


Datenorganisation mit binären Suchbäumen

In Binärbäumen kann man sehr gut Daten verwalten, die nach einem Kriterium geordnet sein sollen. Beispiel: Die Namen Meier, Holbe, Zolke, Albrecht, Nemeth, Zurmann und Cramer sollen in der Reihenfolge eingegeben werden, in der sie hier aufgeführt sind. Sie sollen in einen Binärbaum so angeordnet werden, dass sowohl eine alphabetische Ausgabe als auch eine Suche schnell möglich sind.

Der Name Meier wird als erster eingegeben und steht somit in der Wurzel des Baumes. Wenn man sich dafür entscheidet, im lexikographischen Sinn "kleinere" Namen nach links und lexikographisch "größere" Namen nach rechts zu schreiben, so wird der Name Holbe linker Nachfolger und der Name Zolke rechter Nachfolger von Meier. Da Albrecht im Alphabet vor Meier steht, geht man zum linken Teilbaum von Meier. Jetzt muss mit dem dort schon stehenden Namen Halbe verglichen werden. Albrecht ist kleiner als Halbe, also geht man weiter nach links. Dort steht nichts mehr, und wir können den Namen Albrecht an den Namen Holbe links anhängen. Allgemein kann man dieses Verfahren folgendermaßen formulieren:

Der Inhalt der Wurzel des Baumes wird mit dem neu einzufügenden Element verglichen. Wenn das neue Element kleiner ist als der Wurzelinhalt, geht man zum links anhängenden Teilbaum, sonst zum rechts anhängenden Teilbaum. Dort wiederholt man das gleiche Verfahren für den Teilbaum. Gibt es keinen Teilbaum mehr, so fügt man das neue Element als Blatt an.



Man sieht diesem Baum nicht auf den ersten Blick an, dass er die Namen in einer bestimmten Ordnung enthält. Dies wird erst offensichtlich, wenn man sich überlegt, wie man aufgrund der erhaltenen Struktur alle Namen in alphabetischer Reihenfolge ausgeben kann. Solch einen Durchlauf des gesamten Baumes nennt man *Traversierung*. Eine Sortierung ergibt sich erst aus dem Zusammenspiel von Einordnen und geeignetem Traversieren.

Betrachtet man einen beliebigen Knoten im Baum, so stehen alle Namen, die im Alphabet vor dem im Knoten stehenden Namen liegen, im linken Teilbaum des Knotens. Im rechten Teilbaum stehen nur Namen die größer sind. Deshalb muss an jedem Knoten zuerst der linke Teilbaum ausgegeben werden, dann der Inhalt des Knotens selbst und dann der rechte Teilbaum. Dieses Verfahren nennt man *LWR-Verfahren (Links-Wurzel-Rechts-Verfahren*) oder *Inorder-Verfahren.* Der Algorithmus ist am besten rekursiv zu beschreiben:

Inorder-Verfahren:
WENN der betrachtete Baum nicht leer ist, DANN
    durchlaufe linken Teilbaum nach dem Inorder-Verfahren,
    gib den Wurzelknoten aus,
    durchlaufe rechten Teilbaum nach dem Inorder-Verfahren,
END;

Die Ausgabe lautet: Albrecht, Cramer, Holbe, Meier, Nemeth, Zolke, Zurmann.

Aufgabe

a) Gegeben sind die nachstehenden Buchstabenfolgen:

I) K D S M V B H O A L
II) V S O M L K H D B A
III) A B O D V L M H S K

Die Buchstaben werden jeweils der Reihe nach eingegeben. Kleinere Buchstaben sollen jeweils nach links eingeordnet werden. Zeichne die drei entstehenden Bäume.

b) Welche Knoten der Bäume sind Blätter?

c) Wie tief ist jeder Baum?

d) Traversiere die Bäume nach dem Inorder-Verfahren.

Spezifikation des binären Suchbaums als abstrakter Datentyp

In der *Spezifikation* eines abstrakten Datentyps wird festgelegt, welche Operationen auf dem ADT möglich sind und *Was* diese Operationen bewirken. Es geht nicht um die Frage *Wie* die Operationen das tun. Das ist ein Problem, das der Programmierer in der Phase der Implementierung löst. Grundsätzlich gilt, das eine spezifizierte Operation auf unterschiedliche Arten implementiert werden kann. Den ADT *binärer Suchbaum* spezifizieren wir wie folgt:

ADT binärer Suchbaum

IstLeer	Prüft, ob der Suchbaum leer ist.
Einfuegen	Fügt ein Element in den Suchbaum ein.
Vorhanden	Prüft, ob ein Element im Suchbaum enthalten ist.
Inorder	Gibt die Knoten des Suchbaumes im Inorder-Durchlauf an

Datenstruktur für einen Knoten

Wir unterscheiden beim Binärbaum zwischen den einzelnen Elementen also den Knoten und der Gesamtstruktur also dem eigentlichen Baum. Ein Knoten enthält die Nutzdaten, im Beispiel einfach eine Integer-Zahl, sowie Attribute für die am Knoten hängenden linken und rechten Nachfolgerknoten.

```
class BinKnoten {
  int Zahl;                  // Nutzdaten
  BinKnoten Links, Rechts;   // linker und rechter Nachfolge-Knoten

  public BinKnoten (int Zahl) {
    this.Zahl = Zahl;
    Links = null;
    Rechts = null;
  }
}
```

Bisher haben wir *Rekursion* nur bei Algorithmen gesehen. Dies ist nunmehr ein Beispiel für eine *rekursive Datenstruktur*. Die Klasse *BinKnoten* wird über sich selbst definiert, die Attribute Links und Rechts haben den gleichen Typ wie die zu definierende Klasse.

Datenstruktur für den binären Suchbaum

Die Datenstruktur für den binären Suchbaum enthält den Wurzel-Knoten, einen aktuellen Knoten zur Implementierung von Operationen, sowie die Anzahl der Knoten im Baum.

```
class BinBaum {
  private BinKnoten Wurzel;     // die Wurzel des binären Baumes
  private int Anzahl;           // die Anzahl der Knoten im Baum

  public BinBaum() {
    Wurzel = null;
    Anzahl = 0;
  }

  public void Einfuegen(int Zahl) {
    BinKnoten NeuKnoten = new BinKnoten(Zahl);
    Anzahl++;

    if (Wurzel == null) {
      Wurzel = NeuKnoten;
    } else {
      BinKnoten AktKnoten = Wurzel;
      while (true) {
        if (Zahl < AktKnoten.Zahl) {
          if (AktKnoten.Links == null) {
            AktKnoten.Links = NeuKnoten; break;
          } else {
            AktKnoten = AktKnoten.Links;
          }
        } else {
          if (AktKnoten.Rechts == null) {
            AktKnoten.Rechts = NeuKnoten; break;
          } else {
            AktKnoten = AktKnoten.Rechts;
          }
        }
      }
    }
  }

  private String InOrder(BinKnoten Knoten) {
    if (Knoten == null) {
      return "";
    } else {
      String Li = InOrder(Knoten.Links);
      String Daten = " " + Knoten.Zahl + " ";
      String Re = InOrder(Knoten.Rechts);
      return Li + Daten + Re;
    }
  }

  public String InOrder() {
    return InOrder(Wurzel);
  }

  ... // weitere Methoden

}
```

Aufgaben

1. Analysiere die BinBaum-Klasse.
2. Implementiere die Methoden *IstLeer* und *Vorhanden*.

Lösungen

```
  public boolean IstLeer() {
    return Anzahl == 0;
  }

  public boolean Vorhanden(int Zahl) {
    BinKnoten AktKnoten = Wurzel;
    while ((AktKnoten != null) && (AktKnoten.Zahl != Zahl)) {
      if (Zahl < AktKnoten.Zahl) {
        AktKnoten = AktKnoten.Links;
      } else {
        AktKnoten = AktKnoten.Rechts;
      }
    }
    return (AktKnoten != null);
  }
```

Weitere Aufgaben

1. Der bisherige Einfügen-Algorithmus fügt Doubletten in den binären Suchbaum ein. Implementiere einen zweiten Algorithmus, der eine Zahl, die schon im Suchbaum enthalten ist, nicht noch einmal einfügt.

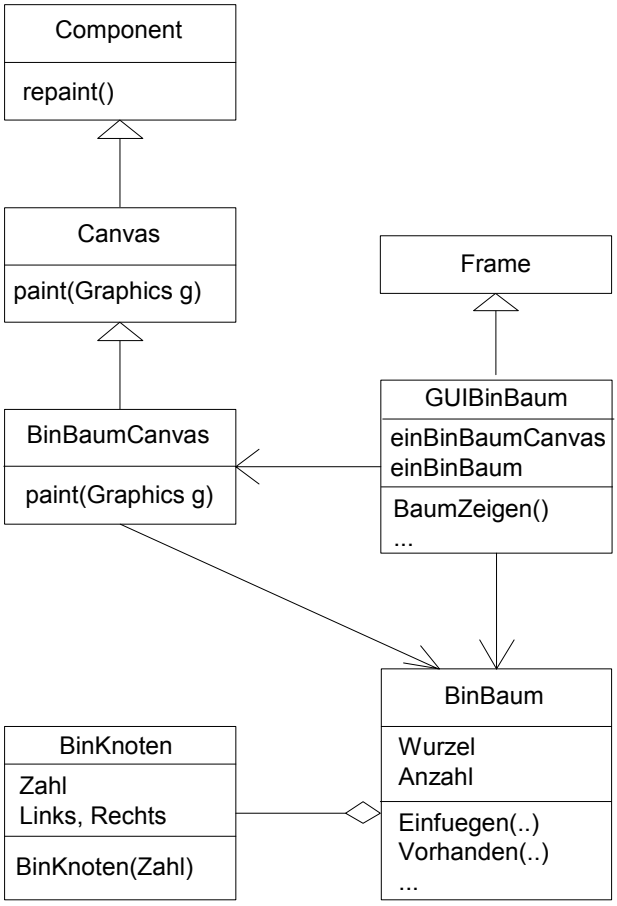
2. Entwickle und implementiere Algorithmen zum Zählen
a) der Knoten
b) der Kanten
c) der Blätter
d) der Knoten mit geraden Zahlen
e) zur Berechnung der Summe aller Zahlen
in einem binären Suchbaum.

3. Analysiere diese Methoden:

```
  private void Einfuegen(BinKnoten AktKnoten, BinKnoten NeuKnoten) {
    if (NeuKnoten.Zahl < AktKnoten.Zahl) {
      if (AktKnoten.Links == null) {
        AktKnoten.Links = NeuKnoten;
      } else {
        Einfuegen(AktKnoten.Links, NeuKnoten);
      }
    } else {
      if (AktKnoten.Rechts == null) {
        AktKnoten.Rechts = NeuKnoten;
      } else {
        Einfuegen(AktKnoten.Rechts, NeuKnoten);
      }
    }
  }

  public void Einfuegen(int Zahl) {
    Anzahl++;
    BinKnoten NeuKnoten = new BinKnoten(Zahl);
    if (Wurzel == null) {
      Wurzel = NeuKnoten;
    } else {
      Einfuegen(Wurzel, NeuKnoten);
    }
  }
```

BinBaumCanvas

Zum Zeichnen benutzt man in Java ein Canvas-Objekt (engl. Leinwand, Zeichenfläche). Ein solches Objekt kann man aber nicht direkt benutzen. Vielmehr muss man von Canvas eine eigene Klasse ableiten. Wir leiten von Canvas die BinBaumCanvas-Klasse ab, die einen binären Suchbaum grafisch darstellen soll.

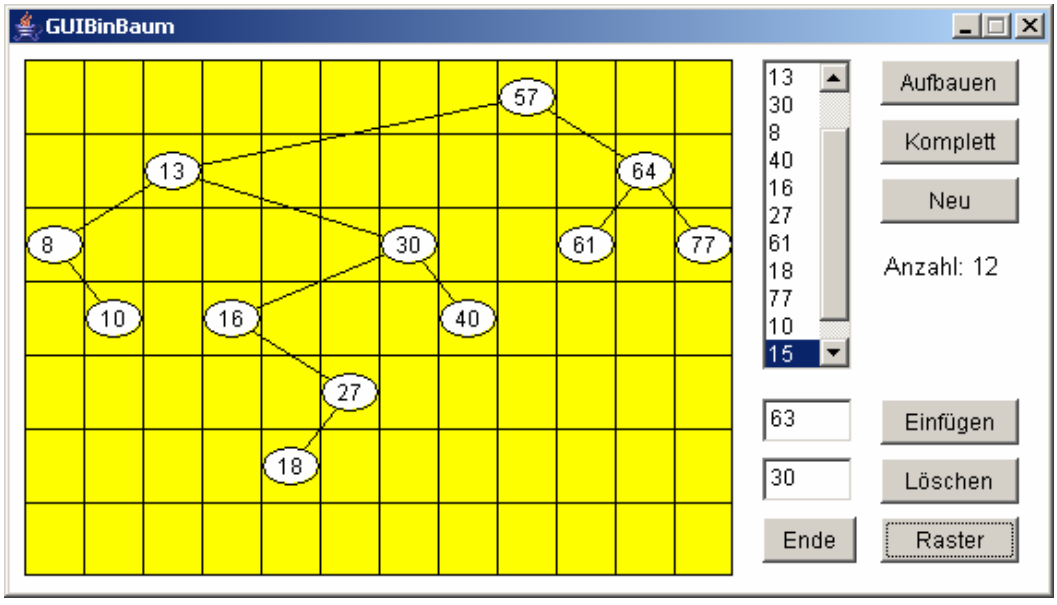


Die Canvas-Klasse hat eine Methode paint(Graphics g), mit der die grafische Ausgabe erzeugt wird. Bei einem Canvas-Objekt wird lediglich die Zeichenfläche gelöscht. Aus diesem Grund muss man eine neue Klasse von Canvas ableiten, um eine eigene Grafikausgabe machen zu können. In der abgeleiteten Klasse überschreibt man die paint(Graphics g)-Methode und lässt sie die gewünschte grafische Ausgabe erzeugen. Die verfügbaren Grafikbefehle findet man in der Klasse Graphics.

Ändert man den binären Suchbaum, so soll er frisch gezeichnet werden. Dazu benutzt man die Methode repaint(). Dies ist eine Methode, die die Klasse Canvas von der Klasse Component erbt.

Zwischen der Klasse BinBaumCanvas und BinBaum besteht eine gerichtete Assoziation, den schließlich muss BinBaumCanvas wissen, welchen binären Suchbaum es darstellen soll. Die Assoziation wird am einfachsten schon beim Erzeugen des BinBaumCanvas-Objekts hergestellt, indem man beim Konstruktor den darzustellenden binräen Suchbaum angibt:

```
public BinBaumCanvas(BinBaum einBinBaum) {
  meinBinBaum = einBinBaum;
}
```



Das GUI-Programm, mit dem der binäre Suchbaum bearbeitet und dargestellt werden kann, hat natürlich ein BinBaumCanvas-Objekt und einen binären Suchbaum als Attribute. Beide sind durch Assoziationen mit dem GUI-Programm verbunden.

Der binäre Suchbaum besteht aus einer Aggregation von BinKnoten.

Löschen im binären Suchbaum

Löschen eines Knotens im Binärbaum zählt zu den schwierigsten Operationen. Drei Fälle müssen unterschieden werden:

- Löschen eines Blatts
- Löschen eines Knotens mit nur einem Teilbaum
- Löschen eines Knotens mit zwei Teilbäumen

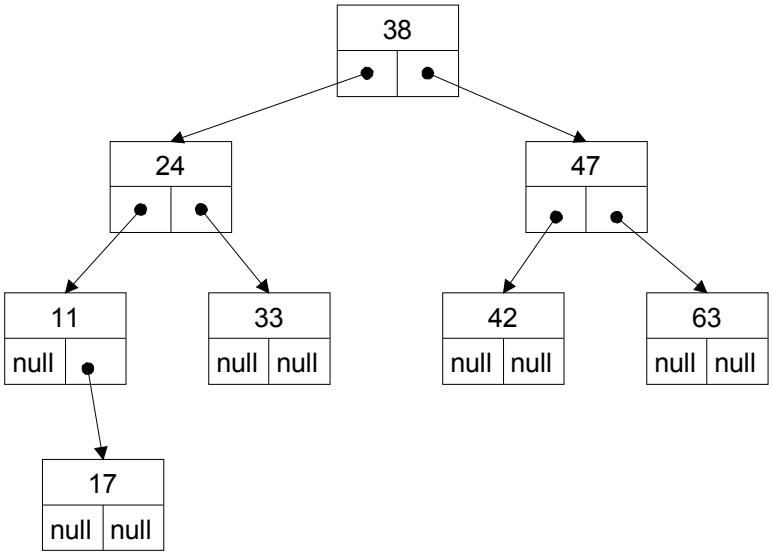
Der erste und zweite Fall sind relativ einfach zulösen. Löscht man aber einen Knoten, an dem zwei Teilbäume dran hängen, hat man ein Problem mit dem zweiten Teilbaum. Wohin damit?

Eine Idee besteht darin, den zweiten Teilbaum an einen passenden Knoten des ersten Teilbaums dran zu hängen. Das führt aber dazu, dass die Höhe des Baumes drastisch zunimmt. Jede nachfolgende Such- und Einfügeoperation dauert dann erheblich länger. Die Löschoperation sollte mit möglichst geringem Aufwand und weitgehender Beibehaltung der Struktur durchgeführt werden kann. Wie geht das? Nun dadurch, dass man den Inhalt des zu löschenden Knotens K_1 durch den Inhalt des nächst kleineren oder nächst größeren Knotens K_2 ersetzt und dafür dann K_2 löscht. Da der Knoten K_2 nur einen Teilbaum hat (warum eigentlich?), kann er leicht gelöscht werden.

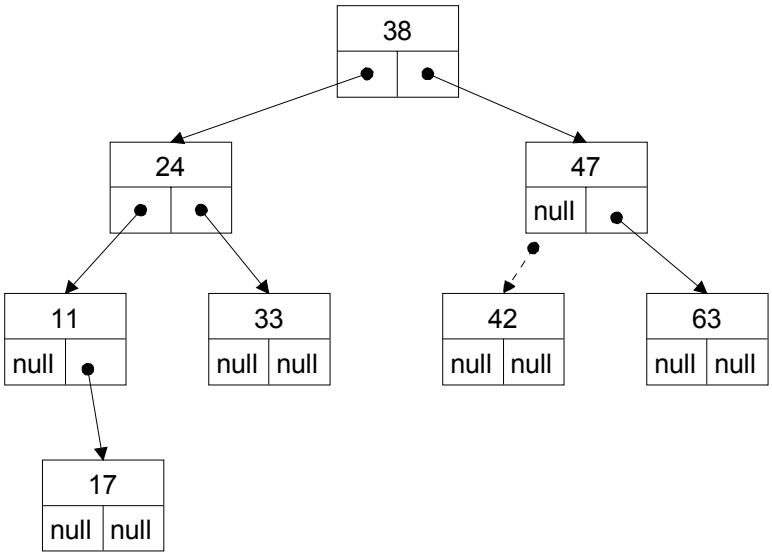
Für die weitere Arbeit ist die nachfolgende Darstellung eines binären Suchbaums sinnvoll. Sie zeigt visuell an, dass ein einzelner Knoten aus drei Bestandteilen besteht:

- den Nutzdaten im Attribut Zahl
- dem Verweis auf den linken Teilbaum
- dem Verweis auf den rechten Teilbaum

Die Verweise sind nichts anderes als die Adressen der Wurzeln der Unterbäume. Diese Adressen werden zur übersichtlichen Darstellung als Pfeile gezeichnet. Die Adresse **null** zeigt an, dass es keinen weiteren Verweis gibt.



Soll nun ein Blatt gelöscht werden, beispielsweise das mit der Zahl 42, so muss der Baum wie im Bild geändert werden. Tragen wir beim Vater des zu löschenden Knotens in die linke Verweiszelle **null** ein, so haben wir keinen Zugriff mehr auf diesen Knoten. Das Java-System entfernt diesen Knoten mit seiner systemeigenen Müllabfuhr (gargabe collection), die alle Objekte entsorgt, auf die das Benutzerprogramm keinen Zugriff mehr hat. Den eigentlichen Löschvorgang des Knotens 42 führen wir also gar nicht durch.



Löschen eines Blatts

Das Löschen eines Blatts ist etwas kompliziert, weil man beim Vaterknoten entweder das Attribut Links oder das Attribut Rechts auf **null** setzen muss. Wenn man schon beim Blatt ist, hat man aber keinen Zugriff mehr auf den Vater-Knoten. Sucht man den Vater eines Knotens, dann ist es etwas diffizil festzustellen, ob denn der linke bzw. rechte Sohn-Knoten ein Blatt ist. Außerdem ist da noch der Sonderfall zu berücksichtigen, dass der Knoten gar keinen Vater hat, also die Wurzel ist.

Man kann sich die Sache dadurch erleichtern, dass man zunächst mal den Knoten sucht, der die zu löschende Zahl einhält und dann prüft, ob es ein Blatt, ein Knoten mit einem oder mit zwei Teilbäumen ist. Ist es es Blatt, so sucht man nochmals von der Wurzel her, bis man den Vater-Knoten gefunden hat.

Löschen eines Knotens mit einem Teilbaum

Einen Knoten zu löschen, der genau einen Teilbaum ist der einfachste Fall. Man überschreibt den zu löschenden Knoten einfach mit der Wurzel des Teilbaumes.

Löschen eines Knotens mit zwei Teilbäumen

Dies ist der schwierigste Fall, denn was soll man mit den beiden Teilbäumen anfangen, die an dem zu löschenden Knoten dranhängen? Viele verschiedene Vorgehensweisen sind denkbar, doch die folgende sorgt dafür, dass man mit möglichst wenig Aufwand und ohne allzu sehr die Struktur des binären Suchbaums zu verändern, löscht. Man ersetzt den Inhalt des zu löschenden Knotens K durch den Inhalt, der in der LWR-Ordnung direkt vor (K_v) oder direkt nach (K_n) dem Knoten K kommt und löscht stattdessen den Knoten K_v bzw. K_n. Diese beiden Knoten haben einen oder keinen Teilbaum, können also nach den ersten beiden Verfahren gelöscht werden.

Vergleich sortiertes Feld – binärer Suchbaum

Im Vergleich zu den Algorithmen auf sortierten Feldern macht die Programmierung der Standardoperationen Einfügen, Suchen und Löschen auf binären Suchbäumen erheblich mehr gedanklichen Aufwand, ist deutlich schwieriger und fehleranfälliger. Wozu also der Aufwand? Ein wesentlicher Grund liegt darin, dass binäre Suchbäume eine dynamische Datenstruktur bilden, die zur Laufzeit des Programms wachsen und schrumpfen kann, während man bei Feldern schon zur Compilezeit die maximale Feldgröße festlegen muss. Der zweite wesentliche Grund liegt in der Zeitkomplexität der Standardoperationen.

Beim Suchen liegen sortiertes Feld und binärer Suchbaum effizienzmäßig gleichauf. Bei n Elementen dauert die Suchoperation durchschnittlich $\log_2(n)$-Vergleiche (siehe Seite 16-17). Zum Einfügen in ein sortiertes Feld wird man erst die Einfügestelle suchen, das macht rund $\log_2(n)$-Vergleiche, dann schafft man eine Lücke und muss dazu im Schnitt n/2-Bewegungen ausführen. Die Zeitkomplexität ist demnach im Wesentlichen durch die Bewegungen bestimmt, sie nimmt somit nicht logarithmisch, sondern linear mit der Anzahl der Elemente im sortierten Feld zu. Dafür schreibt man kurz O(n) (O-Notation). Beim binären Suchbaum geht das Einfügen genauso schnell wie das Suchen, daher ist hier die Zeitkomplexität logarithmisch von der Anzahl der Knoten abhängig, wofür man O(log n) schreibt. Analoge Ergebnisse erhält man bei der dritten Grundoperation, dem Löschen.

im Durchschnitt	**sortiertes Feld**	**binärer Suchbaum**
Einfügen	O(n)	O(log n)
Suchen	O(log n)	O(log n)
Löschen	O(n)	O(log n)

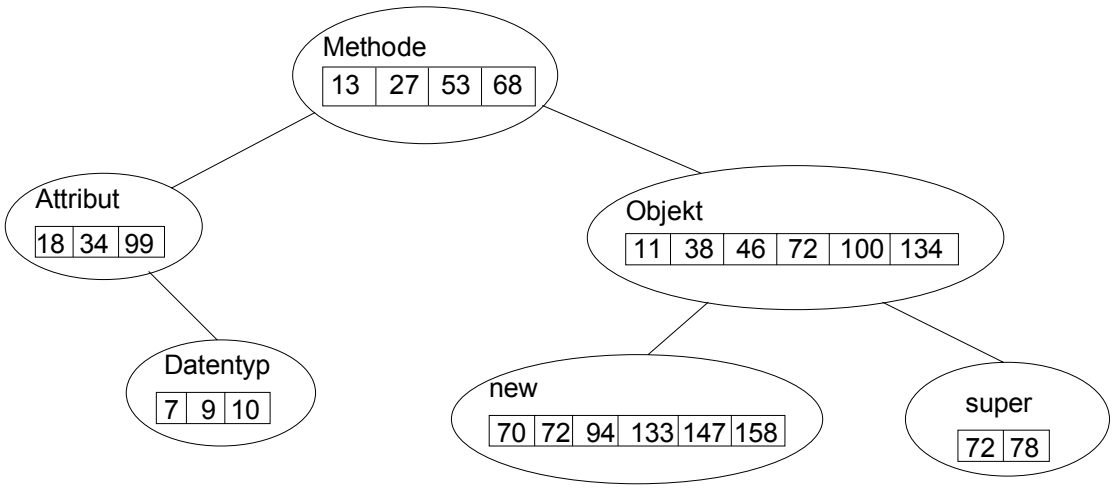
Beim binären Suchbaum haben alle drei Grundoperationen nur logarithmische Zeitkomplexität, weswegen er bei der Verwaltung großer Datenmengen deutlich effizienter als das sortierte Feld ist.

Anwendung des binären Suchbaums - Index-Erstellung

Als komplexeres Anwendungsbeispiel der dynamischen Datenstrukturen betrachten wir die automatisierte Erstellung eines Index. Fachbücher brauchen einen Index der zu Fachbegriffen die Seitenzahlen angibt, an denen der Fachbegriff im Buch vorkommt. Das Javabuch in HTML-Form hat einen Index, da es aber nicht auf Seiten basiert, verweist der Index auf die entsprechenden HTML-Dokumente. In Word kann man über Einfügen/Index und Verzeichnisse/Eintrag Wörter für den Index festlegen. Alternativ gibt man eine Datei der Indexwörter vor. Auf derselben Registerkarte erzeugt der Schalter OK den Index für die zuvor festgelegten Index-Wörter.

Der erzeugte Index muss die Index-Wörter in alphabetischer Reihenfolge auflisten und die zugehörigen Seitenzahlen in numerisch aufsteigender Folge. Es sind natürlich viele verschiedene Ansätze zur Erzeugung eines Index denkbar. Im Hinblick auf unsere vertieften Informatik-Kenntnisse wählen wir den besonders Erfolg versprechenden Weg über einen binären Suchbaum. Fügt man die Index-Wörter aus dem Text in einen binären Suchbaum ein, so erzeugt am Ende die LWR-Ausgabe die gewünschte alphabetische Reihenfolge.

Da Index-Wörter auf mehreren verschiedenen Seiten vorkommen können, müssen wir uns bei jedem Wort eine Liste der Seitenzahlen merken. Diese Liste ist im Prinzip eine Warteliste, kann aber auch einfach als Vector realisiert werden. Im folgenden Bild ist ein Beispiel eines so konstruierten Indexbaumes dargestellt.



Aufgabe

Erstelle ein Programm zur Indexerstellung für einen Text, der als Textdatei vorliegt. Ein Wort, das an einer bestimmten Stelle im Text in den Index aufgenommen werden soll, soll durch ein vorangestelltes ^-Zeichen markiert sein.

Textdateien

Programme können Daten wie z. B. Spielstände in Textdateien abspeichern oder von dort aus einlesen. Textdateien enthalten im Gegensatz zu Textdokumenten nur unformatierten Text. Der Text selbst ist zeilenweise strukturiert. Das Ende einer Zeile wird durch Steuerzeichen festgelegt. Unter DOS und Windows werden dazu zwei Steuerzeichen CarriageReturn (CR, Wagenrücklauf, #13, \r) und Linefeed (LF, neue Zeile, 10, \n) benutzt, unter Unix wird nur das CarriageReturn und beim Mac nur Linefeed verwendet. Im ASCII-Code haben diese Steuerzeichen die Codes 13 bzw. 10, in Java gibt man sie mit \r und \n an.

Java bietet für die Arbeit mit Textdateien die Klassen FileWriter und FileReader im IO-Package an. Beim Aufruf des Konstruktors gibt man den Dateinamen an. Er öffnet dann die angegebene Datei zum Schreiben bzw. Lesen. Dabei gilt es zu beachten, dass der Verzeichnis-Separator von Unix / und nicht der Backslash \ von DOS und Windows benutzt wird. Da die Ein- und Ausgabe von Dateien grundsätzlich zu Laufzeitfehlern führen kann (volle Festplatte, falscher Pfad, fehlende Rechte, ...) müssen die Ein-/Ausgabeoperationen in einem try-catch-Block stehen. Nachdem die Textdatei gelesen oder geschrieben ist, muss man sie mittels close wieder schließen. Mit der Methode write kann man zwar einen ganzen String ausgeben, aber die entsprechende Methode read liest nur ein einziges int-Zeichen ein, das mittels Typcasting (vgl. S. 43) als char-Zeichen interpretiert werden kann.

```
import java.io.*;

public class ReaderWriter {

  public static void main(String[] args) {
    try {
      FileWriter myFileWriter = new FileWriter("c:/temp/test.txt");
      myFileWriter.write("Hallo,\r\n");
      myFileWriter.write("ich gehöre in\r\n");
      myFileWriter.write("eine Textdatei!\r\n");
      myFileWriter.close();
    } catch (IOException e) {
      System.out.println("Fehler: Konnte Datei nicht erstellen!");
    }

    try {
      FileReader myFileReader = new FileReader("c:/temp/test.txt");
      while (myFileReader.ready()) {
        int Zeichen = myFileReader.read();
        System.out.print((char) Zeichen);
      }
      myFileReader.close();
    } catch (IOException e) {
      System.out.println("Fehler: Konnte Datei nicht öffnen!");
    }
  }
}
```

Komfortabler sind die Klassen *BufferedReader* und *BufferedWriter*. Sie puffern mittels eines internen Speichers die Eingabe und Ausgabe ab, was zu weniger Schreib/Lesevorgängen führt und damit die Performance erhöht. Der erhöhte Komfort rührt daher, dass BufferedWriter ein explizites betriebssystemunabhängiges *newLine*() enthält und BufferedReader ganze Zeilen mittles *readLine*() lesen kann.

Die Konstruktoren von BufferedReader und BufferedWriter werden nicht mit den Dateinamen aufgerufen, sondern mit Reader/Writer-Objekten, also für unseren Bedarf mit abgeleiteten FileReader/ FileWriter-Objekten.

```
import java.io.*;

public class BufferedReaderWriter {

  public static void main(String[] args) {
    try {
      BufferedWriter myBufferedWriter =
        new BufferedWriter(new FileWriter("c:/temp/test.txt"));
      myBufferedWriter.write("Hallo,"); myBufferedWriter.newLine();
      myBufferedWriter.write("ich gehöre in\n"); // jetzt mit \n statt newLine()
      myBufferedWriter.write("eine Textdatei!\n");
      myBufferedWriter.close();
    } catch (IOException e) {
      System.out.println("Fehler: Konnte Datei nicht erstellen!");
    }

    try {
      BufferedReader myBufferedReader =
        new BufferedReader(new FileReader("c:/temp/test.txt"));
      while (myBufferedReader.ready()) {
        String Zeile = myBufferedReader.readLine();
        System.out.println(Zeile);
      }
      myBufferedReader.close();
    } catch (IOException e) {
      System.out.println(" Fehler: Konnte Datei nicht öffnen!");
    }
  }
}
```

Erkennung von Wörtern

Zum Erkennen der Wörter in einer Zeile setzen wir einen StringTokenizer ein (siehe S. 42). Diesmal ist allerdings so, dass die Satzzeichen nicht als Token geliefert werden sollen:

```
StringTokenizer mySTN = new StringTokenizer(Zeile, " .:,;?[]!", false);
```

Backtracking - ein allgemeines Suchverfahren

Erbteilung (7. Bundeswettbewerb Informatik, 3. Aufgabe)

Die Baronin von Birlinghoven hat ihren beiden Töchtern eine Truhe voller Goldmünzen hinterlassen. Ihr Testament bestimmt, dass das Gold einem benachbarten Kloster zukommt, falls es den Töchtern nicht gelingt, den Inhalt der Truhe wertmäßig genau in zwei Hälften untereinander aufzuteilen. Die Goldmünzen haben nur ganzzahlige Werte.

Beispiel:

Eine Truhe Goldmünzen mit den Werten 1, 9, 5, 3, 8 Taler könnten die Töchter in die Hälften 1, 9, 3 Taler und 5, 8 Taler teilen.

Aufgabe:

Schreibe ein Programm, das bei Eingabe einer Folge ganzer Zahlen für die in der Truhe vorkommenden Werte die beiden Erbteile aufzählt, anderenfalls das Erbe dem Kloster zuspricht, wenn eine Aufteilung nicht möglich ist.

Schicke uns mindestens 5 Beispiele mit verschiedenen Truheninhalten. Der Inhalt einer Truhe sei: 15, 27, 39, 7, 23, 56, 13, 39, 22, 5, 42, 34

Rucksackproblem

Ein Rucksack mit Proviant zu packen ist gar nicht so einfach, wenn man mehr Proviant hat, als man tragen kann. Dosen sind schwer, haben aber meistens einen kräftigen Inhalt. Zwieback ist leicht, macht aber nicht so satt. Wenn eine weitere Dose den Rucksack zu schwer machen würde, könnte man stattdessen vielleicht noch ein Päckchen Zwieback einpacken.

In der folgenden Tabelle sind Gewichte von Nahrungsmitteln und deren jeweiliger Nährwert angegeben. Welche Nahrungsmittel muss man einpacken, um einen maximalen Nährwert zu erhalten, ohne das zulässige Maximalgewicht von 173 zu überschreiten?

Gewicht	15	18	20	21	22	25	26	28	30	33
Nährwert	18	20	19	25	22	30	24	26	23	28

CD-Problem

In einer Plattenfirma kommt es des Öfteren vor, dass eine Reihe von Musikstücken auf eine Doppel-CD verteilt werden muss. Dabei sollen die Spieldauern beider CDs möglichst gleich lang sein.

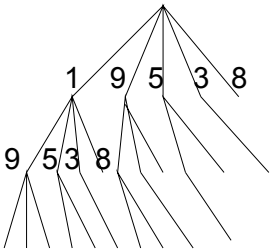
Musikstück Nr.	1	2	3	4	5	6
Spieldauer in min	24	22	19	18	13	11

Das Backtracking-Verfahren

Backtracking ist ein Lösungsverfahren, bei dem man versucht, eine Teillösung eines Problems systematisch zu einer Gesamtlösung auszubauen. Falls in einem gewissen Stadium ein weiterer Ausbau nicht mehr möglich ist (Sackgasse) werden einer oder mehrere der letzten Teilschritte rückgängig gemacht. Die dann erhaltene reduzierte Teillösung versucht man auf einem anderen Weg wieder auszubauen. Das Zurücknehmen von Schritten und erneute Vorangehen wird solange wiederholt, bis eine Lösung des vorliegenden Problems gefunden ist oder man erkennt, dass das Problem keine Lösung besitzt. Die Möglichkeit, in Sackgassen zu laufen und aus ihnen wieder herauszufinden, zeichnet das Backtracking-Verfahren aus.

Die Lösungssuche erfolgt beim Backtracking nicht durch Befolgen einer direkten Vorschrift sondern durch systematisches Versuchen und Nachprüfen (trail-and-error), bei dem alle möglichen Wege zu einer Lösung untersucht werden.

Damit die Lösungssuche *alle* Lösungskandidaten begutachtet muss sie systematisch angelegt sein. Unter zu Hilfenahme eines Suchbaums organisiert man die Systematik. Anschließend nimmt man den stets gleichen Algorithmus zum Absuchen des Suchbaums.



Algorithmus *SucheLösung (k)*

```
für i = k bis Anzahl Taler
    Nimm Taler i
    Falls Erbteilhälfte erreicht
        dann gib Lösung aus
        sonst SucheLösung (i + 1)
    Lege Taler i zurück
```

Scheinbar scheinen Grafik und Algorithmus gänzlich verschiedene Dinge zu sein. Tatsächlich gibt es eine frappierende Entsprechung. Um alle Möglichkeiten an einer Stelle im Baum zu untersuchen benötigt man die for-Schleife. Wird eine Möglichkeit ausgewählt geht es entlang einer Kante im Baum weiter - im Problem bedeutet dies das Nehmen eines Talers. Erreicht man damit eine Lösung gibt man die aus, anderenfalls ruft man rekursiv den Algorithmus für die bislang erreichte Münzauswahl - also für den erreichten Knoten - wieder auf. Führt ein Weg im Baum in eine Sackgasse legt man die zuletzt ausgewählte Münze zurück, geht also im Baum eine Kante zurück und versucht von dort aus weitere Lösungskandidaten. Die Suche kann erheblich eingeschränkt werden, wenn man nur Münzauswahlen kleiner gleich der Erbteilhälfte betrachtet:

Algorithmus SucheLösung(i)

<table>
<tr><td colspan="3">Wiederhole für i=k bis Anzahl Taler</td></tr>
<tr><td colspan="2">ja — Taler annehmbar? — nein</td><td rowspan="4">./.</td></tr>
<tr><td colspan="2">Nimm Taler i</td></tr>
<tr><td colspan="2">ja — Erbteilhälfte erreicht? — nein</td></tr>
<tr><td>Lösung ausgeben</td><td>SucheLösung(i+1)</td></tr>
<tr><td colspan="2">Lege Taler i zurück</td></tr>
</table>