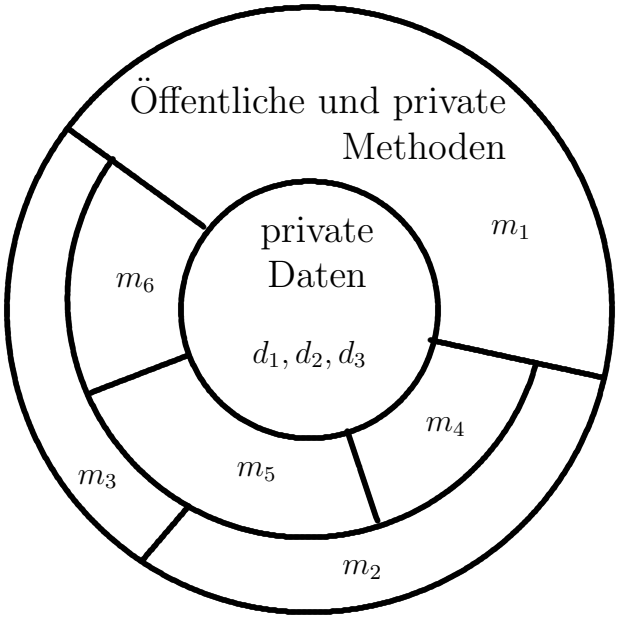
HOCHSCHULE FURTWANGEN UNIVERSITY HFU

Objektorientierte Programmierung in JAVA

Skript zum Modul "Algorithmen und Datenstrukturen" (Studiengang AIN)



Prof. Dr. Wolfgang Rülling

Contents

1 Einleitung

Das Modul **Algorithmen und Datenstrukturen** im zweiten Semester des Studiengangs AIN (Algemeine Informatik) beinhaltet die beiden Themengebiete **Effiziente Algorithmen und Datenstrukturen** und **Objektorientierte Programmierung in Java**. Im vorliegenden Skript wird die objektorientierte Programmierung in Java behandelt. Dabei werden allgemeine C/C++ Programmierkenntnisse, wie sie bereits im 1. Semester vermittelt wurden, als bekannt vorausgesetzt, so dass der Schwerpunkt auf dem Erlernen der objektorientierten Vorgehensweise liegt. Durch die Auswahl der verwendeten Programmierbeispiele sollen zudem die Kenntnisse über effiziente Algorithmen und Datenstrukturen vertieft werden.

Während sich die Programmiersprache C++ insbesondere zur Programmierung von technischen Anwendungen eignet, bei denen es auf eine schnelle Ausführung der Programme ankommt, liegt der Schwerpunkt von Java bei der Hardwareunabhängigkeit der erstellten Programme. Motiviert wurde die Entwicklung von Java durch Internetanwendungen. So sollte es möglich sein, Software auf Webseiten anzubieten, die die Anwender lokal auf ihren jeweiligen Rechnern ausführen können. Insbesondere muss die Software dazu unabhängig vom jeweiligen Betriebssystem (z.B. WINDOWS, LINUX, Solaris, OS/2, ...) und der jeweiligen Datenwortlänge der Maschine immer das gleiche Verhalten zeigen.

Erreicht wird diese Hardwareunabhängigkeit dadurch, dass die Java-Programme nicht in einen Maschinencode übersetzt werden, sondern in eine Interndarstellung, die von einer virtuellen Java-Maschine, dem **Java-Laufzeitsystem interpretiert** wird. Solche Java-Interpreter sind inzwischen für praktisch alle Rechner und Betriebssysteme verfügbar. Insbesondere sind sie in den gängigen Internet-Browsern integriert, so dass man Java-Programme immer dann ausführen kann, wenn ein Internet-Browser verfügbar ist.

Unter anderem hat der Java-Interpreter dafür zu sorgen, dass die arithmetischen Operationen überall mit der gleichen Genauigkeit ausgeführt werden. Das bedeutet beispielsweise, dass eine "elementare Operation" wie etwa eine Multiplikation ganzer Zahlen auf einem Rechner mit zu kleiner Datenwortlänge eventuell durch eine Software-Routine realisiert werden muss, die auf einem Rechner mit ausreichender Datenwortlänge vielleicht mit einem einzigen Maschinenbefehl realisiert wird. Das Beispiel zeigt, dass man je nach Hardware und Betriebssystem entsprechend angepasste Java-Interpreter braucht. Das vom Anwender geschriebene Java-Programm ist jedoch nach seiner Compilierung auf allen Plattformen verwendbar.

Durch den Einsatz des Interpreters ist die Ausführung von Java-Programmen im allgemeinen langsamer als die Ausführung von C++-Programmen. Allerdings wird dieser Nachteil teilweise durch sehr effiziente Bibliotheken und durch die Möglichkeit einer "Just-in-Time"-Compilierung bei der Java-Anweisungen nicht wiederholt interpretiert werden müssen, kompensiert.

1.1 Vergleich zwischen C++ und Java

Die Syntax von Anweisungen ist in Java und C++ nahezu identisch. Beispielweise gibt es in beiden Sprachen Fallunterscheidungen durch "`if...else`"-Anweisungen, und als Schleifenarten die `while`-Schleife, die `do`-Schleife und die `for`-Schleife. Deshalb ist die Sprache Java recht einfach zu erlernen, wenn bereits C oder C++-Kenntnisse vorhanden sind.

Auch die elementaren Datentypen wie `int`, `char`, `float`, ... sehen auf den ersten Blick genauso aus. Hier ist allerdings zu beachten, dass die Datenlänge der Typen in Java exakt definiert ist und nicht vom verwendeten Rechner abhängt. Für den Typ `char` wird beispielsweise eine Unicode-Codierung verwendet, die jedes Zeichen durch 2 Byte darstellt. Dieser Code ist auf den ersten 127 Zeichen mit dem sonst üblichen ASCII-Code kompatibel. Er ist auch außerhalb von Java weit verbreitet, da er groß genug ist um auch viele länderspezifische Sonderzeichen aufzunehmen.

Um die Hardwareunabhängigkeit der Sprache zu gewährleisten, findet in Java eine sehr **strenge Typprüfung** statt. Deshalb ist es beispielsweise nicht möglich, einen Datenwert vom Typ `int` als Wahrheitswert zu interpretieren. Wird dies benötigt, so muss man zuvor für eine explizite Typkonvertierung sorgen.

Während die Typprüfungen bei C++ nur im Compiler stattfinden, so dass man das System gegebenenfalls durch Zusammenbinden getrennt übersetzter Programmteile "überlisten" kann, führt Java gegebenfalls auch **Typprüfungen zur Laufzeit** des Programms durch. Auf diese Weise wird der Programmierer zu einer "sauberen" Programmierung gezwungen, d.h. die erstellten Programme sind weniger fehleranfällig.

Eine weitere Fehlerquelle in C++ ist die **Verwendung von Zeigern**. Hier besteht die Gefahr, dass ein Zeiger falsch initialisiert ist oder auf einen inzwischen nicht mehr reservierten Speicherplatz zeigt. In Java wird diese Problematik dadurch vermieden, dass es keine (expliziten) Zeiger gibt. Insbesondere kann man in Java nicht mit Speicheradressen rechnen. Der Vorteil von Zeigern, dass man über verschiedene Variablen auf die gleichen Daten zugreifen kann, wird stattdessen durch **Referenzvariablen** erreicht.

Bei der **dynamischen Speicherverwaltung** liegt es bei C++ in der Verantwortung des Programmierers, den verwendeten Speicherplatz irgendwann wieder freizugeben. Auch dies ist relativ fehleranfällig und wird oft falsch gemacht. In Java wird der Programmierer deshalb von dieser Aufgabe befreit. Stattdessen gibt es einen **Garbage Collector**, der selbständig feststellt, welche Daten im Speicher nicht mehr benötigt werden und die entsprechenden Speicherbereiche wieder freigibt.

Der wohl wichtigste Unterschied zwischen C++ und Java besteht darin, dass Java eine streng objektorientierte Sprache ist. Das bedeutet, dass es in Java keinerlei Programmcode außerhalb von Klassen gibt. In Java wird der Programmierer also gezwungen, objektorientiert zu programmieren. Das sollte zu besonders leicht wiederverwendbarem Code führen.

1.2 Primitive Datentypen

In Java gibt es einige fest eingebaute **primitive Datentypen**. Im Gegensatz zu C++ ist jedoch die jeweilige Datenlänge, bzw. die Genauigkeit der Daten rechnerunabhängig definiert. Außerdem werden die Variablen standardmäßig je nach Typ mit `0`, `0.0` bzw. `false` initialisiert.

`boolean` Wahrheitswerte sind 1 Byte lang und können nur die Werte `false` und `true` annehmen.

```
boolean b= false;
if (5>3) b=true;
```

`char` Die Zeichen des Zeichensatzes werden als Unicode-Zeichen mit 2 Bytes dargestellt. Zeichenkonstanten werden wie üblich in Hochkomma dargestellt.

```
char c= 'A';
char zeilenende='\n';
char leerzeichen='\u0020'; // Unicode-Escape-Sequenz mit 4 Hexziffern
```

`byte` stellt 1 Byte lange ganze Zahlen im Bereich $[-2^7, 2^7 - 1] = [-128, +127]$ dar.

`short` ist ein Typ für 2 Byte lange Zahlen im Bereich $[-2^{15}, +2^{15} - 1]$.

`int` ist ein Typ für 4 Byte lange Zahlen im Bereich $[-2^{31}, +2^{31} - 1]$.

`long` ist ein Typ für 8 Byte lange Zahlen im Bereich $[-2^{63}, +2^{63} - 1]$.

`float` stellt 4 Byte lange reelle Zahlen dar.
Der Wertebereich ist $[\pm 3.40282347 \cdot 10^{38}]$.

`double` stellt 8 Byte lange reelle Zahlen dar.
Der Wertebereich ist $[\pm 1.79769313486231570 \cdot 10^{308}]$.

Erweiternde Typkonvertierungen von kleineren Zahltypen zu größeren sind implizit möglich: `byte`→ `short`→ `int`→ `long`→ `float`→ `double`
In der umgekehrten Richtung wird die Konvertierung nur durchgeführt, wenn sie explizit durch einen Type-Cast-Operator verlangt wird.

```
int a=15;
float f=a; //implizite Konvertierung von int nach float: f=15.0
f=f/10;    // f=1.5
a= (int)f; // explizite Konvertierung von float nach int: a=1
```

Außer den angegebenen Konvertierungen kann auch implizit von `char` nach `int` konvertiert werden, d.h. Unicode-Zeichen können als Zahlenwerte interpretiert werden (`char`→`int`). Man beachte, dass auch erweiternde Konvertierungen mit einem Informationsverlust verbunden sein können. So sind beispielsweise sehr große Zahlen vom Typ `long` nur näherungsweise durch den Typ `double` oder gar durch `float` darstellbar.

1.3 Ausgabe von Strings

Neben den primitiven Datentypen gibt es sehr viele in Java vordefinierte Klassen. Besonders nützlich ist die Klasse `String`, mit der Zeichenketten dargestellt werden können. Sie soll zunächst nur anhand einfacher Beispiele vorgestellt werden.

```
String s1= "Dies ist ein Beispiel";
String s2= "text";
String s3= s1+s2;
```

Eine in Anführungsstriche eingeschlossene Zeichenkette bewirkt, dass ein Objekt der Klasse `String` erzeugt wird. Im Beispiel werden solche Objekte an die Referenzvariablen `s1` und `s2` zugewiesen. Schließlich wird mit dem Operator `+` ein neuer String erzeugt, der durch das Hintereinanderhängen (Konkatenieren) der Strings `s1` und `s2` entsteht. Die Referenzvariable `s3` entspricht also der Zeichenkette "`Dies ist ein Beispieltext`".
Die Länge eines Strings kann man mit Hilfe der Methode `int length()` ermitteln.

```
int len= s3.length();
```

Die Ausgabe von Strings auf dem Standardausgabegerät kann beispielweise folgendermaßen erreicht werden.

```
System.out.println(s3);
System.out.println("Der Text \"" + s3 + "\" hat die Länge " + len);
```

Die erste Ausgabe erzeugt den Text `Dies ist ein Beispieltext` und beendet die Ausgabe mit einem Zeilenvorschub. Die zweite Ausgabe erzeugt den Text
`Der Text "Dies ist ein Beispieltext" hat die Länge 25`
und beendet ihn ebenfalls mit einem Zeilenvorschub. Interessant ist an der zweiten Ausgabe, dass das Argument von `System.out.println` durch Konkatenation aus mehreren Teilen zusammengesetzt wird und dabei der Ausdruck `len` implizit von `int` nach `String` konvertiert wird.

1.4 Felder

Felder kann man ähnlich wie in C++ durch eckige Klammern definieren und auf ihre Elemente durch Angabe eines Index zugreifen. Bei der Definition ist es jedoch üblich, die Klammern hinter den Typ und nicht hinter den Variablennamen zu schreiben. Wie in C++ werden die Feldelemente bei 0 beginnend indiziert.

```
int[] a;          // Anlegen eines Feldes a noch unbekannter Länge
a= new int[10];   // Erzeugen eines Feldes der Länge 10
                  // und Zuweisung an Variable a
for (int i=0; i<10; ++i)
  a[i]= i*i;

float[] b= new float[100]; // b ist jetzt ein Feld mit 100 Komponenten

b= new float[20];  // Anlegen eines neuen Feldes b
                   // das alte Feld wird automatisch freigegeben
```

Man beachte, dass die Felder immer dynamisch reserviert werden. Die Speicherfreigabe erfolgt automatisch, wenn das Feld nicht mehr über Referenzvariablen zugreifbar ist. Man kann sich dabei vorstellen, dass die Referenzvariablen (intern) durch Zeiger auf die Felder implementiert werden. Insbesondere wird beim Kopieren eines Feldes lediglich ein Zeiger kopiert. Im folgenden Beispiel kann man deshalb mit den Variablen `c` und `d` auf die gleichen Daten zugreifen.

```
int[] c= new int[100];
int[] d= c;
for (int i=0; i<100; ++i)
   c[i]=i;
for (int i=0; i<100; ++i)
   d[i] *= 2;
for (int i=0; i<10; ++i)
   System.out.print(" " + c[i]);  // Liefert 0 2 4 6 8 10 12 14 16 ...
System.out.println();
```

Das Beispiel zeigt, dass die Aussage "In Java gibt es keine Zeiger" streng genommen nicht stimmt. Sowohl Feldzugriffe, als auch Zugriffe auf Objekte werden immer über Zeiger realisiert. Allerdings lassen sich diese Zeiger nicht durch den Programmierer manipulieren.
In einer vereinfachten Notation kann man Felder konstanter Länge auch durch Aufzählung ihrer Elemente erzeugen.

```
int[] x= { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Eine weitere Besonderheit besteht darin, dass jede Referenzvariable eines Feldes eine **Instanzvariable** `length` besitzt, über die man auf die Länge des Feldes zugreifen kann. Auf diese Weise kann man das Feld `x` folgendermaßen ausgeben.

```
for (int i=0; i< x.length; ++i)
   System.out.println(x[i]);
```

Mehrdimensionale Felder werden durch Verschachtelung realisiert. Im folgenden Beispiel besitzt `feld` 4 Komponenten, die ihrerseits wieder Felder unterschiedlicher Länge sind.

```
int[][] feld= {
                {1},       // feld[0] ist ein Feld mit nur einem Element
                {2,3},     // feld[1] hat 2 Einträge
                {4,5,6},   // feld[2] hat 3 Einträge
                {7,8,9,10} // feld[3] hat 4 Einträge
              };
for (int i=0; i<feld.length; ++i)
  {
   for (int j=0; j<feld[i].length; ++j)
      System.out.print(feld[i][j]+" ");
   System.out.println();
  }
```

2 Klassen

2.1 Einführungsbeispiel

Als Einführung in die objektorientierte Programmierung betrachten wir als erstes Beispiel eine Klasse `Counter` zur Realisierung eines Zählers (siehe auch Skript "Programmieren in C++").
Der Zähler besitzt eine interne Variable, die den aktuellen Zählerstand darstellt und von außen nicht direkt zugreifbar ist. Als öffentliche Methoden gibt es einen Konstruktor, der einen auf 0 initialisierten Zähler erzeugt, eine Methode zum Incrementieren, eine zum Zurücksetzen des Zählers und schließlich eine zum Auslesen des aktuellen Zählerstandes.
In Java lässt sich eine solche Klasse folgendermaßen in einer Datei `Counter.java` beschreiben.

```
public class Counter
{
 private int x;                    // Zählerstand

 public Counter() {x=0;}           // Konstruktor
 public void increment() {++x;}    // Hochzählen
 public int get(){return x;}       // Auslesen
 public void reset(){x=0;}         // Zurücksetzen
}
```

Mit Hilfe der **Modifier** `public` und `private` werden die Sichtbarkeiten der deklarierten Größen festgelegt. Die Variable `x` kann nur intern innerhalb der Klasse benutzt werden, die Methoden sind dagegen öffentlich und können von außen aufgerufen werden.
Für dieses Beispiel ist es wichtig, dass die Datei genauso heißt, wie die definierte Klasse. Dabei ist auch die Groß/Kleinschreibung relevant. Zusätzlich erhält der Dateiname lediglich die Endung `.java`. Auf diese Weise ist es für einen Java-Compiler einfach, die Implementierungen aller benötigten Klassen zu finden. Er muss lediglich nach Dateien mit den entsprechenden Namen suchen.
Die Kennzeichnung der Klasse `Counter` mit dem Modifier `public` ist erforderlich, damit die Klasse auch von Programmen in anderen Dateien verwendet werden kann.
Zum Übersetzen der Klasse verwendet man den Compileraufruf `javac Counter.java`
Dabei entsteht als Ergebnis eine Datei `Counter.class` mit der auf beliebigen Rechnern verwendbaren Implementierung der Klasse.
Zur Erprobung benötigen wir nun noch ein Hauptprogramm. Da Java jedoch eine objektorientierte Sprache ist und deshalb kein Programmcode außerhalb von Klassen existiert, müssen wir das Hauptprogramm als Methode einer Klasse definieren.
Beispielsweise geht dies mit folgender Klasse `CounterTest`, in der Datei `CounterTest.java`.

```
public class CounterTest
{
 public static void main(String[] args)
   {
    Counter a= new Counter();           // Erzeugen eines Zählers a
    Counter b= new Counter();           // Erzeugen eines Zählers b
    a.increment();                      // Hochzählen von a
    a.increment();                      // Nochmaliges Hochzählen von a
    System.out.println("a=" + a.get()); // Liefert: a=2
    System.out.println("b=" + b.get()); // Liefert: b=0
   }
}
```

Übersetzt wird diese Klasse mit `javac CounterTest.java`. Dabei erkennt der Compiler die interne Verwendung der Klasse `Counter` und überprüft anhand der Deklarationen in der Datei `Counter.class`, ob die Klasse korrekt verwendet wird. Die Ausführung unseres Beispiels erreichen wir durch den Aufruf `java CounterTest`. Dabei wird der Java-Interpreter für die Datei `CounterTest.class` gestartet. Er sucht in der Klasse `Counter_test` nach einer Methode mit dem Namen `main` und führt diese aus. Dazu muss die Methode `main` genau die im Beispiel verwendete Schnittstelle haben. D.h. als Argument wird ein Feld von Strings erwartet und es wird kein Ergebnis zurückgeliefert. Das Feld von Strings benutzt der Interpreter, um die in der Kommandozeile mitgegebenen Parameter an die Methode zu übergeben. Beispielsweise könnten wir diese Kommandozeilenparameter innerhalb der Methode `main` folgendermaßen zur Kontrolle anzeigen lassen.

```
System.out.print("Kommandozeilenparameter: ");
for (int i=0; i<args.length; ++i)
   System.out.print(args[i]+" ");
System.out.println();
```

Eine weitere Besonderheit der Methode `main` besteht darin, dass sie mit dem Modifier `static` versehen sein muss. Er bewirkt, dass die Methode nicht an ein Objekt gebunden ist, sondern eine **Klassenmethode** darstellt, die aufgerufen werden kann, ohne dass zuvor ein Objekt der Klasse `Counter_test` erzeugt werden muss.

2.2 Vererbung

Ein wichtiges Konzept der objektorientierten Programmierung ist die **Vererbung**, bei der eine neue Klasse Eigenschaften einer bereits vorhandenen Klasse übernimmt. Als Beispiel betrachten wir die Klasse `Counter2`, die einen Zähler realisieren soll, der beliebig initialisiert werden kann und der zusätzlich auch decrementiert werden kann.

```
public class Counter2 extends Counter
{
 public Counter2(int v) {x=v;}   // Konstruktor
 public void decrement()         // Decrementieren
   {if (x>0)
      --x;
   }
}
```

Hier ist beabsichtigt, dass Objekte der **abgeleiteten** Klasse `Counter2` genau die Variablen und Methoden der Klasse `Counter` besitzen und zusätzlich noch die `decrement`-Methode. Allerdings gilt beim Ableiten, dass keine privaten Variablen und Methoden vererbt werden. Deshalb sollte der Modifier `private` der Variablen `x` durch den Modifier `protected` ersetzt werden. Er bewirkt die Sichtbarkeit in der eigenen Klasse und in daraus abgeleiteten Klassen. Schließlich ist noch zu beachten, dass Konstruktoren nicht vererbt werden (siehe auch 2.4).
Zum Testen der neuen Klasse, verwenden wir folgende Beispielanwendung

```
public class CounterTest
{
 public static void main(String[] args)
   {Counter a= new Counter();           // Erzeugen eines Zählers a
    Counter2 b=new Counter2(5);         // Erzeugen eines Zählers b
    a.increment();                      // Hochzählen von a
    a.increment();                      // Nochmaliges Hochzählen von a
    b.increment();                      // Hochzählen von b
    System.out.println("a=" + a.get()); // Liefert: a=2
    System.out.println("b=" + b.get()); // Liefert: b=6
    b.decrement();                      // Runterzählen von b
    System.out.println("b=" + b.get()); // Liefert: b=5
   }
}
```

Man beachte dabei, dass man mit einem Objekt der Klasse `Counter2` alles machen kann, das man auch mit Objekten der Klasse `Counter` machen kann. Insbesondere kann man sie auch an Variablen des Typs `Counter` zuweisen. Um dies zu demonstrieren, erweitern wir die Beispielmethode `main` folgendermaßen.

```
Counter c=b;                 // c verweist auf Objekt b (Klasse Counter2)
c.increment();               // zulässig
c.decrement();               // unzulässig! (decrement() nicht in Counter)
((Counter2) c).decrement();  // zulässig
c=a;                         // c verweist auf Objekt a (Klasse Counter)
((Counter2) c).decrement();  // unzulässiger Typ-Cast! (Laufzeitfehler)
```

Hier nimmt die Referenzvariable `c` vom Typ `Counter` nacheinander die Objekte `b` der Klasse `Counter2` und `a` der Klasse `Counter` an. In jedem Fall ist der Aufruf `c.increment()` zulässig.
Dagegen verweigert der Compiler die Ausführung von `c.decrement()`, weil die Klasse `Counter` keine solche Methode hat. Erst wenn man `c` mit einem Typ-Cast nach `Counter2` konvertiert, wird der Aufruf von `decrement()` vom Compiler zugelassen. Dann wird aber zur Laufzeit des Programms geprüft, ob das vorliegende Objekt tatsächlich eine solche Methode hat.
Dieses Phänomen, dass eine Variable Objekte unterschiedlicher Klassen aufnehmen kann, bezeichnet man als **Polymorphismus**. Dabei ist in Java sichergestellt, dass die korrekte Klassenzugehörigkeit zur Laufzeit festgestellt werden kann. Mit Hilfe des Operators `instanceof` kann der Programmierer auch selber prüfen, ob ein Objekt zu einer bestimmten Klasse gehört.
Dazu wird im folgenden Beispiel ein Feld von Zählern angelegt, die alle einmal incrementiert werden. Anschließend werden diejenigen Zähler wieder decrementiert, die zur Klasse `Counter2` gehören.

```
Counter[] f= new Counter[5];
f[0]= new Counter();
f[1]= new Counter2(7);
f[2]= new Counter();
f[3]= new Counter2(10);
f[4]= new Counter();

for (int i=0; i<f.length; ++i)
   {
    f[i].increment();
    System.out.print(" " + f[i].get());
   }
System.out.println();

for (int i=0; i<f.length; ++i)
   {
    if (f[i] instanceof Counter2)
      ((Counter2) f[i]).decrement();
    System.out.print(" " + f[i].get());
   }
System.out.println();
```

Klassen, die nicht explizit aus einer anderen Klasse abgeleitet werden, sind automatisch aus der Klasse `Object` abgeleitet. Damit lässt sich jede Klasse durch eine Klassenhierarchie letztlich auf `Object` zurückführen. In einer Variablen der Klasse `Object` lassen sich daher beliebige Objekte ablegen.
Insbesondere ist es deshalb auch möglich, in einem Feld Objekte ganz verschiedener Klassen abzulegen.

```
Object[] f= new Object[5];
f[0]= new Integer(8);
f[1]= new Double(3.14);
f[2]= new String("Dies ist ein Beispieltext");
f[3]= new Boolean(true);
f[4]= new Character('A');

for (int i=0; i<f.length; ++i)
   System.out.println(f[i]);
```

Man beachte in diesem Beispiel, dass es sich bei `Integer`, `Double`, ... um Klassen und nicht um primitive Datentypen handelt.

2.3 Überlagern von Methoden

Wenn man in einer abgeleiteten Klasse eine Methode implementiert, die auch in der Vaterklasse existiert, überschreibt die neue Methode die vorhandene.

Als Beispiel implementieren wir sowohl in der Klasse `Counter` als auch in der Klasse `Counter2` eine Methode `String toString()`, die den Zählerstand, einen Namen und eine Typangabe als String zurückliefert.

```
public class Counter
{
 protected int x;                 // Zählerstand
 protected String name;           // Name des Zählers

 public Counter(String s)         // Konstruktor
   {x=0;
    name=s;
   }
 public Counter() {x=0; name="";} // Konstruktor
 public void increment() {++x;}   // Hochzählen
 public int get(){return x;}      // Auslesen
 public void reset(){x=0;}        // Zurücksetzen

 public String toString()
   { return "Counter " + name + "(" + x + ")";
   }
}
```

```
public class Counter2 extends Counter
{
 public Counter2(String s, int v)   // Konstruktor
   {name=s; x=v;}
 public void decrement()            // Decrementieren
   {if (x>0)
      --x;
   }
 public String toString()
   { return "Counter2 " + name + "(" + x + ")";
   }
}
```

Wenn man nun in einem Feld Objekte beider Klassen speichert und für jedes Objekt Ausgaben mit Hilfe der Methode `toString` vornimmt, muss jeweils zur Laufzeit des Programms ermittelt werden, welche Methode zu verwenden ist.

```
public class CounterTest
{
 public static void main(String[] args)
   {
    Counter[] f= new Counter[5];
    f[0]= new Counter( "f0");
    f[1]= new Counter2("f1",7);
    f[2]= new Counter( "f2");
    f[3]= new Counter2("f3",10);
    f[4]= new Counter( "f4");

    for (int i=0; i<f.length; ++i)
      System.out.println(f[i].toString());
   }
}
```

Die hier verwendete explizite Konvertierung mit Hilfe der Methode `toString` ist allerdings nicht unbedingt erforderlich, weil bei Ausgaben mit `print` oder `println` auch implizit mit Hilfe von `toString` konvertiert wird.

2.4 Konstruktoren (und Destruktoren)

Bei der Erzeugung eines Objekt wird automatisch ein geeigneter Konstruktor zur Initialisierung aufgerufen. Natürlich kann es auch mehrere Konstruktoren in einer Klasse geben, die sich dann jedoch in ihren Parameterlisten unterscheiden müssen. In diesem Fall spricht man von **Overloading**. Der Compiler wählt dann anhand der angegebenen aktuellen Parameter einen geeigneten Konstruktor aus.

Bei abgeleiteten Klassen muss man beachten, dass Konstruktoren nicht vererbt werden. Sie müssen also neu definiert werden. Dabei sollte in der ersten Zeile im Rumpf des Konstruktors zunächst ein Konstruktor der Vaterklasse aufgerufen werden. Dies geschieht durch einen Aufruf von `super(...)` mit geeigneten Parametern. Wenn der Programmierer dies vergisst, wird automatisch vom Compiler der parameterlose Konstruktor der Vaterklasse aufgerufen. D.h. es wird implizit der Aufruf `super()` ausgeführt. (Vorsicht: In diesem Fall erhält man eine Fehlermeldung, falls die Vaterklasse keinen parameterlosen Konstruktor besitzt!)
Im Beispiel der Klasse `Counter2` könnte der Konstruktor also beispielsweise folgendermaßen definiert werden.

```
public Counter2(String s, int v)   // Konstruktor
  {super(s);
   x=v;
  }
```

Dabei wird der Name des Objekts mithilfe des Konstruktors der Klasse `Counter` eingetragen und anschließend wird der auf 0 initialisierte Zählerstand mit `v` überschrieben.
Man beachte, dass im allgemeinen das Erzeugen eines Objekts zum Aufruf einer ganzen Kette von Konstruktoren führt. Im vorliegenden Beispiel sind dies nacheinander Konstruktoren der Klassen `Object`, `Counter` und `Counter2`.
Sollen Objekte der Klasse `Counter2` auch ohne Angabe von Parametern erzeugt werden, muss man explizit einen parameterlosen Konstruktor in der Klasse definieren.

```
public Counter2()   // Konstruktor
  {super();         // Defaultaufruf
  }
```

Hinweis: Für den Sonderfall, dass in einer Klasse überhaupt kein Konstruktor definiert ist, wird vom Compiler standardmäßig ein trivialer parameterloser Konstruktor ergänzt.
So wie beim Erzeugen von Objekten automatisch Konstruktoren zur Initialisierung aufgerufen werden, kann das Löschen von Objekten zum Aufruf von **Destruktoren** führen. Sie werden in Java immer als parameterlose geschützte Methoden mit dem Namen `finalize` definiert:

```
protected void finalize()
 {// Rumpf der vor der Speicherfreigabe ausgeführt werden soll
 }
```

Da die Speicherfreigabe in Java automatisch durch einen Garbage-Collector erfolgt und nicht explizit durch den Programmierer vorgenommen werden muss, spielen die Destruktoren in der Praxis keine große Rolle und sollten vermieden werden. Tatsächlich ist nicht definiert, zu welchem Zeitpunkt die Speicherfreigabe erfolgt. Im Extremfall dass das Programm bereits beendet ist, bevor es zum Aufruf des Garbage-Collectors kommt, werden die Destruktoren überhaupt nicht aufgerufen.

2.5 Statische Methoden und Klassenvariablen

Werden Variablen oder Methoden einer Klasse mit dem Modifier `static` versehen, bedeutet dies, dass die betreffenden Größen für diese Klasse genau einmal existieren und nicht an die Existenz eines Objekts gebunden sind.
Auf diese Weise kann man beispielsweise für die Klasse `Counter` einen globalen Zähler einführen, der angibt, wieviele Objekte dieser Klasse bereits erzeugt wurden. Solche statischen Variablen werden auch als **Klassenvariablen** bezeichnet.

```
public class Counter
{
 private static int global=0;     // Anzahl bisher erzeugter Zähler
 protected int x;                 // Zählerstand
 protected String name;           // Name des Zählers
 protected int id;                // Nummer des Zählers

 public Counter(String s)         // Konstruktor
   {x=0;
    name=s;
    ++global;
    id=global;
   }
 public Counter()                // Konstruktor
   {x=0;
    ++global;
    id=global;
   }
 public void increment() {++x;}   // Hochzählen
 public int get(){return x;}      // Auslesen
 public void reset(){x=0;}        // Zurücksetzen

 public String toString()
   { return "Counter_" + id + " " + name + "(" + x + ")";
   }
}
```

Klassenvariablen können bei ihrer Deklaration bereits mit einer Konstanten vom Compiler initialisiert werden. Für den Fall, dass die Initialisierung aufwändiger ist, kann man auch eine namenlose statische Methode zur Initialisierung verwenden:

```
static
{global=0; // oder aufwändiger Berechnungsalgorithmus
}
```

In unserem Beispiel wird die Klassenvariable `global` beim Erzeugen der Klasse auf 0 initialisiert und bei jedem Aufruf eines Konstruktors wird sie incrementiert.

Schließlich wird sie innerhalb der Methode `toString` benutzt, um den jeweils vorliegenden Zähler eindeutig zu identifizieren.
Wäre die Klassenvariable als `protected` statt als `private` definiert worden, könnte sie auch innerhalb von abgeleiteten Klassen zugegriffen werden. Allerdings sollte man dann darauf achten, dass sie dort nicht in den Konstruktoren incrementiert wird, da dies bereits in den Konstruktoren der Vaterklasse geschieht.
Definiert man eine Methode als statisch, dann ist sie nicht an die Existenz eines Objekts gebunden und kann auch ohne Objekt aufgerufen werden. Als erstes Beispiel hatten wir bereits die Methode `void main(String[] args)` kennengelernt, die vom Java-Interpreter aufrufbar war.
Im allgemeinen muss man beim Aufruf einer statischen Methode ihren Klassennamen voranstellen, um die Methode eindeutig zu identifizieren. Als Beispiel betrachten wir im folgenden eine Klasse `Primzahl`, mit einer statischen Methode `boolean isprime(int p)`, die prüfen soll, ob ein gegebenes Argument `p` eine Primzahl ist.

```
public class Primzahl
{
 public static boolean isprime(int p)
   {
    if (p==2)
      return true;
    if (p<2)
      return false;
    int m=(int)Math.round(Math.sqrt(p));
    for (int i=2; i<=m; ++i)
     if ((p%i)==0)
       return false;
    return true;
   }
}
```

Zum Testen dieser Klasse benutzen wir eine Klasse `Primzahltest` mit einem Hauptprogramm, das die Primzahlen im Intervall $[1, 100]$ ausgeben soll.

```
public class Primzahltest
{
 public static void main(String[] args)
  {
   for (int x=1; x<100; ++x)
     if (Primzahl.isprime(x))
       System.out.println(x);
  }
}
```

Man beachte, dass im Hauptprogramm der Methodenaufruf in der Notation `Primzahl.isprime(x)`, also mit vorangestelltem Klassennamen erfolgt.

Innerhalb der Methode `isprime` wurden übrigens die statischen Methoden `sqrt` und `round` aus der Klasse `Math` auf die gleiche Weise benutzt. Auch die Ausgabe auf dem Standardausgabegerät wird nun verständlicher. Dazu benutzen wir die statischen Methoden `println` und `print` der Instanzvariablen `out` in der Klasse `System`.

2.6 Abstrakte Klassen

Im folgenden soll als Beispielanwendung eine Klasse programmiert werden, mit deren Hilfe sich Funktionstabellen in einem vorgegebenen Intervall mit einer vorgegebenen Schrittweite erstellen lassen. Dabei sollte auch die gewünschte Funktion f als Parameter übergeben werden können

Bei einer entsprechenden Implementierung in C++ könnte man einen Zeiger auf die Funktion f übergeben. In Java erreicht man das gleiche dadurch, dass man die Funktion in eine Klasse verpackt und dann ein Objekt der Klasse übergibt. Als Vorbereitung für diese Vorgehensweise müssen wir uns mit **abstrakten Klassen** beschäftigen.

Wenn man in einer Klasse eine Methode deklariert, ohne sie zu implementieren, spricht man von einer abstrakten Methode. Formal gibt man dazu den Kopf der Methode an und schließt ihn mit einem Semikolon ab, anstatt den Rumpf zu programmieren. Zusätzlich verwendet man den Modifier `abstract`.

Eine Klasse, die mindestens eine abstrakte Methode besitzt, wird als **abstrakte Klasse** bezeichnet und muss mit dem Modifier `abstract` gekennzeichnet werden.

Im folgenden wird dies anhand einer Klasse `Funktion` dargestellt, die eine abstrakte Methode `double f(double)` besitzt.

```
abstract public class Funktion
{
 private String name;
 abstract public double f(double x);
 public Funktion(String s) {name=s;}
 public String toString() {return name;}
}
```

Eine solche Klasse kann nicht instantiiert werden, d.h. es können keine Objekte der Klasse erzeugt werden. Sinnvoll ist die Klasse dagegen zum Ableiten anderer Klassen. Dabei bleiben die Klassen solange abstrakt, bis sämtliche abstrakte Methoden durch Implementierungen überschrieben werden.

Beispielsweise können wir zwei Klassen `Quadratwurzel` und `Sinus` aus der abstrakten Klasse ableiten, die beide eine Methode `double f(double)` besitzen müssen.

```
public class Quadratwurzel extends Funktion
{
 public Quadratwurzel() {super("Quadratwurzel");}
 public double f(double x) {return Math.sqrt(x);}
}


public class Sinus extends Funktion
{
 public Sinus() {super("Sinus");}
 public double f(double x) {return Math.sin(x);}
}
```

Die Klasse zur Erzeugung von Funktionstabellen lässt sich nach diesen Vorbereitungen nun folgendermaßen programmieren.

```
public class Funktionstabelle
{
 public static void tabelle(Funktion fobj,
                            double a, double b, double delta)
   {
    double x=a;
    System.out.println("Funktionstabelle: " + fobj.toString());
    System.out.println("--------------------------------------");
    while (x<=b)
      {
       double y= fobj.f(x);
       System.out.println(x + "   " + y);
       x += delta;
      }
    System.out.println();
   }
}
```

Als Parameter `fobj` der Methode `tabelle` kann irgendein Objekt einer aus `Funktion` abgeleiteten Klasse übergeben werden. Dabei ist sichergestellt, dass das Objekt eine Methode `toString` besitzt, über die man den Funktionsnamen erfahren kann und eine Methode `f` mit der man die benötigten Funktionswerte berechnen kann.

Die folgenden Beispielanwendung demonstriert, wie die Methode `tabelle` aufgerufen werden kann.

```
public class Funktionstabellentest
{
 public static void main(String[] args)
  {
   Funktion f1=new Sinus();
   Funktion f2=new Quadratwurzel();
   Funktionstabelle.tabelle(f1, 0, 3.14, 0.2);
   Funktionstabelle.tabelle(f2, 0, 1, 0.1);
   Funktionstabelle.tabelle(f2, 1, 10, 1);
  }
}
```

3 Interfaces

Im Beispiel der Methode `tabelle` der Klasse `Funktionstabelle` wurde ausgenutzt, dass alle übergebenen Objekte zu Klassen gehören, die aus `Funktion` abgeleitet wurden und deshalb sehr ähnliche Eigenschaften haben.

Noch flexibler wäre ein Mechanismus, bei dem man Objekte beliebiger nicht miteinander verwandter Klassen übergeben könnte, solange nur die benötigten Eigenschaften der Objekte vorliegen. Genau zu diesem Zweck werden **Interfaces** eingeführt.

Ein **Interface** ist im Prinzip eine abstrakte Klasse, bei der sämtliche Methoden abstrakt sind. Als Schlüsselwort wird jedoch `interface` statt `class` verwendet. (Der Modifier **abstract** ist im `interface` nicht erforderlich.)

Die im Beispiel der Funktionststabellen benötigten Eigenschaften sind die Existenz der Methoden `toString` und `f`. Deshalb definiert man folgendes Interface.

```
public interface Funktionsbeschreibung
{
 public double f(double x);
 public String toString();
}
```

Wenn man nun bei einer Klassendefinition mit Hilfe der Klausel `implements` angibt, dass die Klasse dieses Interface implementiert, stellt der Compiler sicher, dass die im Interface deklarierten abstrakten Methoden tatsächlich in der Klasse implementiert werden. Beispielsweise können wir jetzt die Klassen `Quadratwurzel` und `Sinus` mithilfe des Interfaces definieren, ohne sie aus einer gemeinsamen Vaterklasse abzuleiten.

```
public class Quadratwurzel implements Funktionsbeschreibung
{
 public double f(double x) {return Math.sqrt(x);}
 public String toString()  {return "Quadratwurzel";}
}


public class Sinus implements Funktionsbeschreibung
{
 public double f(double x) {return Math.sin(x);}
 public String toString()  {return "Sinus";}
}
```

Auf diese Weise haben die beiden Klassen gewissermaßen die Eigenschaften des Interface geerbt. Ein weiterer interessanter Aspekt dieses Mechanismus ist, dass eine Klasse auch mehrere verschiedene Interfaces implementieren kann, also Eigenschaften mehrerer Interfaces erben kann. Mit dem Vererbungsmechanismus durch Ableitung ist dies nicht realisierbar, da in Java keine Mehrfachvererbung zulässig ist.
Die Klasse zur Erzeugung von Funktionstabellen lässt sich nach diesen Vorbereitungen nun folgendermaßen programmieren.

```
public class Funktionstabelle
{
 public static void tabelle(Funktionsbeschreibung fobj,
                            double a, double b, double delta)
   {
    double x=a;
    System.out.println("Funktionstabelle: " + fobj.toString());
    System.out.println("--------------------------------------");
    while (x<=b)
      {
       double y= fobj.f(x);
       System.out.println(x + "   " + y);
       x += delta;
      }
    System.out.println();
   }
}
```

Hier kann für den Parameter `fobj` jedes Objekt übergeben werden, dessen Klasse das Interface `Funktionsbeschreibung` implementiert.
Die folgenden Beispielanwendung demonstriert, wie die Methode `tabelle` aufgerufen werden kann.

```
public class Funktionstabellentest
{
 public static void main(String[] args)
  {
   Sinus f1=new Sinus();
   Quadratwurzel f2=new Quadratwurzel();
   Funktionstabelle.tabelle(f1, 0, 3.14, 0.2);
   Funktionstabelle.tabelle(f2, 0, 1, 0.1);
   Funktionstabelle.tabelle(f2, 1, 10, 1);
  }
}
```

Anmerkung: Die im obigen Beispiel verwendete Methode `String toString()` gibt es auch in der Klasse `Object`. Durch Aufruf dieser Methode kann man für jedes Objekt einen eindeutigen Bezeichner erhalten. Er setzt sich aus dem Klassennamen und einer Hexzahl zusammen. Da jede Klasse auch die Methoden von `Object` erbt, wäre ein Überschreiben dieser Methode gar nicht notwendig gewesen. D.h. der Compiler liefert keine Fehlermeldung, wenn man die Definition der abstrakten Methode `String toString()` vergisst und es wird stattdessen die geerbte Methode von `Object` ausgeführt.

3.1 Sortieren beliebiger Daten

Als ein weiteres Beispiel für Interfaces soll das besonders einfache Sortierverfahren "Sortieren durch Auswahl" so implementiert werden, dass es für beliebige Daten verwendbar ist. Dabei kommt es eigentlich nur darauf an, dass Paare von Daten miteinander verglichen werden können. Deshalb setzen wir voraus, dass es für die Daten eine Methode `int compareTo(Object o)` gibt, mit der ein gegebenes Objekt mit einem anderen Objekt `o` verglichen werden kann. Als Ergebnis soll man eine negative Zahl erhalten, wenn das gegebene Objekt kleiner als `o` ist, einen positiven Wert erhält man, für "größer" und im Fall der Gleichheit soll das Ergebnis 0 sein.

```
public interface Comparable
{
 public int compareTo(Object o);
}
```

Tatsächlich ist dieses Interface bereits in Java definiert und sehr viele Klassen wie etwa `Integer`, `Double`, `String`, ... implementieren dieses Interface.

Das Sortierverfahren kann dann beispielsweise folgendermaßen für beliebige Vektoren mit vergleichbaren Daten implementiert werden.

```
public class Sortierung
{public static void sort(Comparable[] x)
 { // Sortiert den Vektor x
   for (int i=0; i<x.length-1; ++i)
     for (int j=i+1; j<x.length; ++j)
       {// Vergleich von x[i] mit x[j]
        if (x[i].compareTo(x[j])>0)
          {// Austausch der beiden Elemente
           Comparable tmp=x[i];
           x[i]=x[j];
           x[j]=tmp;
          }
       }
 }
}
```

In der folgenden Beispielanwendung werden zwei Vektoren `a` und `b` definiert und dann jeweils mit dem obigen Sortierverfahren sortiert. Damit wird demonstriert, dass die Implementierung tatsächlich für unterschiedliche Datentypen verwendbar ist.

```
public class Sortierungsbeispiele
{ public static void main(String[] args)
   {Integer[] a= {20, 17, 100, 5, 23, 150, 12, 140};
    String[] b= {"Anton", "Egon", "Bernd", "Cäsar", "Cleopatra",
      "Hugo", "Werner", "Klaus", "Peter", "Simone",
      "Ursula", "Sabine", "Erika"};
    Sortierung.sort(a);
    System.out.print("a: ");
    for (int i=0; i< a.length; ++i)
      System.out.print(a[i] + " ");
    System.out.println();
    Sortierung.sort(b);
    System.out.print("b: ");
    for (int i=0; i< b.length; ++i)
      System.out.print(b[i] + " ");
    System.out.println();
   }
}
```

Nach dem bisher gelernten hätte man eigentlich bei der Definition des Feldes `a` die Zahlen in der Form `new Integer(20)` in ein Objekt der Klasse `Integer` konvertieren müssen. Seit `J2SE 5.0` wird diese Konvertierung aber auch automatisch durch den Compiler vorgenommen. Man bezeichnet dies als **Autoboxing**, d.h. für

Daten der Standarddatentypen können automatisch entsprechend initialisierte Objekte der **Wrapperklassen** erzeugt werden. Werden umgekehrt Standarddatentypen statt der Wrapperklassen benötigt, wird die umgekehrte Konvertierung durch **Autounboxing** vorgenommen.

4 Fehlerbehandlung

Bei der Ausführung mancher Anweisungen oder Methoden kann es zu Fehlern kommen. Beispiele sind etwa eine Division durch 0, das Berechnen einer Quadratwurzel aus einer negativen Zahl, ein Lesezugriff auf eine nicht vorhandene Datei, eine Indexüberschreitung bei einem Feldzugriff oder ein Overflow bei einer arithmetischen Operation.
In diesem Kapitel wird behandelt, wie ein Java-Programm auf solche Fehler, bzw. **Ausnahmen** reagiert. Da es in der Praxis nicht akzeptabel ist, dass ein Programm unkontrolliert abstürzt, sollte der Programmierer selber festlegen können, wie ein Fehler zu behandeln ist. Beispielsweise könnte eine Fehlermeldung ausgegeben werden, oder eventuell kann der Fehler auch programmintern korrigiert werden.
Prinzipiell könnte der Programmierer jeweils direkt an der den Fehler verursachenden Anweisung die erforderliche Fehlerbehandlung beschreiben. Dies würde jedoch in der Praxis zu einem schwer lesbaren Code führen, weil der eigentliche Programmcode von der Fehlerbehandlung unterbrochen würde. Besser ist es, die Fehlerbehandlung separat an einer andereren Stelle zu programmieren. Dazu wird im folgenden eine spezielle Terminologie eingeführt.
Ein Fehler wird in Java als **Ausnahme** bzw. **exception** bezeichnet. Wenn zur Laufzeit eine Ausnahme auftritt, so sagt man, dass eine **exception** "geworfen" (engl.: **throwing**) oder "ausgelöst" wird. Die Behandlung der Ausnahme wird als **catching** bezeichnet.
Zur Einführung der speziellen Notation betrachten wir zunächst ein einfaches Beispiel ohne Fehlerbehandlung.

```
public class Beispiel1
  {public static void main(String[] args)
     {int[] feld= new int[20];
      for (int i=0; i<=22; ++i)
         feld[i]=(i*i)/(5-i);

      for (int i=0; i<20; ++i)
         {System.out.print("i="+i+"   ");
          System.out.println(feld[i]);
         }
      System.out.println("Ende des Hauptprogramms");
     }
  }
```

Dieses Beispiel enthält mehrere Fehler. Beispielsweise findet in der ersten `for`-Schleife für $i = 5$ eine Division durch 0 statt, die eine `ArithmeticException` auslöst, so dass das Programm sofort abgebrochen wird. Außer der Fehlermeldung "/ by zero" erhält man keine weiteren Ausgaben.

Im folgenden Beispiel befindet sich die kritische Anweisung in einem `try`-Block, so dass die darin geworfenen Exceptions abgefangen werden können.

```
public class Beispiel2
{
 public static void main(String[] args)
   {
    int[] feld= new int[20];
    for (int i=0; i<=22; ++i)
     try {
          feld[i]=(i*i)/(5-i);
         }
          catch (ArrayIndexOutOfBoundsException e)
            {
             System.out.println("Es ist ein Indexfehler aufgetreten: "
                                + e.getMessage());
            }
          catch (ArithmeticException e)
            {
             System.out.println("Es ist ein arithmetischer Fehler aufgetreten: "
                                + e.getMessage());
            }
    for (int i=0; i<20; ++i)
      {System.out.print("i="+i+"   ");
       System.out.println(feld[i]);
      }
    System.out.println("Ende des Hauptprogramms");
   }
}
```

Wird hier eine Exception geworfen, so werden nacheinander die hinter dem `try`-Block stehenden `catch`-Blöcke abgearbeitet, bis die Exception abgearbeitet ist. Für den Fall der `ArithmeticException` bedeutet dies, dass die zweite `catch`-Anweisung die Ausnahme über das Objekt `e` verfügbar macht und die Fehlermeldung
"`Es ist ein arithmetischer Fehler aufgetreten: / by zero`"
ausgegeben wird. Danach wird die Programmausführung fortgesetzt, wobei für $i \geq 20$ unzulässige Feldzugriffe auftreten, die dann zu den Fehlermeldungen
"`Es ist ein Indexfehler aufgetreten: 20`"
"`Es ist ein Indexfehler aufgetreten: 21`"
"`Es ist ein Indexfehler aufgetreten: 22`"

führen. Anschließend erhält man mit der zweiten (korrekten) `for`-Schleife die Ausgabe der Feldelemente.

Das Beispiel zeigt, dass Laufzeitfehler innerhalb eines `try`-Blocks nicht mehr zum Abbruch des Programms führen, wenn ein entsprechender `catch`-Block ausgeführt werden kann. Anschließend wird das Programm (hinter den `catch`-Blöcken) weiter ausgeführt.

Neben den bereits durch die Sprache Java vorgegebenen Ausnahme-Objekten kann man auch eigene Ausnahme-Objekte definieren und durch einen `throw`-Befehl auslösen. So wird im folgenden Beispiel ein neues Objekt der Klasse `ArithmeticException` erzeugt, das die Meldung "`Wurzel aus negativer Zahl!`" verursacht. Man beachte, dass in diesem Beispiel die komplette `for`-Schleife im `try`-Block enthalten ist, so dass beim ersten auftretenden Fehler die komplette Schleife beendet wird. Danach erhält man noch die Meldung "`Ende des Hauptprogramms`" bevor das Programm normal beendet wird.

```
public class Beispiel3
{
 public static void main(String[] args)
   {try{
        for (int x=10; x > -3; --x)
          {double y=0;
           if (x<0)
             throw new ArithmeticException("Wurzel aus negativer Zahl!");
           else
             y=Math.sqrt((double)x);
           System.out.print("x=" +x);
           System.out.println("  y="+y);
          }
       } catch (ArithmeticException e)
           {System.out.println("Folgender Fehler ist aufgetreten: "
                               + e.getMessage());
            System.out.println("Deshalb wird die Verarbeitung abgebrochen");
           }
    System.out.println("Ende des Hauptprogramms");
   }
  }
```

Ohne die `try`- und `catch`-Blöcke wäre das Programm aufgrund der geworfenen Exception fehlerhaft abgebrochen worden.

Wenn man in einer Methode eine Exception nicht abfangen möchte, sollte die Methode mit Hilfe einer `throws`-Klausel bekannt geben, dass sie eine Exception auslösen kann, die nicht abgefangen wird. In diesem Fall wird die Exception an die Aufrufstelle weitergegeben und es wird erwartet, dass sie dort abgefangen oder ebenfalls weitergegeben wird. Ein Beispiel dafür ist die folgende Klasse `MyVector`, bei der

die Methoden `get` und `put` eventuell ungültige Indexzugriffe durchführen. Sie werden in der Methode `main` abgefangen und erzeugen keine Fehlermeldung. Erst am Programmende wird protokolliert, wieviele Fehler auftraten.

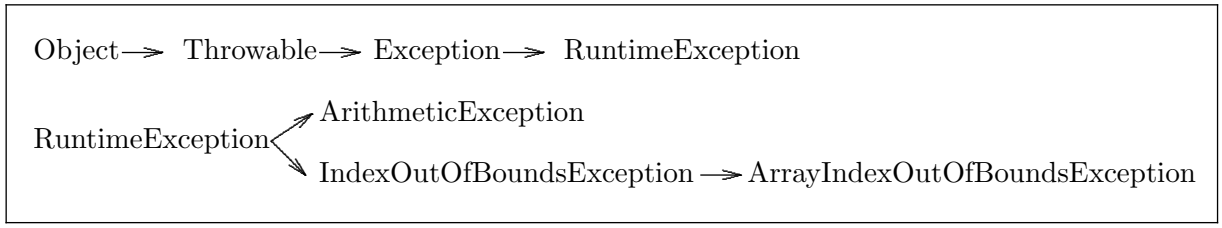
```
public class MyVector
{
 private double[] data;
 public MyVector(int n) { data= new double[n]; }   //Konstruktor

 public double get(int i)
 throws ArrayIndexOutOfBoundsException
   {return data[i];
   }

 public void put(int i, double x)
 throws ArrayIndexOutOfBoundsException
   {data[i]=x;
   }

public static void main(String[] args)
 {
  int counter=0;
  MyVector feld=new MyVector(5);
  for (int i=-2; i<7; ++i)
     try {
          feld.put(i,i*i);
          System.out.println("i=" + i + "  feld[i]=" + feld.get(i));
         } catch (ArrayIndexOutOfBoundsException e)
                {++counter;}
  System.out.println(counter + " Feldkomponenten fehlen");
 }
}
```

Die in den Beispielen verwendeten Exception-Klassen stammen alle aus einer umfangreichen Klassen-Hierarchie. Die folgende Skizze zeigt einen kleinen Auschnitt.



5 Abstrakte Datentypen

In der Praxis kommt es ständig vor, dass man große Datenmengen verarbeiten muss. Um die Daten im Rechner darzustellen, haben wir bisher nur **Felder** (**Arrays**) kennengelernt. Sie erlauben einen schnellen Zugriff auf einzelne Daten über einen Index. Will man jedoch mitten im Feld einen neuen Eintrag einschieben, muss man alle nachfolgenden Einträge entsprechend verschieben. Als Alternative könnte man eine **verkette Liste** von Elementen benutzen, bei der das Einfügen ohne Verschiebung möglich ist, aber dann können die Elemente nicht mehr über einen fortlaufenden Index zugegriffen werden. Andere Implementierungstechniken sind **Baumstrukturen** oder **Hashtabellen**.

Um zu entscheiden, welche Datenstruktur für die jeweilige Anwendung am besten geeignet ist, muss man zunächst betrachten, welche Art von Datenzugriffen (mit welcher Zugriffszeit) benötigt wird. Deshalb hat man in der Informatik den Begriff eines **abstrakten Datentyps** eingeführt. Er beschreibt nicht, wie eine Datenstruktur implementiert ist, sondern welche Zugriffe verfügbar sind. Als extrem einfaches Beispiel betrachten wir im folgenden einen Stack.

5.1 Stack

Einen **Stack** (oder Stapel) benutzt man, um Daten zu speichern die dann in umgekehrter Reihenfolge wieder gelesen werden sollen (**LIFO**-Prinzip: "last in - first out").

Zum Schreiben von Daten benötigt man eine Methode `void push(Object o)` und zum Lesen `Object pop()`. Außerdem sollte man prüfen können, ob noch Elemente im Stack vorhanden sind. Dazu definieren wir eine Methode `boolean is_empty()`.

Die Wirkung eines Stacks kann durch einfache Regeln definiert werden. Beispielsweise gilt für einen beliebig gefüllten Stack `s`, dass man mit einem `pop`-Aufruf, das zuletzt mit `push` geschriebene Element erhält. D.h. das folgende Programmstück liefert $y = x$.

```
s.push(x); y=s.pop()
```

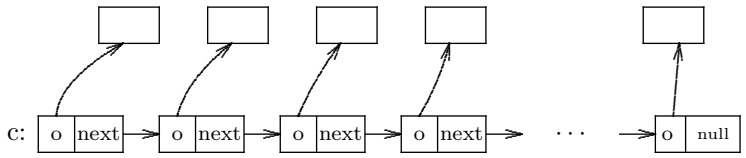
Außerdem müssen die beiden Operationen `push` und `pop` invers zueinander sein, so dass der Stack nach den oben angegebenen Befehlen die gleichen Eigenschaften hat, wie davor. Insbesondere bleibt die "Reihenfolge" der bereits gespeicherten Elemente bei weiteren `push`- und `pop`-Operationen unverändert erhalten.

Man beachte, dass bei einem abstrakten Datentyp nicht festgelegt ist, wie er implementiert werden soll, sondern nur, wie er verwendet wird. Damit liegt eine Ähnlichkeit zu einem Interface vor.

Eine Beispielanwendung könnte folgendermaßen aussehen.

```
public class StackAnwendung
{
 public static void main(String[] args)
   {Stack s= new Stack();
    s.push("Element_1");
    s.push("Element_2");
    s.push("Element_3");
    s.push(new Integer(125));
    while (!s.isEmpty())
       System.out.println(s.pop());
   }
}
```

Für eine einfache Implementierung des Stacks kann eine verkettete Liste benutzt werden. Dazu dient die folgende Datei `Stack.java`.




```
public class Stack
{private Chain c;              // Verweis auf erstes Kettenglied
 public Stack() {c=null};     // Erzeugen eines leeren Stacks
 public void push(Object o)
   {Chain temp= new Chain(); // Erzeugen eines neuen Kettenglieds
    temp.o=o;                // Zuweisen des neuen Stackelements
    temp.next=c;             // Verweis auf voriges Kettenglied
    c=temp;                  // Neues Kettenglied eintragen
   }
 public Object pop()
   {Object o= c.o;
    c=c.next;
    return o;
   }
 public boolean isEmpty() {return (c==null);}

 class Chain
 {private Object o;
  private Chain next;
  Chain() {o=null; next=null;}
 }
}
```

Als Besonderheit beachte man, dass innerhalb der Klasse `Stack` lokal die **innere Klasse** (**inner class**) `Chain` für Kettenglieder definiert wird. Diese Klasse kann nur innerhalb von `Stack` verwendet werden. Dies ist sinnvoll, weil die Kettenglieder nur intern im Stack benötigt werden.
Bei einer **inner class** kann die umfassende **outer class** grundsätzlich auf die Membervariablen der `inner class` zugreifen. Umgekehrt geht das auch.

6 Typisierung von Klassen

Bei der obigen Implementierung eines Stacks konnten beliebige Objekte auf dem Stack abgelegt werden. So wurden in der Beispielanwendung drei Strings und ein Integer auf dem gleichen Stack gespeichert. Oftmals ist dies ein Vorteil, da der Stack auf diese Weise sehr vielseitig verwendet werden kann. Andererseits kann man dann in der Anwendung nicht sicher sein, beim `pop`-Aufruf tatsächlich Daten des korrekten Typs zu erhalten, so dass man eventuell zur Laufzeit zusätzliche Typprüfungen (mit `instanceof`) vornehmen muss.
Deshalb wünscht man oft, dass auf einem Stack nur Daten eines bestimmten Typs abgelegt werden dürfen und der Compiler bereits die Typprüfung vornimmt. Zu diesem Zweck gibt es (seit *J2SE 5.0*) die Möglichkeit **generischer** (**typsicherer**) Klassen. Dazu gibt man im vorliegenden Beispiel bei der Definition der Stack-Klasse in spitzen Klammern einen Bezeichner an, der dann in der Klasse als Typ verwendet werden kann. In der folgenden Implementierung wird dieser Typ als `E` bezeichnet und steht für den Datentyp der Stackelemente, ersetzt also den bisher verwendeten Typ `Object`.

```
public class Stack<E>
 {
  private Chain c;
  public Stack() {c=null;};    // Erzeugen eines leeren Stacks

  public void push(E o)
    {Chain temp= new Chain(); // Erzeugen eines neuen Kettenglieds
     temp.o=o;
     temp.next=c;
     c=temp;
    }

  public E pop()
    {E o= c.o;
     c=c.next;
     return o;
    }
  public boolean isEmpty() {return (c==null);}
```

```
   class Chain
    {
     private E o;
     private Chain next;
     Chain()
      {o=null;
       next=null;
      }
    }
 }
```

In der Beispielanwendung muss dann bei jeder Verwendung des Klassennamens `Stack` in spitzen Klammern der aktuell zu verwendende Datentyp angegeben werden. So kann man beispielsweise in einer Instanz der Klasse `Stack<String>` nur Objekte vom Typ `String` speichern. Für einen anderen Stack mit `Integer`-Elementen würde man entsprechend die Klasse `Stack<Integer>` verwenden.

```
public class StackAnwendung
  {
   public static void main(String[] args)
     {
      Stack<String> s= new Stack<String>();
      s.push("Element_1");
      s.push("Element_2");
      s.push("Element_3");
//    s.push(new Integer(125));   // wäre syntaktisch falsch!

      while (!s.isEmpty())
       System.out.println(s.pop());
      Stack<Integer> si= new Stack<Integer>();
      si.push(1);
      si.push(2);
      si.push(3);
//    si.push("4");               // wäre syntaktisch falsch!

      while (!si.isEmpty())
       System.out.println(si.pop());
     }
  }
```

Anmerkung: Man beachte, dass hier in den Anweisungen `si.push(1)` ... `si.push(3)` die `integer`-Größen jeweils durch **Autoboxing** in Objekte der Klasse `Integer` konvertiert werden.

6.1 Typsichere Verwendung von `compareTo`

Wenn in bisherigen Anwendungen das Interface `compareTo` verwendet wurde, lieferte der Compiler trotz funktionsfähiger Programme oftmals die Warnung

> The method `compareTo(Object)` belongs to the raw type `Comparable`. References to generic type `Comparable<T>` should be parameterized.

Beispielsweise war dies bei der Sortierung von Daten im Abschnitt 3.1 der Fall. Dort wurden in einer Methode `void sort(Comparable[] x)` die Komponenten eines Feldes `x` sortiert. Dazu wurden die Feldkomponenten paarweise mit dem Ausdruck `x[i].compareTo(x[j])` miteinander verglichen.
Der Grund für die dabei vom Compiler gemeldete Warnung ist, dass die Elemente des Feldes `x` zu verschiedenen Klassen gehören können. Beispielsweise könnte das Feld sowohl Integer-Werte, als auch Strings enthalten. Tatsächlich ist im Interface `Comparable` für den Parameter der Methode `compareTo` nur der allgemeine Typ `Object` angegeben.
Zur Erreichung der Typsicherheit schlägt der Compiler deshalb vor, statt dem Interface `Comparable` das folgende typisierte Interface `Comparable<T>` zu verwenden:

```
public interface Comparable<T>
{
 public int compareTo(T o);
}
```

Hier wird festgelegt, dass der übergebene Parameter `o` vom gleichen Typ wie das aktuelle Objekt sein muss. Als Anwendungsbeispiel betrachten wir folgendes Programmstück:

```
Integer a=5;
Integer b=7;
String s="Hugo";
int c= a.compareTo(b);   // ist korrekt
int d= a.compareTo(s);   // Fehlermeldung: Typfehler bei s
```

Da der Compiler für das Object `a` den Typ `Integer` kennt, verwendet er beim Aufruf der `compare`-Methode automatisch das Interface `Comparable<Integer>`, so dass der Vergleich mit `b` syntaktisch korrekt ist, aber der Vergleich mit `s` syntaktisch falsch ist.
Beim Sortierbeispiel haben wir nun das Problem, dass wir einen uns noch unbekannten Typ `T` benutzen müssen, von dem nur klar ist, dass er das Interface `Comparable<T>` implementiert.
Die entsprechende Formulierung in Java sieht dann folgendermaßen aus:

```
public class Sortierung <T extends Comparable<T>>
{public void sort(T[] x)
 {for (int i=0; i<x.length-1; ++i)
    for (int j=i+1; j<x.length; ++j)
      {if (x[i].compareTo(x[j])>0)
         {T tmp=x[i];
          x[i]=x[j];
          x[j]=tmp;
         }
      }
 }
}
```

Diese Klasse wird tatsächlich ohne Warnungen übersetzt. Ein kleiner "Schönheitsfehler" besteht allerdings darin, dass es nicht möglich ist, die typisierte Klasse statisch zu machen. Für den Aufruf der Methode `sort` muss also zunächst ein Objekt der Klasse `Sortierung<T extends Comparable<T>>` erzeugt werden.

```
Integer[] a= {20, 17, 100, 5, 23, 150,12, 140};
new Sortierung<Integer>().sort(a);
for (int i=0; i<a.length; ++i)
  System.out.print(" "+a[i]);
System.out.println();
```

7 Collections

7.1 Das Paket `java.util`

Das Paket `java.util` enthält die Klassen des sogenannten **Collection Frameworks**. Im wesentlichen beinhaltet das Paket die Interfaces

`List` für beliebig große Listen, deren Elemente auch über einen Index zugegriffen werden können,

`Set` zur Darstellung von Mengen und

`Map` für Paare von Daten verschiedenen Typs.

und viele konkrete Implementierungen dieser Schnittstellen. Beispielsweise gibt es sortierte Listen, die über eine Baumstruktur implementiert sind und hinter einer Map kann sich beispielsweise ein Hashverfahren verbergen.

Da diesen Implementierungen sehr effiziente Algorithmen und Datenstrukturen zugrunde liegen, lohnt es meist nicht, eigene Implementierungen vorzunehmen. Im folgenden soll deshalb gezeigt werden, wie man unter Verwendung des Collection Frameworks recht schnell zu effizienten Programmen kommt.

7.1.1 Basisinterface

Sämtliche Collections besitzen gleichartige Zugriffsmethoden, die in einem Basisinterface `Collection` zusammengefasst sind.

Einerseits erreicht man mit diesem Interface, dass trotz der Verschiedenartigkeit der Collections eine einheitliche Schnittstelle verfügbar ist, denn jede Klasse, die `Collection` implementiert, muss natürlich alle Methoden des Interface implementieren.

Andererseits kann es aber auch vorkommen, dass nicht alle der angegebenen Methoden immer sinnvoll sind. Deshalb unterscheidet man zwischen obligatorischen Methoden (zwingend notwendig) und optionalen Methoden, die nicht unbedingt immer verfügbar sein müssen. Will man beispielsweise eine Collection implementieren, die die (optionale) Methode `clear` nicht anbietet, dann kann man die Methode so implementieren, dass ihr Aufruf immer zum Fehler `UnsupportedOperationException` führt, die Methode in der Praxis also nicht genutzt werden kann.

`Collection`-Interface

`int size()`	liefert die Anzahl der Einträge
`boolean isEmpty()`	prüft, ob keine Einträge vorhanden sind
`boolean contains(Object o)`	prüft, ob `o` eingetragen ist
`boolean containsAll(Collection c)`	prüft, ob alle Elemente aus `c` enthalten sind
`Iterator iterator()`	erzeugt einen Iterator
`boolean add(Object o)`	trägt `o` ein (optional)
`boolean addAll(Collection c)`	trägt alle Elemente aus `c` ein (optional)
`boolean remove(Object o)`	entfernt `o` (optional)
`boolean removeAll(Collection c)`	entfernt die in `c` angegebenen Elemente (optional)
`boolean retainAll(Collection c)`	entfernt alle Elemente, außer die in `c` angegebenen (optional)
`void clear()`	entfernt alle Elemente (optional)
`Object[] toArray()`	erzeugt ein Feld mit allen Einträgen
`<T> T[] toArray(T[] a)`	dsgl. mit Verwendung des Feldes `a`
`boolean equals(Object o)`	prüft, ob `o` mit der Collection übereinstimmt
`int hashcode()`	liefert einen Hashcode für die Collection

7.2 Iteratoren

Oft kommt es vor, dass man sämtliche Elemente einer Datenstruktur durchlaufen muss. Zum Beispiel können dies alle Knoten eines Graphen sein, oder alle Elemente einer verketteten Liste. Um für diese Aufgabe unabhängig von der Implementierung der Datenstruktur immer die gleichen Zugriffsmethoden verwenden zu können, führt man Iteratoren ein. Sie haben immer die folgenden Methoden:

`Iterator`-Interface

`boolean hasNext()`	prüft, ob ein weiteres Element existiert
`Object next()`	liefert das nächste Element und schaltet weiter
`void remove()`	löscht das aktuelle Element

Je nach Datenstruktur können weitere Methoden dazukommen. So besitzt beispielsweise das aus `Iterator` abgeleitete Interface `ListIterator` mit dem man doppelt verkette Listen oder Felder durchlaufen kann, folgende zusätzlichen Methoden:

Zusätzlich im `ListIterator`-Interface

`boolean hasPrevious()`	prüft, ob ein Vorgänger existiert
`Object previous()`	liefert das vorherige Element
`int nextIndex()`	liefert den Index des nächsten Elements (oder `size()`)
`int previousIndex()`	liefert den Index des vorherigen Elements (oder -1)
`void add(Object o)`	fügt ein neues Element `o` hinter dem aktuellen Element ein
`void set(Object o)`	ersetzt das aktuelle Element durch `o`

Als Beispielanwendung erzeugen wir im folgenden eine Liste von Strings, in die nacheinander mehrere zusätzliche Elemente eingefügt werden.

```
import java.util.*;

public class Beispiel
  {
   public static void main(String[] args)
     {
      LinkedList<String> li= new LinkedList<String>();
      li.add("Bananen");
      li.add("Äpfel");
      li.add("Birnen");
      li.add("Kirschen");

      // Ausgabe aller Elemente
      for (Iterator it = li.iterator(); it.hasNext(); )
        System.out.println(it.next());
      System.out.println();

      //Einfügen weiterer Elemente vor "Birnen"
      ListIterator<String> it2 = li.listIterator();
      it2.next();
      it2.next();
      it2.add("Orangen");
      it2.add("Zitronen");

      // Erneute Ausgabe aller Elemente
      for (Iterator it = li.iterator(); it.hasNext(); )
        System.out.println(it.next());
     }
  }
```

Die Liste `li` ist ein Objekt der Klasse `LinkedList<String>`, stellt also eine verkettete Liste dar, die nur Strings aufnehmen kann.

Bei den Ausgaben der Liste wird im Beispiel jeweils ein Iterator `it` der Klasse `Iterator` genutzt. Für das Einfügen von Elementen ist dieser Iterator allerdings nicht geeignet, da er über keine `add`-Methode verfügt. Deshalb wird das Einfügen im Beispiel mit Hilfe des Iterators `it2` der Klasse `LinkedList` vorgenommen. Um hier bereits durch den Compiler ein typsicheres Einfügen zu erzwingen, verwendet man noch besser die Klasse `LinkedList<String>`.

7.3 foreach-Schleife

Das Durchlaufen von Daten mit Hilfe von Iteratoren ist eine derart häufige Anwendung, dass dafür in Java (seit `J2SE 5.0`) eine spezielle Notation der **for**-Schleife eingeführt wurde. Für die Ausgabe der Liste `li` aus dem obigen Beispiel kann man deshalb viel kürzer schreiben:

```
for (String s: li)
   System.out.println(s);
```

Dabei durchläuft `s` nacheinander alle Elemente von `li`;
Die allgemeine Syntax dieser **foreach**-Schleife lautet

for (*formalerparameter* : *ausdruck*) *anweisung*;

Dabei steht *formalerparameter* für eine Deklaration der Art `int i` oder `String s`, also der Deklaration einer Laufvariablen. *ausdruck* steht für ein Array oder eine Instanz einer Klasse, die das Interface `java.lang.Iterable` implementiert.
Die Wirkung der **for**-Schleife kann dann durch folgendes äquivalentes Programmstück definiert werden:

```
for (Iterator it = ausdruck.iterator(); it.hasNext(); )
   { formalerparameter = it.next();
     anweisung;
   }
```

Wendet man die **foreach**-Schleife auf Felder an, dann entspricht beispielsweise die Formulierung

```
int[] v= {1,3,5,7,9};
for (int x: v)
   System.out.println(x);
```

der folgenden herkömmlichen Notation

```
int[] v= {1,3,5,7,9};
for (int i=0; i < v.length; ++i)
  {int x= v[i];
   System.out.println(x);
  }
```

7.4 Listen

Das für Listen verfügbare Interface umfasst natürlich das Basis-Interface `Collection` und enthält folgende weitere Methoden:

`List`-Interface

`Object get(int i)`	liefert das Element an Position `i` (ohne es zu entfernen)
`int indexOf(Object o)`	liefert den Index des ersten Vorkommens von `o` oder -1
`int lastIndexOf(Object o)`	liefert den Index des letzten Vorkommens von `o` oder -1
`ListIterator listIterator()` `ListIterator listIterator(int i)`	erzeugt einen Listeniterator erzeugt einen Listeniterator für die Elemente ab Position `i`
`boolean addAll(int i, Collection c)`	fügt alle Elemente aus `c` an Position `i` beginnend ein
`Object remove(int i)`	entfernt das Element an Position `i` (und liefert es zurück)
`Object set(int i, Object o)`	ersetzt das Element an Position `i` durch `o` und liefert das ersetzte Element zurück
`List sublist(int from, int to)`	stellt die Listenelemente mit Index $from \leq i < to$ als Liste zur Verfügung

Das Interface `List` ist recht allgemein gehalten und gilt für beliebige Implementierungen, wie z.B. Felder und verkettete Listen. Spezielle Implementierungen haben noch zusätzliche Methoden.

Einige zusätzliche Methoden der Klasse `LinkedList` (doppelt verkettete Listen)

`Object element()`	liefert das erste Listenelement (ohne es zu entfernen)
`Object getFirst()`	liefert das erste Element (ohne es zu entfernen)
`Object getLast()`	liefert das letzte Element (ohne es zu entfernen)
`void addFirst(Object o)`	Fügt `o` als erstes Element der Liste ein
`void addLast(Object o)`	Hängt `o` als letztes Element der Liste an
`Object removeFirst()`	liefert das erste Element und löscht es aus der Liste
`Object removeLast()`	liefert das letzte Element und löscht es aus der Liste
`int lastIndexOf(Object o)`	liefert den Index des letzten Vorkommens von `o` oder -1 falls `o` nicht enthalten ist

Als Beispiel einer abstrakten Datenstruktur haben wir im Abschnitt 5.1 eine Implementierung der Klasse `Stack<E>` angegeben. Sie bestand im wesentlichen aus einer verketteten Liste. Der `push`-Operator hat ein Element am Listenende angehängt und der `pop`-Operator hat ein Element am Listenende entfernt.

Unter Ausnutzung der Klasse `LinkedList` lässt sich der Stack natürlich wesentlich einfacher implementieren:

```
import java.util.*;

public class Stack<E>
 {private LinkedList<E> c;                  // lokale Liste

  public Stack() {c=new LinkedList<E>();}  // Erzeugen einer leeren Liste
  public void push(E o) {c.addLast(o);}    // neues Element anhängen
  public E pop() {return c.removeLast();}  // letztes Element zurückgeben
                                           // und entfernen
  public boolean isEmpty()                 // Test auf leere Liste
    {return (c.isEmpty();}
 }
```

Als Alternative zur Klasse `LinkedList` gibt es die Klasse `ArrayList` bei der sehr schnelle Elementzugriffe über einen Index möglich sind. Ein eventuelles Verlängern der Liste ist jedoch aufwändiger.

7.5 Sets (Mengen)

Ein wesentlicher Unterschied zwischen Mengen und Listen besteht darin, dass Mengen die gleichen Elemente nicht mehrmals enthalten können. Versucht man beispielsweise bei einem `Set` ein bereits vorhandenes Element erneut mit der Methode `add` einzufügen, so wird das Element nicht eingefügt und man erhält als Rückgabewert der Methode `false` zurück.

Ein weiterer Unterschied zwischen Mengen und Listen ist, dass bei Mengen keine Reihenfolge der Elemente festgelegt ist. Entsprechend werden die Mengenelemente von Iteratoren in einer willkürlichen Reihenfolge durchlaufen.

Die Implementierung eines `Set` kann beispielsweise mit Hilfe einer Hashtabelle erfolgen. In Java entspricht dies der Klasse `HashSet`.

Als Beispielanwendung werden im folgenden die Vielfachen von 6 und die Vielfachen von 8 in ein `HashSet` eingetragen. Anhand der Kontrollausgaben erkennt man, dass tatsächlich keine Zahl mehrfach eingetragen wird. Insbesondere ist der Wert $24 = 4 \cdot 6 = 3 \cdot 8$ nur einmal eingetragen.

```
HashSet<Integer> hs = new HashSet<Integer>();
 for (int i=1; i<10; ++i)
    {if (hs.add(6*i))
       System.out.println("Eintrag von " + 6*i);
     if (hs.add(8*i))
       System.out.println("Eintrag von " + 8*i);
    }
 System.out.print("hs=");
 for (Integer x:hs)
   System.out.print(" "+x);
 System.out.println();
```

Bei der Ausgabe der kompletten Menge erhält man beispielsweise
`hs= 30 36 8 40 72 16 18 48 54 12 64 32 6 24 42 56`

Alternativ kann man eine Menge auch durch eine Baumstruktur implementieren. Das geschieht in Java mit Hilfe der Klasse `TreeSet`. Obiges Beispiel der Vielfachen von 6 oder 8 sieht dann folgendermaßen aus.

```
TreeSet<Integer> ts = new TreeSet<Integer>();
for (int i=1; i<10; ++i)
   {if (ts.add(6*i))
      System.out.println("Eintrag von " + 6*i);
    if (ts.add(8*i))
      System.out.println("Eintrag von " + 8*i);
   }

System.out.print("ts=");
for (Integer x:ts)
  System.out.print(" "+x);
System.out.println();
```

Als Ausgabe der kompletten Menge erhält man nun
`ts= 6 8 12 16 18 24 30 32 36 40 42 48 54 56 64 72.`
Ein Nebeneffekt der Baumstruktur ist anscheinend, dass die Elemente sortiert abgelegt werden.

Tatsächlich implementiert die Klasse `TreeSet` nicht nur das Interface `Set`, sondern sogar das daraus abgeleitete Interface `SortedSet`. Deshalb stehen hier folgende zusätzlichen Methoden zur Verfügung

Zusätzlich im `SortedSet`-Interface

`Object first()`	liefert das erste Element bzgl. der Sortierung
`Object last()`	liefert das letzte Element bzgl. der Sortierung
`SortedSet headSet(Object to)`	liefert die Menge aller Elemente, die kleiner als `to` sind.
`SortedSet tailSet(Object from)`	liefert die Menge aller Elemente, die größer oder gleich `from` sind.
`SortedSet subSet(Object from, to)`	liefert die Menge aller Elemente, die größer oder gleich `from` und kleiner `to` sind.

Bei sortierten Collections wird vorausgesetzt, dass die Elemente das Interface `Comparable` implementieren, also die Methode `int compareTo(Object o)` besitzen, mit der zwei Objekte miteinander verglichen werden können. Durch Änderung der Vergleichsmethode kann die Art der Sortierung verändert werden.
Eine einfachere Möglichkeit die Sortierungsmethode festzulegen, besteht darin, dem Konstruktor von `TreeSet` ein Objekt mitzugeben, das das Interface `Comparator` mit der zu verwendenden Vergleichsmethode implementiert. In diesem Fall wird innerhalb der Klasse `TreeSet` statt mit `int compareTo(Object o)` mit der Methode `int compare(Object o1, Object o2)` gearbeitet.
Im folgenden Beispiel wird diese Technik eingesetzt, um eine Zahlenfolge monoton fallend in einer Baumstruktur zu speichern.

```
TreeSet<Integer> tsr = new TreeSet<Integer>(new ReversInteger());
for (int i=1; i<10; ++i)
   {if (tsr.add(6*i))
      System.out.println("Eintrag von " + 6*i);
    if (tsr.add(8*i))
      System.out.println("Eintrag von " + 8*i);
   }

System.out.print("tsr=");
for (Integer x:tsr)
  System.out.print(" "+x);
System.out.println();
```

Dabei soll die Klasse `ReversInteger` folgendermaßen definiert werden

```
class ReversInteger implements Comparator
{public int compare(Object o1, Object o2)
   {return ((Integer)o2).compareTo((Integer)o1);}
}
```

7.6 Maps (Abbildungen)

Collections des Typs `Map` verwendet man, um Paare von Daten zu speichern. Dabei besteht ein Datenpaar aus einem *Schlüssel* und einem *Wert*. Als Beispiel denke man etwa an ein Telefonbuch, in dem man zu Namen die dazugehörigen Telefonnummern finden kann. Die Namen dienen dabei als Schlüssel und die Telefonnummern sind die dazugehörigen Werte.
Grundsätzlich kann eine Map mit Hilfe eines balancierten Baumes implementiert werden, in dem man die Datenpaare nach dem Schlüssel sortiert einträgt. Dabei ergibt sich eine logarithmische Zugriffszeit auf die Daten. Noch schneller ist die Implementierung mit einem Hashverfahren.
Ein von der gewählten Implementierung unabhängiges allgemein verwendbares Interface `Map` enthält folgende Methoden.

`Map`-Interface

`int size()`	liefert die Anzahl der Einträge
`boolean isEmpty()`	prüft, ob keine Einträge vorhanden sind
`boolean containsKey(Object key)`	prüft, ob ein Datenpaar mit Schlüssel `key` eingetragen ist
`boolean containsValue(Object value)`	prüft, ob ein Datenpaar mit Wert `value` eingetragen ist
`Object get(Object key)`	liefert den eingetragenen Wert zum Schlüssen `key`
`Object put(Object key, Object value)`	trägt das Datenpaar(`key`,`value`) ein (optional)
`void putAll(Map t)`	trägt alle Datenpaare aus `t` ein (optional)
`boolean remove(Object key)`	entfernt das Datenpaar mit Schlüssel `key` (optional)
`void clear()`	entfernt alle Datenpaare (optional)
`Set<K> keySet()`	stellt die eingetragenen Schlüssel als Menge dar (keine Kopie!)
`Collection<V> values()`	stellt die eingetragenen Werte als Collection dar (keine Kopie!)
`Set<Map.Entry<K,V>> entrySet()`	stellt die eingetragenen Datenpaare als Menge dar (keine Kopie!)
`boolean equals(Object o)`	vergleicht die Map mit `o`
`int hashcode()`	liefert den Hashcode der Map

Auffällig ist, dass eine `Map` keinen Iterator zum Durchlaufen der Einträge besitzt. Stattdessen hat man die Möglichkeit, die Schlüssel, Werte oder Datenpaare mit Hilfe der Methoden `keySet`, `values` bzw. `entrySet` als Menge oder Collection darzustellen und für diese dann Iteratoren zu erzeugen. Dabei greift man weiterhin auf die Originaleinträge der Map zu, d.h. eventuelle Änderungen in der Map wirken sich beispiel-

sweise auch auf die von `keySet` erzeugte Schlüsselmenge aus.
Als Beispiel programmieren wir ein Telefonbuch mit Hilfe einer `Map`. Als Implementierung der Map benutzen wir dir Klasse `HashMap`, bei der die Datenpaare mit Hilfe einer Hashtabelle gespeichert werden, so dass die einzelnen Datenzugriffe im Durchschnitt in konstanter Zeit erfolgen.

```
import java.util.*;
public class testmap
{public static void main(String[] args)
  {Map<String,String> telefonbuch = new HashMap<String,String>();

   telefonbuch.put("Holger",  "504030");
   telefonbuch.put("Bernd",   "12345");
   telefonbuch.put("Michael", "654902");

   Set<String>        namen= telefonbuch.keySet();
   Collection<String> nummern= telefonbuch.values();

   telefonbuch.put("Monika", "151720");
   telefonbuch.put("Ursula", "980145");

   // Ausgabe aller eingetragenen Namen
   System.out.print("Namen:");
   for (String s: namen)
     System.out.print(" " + s);     // Holger Michael Ursula Monika Bernd
   System.out.println();

   // Ausgabe aller eingetragenen Nummern
   System.out.print("Telefonnummern:");
   for (String s: nummern)
     System.out.print(" " + s);     // 504030 654902 980145 151720 12345
   System.out.println();

   // Beispielzugriffe auf das TelefonBuch
   String[] v= {"Monika", "Martin", "Jürgen", "Bernd"};
   for (String s: v)
     if (telefonbuch.containsKey(s))
       {String n= telefonbuch.get(s);
        System.out.println(s + " hat die Telefonnummer " + n);
       }
     else
       System.out.println(s + " ist nicht im Telefonbuch zu finden");
   }
}
```

Man beachte, dass die hier verwendeten Iteratoren die Elemente in einer willkürlichen Reihenfolge ausgeben, da die Einträge einer Hashtabelle naturgemäß nicht sortiert sind.
Alternativ hätten wir statt dem Interface `Map` auch `SortedMap` verwenden können. Es ist ähnlich wie `SortetSet` aufgebaut und enthält folgende Methoden:

Zusätzlich im `SortedMap`-Interface

`Object first()`	liefert das erste Element bzgl. der Sortierung
`Object last()`	liefert das letzte Element bzgl. der Sortierung
`SortedMap headMap(Object to)`	liefert die Menge aller Elemente, die kleiner als `to` sind.
`SortedMap tailMap(Object from)`	liefert die Menge aller Elemente, die größer oder gleich `from` sind.
`SortedMap subMap(Object from, to)`	liefert die Menge aller Elemente, die größer oder gleich `from` und kleiner `to` sind.

Als Implementierung dieses Interfaces gibt es die Klasse `TreeMap`. Hier werden die Datenpaare in einem balancierten Baum gespeichert. Dabei wird die natürliche Ordnung der Schlüssel für die Sortierung genutzt. Die Methoden `keySet` und `entrySet` liefern dabei Collections, deren Iteratoren die Elemente aufsteigend durchlaufen.
Das folgende Programm ist ein Anwendungsbeispiel, bei dem sämtliche in der Kommandozeile stehenden Strings durchlaufen werden. Dabei wird gezählt, wieoft jeder String vorkommt. Anschließend werden die Strings alphabetisch sortiert mit ihren Häufigkeiten ausgegeben.

```
public class Textanalyse
{public static void main(String[] args)
  {SortedMap<String,Integer> statistik = new TreeMap<String,Integer>();
   for (String s: args)
      if (statistik.containsKey(s))
        {int n=statistik.get(s);
         statistik.put(s,n+1);      // Zähler für s erhöhen
        }
      else
        statistik.put(s,1);         // erstes Vorkommen von s

   // Ergebnisausgabe
   for (String s:  statistik.keySet())
      System.out.println(s + ": "+ statistik.get(s));
  }
}
```

7.7 Tabelle implementierter Collections

Die folgende Tabelle gibt eine (unvollständige) Übersicht über die verfügbaren Implementierungen von Collections:

`Collection-Typ`	Beschreibung
`ArrayList`	Indizierte Liste, deren Größe dynamisch verändert werden kann. Indexzugriffe sind schnell, Größenänderungen sind aufwändig.
`LinkedList`	Verkettete Liste. Indexzugriffe sind langsam, Einfügen und Löschen ist schnell.
`HashSet`	ungeordnete Datenmenge (ohne Duplikate)
`TreeSet`	Sortierte Menge.
`EnumSet`	Menge für Aufzählungstypen.
`LinkedHashSet`	Eine Menge mit zusätzlicher Information über die Reihenfolge der Einfügungen.
`PriorityQueue`	Datenmenge, die das Entfernen des kleinsten Elements effizient ermöglicht.
`HashMap`	Menge von (Schlüssel,Wert)-Paaren
`TreeMap`	nach Schlüsseln sortierte Menge von (Schlüssel,Wert)-Paaren
`EnumMap`	Map bei der die Schlüssel zu einem Aufzählungstyp gehören
`LinkedHashMap`	Map, mit zusätzlicher Information über die Reihenfolge der Einfügungen.
`WeakHashMap`	Map, bei der der Garbage-Collector nicht mehr benötigte Elemente entfernen kann.
`IdentityHashMap`	Map, bei der die Schlüssel mit `==` statt mit `equals` verglichen werden.

7.8 Typkompatibilität bei Collections

Wir wissen, dass man Objekte einer Unterklasse auch an Referenzvariablen einer entsprechenden Oberklasse zuweisen kann. Als Beispiel betrachten wir die Klasse `Number`, die Oberklasse von `Integer`, `Double`, `Float`, ... ist. Deshalb kann im folgenden Beispiel die Variable `n` unter anderem Objekte der Klassen `Integer` und `Double` aufnehmen.

```
Integer i = new Integer(13);
Double d = new Double(3.14);
Number n = d;        // erlaubte Zuweisung
n= i;                // erlaubte Zuweisung
```

Versucht man diesen Mechanismus auf typisierte Collections anzuwenden, treten unerwartete Probleme auf. Im folgenden soll dies am Beispiel der Klasse `Vector<E>` demonstriert werden. Sie implementiert ebenfalls das Collection-Interface und kann ähnlich wie `ArrayList` verwendet werden. Beispielsweise zeigt sich, dass die Klasse `Vector<Number>` keine Oberklasse von `Vector<Integer>` und `Vector<Double>` ist.

Anderenfalls könnte man nämlich mit dem folgenden Programmstück `Double`-Werte in eine `Integer`-Vektor schreiben.

```
Vector<Integer> vi= new Vector<Integer>();
Vector<Number> vn= vi;    // unzulässige Anweisung
vn.add(new Double(3));    // wäre ein Konflikt zu vn=vi
```

Genauso sieht man, dass `Vector<Object>` keine Oberklasse von `Vector<Integer>` und `Vector<Double>` ist. Zum Definieren einer Oberklasse mit der Bedeutung "Vektor mit Elementen eines beliebigen Typs" benutzt man die Schreibweise `Vector<?>`. Der unbekannte Typ der Komponenten wird hier also durch den **Wildcard** ? dargestellt. Als Beispiel betrachten wir die Ausgabe eines beliebigen Vektors:

```
public static void print (Vector<?> v)
{
 for (int i=0; i<v.size(); ++i)
   System.out.print(" " + v.get(i));
 System.out.println();
}
```

Will man dagegen beispielsweise eine Funktion schreiben, die das Maximum der Vektorkomponenten ermittelt, muss der Typ der Vektorkomponenten das Interface `Comparable` implementieren.
Diese Nebenbedingung kann durch `Vector<? extends Comparable>` formuliert werden. Man spricht dann von einem "gebundenen Wildcard".

```
public static Comparable max(Vector<? extends Comparable> v)
{
 Comparable result = v.get(0);
 for (Comparable x : v)
   if (result.compareTo(x) < 0)
     result = x;
 return result;
}
```

Entsprechend könnten wir den Typ auch durch `Vector<? extends Number>` auf die von `Number` abgeleiteten Klassen beschränken.

```
public static Number max(Vector<? extends Number> v)
{
 Number result = v.get(0);
 for (Number x : v)
   if (result.compareTo(x) < 0)
     result = x;
 return result;
}
```

Beide Implementierungen haben aber den "Schönheitsfehler", dass der Ergebnistyp nicht mit dem Elementtyp des Vektors übereinstimmt, sondern nur eine Oberklasse darstellt. Deshalb sollte man besser wie im folgenden Beispiel dargestellt, den variablen Typ durch einen Typparameter darstellen.

```
public static <T extends Comparable<T>> T max(Vector<T> v)
{T result = v.get(0);
 for (T x : v)
  if (result.compareTo(x) < 0)
 result = x;
 return result;
}
```

Man beachte, dass hier auch das im Abschnitt 6.1 vorgestellte typsichere Interface `Comparable<T>` verwendet wird, so dass die Typprüfung im Ausdruck `result.compareTo(x)` bereits vom Compiler vorgenommen werden kann und nicht erst zur Laufzeit des Programms erfolgt.
In allen angegebenen Beispielen mit Verwendung von Wildcards werden nur Lesezugriffe auf Vektoren vorgenommen. Tatsächlich sind Schreibzugriffe auf Parameter grundsätzlich nicht zulässig, wenn der Parametertyp mit einem Wildcard beschrieben ist. Auf diese Weise wird in Java das oben beschriebene Schreibproblem vermieden. (Eintrag einer `Double`-Größe in eine Collection, die nur Objekte der Klasse `Integer` enthalten darf, über den Umweg einer `Number`-Referenzvariable.)
Folgendes Programmstück ist deshalb nicht korrekt:

```
Vector<Integer> vi = new Vector<Integer>();
vi.add(2);
vi.add(4);
vi.add(6);
Vector<? extends Number> xv= vi;
xv.add(new Double(1.5));   // Unzulässiger Schreibzugriff
for (Number local:xv)
 System.out.println(local);
```

8 Dokumentation von Java-Programmen

Üblicherweise dokumentiert man Programme mit Hilfe von Kommentaren. Dazu verwendet man in Java die gleiche Notation wie in C++. Einzeilige Kommentare werden mit `//` eingeleitet und mehrzeilige Kommentare werden in `/*` und `*/` eingeschlossen. Zusätzlich gibt es in Java aber auch spezielle Kommentare, aus denen automatisch mit Hilfe des Tools **javadoc** eine Dokumentation im `HTML`-Format generiert werden kann. Beispielsweise wird die unter `http://docs.oracle.com/javase/1.5.0/docs/api/` verfügbare Dokumentation der Java-API (API=application programming interface) auf diese Weise erzeugt.
Dokumentationskommentare werden mit `/**` eingeleitet und mit `*/` beendet. Zur besseren Lesbarkeit ist es üblich, die zwischen diesen Begrenzern liegenden Zeilen jeweils mit einem `*` einzuleiten, notwendig ist dies jedoch nicht.
Als Beispiel betrachten wir die in Tabelle 1 angegebene Implementierung der Klasse `Counter`, die mit Dokumentationskommentaren versehen ist. Man erkennt, dass die Kommentare jeweils **vor** dem beschriebenen Programmteil stehen. Das Programm `javadoc` extrahiert die Kommentare für öffentliche Klassen, Schnittstellen und Methoden, entfernt die Kommentarzeichen und ordnet die Texte jeweils der Programmeinheit zu.
Die wichtigsten Makros für die Dokumentation sind in folgender Tabelle zusammengefasst.

`@param`	Angabe des Namens und einer Beschreibung eines Parameters Beispiel: `@param ch the character to be tested`
`@return`	beschreibt den Rückgabewert einer Methode
`@throws`	Angabe der zu behandelnden Exceptions
`@author`	Autor des Programms
`@version`	Programmversion
`@see`	Verweis auf andere Teile der Dokumentation

Der erste Satz eines Dokumentationskommentars hat die Bedeutung einer Überschrift. Er wird für Kurzbeschreibungen verwendet. Danach folgt die ausführliche Beschreibung. Mit `@` werden Makros eingeleitet, die die Angaben strukturieren. So wird beispielsweise nach `@author` der Autor des Programms angegeben und nach `@version` die Beschreibung der Programmversion.
Am Beispiel des Konstruktors `public Counter(String s)` sieht man, wie sich die Bedeutungen von Parametern mit dem `@param`-Makro beschreiben lassen.
Zur Strukturierung und zum Hervorheben von Textstellen kann man in den Dokumentationskommentaren beliebige `HTML`-Befehle einsetzen. Beispielsweise sollte man Klassennamen, Methodennamen und andere im Java-Programm vorkommende Bezeichner in den Kommentaren in `<code>` ... `</code>`-Blöcke einschließen, um sie optisch hervorzuheben. Im Beispiel wurde diese Technik in der Beschreibung des Hauptprogramms für die Referenzvariable `a` und für `System.out` verwendet.

```
/**
 * This class provides the functionality of a counter. The internal state
 * of a counter is an integer value that can be incremented or resetted.
 * Other modifications are not permitted.
 *
 * @author Wolfgang Rülling
 * @version 1.0 (November 2006)
 */

public class Counter
{protected int x;
 protected String name;

 /**
 * Creates a counter initialized to 0.
 */
 public Counter() {x=0;}

 /**
 * Creates a named counter intialized to 0.
 *
 * @param s name of this counter
 */
 public Counter(String s) {x=0; name=s; }

 /**
 * Increments the value of a counter.
 */
 public void increment() {++x;}

 /**
 * Method to get the value of a counter.
 * @return the integer value of this counter
 */
 public int get() {return x;}

 /**
 * Resets the value of a counter.
 */
 public void reset(){x=0;}

 /**
 * Method to get a description of a counter.
 * @return a string describing this counter. It consits of the
 * counter's name and it's value enclosed in brackets.
 * For example the result may look like "Counter a(1)"
 * for a counter with name "a" and value 1
 */
 public String toString() { return "Counter " + name + "(" + x + ")"; }
}
```

```
 /**
 * Program to demonstrate the behavior of counters.
 *
 * At first two counters are created named "a" and "b".
 * Then <code>a</code> is incremented and the values of both counters
 * are written to <code>System.out</code>.
 * Finally the name of counter <code>a</code> is written
 * to <code>System.out</code>.
 *
 * @param args commandline arguments (unused)
 */
 public static void main(String[] args) {
 System.out.println();
 Counter a= new Counter("a");
 Counter b= new Counter("b");
 a.increment();
 System.out.println("a="+a.get());
 System.out.println("b="+b.get());
 System.out.println(a.toString());
 }
}
```

Table 1: Programmbeispiel mit Verwendung von Dokumentationskommentaren

Außerdem bietet sich das HTML-Tag `<p>` zum Trennen von Absätzen an. Gelegentlich ist es auch sinnvoll, Sonderzeichen in der bei HTML üblichen Notation zu schreiben. Dies ist zum Beispiel sinnvoll, wenn der erste Satz eines Dokumentationskommentars intern einen Abkürzungspunkt enthält. Da ein von einem Trennzeichen (Leerzeichen oder Zeilenumbruch) gefolgter Punkt als Satzendepunkt interpretiert wird, sollte man nach dem Abkürzungspunkt ein HTML-Sonderzeichen einfügen. Als Beispiel betrachte man folgenden Kommentar.

```
/** Algorithmus von V. Strassen zur Multiplikation von Matrizen.
 */
```

Damit dieser Satz nicht nach `V.` endet, sollte er folgendermaßen formuliert werden.

```
/** Algorithmus von V. Strassen zur Multiplikation von Matrizen.
 */
```

In Tabelle 2 ist auf den folgenden drei Seiten auszugsweise dargestellt, wie die für die Klasse `Counter` aus dem Programmbeispiel 1 erzeugte `HTML`-Dokumentation aussieht. Sie ist tatsächlich genauso aufgebaut wie die offizielle Java-Dokumentation und besitzt auch den gleichen Benutzungskomfort durch ein generiertes Indexverzeichnis, ein Verzeichnis aller Klassen, sowie der Möglichkeit über **Links** zwischen verschiedenen Teilen der Dokumentation zu wechseln.

Class Counter

java.lang.Object


Counter

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

This class provides the functionality of a counter. The internal state of a counter is an integer value that can be incremented or resetted. Other modifications are not permitted.

Version:
1.0 (November 2006)
Author:
Wolfgang Rülling

Constructor Summary

Counter() Creates a counter initialized to 0.
Counter(java.lang.String s) Creates a named counter intialized to 0.

Method Summary

int	**get**() Method to get the value of a counter.
void	**increment**() Increments the value of a counter.
static void	**main**(java.lang.String[] args) Program to demonstrate the behavior of counters.
void	**reset**() Resets the value of a counter.
java.lang.String	**toString**() Method to get a description of a counter.

Constructor Detail

Counter

```
public Counter()
```

Creates a counter initialized to 0.

Counter

```
public Counter(java.lang.String s)
```

Creates a named counter intialized to 0.

Parameters:
`s` - name of this counter

Method Detail

increment

```
public void increment()
```

Increments the value of a counter.

---

get

```
public int get()
```

Method to get the value of a counter.

Returns:
the integer value of this counter

---

reset

```
public void reset()
```

Resets the value of a counter.

---

toString

`public java.lang.String` **`toString`**`()`

Method to get a description of a counter. It consits of the counter's name and it's value enclosed in brackets. For example the result may look like "Counter a(1)" for a counter with name "a" and value 1

Overrides:

`toString` in class `java.lang.Object`

---

main

`public static void` **`main`**`(java.lang.String[] args)`

Program to demonstrate the behavior of counters. At first two counters are created named "a" and "b". Then `a` is incremented and the values of both counters are written to `System.out`. Finally the name of counter `a` is written to `System.out`.

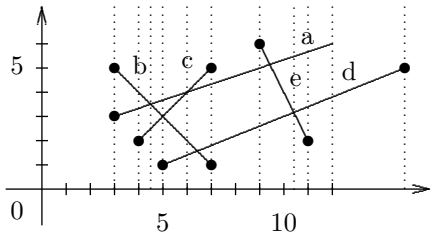
Parameters:

`args` - commandline arguments (unused)

Table 2: Ausschnitte aus der von `javadoc` generierten HTML-Dokumentation

9 Beispielanwendung: Algorithmus von Bentley-Ottmann

Als Anwendungsbeispiel für die Verwendung von Collections betrachten wir das Problem zu einer Menge von n Liniensegmenten in der Ebene die Menge aller Schnittpunkte zu ermitteln.



Beispielsweise ergeben die in der Skizze dargestellten fünf Liniensegmente a, b, c, d und e sechs Schnittpunkte. Die einfachste Methode die Menge der Schnittpunkte zu berechnen, besteht darin paarweise für je zwei Liniensegmente zu prüfen, ob sie sich schneiden. Diese Methode hat also die Laufzeit $T = \mathrm{O}(n^2)$. Insbesondere für den Fall, dass man nur relativ wenige Schnittpunkte erwartet, ist diese Laufzeit unnötig hoch.

Der im folgenden behandelte **Algorithmus von Bentley-Ottmann** benötigt stattdessen nur die Laufzeit $T = \mathrm{O}((n + k) \cdot \log n)$, wobei k die (unbekannte) Anzahl der Schnittpunkte ist.

Die Grundidee des Algorithmus besteht darin, eine Sweepline von links nach recht über die Ebene zu bewegen und dabei zu beobachten, welche Segmente jeweils auf der Sweepline liegen. Im Beispiel überstreicht die Sweepline beispielsweise für $x = 4$ nur die Segmente c, a und b (in dieser Reihenfolge von unten nach oben). Bevor man mit der Sweepline einen Schnittpunkt erreicht, treten die betroffenen Liniensegmente als Nachbarn in der Sweepline auf. Deshalb muss der Algorithmus jeweils nur für benachbarte Segmente prüfen, ob sie sich schneiden.

Eine weitere wesentliche Idee besteht darin, dass man die Sweepline nicht kontinuierlich verschiebt, sondern sie von einem relevanten Punkte zum nächsten springen lässt. Diese Punkte nennt man dann Events. Zunächst sind alle Anfangs- und Endpunkte von Segmenten solche Events. Immer wenn man während des Ablaufs einen Schnittpunkt für benachbarte Segments berechnet, fügt man diesen zusätzlich in die Menge der Events ein.

Beim Verschieben der Sweepline zu einem solchen Schnittpunkt muss man dann die beiden betroffenen Segmente austauschen, um ihre Reihenfolge in der Sweepline zu korrigieren.

Im folgenden ist der Pseudocode dieses Algorithmus dargestellt. Die Initialisierung vor der while-Schleife kann in Zeit $\mathrm{O}(n \cdot \log n)$ ausgeführt werden und der Rumpf der while-Schleife wird insgesamt $\mathrm{O}(n + k)$-mal ausgeführt. Die geforderte Gesamtlaufzeit von $T = \mathrm{O}((n+k) \cdot \log n)$ wird also eingehalten, wenn alle Anweisungen des Schleifenrumpfs jeweils mit Laufzeit $\mathrm{O}(\log(n))$ implementiert werden können.

9.1 Pseudocode

```
eventQueue = alle Endpunkte von Segmenten;
Sortiere die eventQueue aufsteigend nach x und y;
sweepLine = leere Liste;
result = leere Menge;
while (eventQueue ≠ ∅)
  {e = nächster event der eventQueue;
   entferne e aus der eventQueue;
   if (e ist Segmentanfang)
      {seg = zu e gehöriges Segment;
       füge seg sortiert in sweepLine ein;
       sega = Segment unterhalb von seg;
       segb = Segment oberhalb von seg;
       if (Schnittpunkt ia zwischen seg und sega existiert)
          füge ia in eventQueue ein;
       if (Schnittpunkt ib zwischen seg und segb existiert)
          füge ib in eventQueue ein;
      }
   else if (e ist Segmentende)
      {seg = zu e gehöriges Segment;
       sega= Segment unterhalb von seg;
       segb= Segment oberhalb von seg;
       entferne seg aus sweepLine;
       if (Schnittpunkt i zwischen sega und segb existiert)
          füge i in eventQueue ein;
      }
   else
      {// e gehört zu einem Schnittpunkt
       füge den Schnittpunkt e in result ein;
       seg1 < seg2 seien die beiden zu e gehörenden Segmente;
       Vertausche die Positionen von seg1 und seg2 in sweepLine;
       // seg1 liegt nun oberhalb seg2
       sega= Segment unterhalb von seg2;
       segb= Segment oberhalb von seg1;
       if (Schnittpunkt ia zwischen sega und seg2 existiert)
          füge ia in eventQueue ein;
       if (Schnittpunkt ib zwischen segb und seg1 existiert)
          füge ib in eventQueue ein;
      }
  }
// result enthält nun die Liste aller Schnittpunkte von Liniensegmenten
```

Die `eventQueue` stellt im wesentlichen eine sortierte Liste von Punkten dar. Die benötigten Operationen sind das Lesen und Entfernen des ersten Elements und das sortierte Einfügen einzelner Punkte. Mit Hilfe eines `TreeSet` sind diese Operationen jeweils in Zeit $O(\log n)$ möglich. Zu beachten ist, dass die Elemente nach *Segmentanfang Segmentende* und *Schnittpunkt* unterschieden werden müssen. In Java können dazu drei Klassen `LeftEnd`, `RightEnd` und `Intersection` aus einer Basisklasse `Event` abgeleitet werden. Da zu jedem Event auch die dazugehörigen Segmente zugegriffen werden müssen, sollten die Segmente als Membervariablen gespeichert werden. (Alternativ könnte man sie über HashMaps den Punkten zuordnen.)
Das Ergebnis `result` ist eine Menge von Schnittpunkten. Hier wird lediglich das Einfügen von Elementen benötigt. Dabei muss natürlich das doppelte Eintragen von Elementen vermieden werden. Für die Implementierung kommt ein `TreeSet` oder `HashSet` infrage.
Am aufwändigsten ist die `sweepLine`. Sie stellt eine sortierte Liste von Segmente dar und muss das Einfügen und Löschen beliebiger Elemente unterstützen. Zusätzlich wird das Austauschen benachbarter Einträge (beim Weiterschalten der `sweepLine` an einem Schnittpunkt) benötigt. Außerdem muss man zu Einträgen den direkten Vorgänger und Nachfolger bezüglich der Sortierung zugreifen können.
Eine ideale Implementierung wäre eine verkettete Liste, in der man zusätzlich Einträge über ihre `y`-Koordinate mit Binärsuche zugreifen kann. Da nicht offensichtlich ist, wie diese Zugriffe direkt mit Java-Collections ausgeführt werden können, benutzen wir für die Implementierung zunächst eine Klasse `SweepLine`, die später zu implementieren ist.
Um den Algorithmus einfach aufrufen zu können, soll folgende Methode definiert werden.

```
public Set<Point> BentleyOttmannAlgorithmus(Set<Segment> segs);
```

Die für die Schnittstelle benötigten Klassen `Point` und `Segment` können folgendermaßen sehr einfach implementiert werden.

```
public class Point implements Comparable
{public double x,y;
 public Point(double x, double y) {this.x=x; this.y=y;}
 public Point(Point p) { x=p.x; y=p.y;}

 public int compareTo(Object o)
  {if (x<((Point)o).x) return -1;
   if (x>((Point)o).x) return +1;
   if (y<((Point)o).y) return -1;
   if (y>((Point)o).y) return +1;
   return 0;
  }
}
```

```
public class Segment
{public String name;
 public Point p,q;
 public Segment(String name, Point p, Point q)
  {this.name=name; this.p=p; this.q=q;}
 public Segment(Segment s)
  {this.name=s.name; this.p=s.p; this.q=s.q;}
}
```

Die weiteren benötigten Klassen haben nur interne Bedeutung und sollten deshalb lokal als **innere Klassen** definiert werden.

```
class Event implements Comparable
{
 Point p;
 public Event(Point p){this.p=p;}
 public int compareTo(Object o)
 {// Events werden nach ihrem Punkt sortiert
  Point p1= p;
  Point p2= ((Event)o).p;
  return p1.compareTo(p2);
 }
}

class LeftEnd extends Event
{Segment s;
 LeftEnd(Point p, Segment s) {super(p); this.s=s;}
}

class RightEnd extends Event
{Segment s;
 RightEnd(Point p, Segment s) {super(p); this.s=s;}
}

class Intersection extends Event
{Segment s1, s2;
 Intersection(Point p, Segment s1, Segment s2)
    {super(p); this.s1=s1; this.s2=s2;}
}
```

9.2 Java-Implementierung

Damit sind wir nun in der Lage den Pseudocode des Algorithmus direkt in Java zu übersetzen.

```
public Set<Point> BentleyOttmannAlgorithm(Set<Segment> segs)
{// Initialisierung der eventQueue mit allen Endpunkten von Segmenten
 SortedSet<Event> eventQueue = new TreeSet<Event>();
 for (Segment s:segs)
    {if (s.p.compareTo(s.q) <= 0)
       {eventQueue.add(new LeftEnd(s.p, s));
        eventQueue.add(new RightEnd(s.q, s));
       }
     else
       {eventQueue.add(new LeftEnd(s.q, s));
        eventQueue.add(new RightEnd(s.p, s));
       }
    }
 SweepLine sweepLine = new SweepLine(eventQueue.first().p.x);
 Set<Point> result= new TreeSet<Point>();  // leere Ergebnismenge
 while (eventQueue.size()>0)
    {Event e= eventQueue.first();  // erstes event holen
     eventQueue.remove(e);         // und entfernen
     sweepLine.setX(e.p.x);        // Position der sweepLine festlegen
     Segment s,low,high;

     if (e instanceof LeftEnd)
       {s=((LeftEnd)e).s;
        sweepLine.add(s);
        low  = sweepLine.getPrevious(s); // unterer Nachbar
        high = sweepLine.getNext(s);     // oberer Nachbar
        // Schnittpunkte berechnen und eintragen
        insert_event(low,s,  e.p,eventQueue);
        insert_event(s,high, e.p,eventQueue);
       }
      else if (e instanceof RightEnd)
       {s=((RightEnd)e).s;
        low  = sweepLine.getPrevious(s); // unterer Nachbar
        high = sweepLine.getNext(s);     // oberer Nachbar
        sweepLine.remove(s);
        // Schnittpunkt berechnen und eintragen
        insert_event(low,high,  e.p,eventQueue);
       }
```

```
     else
       {// e ist ein intersection event
        Segment s1=((Intersection)e).s1;
        Segment s2=((Intersection)e).s2;
        low=sweepLine.getPrevious(s1); // unterer Nachbar
        high=sweepLine.getNext(s2);    // oberer Nachbar

        sweepLine.actualize(s1,s2); // Positionen in sweepLine vertauschen

        // Schnittpunkte berechnen und eintragen
        insert_event(low,s2,  e.p,eventQueue);
        insert_event(s1,high, e.p,eventQueue);
       }
     }
 return result;
}
```

Zum Berechnen von Segmentschnittpunkten und zum Eintragen in die `eventQueue` werden die folgenden Funktionen benutzt:

```
    void insert_event(Segment a, Segment b, Point e, SortedSet<Event> eventQueue)
    {// Berechnet den Schnittpunkt der Segmente a und b und trägt ihn
     // als Event in eventQueue ein, falls er größer als e ist
        if ((a!=null)&&(b!=null))
        {Point i=intersection(a,b);
         if (i!=null)
            if (i.compareTo(e)>0)
              eventQueue.add(new Intersection(i,a,b));
        }
    }


    public Point intersection(Segment a, Segment b)
    {// Ermittelt den Schnittpunkt der beiden Segments a und b
     double am=(a.q.y-a.p.y)/(a.q.x-a.p.x); // Steigung von a
     double bm=(b.q.y-b.p.y)/(b.q.x-b.p.x); // Steigung von b
     double x= (b.p.y-a.p.y-bm*b.p.x + am*a.p.x)/(am-bm); // Geradenschnittpunkt
     double y= a.p.y+am*(x-a.p.x);
     if ((a.p.x<= x) && (x <= a.q.x))      // Bereichsprüfung
            return new Point(x,y);
     else
        return null;
    }
```

Damit steht nur noch die Implementierung der Klasse `SweepLine` aus. Die benötigten Methoden gemäß obiger Implementierung sind:

`SweepLine(double start)`: Konstruktor, der eine leere SweepLine an die x-Koordinate `start` positioniert.

`void setX(double x)`: verschiebt die SweepLine zur Position `x`. Dabei kann vorausgesetzt werden, dass `x` größer oder gleich der zuletzt verwendeten Position ist. Außerdem kann vorausgesetzt werden, dass zwischen der vorherigen Position und der neuen Position die Reihenfolge der Segmente in der SweepLine unverändert bleibt.

`void add(Segment s)`: fügt ein Segment `s` nach `y`-Koordinate sortiert in die SweepLine ein. Dabei ist zu beachten, dass die `y`-Koordinate von der aktuellen Position der SweepLine abhängt.

`void remove(Segment s)`: entfernt das Segment `s` aus der SweepLine.

`Segment getPrevious(Segment s)`: liefert den Vorgänger des Segments `s` in der SweepLine.

`Segment getNext(Segment s)`: liefert den Nachfolger des Segments `s` in der SweepLine.

`void actualize(Segment a, Segment b)`: vertauscht die Segmente `a` und `b` in der SweepLine. Dabei kann vorausgesetzt werden, dass die beiden Segmente in der SweepLine benachbart sind. Die alte Anordnung entspricht ihrer Reihenfolge "etwas" links von der SweepLine und die neue Anordnung entspricht ihrer Reihenfolge "etwas" rechts von der SweepLine.

Wie bereits erwähnt, dürfen sämtliche Methoden höchstens logarithmische Laufzeit in der aktuellen Größe der SweepLine haben. Aus der Beschreibung der benötigten Methoden wird erkennbar, warum die effiziente Implementierung der Klasse `SweepLine` nicht trivial ist. Einerseits gibt es Elementzugriffe wie `getPrevious` und `getNext`, die ideal mit einer verketteten Liste durchgeführt werden können, andererseits braucht man für `add` eine **Binärsuche**, die in einer verketteten Liste nicht effizient machbar ist.
Als Lösung versuchen wir deshalb sämtliche Operationen in Java mit Hilfe der Klasse `TreeSet` zu realisieren. Da der dazu erforderliche Vergleichsoperator auf Segmenten auch die aktuelle Position der SweepLine berücksichtigen muss, definieren wir ihn in einer separaten Klasse `SegmentComparator`, die die aktuelle Position der SweepLine in einer lokalen Membervariable `currentTime` speichert.

```
class SegmentComparator<S> implements Comparator<S>
{private double currentTime;
 public void set(double time) {currentTime=time;}
 public int compare(S a, S b)
 {Segment sa=(Segment)a;
  Segment sb=(Segment)b;
  double ma=(sa.q.y-sa.p.y)/(sa.q.x-sa.p.x);
  double mb=(sb.q.y-sb.p.y)/(sb.q.x-sb.p.x);
  double ya= sa.p.y + (currentTime-sa.p.x)*ma;
  double yb= sb.p.y + (currentTime-sb.p.x)*mb;
  if (ya<yb) return -1;
  if (ya>yb) return +1;
  if (ma<mb) return -1;
  if (ma>mb) return +1;
  return 0;
 }
}
```

Die Klasse `SweepLine` besitzt dann eine Membervariable `currentX` mit der aktuellen Position der SweepLine, sowie eine Variable `sl` der Klasse `TreeSet`, die die Segmente der SweepLine aufnimmt.

```
class SweepLine
    {private double lastSmallerX;               // vorige kleinere x-Position
     private double currentX;                   // aktuelle x-Position
     private SegmentComparator<Segment> sc;     // Segmentvergleiche
     private SortedSet<Segment> sl;             // aktuelle Segmente

     SweepLine(double start)
     {lastSmallerX=start;
      sc = new SegmentComparator<Segment>();
      sl = new TreeSet<Segment>(sc);
     }
```

In der Membervariablen `lastSmallerX` soll die jeweils vorherige Position der SweepLine gespeichert werden. Die Aktualisierung geschieht in der Methode `SetX`.

```
    void setX(double x)
     {if (currentX<x)    // Verschiebungen nach links sind unzulässig
        {lastSmallerX=currentX;
         currentX=x;
        }
     }
```

Sofern die Position der SweepLine richtig eingestellt ist, kann das Einfügen und

Entfernen von Segmenten einfach über den `TreeSet sl` durchgeführt werden.

```
    void add(Segment s) {sl.add(s);}
    void remove(Segment s) {sl.remove(s);}
```

Schwieriger ist das Vertauschen von Segmenten, das jeweils am Schnittpunkt der Segmente aufgerufen wird. Es kann dadurch realisiert werden, dass die SweepLine zunächst (intern) leicht nach links verschoben wird, um eine eindeutige Reihenfolge der Segmente zu garantieren. Dann werden die Segmente aus dem Baum entfernt und schließlich etwas rechts vom Schnittpunkt wieder in die SweepLine eingefügt.

```
    void actualize(Segment a, Segment b)
     {sc.set((currentX+lastSmallerX)/2);   // Sweepline nach links schieben
      sl.remove(a);                        // Segmente entfernen
      sl.remove(b);
      sc.set(currentX+Math.ulp(currentX)); // Sweepline nach rechts schieben
      sl.add(a);                           // Segmente wieder einfügen
      sl.add(b);
      sc.set(currentX);                    // Sweepline wieder korrekt einsetzen
     }
```

Schließlich müssen noch Nachbarsegmente verfügbar gemacht werden. Das lässt sich über die Methoden `headSet` und `tailSet` erreichen.

```
    Segment getPrevious(Segment s)
     {sc.set((currentX+lastSmallerX)/2); // Sweepline etwas nach links schieben
      SortedSet<Segment> help=sl.headSet(s);
      Segment result=null;
      if (help.size()>0)
        result=help.last();
      sc.set(currentX);                  // Sweepline wieder korrekt einsetzen
      return result;
     }
    Segment getNext(Segment s)
     {sc.set((currentX+lastSmallerX)/2); // Sweepline etwas nach links schieben
      Segment result=null;
      SortedSet<Segment> help=sl.tailSet(s);
      if (help.size()>1)
        {Iterator<Segment> its=help.iterator();
         if (its.hasNext()) its.next();         // s überspringen
         if (its.hasNext()) result=its.next();
        }
      sc.set(currentX);                  // Sweepline wieder korrekt einsetzen
      return result;
     }
}
```

Damit ist die Klasse `SweepLine` vollständig definiert.

10 Die Klasse BigInteger

Zum Rechnen mit extrem großen Zahlen gibt es im Paket `java.math` die Klassen `BigInteger` für ganze Zahlen und `BigDecimal` für Fließkommazahlen. Zum Erzeugen von Objekten übergibt man dem Konstruktor die gewünschte Zahl in Form eines Strings und kann dann mit Methoden `add`, `subtract`, `multiply`, `divide`, `remainder`, ... arithmetische Operationen ausführen.
Statt einer genauen Definition der Klasse betrachten wir nur ein paar einfache Anwendungsbeispiele. Zunächst sollen die Fakultäten der Zahlen von 1 bis 60 ausgegeben werden.

```
BigInteger eins= new BigInteger("1");
BigInteger a= new BigInteger("1");
BigInteger result=new BigInteger("1");
for (int i=0; i<60; ++i)
   {result=result.multiply(a);
    System.out.println("a=" + a + "  a!="+ result);
    a=a.add(eins);
   }
```

Das Ergebnis sieht dann folgendermaßen aus:

```
a=1  a!=1
a=2  a!=2
a=3  a!=6
a=4  a!=24
a=5  a!=120
a=6  a!=720
...
a=55  a!=12696403353658275925965100847566516959580321051449436762275840000000000000
a=56  a!=710998587804863451854045647463724949736497978881168458687447040000000000000
a=57  a!=40526919504877216755680601905432322134980384796226602145184481280000000000000
a=58  a!=2350561331282878571829474910515074683828862318181142924420699914240000000000000
a=59  a!=138683118545689835737939019720389406345902876772687432540821294940160000000000000
a=60  a!=8320987112741390144276341183223364380754172606361245952449277696409600000000000000
```

Als zweites Argument kann man beim Konstruktor auch eine gewünschte Zahlenbasis angeben. Beispielsweise kann man folgendermaßen Zweierpotenzen erzeugen.

```
BigInteger x= new BigInteger("10000000000", 2);  // 2^10
System.out.println(x); // 1024
```

11 Datei- Ein/Ausgabe

11.1 Ausgabe (Character-Stream)

Für die sequentielle Ausgabe von Daten gibt es im Paket `java.io` die abstrakte Klasse `Writer` mit folgenden Methoden

`Writer()`: Konstruktor zum Öffnen des Ausgabestroms
`abstract void close()`: schließen des Ausgabestroms
`abstract void flush()`: leeren des Ausgabpuffers
`void write(int c)`: schreibt ein Zeichen
`void write(char[] cbuf)`: schreibt ein Feld mit Zeichen
`void write(String str)`: schreibt einen String
`abstract void write(char[] cbuf, int off, int len)`:
schreibt `len` Zeichen des Feldes `cbuf` ab Offset `off`
`void write(String str, int off, int len)`:
schreibt `len` Zeichen des Strings `str` ab Offset `off`

Zumindest die Methoden `write(char[] cbuf, int off, int len)`, `close()` und `void flush()` müssen von allen abgeleiteten Klassen implementiert werden. Zu den vorgegebenen abgeleiteten Klassen gehören unter anderem `PrintWriter` zur Ausgabe aller Typen im Textformat, `StringWriter` zum Schreiben in einen String, `FileWriter` zur Ausgabe in eine Datei und `BufferedWriter` für eine gepufferte Ausgabe. Im folgenden soll an einem kleinen Beispiel gezeigt werden, wie man Daten in eine Datei schreiben kann.

```
import java.io.*;
public class Beispiel
  {public static void main(String[] args)
    {FileWriter f=null;
     try {
          f= new FileWriter("listing");
          f.write("Die ersten 10 Qudratzahlen:");
          for (int i=1; i<=10; ++i)
             f.write(" "+ i*i);
          f.write("\r\n");
          f.write("Ende des Beispiels\r\n");
          f.close();
         }
     catch (IOException e)
         {
          System.out.println("Fehler bei der Ausgabe");
         }
    }
  }
```

Aus Effizienzgründen sollte man jedoch nicht bei jeder Ausgabe auf die Datei zugreifen, sondern die Ausgaben zunächst puffern. Dazu verwendet man ein Objekt der Klasse `BufferedWriter`, das seinerseits die Daten an einen `FileWriter` weiterleitet.

```
...
Writer f1;
BufferedWriter f2;
...
f1= new FileWriter("listing");
f2= new BufferedWriter(f1);
...
f2.write("Die ersten 10 Qudratzahlen:");
...
f2.close();
f1.close();
```

Um die umständliche Unterscheidung zwischen den beiden Objekten `f1` und `f2` zu vermeiden, definiert man den Writer besser namenlos beim Aufruf des Konstruktors.

```
import java.io.*;
public class Beispiel
  {public static void main(String[] args)
    {BufferedWriter f=null;
     try {
          f= new BufferedWriter(new FileWriter("listing"));
          f.write("Die ersten 10 Qudratzahlen:");
          for (int i=1; i<=10; ++i)
             f.write(" "+ i*i);
          f.write("\r\n");
          f.write("Ende des Beispiels\r\n");
          f.close();
         }
     catch (IOException e)
         {
          System.out.println("Fehler bei der Ausgabe");
         }
    }
  }
```

In den bisherigen Beispielen wurde ausgenutzt, dass die auszugebenden Daten mit Hilfe des Konkatenationsoperators `+` und der Methode `toString` in Strings konvertiert wurden. Als Alternative kann man auch einen `PrintWriter` verwenden, der die primitiven Datentypen mit Hilfe spezieller `print`- und `println`-Methoden in Strings konvertiert an den BufferedWriter weitergeben kann. Die typische Definition lautet dann

```
PrintWriter f= new PrintWriter(new BufferedWriter(new FileWriter("listing")));
```

Beispielanwendung:

```
for (int i=1; i<10; ++i)
  {f.print("i=");
   f.print(i);
   f.print(" 1.0/i=");
   f.println(1.0/i);
  }
```

11.2 Eingabe (Character-Stream)

Entsprechend der abstrakten Klasse `Writer` für die Ausgabe gibt es für die Eingabe die abstrakte Klasse `Reader` mit folgenden Methoden.

`Reader()`: Konstruktor zum Öffnen des Eingabestroms

`int read()`: liest das nächste Zeichen aus dem Eingabestrom und gibt es als `int`-Wert im Bereich $[0, 65535]$ zurück. Ist das Ende des Eingabestroms erreicht, erhält man -1.

`int read(char[] cbuf)`: liest Zeichen in das Feld `cbuf` ein und liefert die Anzahl gelesener Zeichen bzw. -1 am Ende des Eingabestroms.

`abstract int read(char[] cbuf, int off, int len)`: liest Zeichen in ein Teilfeld von `cbuf` ein und liefert die Anzahl gelesener Zeichen bzw. -1 am Ende des Eingabestroms.

`long skip(long n)`: überspringt `n` Zeichen im Eingabestrom (blockierend)

`boolean ready()`: liefert true, falls ein weiteres Zeichen (ohne Blockierung) gelesen werden kann.

`abstract void close()`: schließen des Eingabestroms.

`void mark(int ReadAheadlimit)`: markieren einer Stelle im Eingabstrom, um später von dort nach `reset` erneut zu lesen. Der Parameter `ReadAheadlimit` gibt an, nach wievielen gelesenen Zeichen das Zurückpositionieren noch möglich sein soll.

`boolean markSupported()`: prüft, ob das Markieren vom Eingabestrom unterstützt wird.

`void reset()`: positioniert im Eingabestrom auf die letzte mit `mark` markierte Stelle zurück (falls möglich).

Zumindest die Methoden `read(char[] cbuf, int off, int len)` und `close()` müssen von allen abgeleiteten Klassen implementiert werden. Zu den vorgegebenen abgeleiteten Klassen gehören unter anderem `FileReader` zum Einlesen aus einer Datei, `StringReader` zum Lesen aus einem String und `BufferedReader` zur gepufferten Eingabe und zum Lesen einer ganzen Eingabezeile. Weiter gibt es die Klasse `PushbackReader`, bei der bereits gelesene Zeichen wieder in den Eingabestrom zurückgegeben werden können, um sie erneut zu lesen. Im folgenden soll an einem kleinen Beispiel gezeigt werden, wie man Daten aus einer Datei lesen kann.

```
import java.io.*;
public class Beispiel
  {public static void main(String[] args)
    {FileReader f=null;
     int c;
     try {
          f= new FileReader("/etc/hosts");
          while((c=f.read()) != -1)
            System.out.print((char)c);
          f.close();
         }
     catch (IOException e)
         {
          System.out.println("Fehler beim Lesen der Datei");
         }
    }
  }
```

Wie auch bereits bei der Ausgabe festgestellt, sollte man aus Effizienzgründen nicht zeichenweise auf ein Gerät zugreifen, sondern beispielsweise zeilenweise lesen. Dazu verwenden wir im folgenden Beispiel die Klasse `BufferedReader`.

```
import java.io.*;
public class Beispiel
  {public static void main(String[] args)
    {BufferedReader f=null;
     String line;
     try {
          f= new BufferedReader(new FileReader("/etc/hosts"));
          while((line=f.readLine()) != null)
            System.out.println(line);
          f.close();
         }
     catch (IOException e)
         {
          System.out.println("Fehler beim Lesen der Datei");
         }
    }
  }
```

Als weiteres Beispiel betrachten wir ein Programm, das alle in einer Textdatei vorkommenden Zahlen ausgeben soll, aber die sonstigen Textbestandteile ignorieren soll. Dazu schreiben wir eine Funktion `skipText(PushbackReader f)`, die solange zeichenweise liest, bis eine Ziffer gefunden wird. Dann wird die gelesene Ziffer mit der Methode `void unread(int c)` wieder in den Eingabestrom zurückgegeben.

Auf diese Weise kann anschließend die vollständige Zahl mit einer Hilfsfunktion `long read(PushbackReader f)` vom Eingabestrom gelesen werden. Das hinter der gelesenen Zahl stehende Zeichen wird wiederum zur erneuten Analyse in den Eingabestrom zurückgegeben.

Man beachte, dass diese Hilfsfunktionen so allgemein implementiert sind, dass sie sowohl aus einem String, als auch aus einer Datei lesen können. Sie sind also sehr vielseitig einsetzbar.

```
import java.io.*;
public class Beispiel
{
 static boolean isZiffer(char c)
  { return (c>='0')&&(c<='9');
  }
 static long read(PushbackReader f)
 throws IOException
  {// liest eine long Zahl von f
   long zahl=0;
   char c;
   int ci;
   do {ci=f.read();
       c=(char)ci;
       if (isZiffer(c))
         zahl = 10*zahl + (c-'0');
      }while (isZiffer(c));
   if (ci!=-1)
     f.unread(c);
   return zahl;
  }
 static int skipText(PushbackReader f)
 throws IOException
  {// überliest Zeichen aus f, bis eine Ziffer auftritt
   // und liefert das zuletzt gelesene Zeichen zurück
   char c;
   int ci;
   do {ci=f.read();
       c=(char)ci;
      } while ((ci!=-1)&& !isZiffer(c));
   if (ci!=-1)
     f.unread(c);  // gelesene Ziffer zurückgeben
   return ci;
  }
```

```
 public static void main(String[] args)
  {try {PushbackReader f= new PushbackReader(new FileReader("Beispieldatei.txt"));
        // PushbackReader f=new PushbackReader(new StringReader("Beispiel123"));
        int ci;
        do {ci=skipText(f);
            if (ci!=-1)
              {long x= read(f);
               System.out.println("\n Der Text enthält die Zahl "+ x);
              }
           } while (ci!=-1);
        f.close();
       }
   catch (IOException e)
       {
        System.out.println("Fehler beim Lesen der Datei");
       }
  }
}
```

Man beachte auch, dass die Hilfsfunktionen die beim Lesen eventuell geworfenen Exceptions nicht abfangen, sondern an den Aufrufer weiterreichen. Im vorliegenden Fall werden die Exceptions im Hauptprogramm behandelt.

11.3 Ein/Ausgabe von Byte-Streams

Bei der bisher verwendeten Ein/Ausgabe wurde immer mit Textdateien gearbeitet, d.h. die auszugebenden Daten wurden in "lesbare Darstellungen" konvertiert, bzw. es wurden solche Eingaben erwartet. Effizienter ist es natürlich, die Daten genauso zu speichern, wie sie programmintern im Speicher abgelegt sind. Da Java jedoch hardwareunabhängig funktionieren soll, geht man hier so vor, dass man sämtliche Daten in eindeutig definierte Bytefolgen konvertiert und diese auf einer Datei ablegt. Für diese Art der Ein/Ausgabe ist ebenfalls das Paket `java.io` zuständig.
Für die Ausgabe gibt es die Basisklasse `OutputStream` und für die Eingabe gibt es `InputStream`. Die `write`- und `read`-Methoden sind ähnlich, wie bei Character-Streams, aber statt dem Typ `char` wird immer `byte` verwendet.
Mit den abgeleiteten Klassen `DataInputStream` und `DataOutputStream` können alle **primitive Datentypen** gelesen und geschrieben werden. Noch mächtiger sind die Klassen `ObjectInputStream` und `ObjectOutputStream`, die zusätzlich zu den primitiven Datentypen auch beliebige (serialisierbare) Objekte verarbeiten können. Statt detailiert auf alle Klassen und Methoden einzugehen, betrachten wir im folgenden ein einfaches Anwendungsbeispiel.
Im ersten Teil des Programms werden die Größen `xa`, `xb`, `xv` und `xts` in die Datei `test.dat` geschrieben.

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

public class Beispiel
{
 public static void main(String[] args)
 {
  int xa=15;
  double xb=30.105;
  int[] xv= {12,13,14,15,16};

  TreeSet<String> xts= new TreeSet<String>();
  xts.add("Zitronen");
  xts.add("Birnen");
  xts.add("Äpfel");
  xts.add("Apfelsinen");
  xts.add("Nüsse");

  ObjectOutputStream fout=null;

  try {fout=new ObjectOutputStream(
            new FileOutputStream("test.dat"));
       fout.writeInt(xa);
       fout.writeDouble(xb);
       fout.writeInt(xv.length);
       for (Integer i:xv)
         fout.writeInt(i);
       fout.writeObject(xts);
       fout.close();
      }
  catch (IOException e)
  {System.out.println("Fehler beim Schreiben der Datei");
  }
```

Man beachte, dass das Schreiben beliebiger Objekte eine relativ komplexe Aufgabe ist. Dazu muss festgestellt werden, welche Membervariablen das Objekt hat und wie diese jeweils (rekursiv) geschrieben werden können. Insgesamt ergibt sich dabei eine serialisierte Darstellung des Objekts. Ein besonderes Problem stellt dabei die Behandlung von Verweisen auf andere Objekte dar. Insbesondere müssen zyklische Verweise beim Schreiben erkannt werden.

Im folgenden zweiten Programmteil werden die Größen `ya`, `yb`, `yv` und `yts` aus der Datei `test.dat` eingelesen.

```
  // Zurücklesen der geschriebenen Daten
  int ya=0;
  double yb=0;
  int[] yv=null;
  TreeSet<String> yts=null;
  ObjectInputStream fin=null;
  try {fin= new ObjectInputStream(
            new FileInputStream("test.dat"));
       ya= fin.readInt();
       yb= fin.readDouble();
       yv= new int[fin.readInt()];
       for (int i=0; i< yv.length; ++i)
          yv[i]=fin.readInt();
       yts=(TreeSet<String>)(fin.readObject());
       fin.close();
      }
  catch (IOException e)
      {System.out.println("Fehler beim Lesen der Datei");
      }
  catch (ClassNotFoundException e)
      { System.out.println("Unbekannte Klasse"+e);
      }
  System.out.println("ya=" + ya + " yb=" + yb);
  for (int i:yv)
     System.out.print(" "+ i);
  System.out.println();
  for (Object o:yts)
     System.out.print(" "+o);
  System.out.println();
 }
}
```

12 Pakete

Jede Klasse gehört zu einem **Paket**. Der vollständige Klassenname hat die Form *Paketname*`.`*Klassenname*, wobei der Paketname wiederum Punkte enthalten kann. Beispielsweise bezeichnet `java.math.BigInteger` die Klasse `BigInteger` im Paket `java.math`. Um die umständliche vollständige Namensangabe zu vermeiden, kann man mit Hilfe der `import`-Anweisung einzelne Klassen oder sämtliche Klassen eines Pakets bekanntmachen. Eine Ausnahme stellen die Klassen des Pakets `java.lang` dar. Sie sind automatisch bekannt.

```
import java.math.BigInteger;  // macht die Klasse BigInteger bekannt
import java.math.*;           // alle Klassen des Pakets java.math
```

Die Punkte innerhalb eines Paketnamens geben eine Verzeichnisstruktur wieder. `math` ist also ein Verzeichnis unterhalb von `java`. Die Wurzel des Verzeichnisbaums wird bei der Installation des Java-Systems automatisch festgelegt. Um auch eigene Pakete verwenden zu können, muss man mit Hilfe der Environmentvariablen `CLASSPATH` angeben, auf welchen Verzeichnissen nach zusätzlichen Paketen gesucht werden soll.
Will man ein eigenes Paket erzeugen, dann legt man für das Paket ein entsprechend benanntes Verzeichnis an, in dem man die Source-Dateien ablegt. Zusätzlich gibt man in jeder dieser Source-Dateien als ersten Befehl eine `package`-Anweisung an, um die Zugehörigkeit zu dem Paket zu beschreiben.
Fehlt die `package`-Anweisung, so wird die Datei standardmäßig dem Default-Paket zugeordnet. Da die Klassen des Default-Pakets auch ohne explizite `import`-Anweisung genutzt werden können, eignet sich diese Vorgehensweise insbesondere für kleinere Programmbeispiele.

> Genau dann kann eine Klasse eingebunden werden, wenn sie zum gleichen Paket gehört oder als `public` deklariert ist.

Die Suche nach Klassen ist in der Praxis recht einfach, da der Paketname den Verzeichnispfad darstellt und der Klassenname dem Dateinamen entspricht. Deshalb ist es wesentlich, dass es in jeder Source-Datei genau eine Klasse gibt, die das Attribut `public` trägt.
Aus dem gleichen Grund ist es nicht möglich, eine Klasse auf mehrere Dateien zu verteilen. Umgekehrt kann man zwar mehrere Klassen in einer Datei ablegen, aber dann ist nur eine davon von außen sichtbar. Die anderen können nur lokal verwendet werden. Beispielsweise waren im Kapitel 9 die Klassen `Event` und `SegmentComparator` nur innerhalb des Pakets nützlich. Da sie ohne das Attribut `public` deklariert wurden, waren sie nur innerhalb des Pakets verwendbar (Standard-Sichtbarkeit).

13 Archive

Zur Weitergabe eines Java-Programms muss man sämtliche benötigten Klassen in compilierter Form durch `.class`-Dateien zur Verfügung stellen. Dabei stellt sich jedoch das Problem, dass diese Dateien auf dem Zielrechner wieder in der ursprünglichen Verzeichnishierarchie gespeichert werden müssen, damit die Klassen vom Java-System gefunden werden können.
Soll etwa die Methode `main` der Klasse `Alpha` im Unterverzeichnis `v` des Pakets `progs` gestartet werden, benutzt man folgenden Aufruf.

```
java -cp progs progs.v.Alpha
```

Dabei definiert die Option `-cp progs` (oder `-classpath progs`) den lokal zu verwendenden Suchpfad. Diese Angabe überschreibt die Environmentvariable `CLASSPATH`.
Da es recht aufwändig sein kann die komplette Verzeichnistruktur eines Pakets auf dem Zielrechner nachzubilden, bietet sich als einfachere Alternative an, die komplette Verzeichnisstruktur in einem `jar`-Archiv zusammenzupacken.
Beispielsweise packt das folgende Kommando sämtliche im Verzeichnis `progs` und darunter vorhandenen Dateien in das Archiv `bsp.jar`. (Vorsicht: wenn die Quelldateien nicht ausgeliefert werden sollen, muss man sie vorher entfernen, oder einen (komplizierteren) `jar`-Aufruf benutzen, der nur die `*.class`-Dateien übernimmt.)

```
jar -cvf bsp.jar progs
```

Bindet man zusätzlich in das Archiv auch noch eine Information darüber ein, von welcher Klasse die `main`-Methode ausgeführt werden soll, wird das Archiv direkt ausführbar. Diese Information wird in einer Datei `manifest.mf` im Unterverzeichnis `meta-inf` des Archivs abgelegt. Im vorliegenden Beispiel muss diese Datei folgende Zeile (mit Zeilenvorschubzeichen am Ende) enthalten:

```
Main-Class: progs.v.Alpha
```

Erstellt man die Manifest-Datei beispielsweise mit einem Texteditor als Datei `manifest.txt`, so kann sie folgendermaßen mit Hilfe der `jar`-Option `-m` ins Archiv übernommen werden.

```
jar -cvmf manifest.txt bsp.jar progs
```

Die Ausführung des Programms geschieht dann mit dem Befehl

```
java -jar bsp.jar
```

14 Literatur

Eine zweistündige Vorlesung reicht bei weitem nicht aus, um die komplette Sprache Java und die verfügbaren Pakete kennenzulernen. Der Schwerpunkt des vorliegenden Skripts liegt auf der Darstellung der Objektorientierten Programmierung. Dabei wurde bewusst auf Spezialgebiete wie graphische Benutzeroberflächen, Datenbanken und verteilte Programmierung verzichtet. Diese Gebiete sollten in weiteren Vorlesungen behandelt werden.
Sowohl als Einführung in die Programmierung mit Java, als auch als weiterführende Literatur können folgende Bücher empfohlen werden.

Guido Krüger: *Handbuch der Java-Programmierung*, O'Reilly Verlag Köln, 2014,
ISBN 978-3-95561-514-7
(kostenlose HTML-Version unter `www.javabuch.de` erhältlich)

Ralf Kühnel: *Die Java 2 Fibel*, Addison-Wesley, 1999,
ISBN 3-8273-1410-0

Tutorials zur Sprache Java (Standard Edition) findet man im Internet z.B. unter
`http://docs.oracle.com/javase/tutorial/index.html`.

Der Direktzugriff auf die Dokumentation (API) akueller Java-Versionen ist unter folgenden Links zu finden
`http://docs.oracle.com/javase/6/docs/api/`
`http://docs.oracle.com/javase/7/docs/api/`
`http://docs.oracle.com/javase/8/docs/api/`